TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Department of Computer Engineering

Dissertation

GENERATION OF APPLICATION SPECIFIC HARDWARE EXTENSIONS FOR HYBRID ARCHITECTURES

The Development of PIRANHA – A GCC Plugin for High-Level-Synthesis

Submitted to the Faculty of Computer Science
in partial fulfillment of the requirements for the degree of
Doctor of Engineering (Dr.-Ing.)
at the
DRESDEN UNIVERSITY OF TECHNOLOGY

Author:	Gerald Hempel
Date of birth:	May 3, 1981
Place of birth:	Zittau
First Reviewer:	Prof. Dr.-Ing. Christian Hochberger Technische Universität Darmstadt
Second Reviewer:	Prof. Dr.-Ing. habil. Andreas Koch Technische Universität Darmstadt
Expert consultant:	Prof. Dr.-Ing. habil. Klaus Kabitzsch Technische Universität Dresden
Submitted on:	December 20th, 2017

Acknowledgment

An academic thesis is never the work of a single person, therefore I want to take this opportunity to thank all people who have contributed in various ways to the successful conclusion of my dissertation.

First and foremost, I would like to express my sincere gratitude to my PhD supervisor Professor Dr. Christian Hochberger, who has always helped me with his large wealth of knowledge and numerous constructive discussions. I would also like to thank him for the confidence and the freedom he has provided me throughout the development of my dissertation.

I would like to acknowledge the valuable input of Professor Koch and Professor Kabitzsch, who reviewed my work as second supervisor and expert consultant, respectively. Also, I would like to recognize the work of Professor Castrillon and Professor Baader, who participated in the examination committee.

A major part of the success of this dissertation is based on contributions from student's diploma and study theses. In particular Jan Hoyer and Michael Raitza have to be mentioned here, who implemented the foundations of my work. Also, I would like to thank Jonas Wielicki, who made an important contribution to the success of this dissertation with his work on the analysis of memory accesses. My special thanks also go to Johanna Rohde and Candy Lohse, who implemented particular synthesis optimizations. Moreover Andreas Wiese have to be mentioned here, who helped to complete the context of my dissertation with the integration of generated hardware accelerators into the infrastructure of modern operating systems.

This acknowledgment would not be complete without mentioning my research colleagues and staff members of the Department of Computer Engineering of the TU Dresden and TU Darmstadt, who contributed to this dissertation with numerous interesting debates and ideas. My special thanks go to my friend and colleague Markus Vogt, who read many chapters of my thesis and helped me to improve it technically and linguistically. I further thank him for the maintenance and provision of the tool infrastructure that made this dissertation possible.

The past few years have been stressful for my wife Susanne and my family, who always encouraged and supported me and my work at all stages. You have my special thanks.

CONTENTS

I Introduction **5**

1 Introduction **6**

1.1 Motivation 8

1.2 Proposed Work-Flow 10

1.3 Aims and Objectives 11

1.4 Thesis Outline 13

II Technical Background **15**

2 HLS for Configurable Systems **16**

2.1 Concepts of Hardware Generation for Digital Systems 17

2.2 C as Input Language for HLS 20

2.3 Concepts of HLS for Configurable SoCs 21

2.4 HLS Projects Using Special Languages 25

2.5 HLS Projects Using HDL-like Languages 27

2.6 HLS Projects Based on Object-Oriented Languages 28

2.7 HLS Projects Using OpenCL Based Languages 30

2.8 HLS Projects Using C or C++ 33

2.9 GCC PIRANHA 43

3 Target Platform **48**

3.1 Configurable Platforms 49

3.2 Soft-Core Processors 52

3.3 FPGA SoC 58

4 GCC Framework **61**

4.1 Compilation Flow 62

4.2 Intermediate Representation 65
4.3 Optimization Passes 72
4.4 GCC Plugin Interface 75

III Application Acceleration 79

5 Application Analysis 80
5.1 Function Data Collection 82
5.2 Loop Data Collection 82
5.3 Processing the Transcript File 85
5.4 Performance Estimation 88
5.5 Theory of Memory Access Analysis 92
5.6 Implementation of Memory-Access Analysis 102

6 Application Modification 115
6.1 Modifying the GIMPLE Structure 115
6.2 Accelerator Function 125
6.3 Runtime Alias Analysis 135
6.4 Generated Files 138
6.5 OS Integration of Accelerators 140
6.6 Base Address Assignment 141

7 High-Level Hardware Synthesis 144
7.1 Variants of Processor Customizations 146
7.2 From GIMPLE to HDL 146
7.3 Generation of the State Machine 149
7.4 Optimization Strategies 154

8 Generated Host Processor Interface 166
8.1 Parameter Interface 167

8.2 Memory Interface 173
8.3 FIFO Interface 177
8.4 Accelerator Address Map 180

IV Evaluation **182**

9 Evaluation **183**
9.1 Implementation Example 183
9.2 Evaluation of Memory Access Analysis 190
9.3 Benchmarks 195
9.4 Influence of Register-Allocation Strategies 197
9.5 Influence of HLS Optimizations 201

10 Conclusion **212**
10.1 Realization of Aims and Objectives 213
10.2 Limitations and Future Work 216
10.3 Summary 218

A Examples and Code Listings **220**
A.1 Alias Set Generation for Interleaved Structure Access 221
A.2 GIMPLE Example 222

B Platform **225**
B.1 SpartanMC Instruction Set 225
B.2 SpartanMC Pipeline 226

C GCC Plugin **228**
C.1 PIRANHA Parameters 229
C.2 Heuristic Delay Times for Operations 232
C.3 Generated Structures for Memory Analysis 232

C.4 GIMPLE Statements 233
C.5 Tree Types 234
C.6 Operations 236

D OS Integration **237**

E Evaluation Results **247**

E.1 Comparison of Register-Allocation Strategies 247
E.2 State Machine Evaluation 249
E.3 Performance Evaluation 253
E.4 Performance Estimation 254
E.5 Data Transfer Evaluation 255

List of Figures **256**

List of Tables **260**

List of Algorithms **261**

Part I

Introduction

1 INTRODUCTION

The ongoing digitalization of our everyday life entails several challenges for future computing systems. Particularly in the area of embedded systems there has been an increasing need for better computing performance along with improved energy efficiency. For a long time, this was achieved by downsizing the underlying semiconductor technology and increasing the clock frequency of circuits. Both approaches have recurrently been pushing the manufacturing process of such circuits to its technological limits. In recent years, the rise of circuit frequencies has stagnated due to the physical limits of current semiconductor technology. Since increasing the clock frequency is no longer an easy way to achieve performance improvements, the design paradigm has changed to real parallel systems implementing multiple-processor cores on a single chip. This has revealed new bottlenecks such as the communication and synchronization overhead of programs. Even worse, designers and tool developers are now forced to rethink the software design process as performance improvements are no longer achievable by simply executing the old program on the new architecture.

Nonetheless, the introduction of multi-core systems has been a success for the consumer market as well as for scientific applications. Since then, programs are being divided into different tasks running on dedicated processor cores. This helps with arising performance problems but also increases the energy consumption of the whole system.

General-purpose processors are able to solve nearly all kinds of computational problems, this often comes at the price of poor power efficiency. Therefore, embedded systems with strict energy constraints or an inefficient main processor often resort to additional hardware units that provide the best energy efficiency for a specific task. The next stage of evolution for such systems has been the customization of hardware to the particular needs of the program. This development has been strongly supported by the availability of low-cost reconfigurable devices. This introduces several new challenges for system designers. Reconfigurable systems are no longer based on static hardware. Thus, new design methodologies are required in order to unify the software and hardware design process.

This thesis presents a tool-flow that helps system designers to benefit from application-specific hardware accelerators without relying on special hardware knowledge.

1.1 Motivation

In recent decades, embedded systems have become a ubiquitous factor in nearly all aspects of our lives. The classical definition describes an embedded system as part of a larger technical system that is designed for one specific task [92, p. 1]. This definition is still true for many embedded systems that work within products designed for a single purpose for instance, the bending light control of a car. Typically, they are designed for one task or a fixed range of functionalities that do not change over the complete life cycle of the product.

Today's challenges emerge from the idea of closely meshed applications and service architectures often referred to as the *Internet of Things*, and will undoubtedly increase the need for embedded systems. However, the classical definition for such systems needs refining. On the one hand, the large number of different tasks and devices impedes the successful economical design of tailored systems for single tasks. On the other hand, especially small devices still require special hardware in order to meet real-time conditions or energy constraints. Moreover, tightly coupled devices cannot operate in a closed environment; they have to stay prepared for changes in the surrounding systems or even for deliberate attacks. It is not always desirable to replace all of them when security, communication or coding standards have changed. For this reason, the need for more flexible systems that can adapt their software and hardware will increase.

These challenges have partly been met by the design of system on chip (SoC) architectures. An SoC combines all functionalities required by the system on a single chip. A popular example is the Qualcomm Snapdragon architecture [111] used in current smartphones. It contains nearly all system components on a single die. These include, among others, a general-purpose processor, an H.265 MPEG decoder and a graphics processing unit (GPU). This allows its usage in a multi-purpose device where the actual requirements are specified by the end user. It saves the system designer from the tedious and error-prone process of

designing a system with individual components. This kind of flexibility, however, is not sufficient for some applications, as it does not allow for a change in the behavior of the hardware components, if necessary. For instance, the current H.265 video decoder implemented on the Snapdragon will be nearly useless as soon as the new standard H.266 is fully established. Additionally, for many companies, it is not economically feasible to design and manufacture such a complex SoC architecture[1].

A promising alternative to raise the degree of hardware flexibility is the use of field programmable gate arrays (FPGA). These enable a change in the hardware even in a shipped system and involve a reasonable price for custom designs. In comparison to ASIC design, they reduce development efforts and time-to-market. At present, FPGAs are already the platform of choice for many embedded projects. However, classical FPGAs cannot handle the growing complexity of todays applications, as pure hardware layouts lack the flexibility and expressiveness of software designs. Hence, it is hardly surprising that, according to a Mentor Graphics functional verification study [107], more than 56% of FPGA designs contain at least one general-purpose processor. Combining both, processors and reconfigurable logic, has the potential to enable the design of complex software in perfect interaction with custom hardware. Besides the gained flexibility, this also has the potential to improve the platform's energy efficiency as well as the execution speed of the software.

Despite that, the price for this benefit is a complex and time-consuming design-flow that forces the designer to implement the system on a degree of abstraction that requires comprehensive knowledge of the underlying hardware. On these grounds, many developers flinch from using reconfigurable logic.

Consequently, today's challenge is the seamless and transparent integration of the hardware design-flow into software development.

[1] The mask costs of an application-specific integrated circuit (ASIC) based on 20 nm technology amount to around five to eight million US dollars [86].

1.2 Proposed Work-Flow

In order to bring the hardware design-flow as close as possible to typical software design-flows, the integration of the hardware generation into a full-featured software compiler is a natural choice. The chosen compiler environment for this thesis was the Gnu Compiler Collection (GCC) framework. This compiler is widely used for industrial applications, provides a rich set of optimization strategies, and supports several embedded platforms. Moreover, it is an open source project that allows modifications and custom extensions by default.

The basic idea of this thesis is to integrate hardware generation into the GCC compilation process in order to make it transparent to the developer. There should be no major differences in comparison to compiling a regular software project. Figure 1.1 outlines the proposed work-flow. The input for the compilation is a plain C application. The following software compilation is carried out by the GCC while the hardware generation is handled by an additional GCC-plugin. The integration of the plugin is implemented in such a way that the hardware generation benefits from most optimizations provided by the GCC software compilation flow. Finally, the result is divided into two parts: the application binary and several hardware description files. The latter require further processing by platform-specific tools in order to generate a bit-file for the programmable logic. The proposed target machine consists of a fixed part containing a general-purpose processor and a configurable part containing the generated logic. The generated logic acts as a special peripheral unit executing suitable hotspots of the software application. Such peripheral units will be later referred to as *accelerators*.

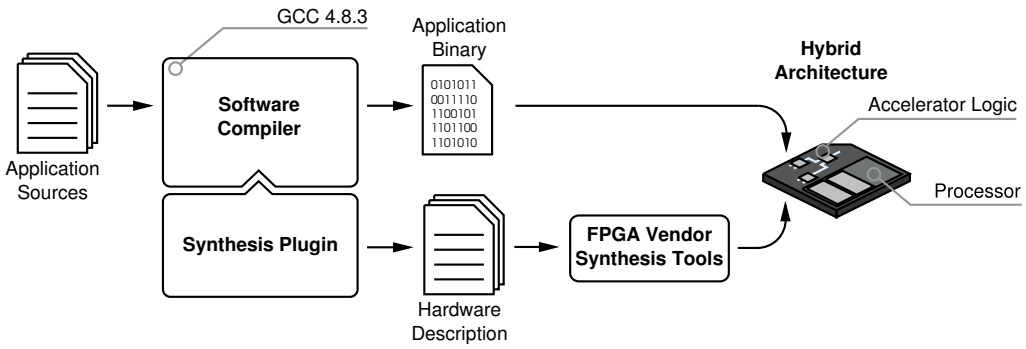

Figure 1.1: Operating principle of the hybrid hardware/software work-flow

The general-purpose processor can be implemented in two ways. First, it is possible to use a classical FPGA programmed with a hardware design that complies with the specification of a processor. This is also called a soft-core processor. Although the soft-core is quasi-static from the perspective of accelerator generation, the complete design is still customizable. This allows the adaptation of critical system components for instance, the peripheral interface to fit seamlessly with the generated accelerators.

Second, it is possible to use special platforms that already contain a processor core in silicon in combination with programmable logic. Such ASIC-like system components are called hard-cores. Even though such processor platforms are less flexible, they provide the typical advantages of highly integrated circuit designs for example excellent power efficiency and a high clock frequency. The combination of hard processor cores, typically with a rich set of standard peripherals, and an FPGA fabric is also referred to as FPGA-SoC.

In this thesis, the term "hybrid architecture" comprises FPGA-SoCs as well as FPGAs containing soft-core processors.

1.3 Aims and Objectives

At present, the standard way of creating hardware accelerators is to develop them manually. This process is time-consuming and error-prone.

The goal of this thesis is to implement a tool-flow that helps the developer to create hardware accelerators automatically, starting from a high-level software description. The tool-flow should provide the following features:

- The possibility to handle plain C as input language. The target C code should require neither refinements nor special annotations. This condition lowers the entry barrier for non-hardware developers considerably and further enables support for legacy application code.
- Seamless integration into a standard software compiler, namely the GCC. This should allow developers to take advantage of optimizations in the software compilation flow for the generation of hardware. Reuse of the GCC front-end should further reduce the effort for syntax analysis and code parsing.

Furthermore, the usage of the GCC helps to provide a transparent tool-flow for most software developers.

- Automatic selection of promising code sequences. It should not be possible or necessary to map the whole application code to hardware. Code sequences containing promising application hotspots should be extracted automatically. There is no need for manual code annotations or special language extensions in order to select parts of the application.

- Automatic patching of the software in order to call generated accelerators. The invocation should remain completely transparent to the user. It should also include an optional fall-back to software execution.

- Automatic generation of a memory and a host-processor interface.

- Portability to other processor architectures. This means a modular interface concept and a platform-agnostic generation process for hardware accelerators are required.

The generation of hardware accelerators from unmodified C code poses many challenges for an automatic tool-flow approach. Most likely, some of these challenges can never be overcome due to the lack of hardware-specific information within the C language definition. However, the idea of this work is to consider only those code sequences that provide sufficient information for a good hardware generation. This comprises two assumptions.

First, there must be enough code sections in legacy C code that comply with the requirements for hardware generation. Second, the static code analysis during program compilation provides sufficient information for accelerator generation. Note that these two conditions are not completely independent of each other. A good static code analysis allows identifying more suitable code sections, whereas a suitable initial application code reduces the effort for the required code analysis and hardware generation. This leads to one key question: Which improvements in the application performance can be achieved with a static code analysis for a given application and target platform?

1.4 Thesis Outline

This thesis is divided into four parts. After Part I introducing the work, Part II which encompasses Chapters 2 to 4, gives an overview of the technical background. Part III, comprising Chapters 5 to 8, describes substantial parts of the presented tool-flow and gives detailed information about its implementation. Finally, the results of this thesis are evaluated and summarized in Part IV.

Part II starts with the second chapter, which emphasizes the challenges and opportunities for system development using high-level-synthesis (HLS). Hence, the impact of different input languages for HLS is discussed and an overview of existing HLS approaches for configurable platforms are given. Finally, the HLS compiler plugin that forms the basis of this thesis is introduced.

The first part of Chapter 2 discusses possible hardware architectures for the proposed work-flow. The second part will introduce the hardware platforms that were actually used in the context of this work. This includes an explanation of the features and architecture of the soft-core processor SpartanMC and the commercial Xilinx Zynq-7000 platform.

Chapter 4 gives an overview on the GCC framework. The GCC is the underlying compiler for the proposed HLS plugin. Consequently, most of the used structures and programming paradigms are tightly coupled to the software architecture of the GCC. This chapter will give an overview of the compilation flow itself, as well as on the intermediate representation and the optimization used by the compiler. Finally, the plugin interface of the used GCC version is introduced.

Part III, comprising the following four chapters, describes the different steps that are necessary to generate hardware accelerators for the given target platforms.

Chapter 5 explains of the application analysis to determine parts of the application that are suitable for acceleration. The selection of application hotspots is explained as well as different methods to determine the presumable efficiency of the extracted accelerator at compile time.

After the selection of hotspots, the application must be modified in order to call the generated hardware accelerators. This problem is tackled in Chapter 6. First, techniques for modifying the intermediate representation are explained. Afterwards, a description of the integration of accelerators in the software application

with respect to different target architectures is given. The chapter closes by introducing methods for the integration of generated accelerators into a full-featured operating system.

Chapter 7 presents the algorithms required for the high-level-synthesis of accelerators at compile time. It describes the transformation of an application hotspot from the internal compiler representation into the hardware description of a finite-state machine (FSM).

Chapter 8 presents three ways of interfacing with the respective host processor of the given target platforms. The interfaces will be discussed in the context of the host processor architecture and the underlying software environment.

Finally, Part IV concludes this thesis. The presented approaches are evaluated in Chapter 9, starting with a case study to show the operability of the presented work-flow. Additionally, the effectiveness of the application analysis is investigated using a considerable set of benchmark applications. Finally, the overall performance is evaluated for both target architectures.

Chapter 10 discusses the achievements of the presented thesis. The conclusion also addresses limitations of the current approach and gives recommendations for future improvements.

Part II

Technical Background

2 HLS FOR CONFIGURABLE SYSTEMS

2.1 Concepts of Hardware Generation for Digital Systems

In this thesis, the generation of hardware for digital systems means the mapping of an abstract algorithmic problem to the specific physical structure of a circuit. In the early years of circuit design this was done by utilizing elements of the target platform typically on the granularity of logic gates or even single transistors. This approach is no longer feasible for todays chip technology. In order to raise the level of abstraction, nowadays designers strive for behavioral descriptions of the desired algorithm without any reference to the structures of the underlying target platform.

A well-known abstraction model for circuit design is the Y-chart shown in Figure 2.1, which was introduced by Gajski et al. [98] and refined by Walker et al. [118]. This chart allows visualizing design views as well as design hierarchies. The characteristic Y-form arises from three radial axes showing three different design domains: *behavioral description*, *logical structure* and *physical structure*.

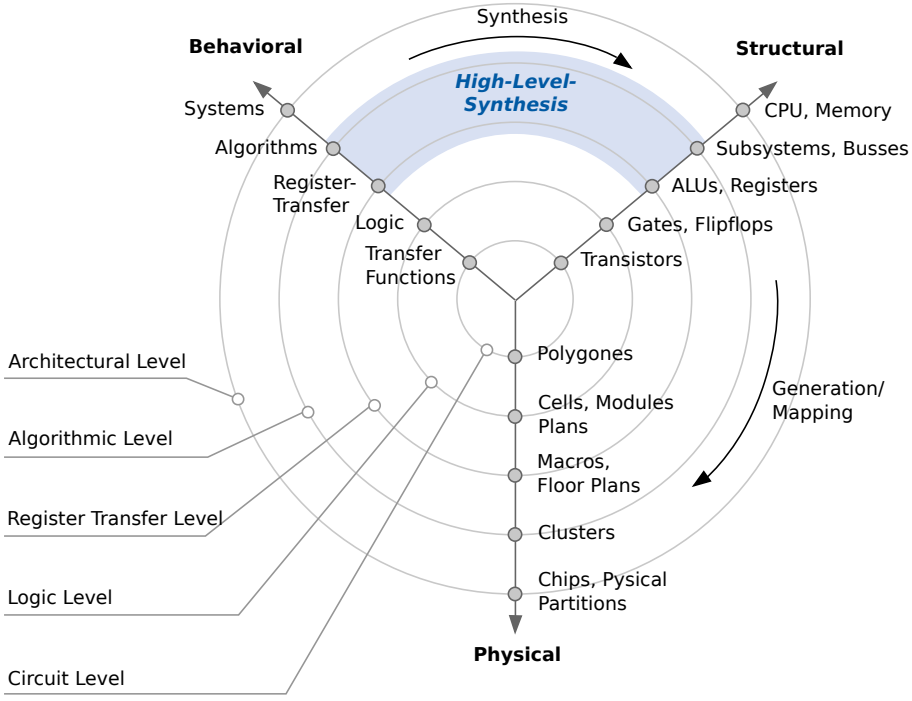

Figure 2.1: Gajski and Walker Y-chart showing high-level-synthesis

Five concentric circles represent the hierarchical levels of abstraction within the design process. The abstraction level increases from the innermost to the outermost circle. Within this chart a system can be specified for all three domains by using the following abstraction levels:

Architectural Level Abstract behavioral specification of system requirements and its basic concepts.

Algorithmic Level Functional description of subsystems and their interaction.

Register-Transfer Level Detailed description of data-flows and transformations on register level.

Logic Level Specification of the design on the level of logic gates and flip flops.

Circuit Level Specification of the system on the level of basic physical elements. This level can be used to implement the actual hardware device.

The process of circuit design is modeled by specializing the problem description by moving from the outermost abstraction level to the center of the chart. In reality, this process often requires transition between domains. The transition from the behavioral to the structural domain is called synthesis.

2.1.1 High-level-Synthesis

The synthesis steps performing the specialization from algorithmic level to register-transfer level are of particular interest within this thesis. This part is called high-level-synthesis. According to [115], the main task of HLS is the decomposition of a given behavioral or algorithmic description into a datapath and control-flow. The description can be either a program written in a high-level language or an abstract behavioral model, as provided by several tools e.g. MatLab [54, 30] and LabView [23]. An abstract work-flow is given in Figure 2.2 and consists of the following four tasks:

Compilation Translating the behavioral/algorithmic description into a formal model, for instance, a control data-flow graph (CDFG).

Allocation Identifying the quantity and requirements of hardware resources for the operations of the algorithm. In addition to functional units like arithmetic

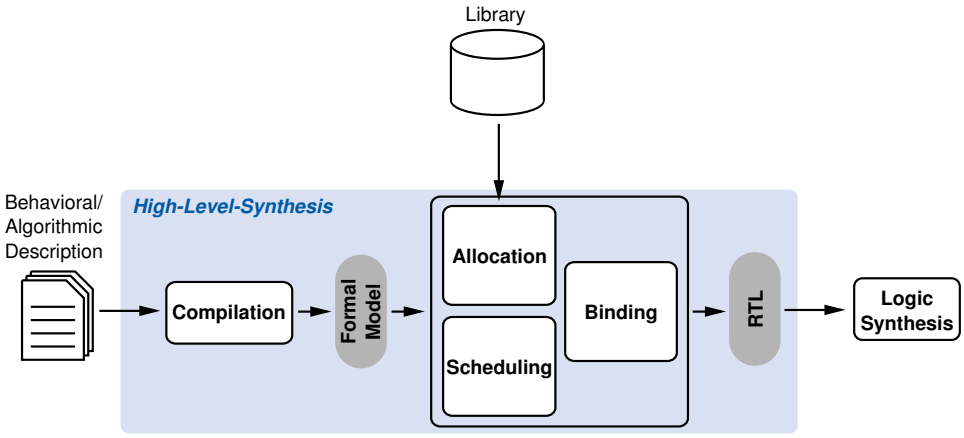

Figure 2.2: High-Level-Synthesis design steps

logic units (ALUs), this includes registers, memories, and communication resources. The allocation can be supported by libraries providing optimized functional units.

Scheduling Assigning all operations to a specific time interval with respect to given resource and timing constraints.

Binding Assigning all variables/data to corresponding memory resources (e.g. registers). Furthermore, operations are assigned to functional units, and data transfers to communication resources.

The usual result of an HLS tool is a problem description on register-transfer level, which is further synthesized to gate level by the logic synthesis. Typically, the register-transfer level is expressed in hardware description language (HDL).

While traditional HDLs, like VHDL [85] and Verilog [84], are good at describing detailed hardware-properties such as timing behavior, they are generally cumbersome in expressing higher-level abstractions. Hence, HDLs may not be the first choice to meet future productivity requirements for large and complex hardware designs. Furthermore, they are hard to use for most software programmers. Thus, it is worthwhile to focus on HLS tools that allow algorithmic descriptions using a much more abstract input language.

2.2 C as Input Language for HLS

There is an ongoing discussion in the research community about suitable input languages for HLS. First attempts extended established HDLs like Verilog and VHDL with behavioral language features. The resulting behavioral-HDLs [112, 94] were processed by the first commercial HLS tools, e.g. the Synopsys Behavioral Compiler [38].

Although those languages enable the desired higher level of abstraction, the acceptance from the user community is rather low. Consequently, the next step for HLS tools was the usage of a widely used input language. The obvious choice was C, which is accepted and frequently used for industrial and research projects.

Typical HDLs define a set of statements simultaneously taking effect after a defined trigger condition is met. Naturally, this is useful for hardware development, as it allows the description of parallel structures as well as strict timing behaviour. However, C is a procedural language optimized for serial execution on a single general-purpose processor. For usage in HLS, C has several stumbling blocks:

- The width of operators and operations is fixed and only defined by the used data type. This prevents the synthesis tool from generating registers and buses of arbitrary width.
- Some C constructs are not suitable for synthesis, especially code working on an unpredictable amount of resources, e.g. dynamic memory management or recursion.
- The lack of information about timing and input/output (IO) behavior requires special treatment of such code sections.
- Instruction-level parallelism (ILP) is not directly exposed. Typically, this is the first source of speedup for custom hardware; thus, it is mandatory to exploit such parallel operations. The second form of parallelism is thread-level parallelism (TLP). Naturally, threads are directly utilized by special library calls, which make them easy to find for synthesis tools. But in most cases the granularity of whole threads is not suitable for direct automatic hardware generation, as they usually contain long and complex code fragments. Nevertheless, TLP may be useful for hardware acceleration as it provides the

potential for real parallel code execution between processor and hardware accelerators.

- Arbitrary pointer access is a problem for most HLS tools. It prevents synthesis tools from determining a predictable pattern for memory accesses. This aggravates the typical memory bottleneck for hardware accelerators.
- One of the major challenges for C-based HLS is the lack of information about the aliasing of memory accesses (cf. Section 5.5). Aliasing is not explicitly exposed or prevented in C code as it is nearly irrelevant for serial execution. Unfortunately, it prevents all attempts of data prefetching or parallelization of memory accesses in HLS. Thus, aliasing is a major issue for HLS that requires special attention.

Some of these issues, e.g. the unknown operator width, result in slightly sub-optimal hardware. For this reason, they are ignored by most synthesis tools. Others, like recursion and dynamic memory management, are often suppressed by the design-rules of the HLS tool.

2.3 Concepts of HLS for Configurable SoCs

The following sections will give an impression of tool-flows that provide a unique methodology or have a major impact on the field of automatic HLS research. The overview is given with respect to the main objectives of this thesis. At the end of each survey a brief summary categorizes the approach by its input language, the proposed target architecture, its level of automation, and its design objective.

The design objective describes the ability of the proposed tool-flow to generate a consistent SoC containing both custom hardware and software parts. Therefore, a tool-flow requires the capability for an automated hardware/software co-design. In this thesis, three varieties of design objectives have been distinguished.

Single IP[1] **Mapping**: Tool-flows that are based on this approach are specialized to map the whole application or a well defined part of the application to a custom hardware design. It requires additional effort to integrate the resulting hardware into the surrounding system. Typically, such approaches

[1] IP stands for **I**ntellectual **P**roperty which denotes a reusable component of a circuit design

provide generic interface logic that must be adapted to the given hardware. The choice of suitable application hotspots is delegated to the developer. Both tasks require profound knowledge of the hardware platform.

Software Back-Delegation: The concept of software back-delegation is much more convenient for the developer, as it aims to generate complete SoCs with software and hardware parts. Besides the configurable platform, systems using this concept provide a general-purpose processor, which executes the program sections that failed to meet performance requirements or contain unsynthesizable program statements. Potentially required interfaces are generated automatically with respect to an underlying target platform. Having the opportunity to switch back to software enlarges the potential code base for accelerators but also introduces additional overhead for data transfers.

Hybrid Architecture: Similar to software back-delegation, this concept aims at platforms consisting of a general-purpose processor and a configurable part. The application basically runs on the software processor while only fully synthesizable hotspots are mapped to customized hardware units. In contrast to software back-delegation the resulting hardware accelerators might be smaller and the overhead of data-transfers, often a crucial point when aiming at speedups, can be minimized.

All tool-flows that were examined within this thesis are intended for the offline synthesis of hardware units. Besides these approaches, there are several other projects taking advantage of online hardware synthesis. Typically, they are based on coarse-grain reconfigurable architectures [10]. These architectures require very little configuration data and allow fast reconfiguration during application runtime. Though, the used algorithms and mechanisms are often similar, such projects [56, 49, 26] represent another research field and will not be discussed in this thesis.

2.3.1 Taxonomy of Existing HLS Tool-Flows

In the last 25 years various HLS work-flows with scientific and industrial backgrounds have been published. Usually, they follow different philosophies to

achieve their goals, even though the main objective namely, raising the level of abstraction for hardware development is identical. However, many work-flows arose from individual challenges. As a result, they provide a large variety of qualities that are not relevant in context of this thesis. Hence, several properties are not addressed in the following survey.

The used taxonomy considers five major attributes to distinguish the presented HLS approach from past and recent HLS work-flows:

Input Language: The language or behavioral description that is required to hand the algorithmic problem to the HLS tool. It is either an established high-level programming language or a special language typically adapting features from other languages. The latter is sometimes achieved by modifying a given language. In such cases the following survey gives a hint to the used enhancements or constraints.

Concept: The concept characterizes the scope covered by the proposed work-flow. Some work-flows are used to design a complete system containing both software and hardware parts while others are limited to pure hardware design. The distinguished design concepts *Single IP Mapping*, *Software Back-Delegation* and *Hybrid Architecture* were described in Section 2.3.

Flexibility: The flexibility describes the capability of the approach to address problems from different application domains. Typically, this attribute is strongly influenced by the static part[2] of the given target platform or the input language. Due to their respective architecture templates some approaches can not support certain interface types (e.g. streaming) or the language restricts the usage of language features (e.g. arbitrary pointers). Obviously, such constraints limit the flexibility.

- □□□ Almost a domain-specific language or architecture.
- ■□□ Usable for HW-design only or introduces major limitations.
- ■■□ HLS tool requires minor refining of programs.
- ■■■ All features of a high-level language are supported.

[2] The part of the hardware that is invariant to application-specific hardware customizations, e.g. buses, host processor or memory layout.

HW/SW Co-Design: The hardware (HW)/software (SW) co-design attribute describes the degree of automation provided by the tool to support the user with the selection of hardware accelerators. Obviously, this feature is not required for approaches that are based on the *Single IP Mapping* concept.

- ○○○ HW/SW co-design not in scope of the tool.
- ●○○ HLS tool allows manual selection of accelerators.
- ●●○ HLS tool supports the user by suggesting suitable code section.
- ●●● Selection of accelerators is completely transparent to the user.

Interface The interface attribute specifies the ability of the tool to automatically choose different interface types.

- ○○○ Interfaces must be implemented manually.
- ●○○ The tool provides a library of interfaces.
- ●●○ Interfaces were suggested after memory analysis.
- ●●● The selection of suitable interfaces is transparent to the user.

2.4 HLS Projects Using Special Languages

Due to the known issues with C for HLS, several research groups tried to find other ways to raise the productivity of hardware design. Some introduced completely new languages that circumvent arising problems with the existing ones. Others tried to adapt major languages with domain-specific constructs that potentially fit better with the needs of hardware development than C.

2.4.1 HLS Projects Based on Functional Languages

The idea of using a functional language to describe hardware is quite popular in academic HLS-projects [55, 41, 12]. It must be admitted that functional languages have some points in common with HDLs. For example, the model that describes combinatorial logic is a mathematic equation that is easy to decompose by using functional languages. Another important point is, that the order in which a functional language is evaluated does not matter as long as data dependencies are respected. For that reason, it is easy to exploit parallelism from such languages.

Haskell is a widely used functional language and the basis for several HLS-projects [12, 44, 9]. All these projects aim at a new way of describing hardware in order to supplement or even replace classical HDLs as input language. The principle of operation for those languages is to provide a library of Haskell functions and types. They form the language primitives of a domain specific language tailored to the hardware description.

2.4.2 CλaSH

A prominent example for the use of a functional language for hardware development is CλaSH [9], which stands for **C**AES (Computer Architecture for Embedded Systems) **La**nguage for **S**ynchron-ous **H**ardware. The CλaSH-compiler allows translating Haskell modules into VHDL. The modules have to be written in a very concise way using a hardware-oriented subset of Haskell. Thus, CλaSH tries to combine the best of two worlds. On the one hand, it provides the flexibility of a functional language comprising a template language for introducing new HDL primitives, it also provides the possibility to define *higher-order* functions for procedural hardware generation. On the other hand, CλaSH provides the low-level expressiveness of typical HDLs by enabling the description of bit-accurate hardware structures.

Hardware designs written in CλaSH are comparable to hand-optimized VHDL designs in terms of size and performance. The use of Haskell code allow a concise description of complex functions. However, pure functional languages are not widely accepted in the industry even though they are elegant. This is caused by the fact that such approaches typically provide ways for effective implementations of simple transformational algorithms but lack expressiveness or productivity for complex stateful algorithms or whole system descriptions.

CλaSH Summary		
Input Language:	*Haskell with libraries*	
Design Concept:	*Specialized IP core*	
Flexibility:	*Arbitrary hardware-design*	■■□
HW/SW CO-Design:	*No direct support for HW/SW co-design*	□□□
Interface:	*Arbitrary interfaces*	□□□

2.5 HLS Projects Using HDL-like Languages

Some approaches refine the design flow and structure of current HDL-languages. They borrow features from other programming languages and typically create a new one. Examples of this approach are SystemVerilog [2] and Bluespec SystemVerilog [46].

2.5.1 Bluespec SystemVerilog

While SystemVerilog extends Verilog with new language features, Bluespec SystemVerilog uses a fundamentally different approach. It reuses elements of functional languages, which thereby find their way into commercial tools. Bluespec Inc. was founded as a company in 2003 to market HLS tools based on the Bluespec System Verilog HLS language. The Bluespec language is strongly inspired by Haskell but also has adapted features and syntax that resembles Verilog.

On the one hand, it uses functional rules similar to Haskell and provides a strong type system that allows language features like polymorphism. On the other hand, it provides timing and control-flow structures similar to Verilog. An additional feature of Bluespec System Verilog is the special syntax for atomic transactions. It allows to describe complex control structures in an elegant and compact way, and ensures a precise and well-defined concept of chronological sequences. In this paradigm, hardware is described as a data-flow network consisting of atomic rules. State changes happen simultaneously when rules are fired. Parallelism is achieved through concurrent execution of non-conflicting rules.

Bluespec Sytem Verilog Summary		
Input Language:	*HDL-inspired Language*	
Design Concept:	*Specialized IP core*	
Flexibility:	*Arbitrary hardware-design*	■■□
HW/SW CO-Design:	*No direct support for HW/SW co-design*	□□□
Interface:	*Arbitrary interfaces*	□□□

2.6 HLS Projects Based on Object-Oriented Languages

There are several projects that translate high-level languages (other than C) into hardware. Examples are Kiwi [57] (based on C#) and Lime [8] (based on Java). In order to enable hardware-specific features, such languages typically provide a customized API to the developers. It allows describing a whole system in a hardware friendly but still object-oriented manner.

2.6.1 Liquid Metal and Lime

Besides the object-oriented features of Java, Lime programs expose parallelism and enable bit-level analysis. Programs written in Lime can use a restricted form of Java `enum` types that guarantee their immutability. Together with the qualifier `local`, Lime allows the isolation of computation and eases the mapping of code sections to different (hardware) units. Moreover, Lime provides keywords for abstract parallelism (`task`), streaming (`=>`), and data parallel constructs like map (`@`) and reduce (`!`).

Using the Lime language together with the Liquid Metal project [33] enables programmers to compile the same code into either pure software binaries or software/hardware co-designs. The proposed work-flow is shown in Figure 2.3.

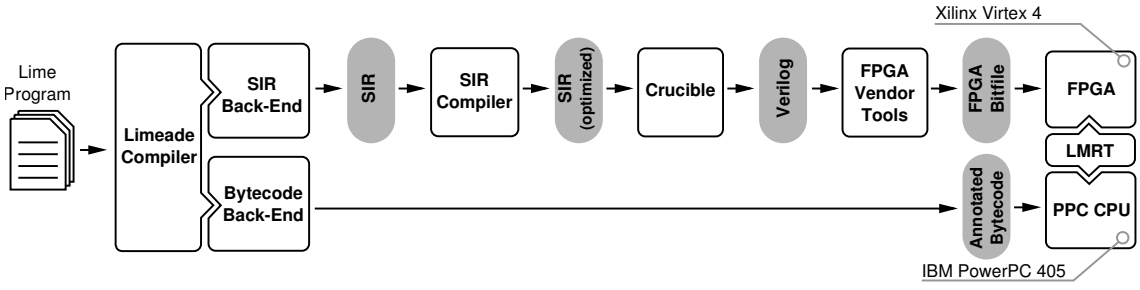

Figure 2.3: Liquid metal compilation and runtime system (derived from [33])

As Lime is basically Java code with some additional features, it can be flawlessly translated into Bytecode. In a first step, the Limeade compiler (cf. Figure 2.3) checks for compliance with the Lime coding-rules. These rules are required for hardware compilation in order to avoid problematic program structures for instance, object aliasing. Suitable parts of the program will be translated into a synchronous dataflow model that will be further optimized by the synchronous intermediate representation (SIR) compiler. Finally, the mapping into HDL is carried out by the Crucible compiler. The resulting code runs on an FPGA-SoC[3]. Depending on the specification in the Lime code, a tailored interface to transfer data between the hardware and software part of the application is generated. It is utilized by the Liquid Metal runtime (LMRT) in the resulting system. Consequently, the communication with the hardware accelerator is transparent for the system designer as long as the used communication pattern is implemented in the LMRT.

Liquid Metal with Lime Summary		
Input Language:	*Java with extensions*	
Design Concept:	*Heterogeneous architecture (CPU,GPU,FPGA)*	
Flexibility:	*Plain Java but requires special keywords and coding rules for hardware parts*	■■□
HW/SW CO-Design:	*Lime-conform code sections determine future hardware parts*	■□□
Interface:	*Runtime environment using various interfaces*	■■■

[3]The configurable platform used in [33] is a Virtex 4 FPGA. The software runs on an embedded IBM PowerPC 405.

2.7 HLS Projects Using OpenCL Based Languages

Open Computing Language (OpenCL) is an open industry standard for programming heterogeneous and parallel computing platforms such as GPUs, DSPs, and FPGAs. Although OpenCL is largely used for GPU programming, the design paradigm is similar to all HLS design flows targeting streaming problems for FPGAs namely HLS projects based on Impulse-C [61] or CUDA [29, 48].

2.7.1 Altera OpenCL

One of the most popular examples for a commercial tool-flow using OpenCL [35, 22, 21] was created by Altera. Programs using this language demand a hierarchical memory layout similar to the hardware layout of GPUs. Although the program is mapped to an FPGA, many parts of the architecture are generated from fixed modules. Altera OpenCL relies on the template FPGA architecture sketched in Figure 2.4. It consists of modules that have been manually optimized for the underlying reconfigurable architecture. In particular, performance-critical parts, for instance, the external memory interface benefit from this strategy. The only custom-generated parts of the design are the kernel pipelines, which contain the datapath and the control unit of the computational kernels.

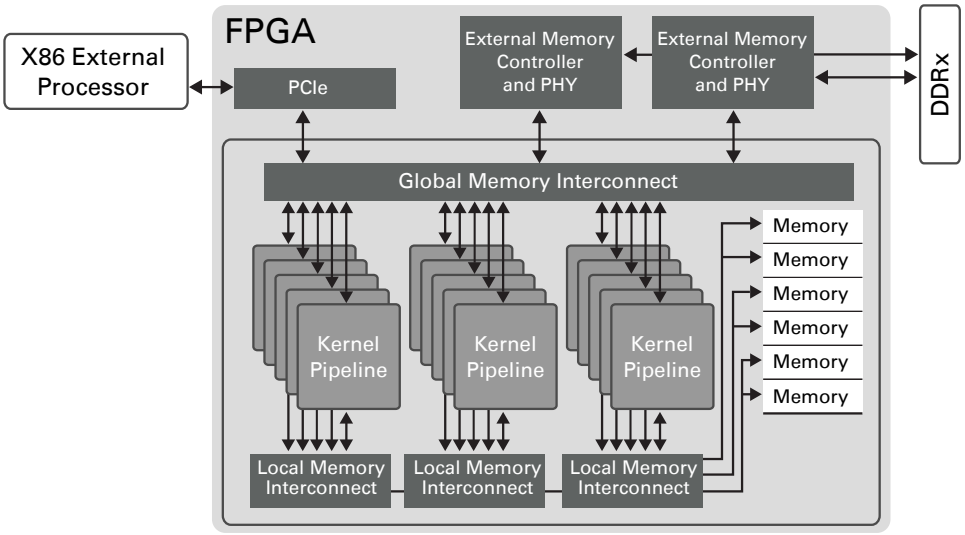

Figure 2.4: Altera OpenCL hardware architecture [7]

Virtually, the proposed work-flow (Figure 2.5) starts with two separate programs. First, is the classical C/C++ part that is running on the host machine (mostly an x86 processor). It is responsible for queuing and transferring data to the OpenCL computing device. Second, is the OpenCL-C part that is running on the programmable hardware. Only the second part requires the special HLS tool-flow that is provided by the Altera OpenCL compiler. Hence, the software/hardware partitioning is inherent to the code structure containing dedicated parts for the host processor and the configurable hardware. In order to introduce existing hardware kernels to the host compiler, the auto-discovery module is used. It provides information about the kernel itself and its interfaces.

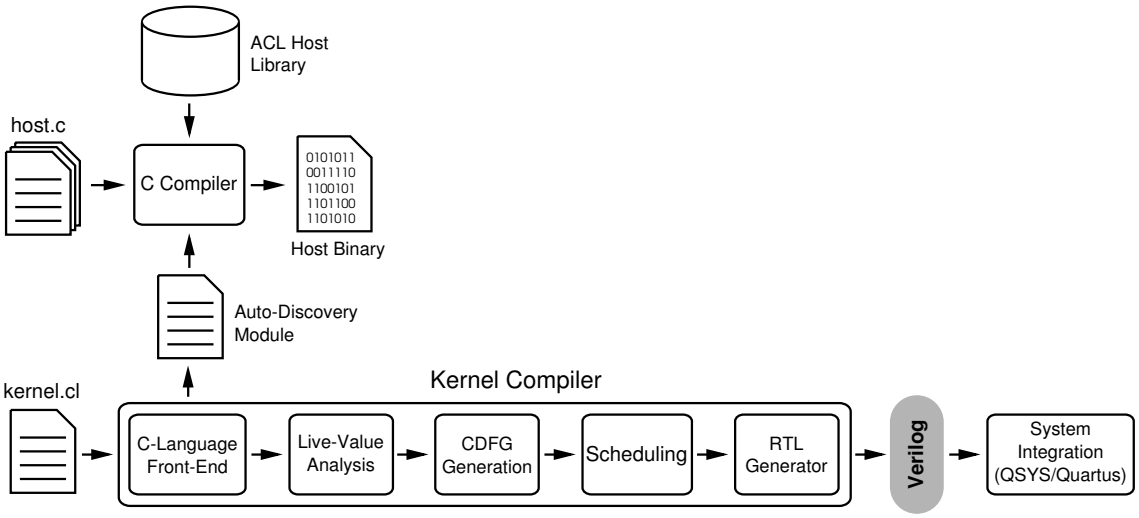

Figure 2.5: Altera OpenCL tool-flow [22]

As described in [7], the OpenCL code itself is a subset of the ISO C99 standard with additional support for parallelism. On the one hand, it provides extensions for special data types, e.g. vectors or 2D/3D work items, on the other hand it prohibits function-pointers, recursion and dynamic arrays. A special feature of OpenCL is the hierarchical memory structure that is reflected by the C code through special key words for variable declaration (e.g. `__host`, `__global`, `__local`). Together with the special data types, this enables data locality and compensates for the lack of information about memory accesses.

The feasibility of OpenCL as an input language for HLS has been evaluated in

several case studies. If used properly, it leads to impressive speedups on reconfigurable hardware. It even outperforms comparable implementations on GPUs [53] in terms of speedup and/or energy consumption. Due to the flexibility provided by the reconfigurable architecture it is possible to fit problems accurately to the provided memories, which makes it easy to beat a GPU architecture with a fixed memory size.

In summary, the Altera OpenCL tool-flow provides good speedups connected with a well-estab-lished programming paradigm. However, this comes at the cost of a strict program structure targeting a more or less static hardware architecture. Currently, the proposed architecture is not designed for usage with SoCs.

Altera OpenCL Summary		
Input Language:	*C/C++ and OpenCL extensions*	
Design Concept:	*Fixed architecture template for FPGA*	
Flexibility:	*Tailored for streaming applications → adapting new algorithms requires substantial refinement*	■□□
HW/SW CO-Design:	*Separate development of hardware and software (using seperate files for each part)*	□□□
Interface:	*FPGA connected with general-purpose processor via PCIe*	■■□

2.8 HLS Projects Using C or C++

In order to overcome the issues with C as input language for HLS (cf. Section 2.2), many approaches revisit the usage of C by using a subset of the language or by modifying the language to meet the requirements of hardware generation.

2.8.1 SystemC

The basic idea of SystemC [4] is the usage of C++ as a unified language for hardware and software. As full-featured C++ would be too complex for hardware generation, SystemC provides a C++ class library tailored to the special needs of hardware design. Developers either use parts of the library that are suitable for hardware-mapping or implement their code using the complete set of C++ features. The latter implies that the code has to run in software. The resulting system can be regarded as hardware/software co-design containing parts of the application designed for software or simulation purposes and containing parts for hardware generation.

```
#include "systemc.h"
#define WIDTH  4
SC_MODULE(adder) {
    sc_in<sc_uint<WIDTH> > a, b;
    sc_out<sc_uint<WIDTH> > sum;

    void do_add() {
        sum.write(a.read() + b.read());
    }
    SC_CTOR(adder)       {
        SC_METHOD(do_add);
        sensitive << a << b;
    }
};
```

Listing 2.6: SystemC design of a 4-bit adder [114]

The SystemC class library makes use of C++ class templates to provide new language features for typical hardware constructs e.g. modules, parallelism, special bit operations or timing specifications. Listing 2.6 shows a sample SystemC

implementation of a 4-bit adder. It uses several special templates (e.g. `sc_in`, `sc_out`) and functions (e.g. `SC_CTOR`, `SC_METHOD`) to provide the expressiveness of an HDL within C++. Hence, to design the hardware part of an application, one requires the same knowledge of the target platform as for a direct HDL design. As a consequence, this method offers only a small rise in abstraction. Moreover, the automatic selection of suitable hardware parts is not supported and still belongs to the designer.

Nevertheless, SystemC has the advantage of giving the developer the opportunity to design all parts of an application within the same tool and the same language. This makes it easy to elaborate the design space for hardware and software mapping. Additionally, it allows the simulation and verification of the whole system during an early stage of development. Hence, SystemC does not claim to be an HLS tool. Rather, it is recognized as a tool for electronic system level design[4].

SystemC Summary		
Input Language:	*C++ with HDL-inspired libraries*	
Design Concept:	*Heterogeneous architecture or arbitrary hardware design*	
Flexibility:	*Arbitrary hardware architecture aiming at SoCs*	■■■
HW/SW CO-Design:	*Determined by language-structure*	■□□
Interface:	*Arbitrary interfaces*	■□□

[4]Methods for the development of electronic systems covering the upper three levels of abstraction (cf. Y-Chart in Figure 2.1), including hardware/software co-design

2.8.2 Xilinx Vivado

In 2013, Xilinx released the Vivado Design Suite [60] after many years of development. It can be regarded as an integrated design environment (IDE) with several components tailored for circuit design. Among others, it contains the Vivado HLS compiler, which enables C,C++, and SystemC programs to be directly translated to application-specific hardware. The proposed design-flow is presented in Figure 2.7.

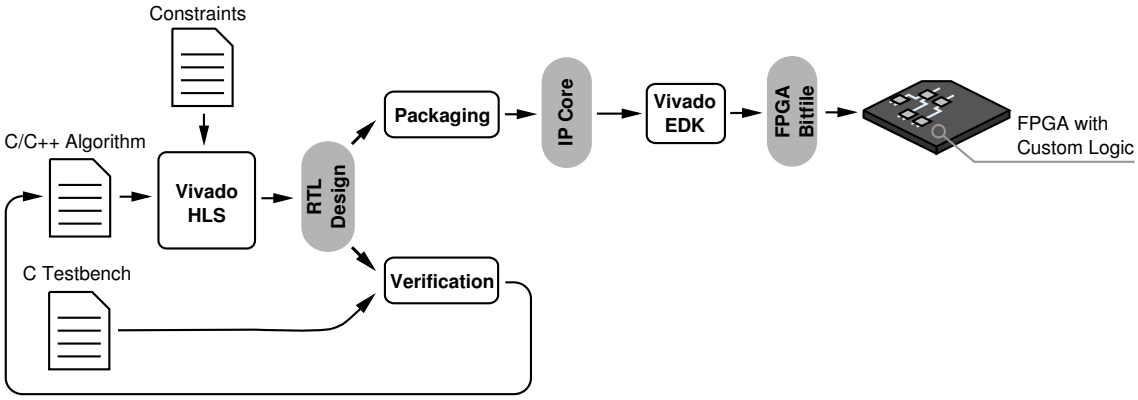

Figure 2.7: Vivado design-flow (adapted from [60]

The concept of Vivado HLS follows the idea of mapping a number of complete C-functions to a hardware design. The desired behavior can be specified in a C testbench that allows verification of the design on C-level by using a common software compiler. The selection of C code and the interfacing must be carried out manually. However, Vivado provides several features that support the developer during the integration and development of custom IP cores. The tool-suite enables design space exploration by allowing the designer to specify constraints, e.g. area/speed optimization. In addition, it generates various solutions for each design. It belongs to the designer to choose which solution fits the requirements best. In order to generate reusable custom hardware, the IP packager can be used to turn the design into a self-contained IP core. The IP packager provides several high-throughput interfaces that are tailored to the Xilinx in-house processor cores and FPGA architectures. The generated IPs can be further reused by Xilinx system builder tools like Vivado EDK.

Besides mentioning problematic code structures like recursion, Vivado HLS gives neither explicit coding rules nor any best practices for development. Hence, programs usually require considerable refinements to achieve optimal results.

The assignment of data to dedicated interfaces or memories is recommended in order to help Vivado with the generation of an IP core interface. If such refinements are ignored or not possible due to the structure of the application, the resulting design can even introduce slowdowns. This fact was exemplified in a case study for a tree-traversal algorithm [65]. In this work, the use of random pointers prohibits the use of fast interfaces and finally leads to substantial slowdowns. Although this is not a typical use case for hardware acceleration, it shows that code refinement is essential when using Vivado HLS.

Xilinx Vivado HLS Summary		
Input Language:	*C/C++/SystemC*	
Design Concept:	*Generate specialized IP core*	
Flexibility:	*Arbitrary hardware architecture targeting FPGAs/SoC-FPGAs*	■■■
HW/SW CO-Design:	*Separate development of hardware and software*	■□□
Interface:	*Provides several interface templates*	■□□

2.8.3 Nymble

The Nymble compiler [34] was introduced in 2013 and was inspired by the COMRADE [39] compiler. Both approaches use a technique for software back-delegation. The idea of giving the control back to software originates from GarpCC [16] but has been strongly improved by COMRADE and Nymble. The proposed hardware/software co-compilation flow is shown in Figure 2.8.

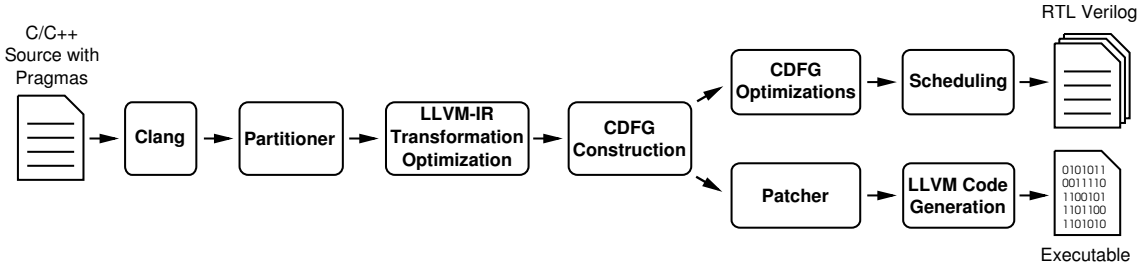

Figure 2.8: Nymble hardware/software co-compilation [34]

The Nymble compiler is based on the Low Level Virtual Machine (LLVM) but uses a modified clang C/C++ front-end, which accepts custom pragma directives. These pragmas are used to define the hardware/software partitioning. The future hardware part is translated into a separate function that, after applying optimizations in LLVM, is processed by a specialized HLS flow that generates Verilog. The remaining application is patched to call the hardware function, and is further translated into an executable. The concept of software back-delegation requires a tight coupling between processor and accelerator memory. For this reason, Nymble uses a sophisticated (cache)-Memory Architecture for Reconfigurable Computers (MARC II) [105] to interface between hardware and software.

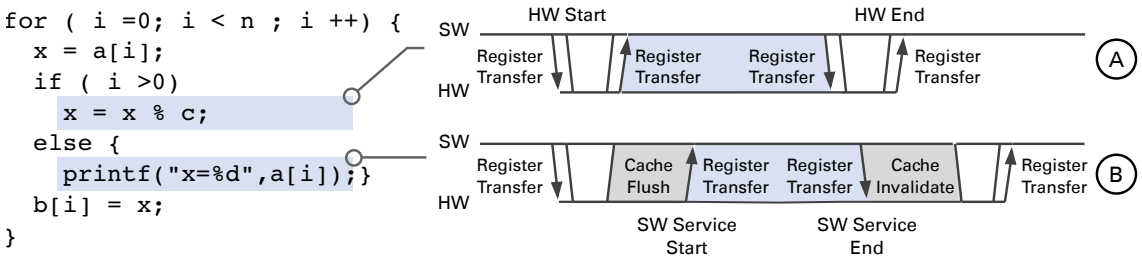

Figure 2.9: Switch between hardware and software [34]

The kernel[5] selection is typically done on the granularity of loops. Because of the back-delegation feature it is possible to translate loops even if they contain real function calls, which allows the mapping of large portions of C code to hardware. However, frequent switching between hardware and software introduces an overhead to the execution time. This may be acceptable for the transfer of small amounts of data, e.g for a complex arithmetic operation (cf. Figure 2.9 *(A)*). But for operations with unknown side-effects – like function calls – back-delegation is often expensive. The used data cache between hardware and software needs to maintain its data consistency for each software/hardware switch (cf. Figure 2.9 *(B)*). Besides additional hardware resources, this introduces long delays due to the overhead of data transfers.

Nymble Summary		
Input Language:	*C/C++*	
Design Concept:	*Cores for hybrid-architectures using software back-delegation*	
Flexibility:	*Complete system design targeting FPGAs/SoC-FPGAs*	■■■
HW/SW CO-Design:	*Manual code annotation of suitable hardware sections*	■□□
Interface:	*Generated cache for accelerators*	■■■

[5] In the context of HLS, the term "kernel" refers to that part of the application code which is considered for hardware mapping.

2.8.4 LegUp

LegUp [17, 18] was introduced in 2011 but is still under development. The original approach was tailored to a hybrid architecture using the Tiger MIPS soft-core processor [110] for Altera-FPGAs. Currently, the HLS-flow is adapted to Altera Cyclone V SoCs, which contain a hard IP Advanced RISC Machine (ARM) processor core [24]. Besides hybrid architectures, the LLVM-based HLS tool can also be used standalone in order to map C functions to single IPs. LegUp does not support software back-delegation within accelerators; thus, it requires fully synthesizable C sections for hardware generation.

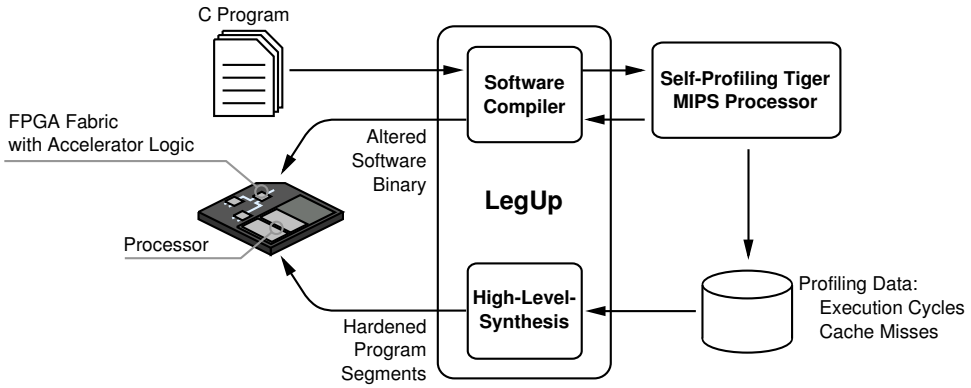

Figure 2.10: LegUp design-flow [17]

A special feature of LegUp is the automatic profiling of applications in order to identify computational hotspots. To make use of this feature, the C application must be implemented for a soft-core processor running on the FPGA (cf. Figure 2.10). A built-in hardware profiler [6] is then used to generate an application trace. Using this trace data allows LegUp to estimate the possible speedup of an accelerator. Afterwards, the user can mark the discovered critical functions in the source code for synthesis. The selected function sources will be recompiled with LegUp-HLS while the remaining source code is patched with accelerator calls and translated by a regular software compiler.

As traces depend on runtime data, the gathered information is naturally not complete. The number of loop iterations, the control flow, and the memory accesses might differ in actual application runs. In order to avoid an inaccurate

hardware-software partitioning, the final selection of hotspots still belongs to the developer. The interface between the Tiger MIPS soft-core processor and the hardware accelerator is implemented by using shared memories. Therefore, the Avalon system bus is used in order to connect the processor and the accelerator with multi-port memories provided by the FPGA platform.

LegUp Summary		
Input Language:	*C/C++*	
Design Concept:	*Cores for hybrid-architectures. The general-purpose processor requires a profiling interface or a cycle-accurate emulation*	
Flexibility:	*Complete system design targeting FPGAs/SoC-FPGAs*	■■■
HW/SW CO-Design:	*Suitable hardware sections are determined via profiling*	■■□
Interface:	*Using shared memory*	■■□

2.8.5 Overview of Other HLS Projects

The presented projects are only a few examples from the numerous existing HLS tools and work-flows. The following by no means exhaustive Tables 2.-8 and 2.-9 give a quick survey of further approaches for HLS tools. The chosen taxonomy adapts the criteria of the previously presented projects.

Table 2.-8: Overview of Related HLS Projects (1)

HLS-Project	Input Language	Design Concept	Flexibility	HW/SW Co-Design	Interfacing
COMRADE [39, 25]	C/C++	SW Back-Delegation	■■■	■■□	■■■
Catapult-C [45, 13]	C/C++	Single IP	■■□	□□□	■□□
CASH [15]	C (Annotated)	Single IP	■□□	□□□	■□□
CHiMPS [52, 40]	C (Subset)	Hybrid Architecture	■□□	■□□	■■□
GarpCC [16, 32]	Ansi C	Hybrid Architecture	■□□	■■□	■■□
GAUT [43, 20]	C/C++	Single IP	■□□	□□□	■■□
Handel-C [14, 63]	HDL-inspired	Single IP	■■□	□□□	■□□
HercuLeS [37, 36]	C/NAC	Single IP	■■□	□□□	■□□
Matlab HLS [54]	Model-Based	Hybrid Architecture	■□□	■■□	■■■
LabView HLS [23]	Model-Based	Hybrid Architecture	■□□	■■□	■■■
Nimble [31, 42]	Ansi C	Hybrid Architecture	■□□	■■□	■■□
Panda/Bambu [1, 50]	Ansi C	Single IP	■■□	□□□	■■□
ROCCC [64]	C + Library	Single IP	■□□	□□□	■■□
Streams-C [28]	C (Extended)	Hybrid Architecture	■□□	□□□	■■□

Table 2.-9: Overview of Related HLS Projects (2)

HLS-Project	Input Language	Design Concept	Flexibility	HW/SW Co-Design	Interfacing
Lava [12]	Haskell	Single IP	■■□	□□□	■■■
muFP [55]	Functional	Single IP	■■□	□□□	■□□
HML [41]	Functional	Single IP	■□□	□□□	■□□
C2Verilog [59]	Ansi C	Single IP	■□□	□□□	■□□
C-to-Verilog [11]	Ansi C	Single IP	■■□	□□□	■□□
C-to-Silicon [3]	SystemC/OSCI TLM	Single IP	■□□	□□□	■■□
Kiwi [57]	C#	Hybrid Architecture	■■□	■□□	■□□
Impuls-C [61]	HDL-inspired C	Single IP	■■□	□□□	■□□
eXCite [66, 67]	Ansi C (Extended)	Single IP	■□□	□□□	■□□
xPilot [19]	C/SystemC	Single IP	■□□	■■□	■■■
FPGA Brook [51]	C (Extended)	Hybrid Architecture	■□□	■□□	■■□
DIME-C [27]	Ansi C (Extended)	Single IP	■■□	□□□	■□□
Carte/MAP [5, 58]	C + Library	Single IP	■□□	□□□	■■□
Daedalus [47, 62]	C (Extended)	Hybrid Architecture	■□□	■■□	■■□

2.9 GCC PIRANHA

The following section describes the structural decisions that have been taken in order to create a new HLS tool-flow. Although the content of this section does not directly belong to the technical background, it helps the reader to classify the implemented approach with respect to the existing ones. Furthermore, it motivates the following background information about the target platforms and the GCC compiler framework.

GCC PIRANHA stands for GCC **P**lugin for **IR AN**alysis and **H**ardware **A**cceleration. It was developed as HLS tool-flow that is tightly coupled with the software compilation process. It targets hybrid architectures and was originally designed to generate accelerators for the SpartanMC soft-core processor [139]. The tool-flow is based on the GCC framework. While the first version of the tool [142] uses GCC version 4.4.5, the refined version [145] was implemented using the GCC plugin interface of GCC 4.8.3.

2.9.1 Manageable Language Constructs

By using GCC as the underlying compiler framework for the proposed HLS approach, it is obvious to use C as the preferred input language. On the one hand, C is a commonly used language in the area of embedded systems and is well accepted by the industry and the research community. On the other hand, as described in Section 2.2, the use of C introduces various challenges to the whole HLS process. Some of the required information, e.g. the patterns of memory accesses or memory aliasing are not expressed within the language. Other important information like ILP or the optimal operator width could be extracted but requires very complex analysis techniques that sometimes cannot be handled with acceptable effort. Hence, the philosophy behind the proposed work-flow is to consider only those code sections for HLS that seem to be manageable by the plugin and lead to correct synthesis results.

Typically, most of the execution time of embedded applications is spent in loops. Consequently, the analysis of PIRANHA focuses on loops. In contrast to other HLS flows (e.g.: LegUp [17]), complete functions and linear code sections are not considered for hardware generation. Within the loop body, PIRANHA can handle

arbitrary control flow and even subordinated loops. An exception are function calls, which cannot be mapped to hardware without further effort. As C function calls cannot be considered as free of side effects, the inclusion of these statements finally requires a mechanism for software back-delegation, which introduces additional hardware effort. The lack of software back-delegation is partially compensated by the inlining capabilities of the GCC. Both granularities, loops and functions, entail different advantages and disadvantages: Obviously, the parameter list of function-based accelerators is clearly defined and often smaller than for loop-based accelerators. On the downside, the chance for the occurrence of unsuitable code sections, like library calls, is typically higher. For the proposed work-flow, the granularity of functions would lead to a higher number of rejected hardware kernels. For this reason, PIRANHA considers only loops for acceleration.

Besides function calls, the proposed work-flow also declines code sections containing unsupported operations, e.g. floating point arithmetic or divisions. Such features are not relevant for the current implementation but can be upgraded later with little effort by using existing interfaces.

2.9.2 Translation Work-Flow

As mentioned in Section 1.2, the idea behind PIRANHA is to provide a transparent tool-flow for software programmers in order to considerably lower the entry barrier for hardware development. Hence, the selection of suitable accelerators as well as the integration of accelerators into the host system is completely automated. PIRANHA consists of two separated GCC-plugins collectively performing four processing steps. Figure 2.11 gives an overview of the tool-flow and its processing steps.

1. The *loop data collection* performs a whole-program analysis and collects information about all loops of the program. This step must be carried out in a separate compilation run in order to ensure that all files of the program can be analyzed. Thus, it is solely implemented within the first plugin. The result of this run is an analysis transcript of all loops within the program.

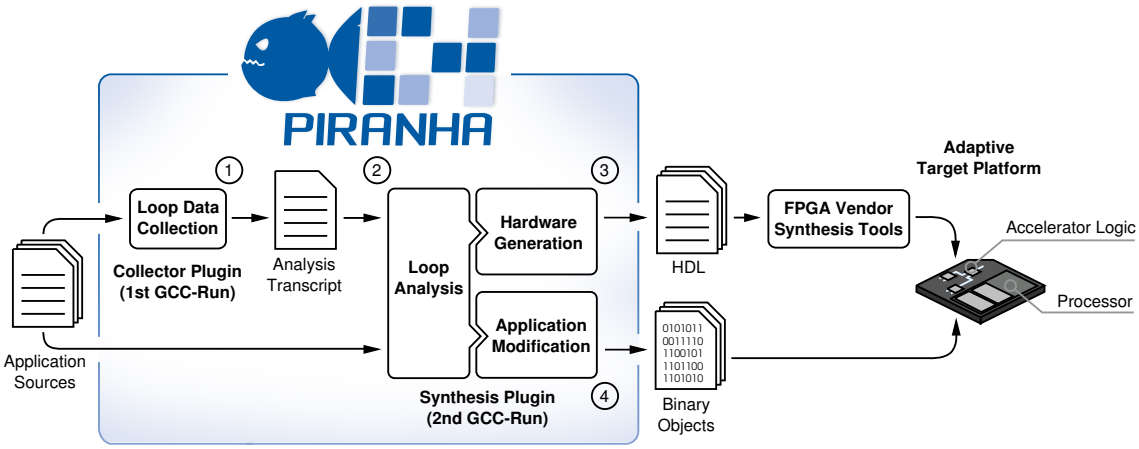

Figure 2.11: GCC plugin-based HLS work-flow for configurable architectures

2. The *loop analysis* is performed as a first step from the second plugin during the second compilation run. Its task is to analyze each loop of the source code by using the gathered data from Step 1. Besides this information, it analyzes the feasibility of hardware generation and uses a cost model to estimate the speedup of loops with respect to the target platform.

3. The *hardware generation* performs the HLS for loops that were selected in the previous step. The result of this step is a hardware description of the generated accelerators.

4. The *application modification* adapts the original software code in order to integrate the hardware accelerator. According to the target platform, additional C code is generated to perform the communication between host processor and accelerator interface. Afterwards, the whole application is translated and linked to generate a program binary.

GCC PIRANHA invests substantial effort to analyze the application in order to identify suitable code sections. For most of the existing HLS approaches, this task is assigned to the system developer, which may lead to better overall results but also to a much higher entry barrier for the user of the system. Hence, GCC PIRANHA has taken a completely new direction for HLS tool-flows. It works on whole applications and requires neither program rewriting nor special annotation or language features.

2.9.3 Runtime Work-Flow

After generating a modified binary file in Executable and Linking Format (ELF) and a bit file containing the hardware design, both files can be used to run the application on the target platform. Therefore, the ELF binary is executed on the general-purpose processor while the bit file is used to configure the programmable logic part of the target platform. PIRANHA modifies the ELF binary in such a way that running the whole application in software is still possible even if the hardware accelerator is temporarily unavailable. This feature is required for rather complex systems that share several accelerators between multiple applications. For instance, this may occur for accelerated library functions. In general, the execution of an accelerator is divided into three steps as shown in Figure 2.12.

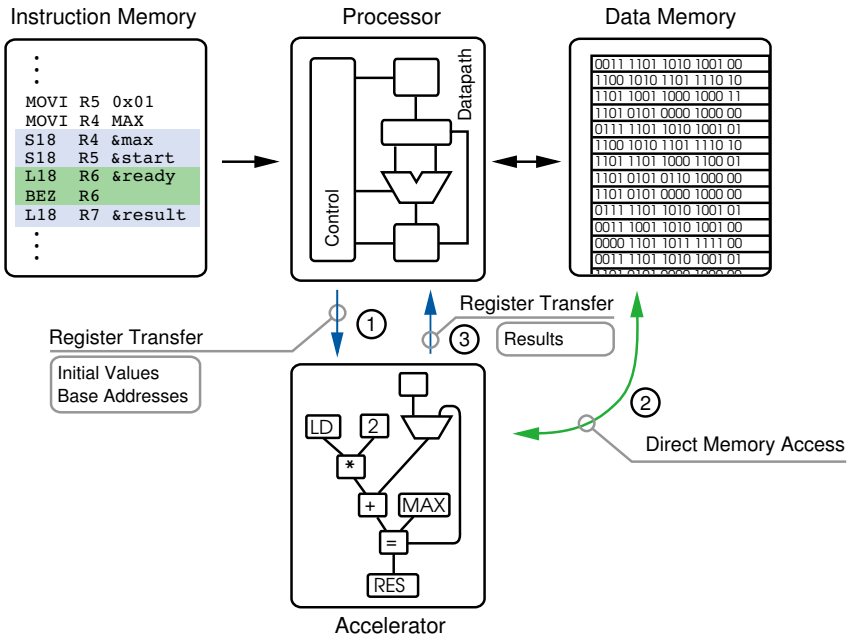

Figure 2.12: GCC plugin-based HLS work-flow for FPGA-based architectures

1. The entry point of the loop is identified and replaced by a call of the accelerator wrapper function. The first task of that function is to pass input parameters to the hardware accelerator. Such input parameters are typically variables that are used within the accelerated loop. The transfer is initiated by the host processor. Afterwards, the host processor hands control to the accelerator by setting a start bit in the accelerator control register.

2. After the transfer of input parameters the accelerator starts its loop execution while the host processor polls the accelerator status register. All array and pointer accesses within the loop require a direct memory interface. The default behavior of the accelerator is to generate a direct memory access (DMA) for each single value, but the actual method can be adapted to the special requirements of the target architecture and the C code of the loop. This memory interface requires special attention in order to avoid slow downs of the accelerator.

3. Finally, the accelerator uses the status register to signal its completion to the host processor. Afterwards, the processor transfers results from the accelerator and continues the regular software execution. One of the generated result values indicates the software entry point that is used to continue the program. If the accelerator has run successfully, the loop must be bypassed by setting this entry point behind the loop. Otherwise, the entry point is set above the loop that executes the software loop without further modifications.

2.9.4 Target Architecture

GCC PIRANHA is tailored for hybrid architectures containing a general purpose processor and configurable logic. The current implementation was evaluated on two target architectures of different complexities: first, on a soft-core processor tailored for SoCs on FPGAs running bare metal applications, and second, on an FPGA platform with an embedded ARM processor running a full-featured operating system. Both host processors provide a GCC port for their architecture. Hence, the whole approach is portable without modifications on the hardware/software co-compilation process, except the hardware and software interface for memory and peripheral access.

3 TARGET PLATFORM

The following chapter gives an overview on the technical background of the used target platforms. The proposed work-flow is based on configurable platforms, consequently, the used FPGA technology is briefly introduced. The GCC plugin was considered to be target-agnostic as long as the related platform provides configurable logic resources. Thus, it seemed natural to prove the practicability of the approach by implementing it for two target architectures. Both architectures are introduced in the following sections.

3.1 Configurable Platforms

Application-specific architectures can be implemented by using various technologies. Together with other individual features, each technology provides its own level of granularity for application-specific customizations.

ASICs allow the adoption of nearly all design variables, starting from the circuit layout down to specific technology parameters. The result is usually a tailored circuit with nearly optimal properties in terms of area, speed and energy consumption. The downside of this technology is the design process, which includes all development steps required for a complete semiconductor fabrication flow. Thus, for many projects, building an ASIC is often too expensive and time consuming. Another approach to custom circuit design is the use of application-specific standard products (ASSPs). The key of this technology is the usage of "off-the-shelf" components in order to simplify the design process. However, even such standard cell designs require a minimum quantity of pieces to be economically advisable. The hardware accelerators presented in this thesis are solely implemented on FPGAs. Such platforms are ideal for prototypes with very low quantities and have become increasingly popular due to decreasing costs per unit. In contrast to ASICs and ASSPs, FPGAs provide the possibility to change the design after fabrication in the "field".

However, the design concepts presented in this work can be adapted to all of these technologies and may also be suitable for future reconfiguration techniques like silicon nano wire transistors [116].

3.1.1 Field Programmable Gate Arrays

The key feature of FPGAs is their ability to change the behavior of the chip even after fabrication. Therefore, the static architecture consists of homogeneous logic modules and an arbitrary routing network. Most of the configurable logic blocks (CLBs) and the routing network contain programmable switches[1] or programmable memory elements[2], which can be used to adapt the hardware layout to implement a certain behavior. Although the actual implementation of the CLBs and the routing network depend on the FPGA vendor, it is often similar with respect to its logical structure. Figure 3.1 shows the architecture of a Xilinx 7 Series FPGA [126], which is used for the implementations in this thesis.

Similar to other FPGA types, the essential element of this architecture is the CLB. The Xilinx CLB architecture is organized in slices that provide resources to emulate nearly all basic logic functions. The core pieces of these slices are the look up tables (LUTs). These SRAM cells are used to specify Boolean functions by using the address bits as input variables and to determine the output through the assigned contents of the memory. Furthermore, a slice contains adders that form a carry-chain with the neighboring slices, and several flip-flops that are used to store data in order to create synchronous logic. The multiplexers (cf. Figure 3.1 for the *slice structure*) are required as switches to select or bypass single elements. The input and output signals of each CLB are connected to the configurable routing network. Additionally to the generic CLBs, FPGAs provide different special blocks implemented as fixed hardware, e.g. dual-ported memories (BRAM), DSP blocks[3], serial transceivers, and IO blocks. On current Xilinx FPGAs such modules are organized in columns and stacked above a silicon interposer by using conducting "bumps". The silicon interposer provides the static connections between these columns. This technique allows a smaller footprint and shortens the distances between modules, which decreases the latency for interconnects. The resulting advanced silicon modular block layout (ASMBL) allows Xilinx to adapt its FPGA portfolio to new application domains with little effort.

[1] Some manufactures also use fuses here.
[2] This can be either static random access memory (SRAM) cells or Flash cells
[3] Digital signal processors (DSP-blocks) [124] are execution units containing a parametrizable adder and multiplier.

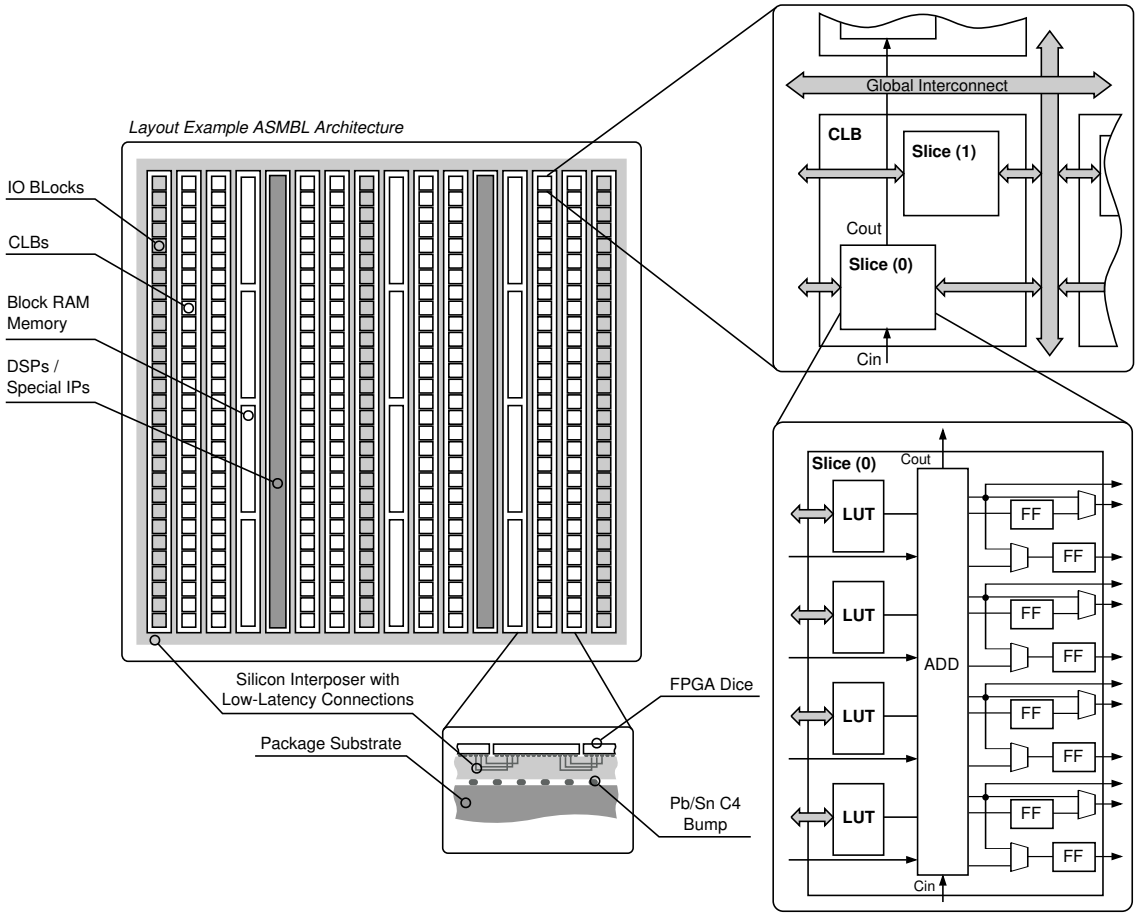

Figure 3.1: Xilinx Artix FPGA layout [125, 113]

In the 1980s, the first FPGAs were mainly used for glue and control logic. At present, FPGAs have evolved into highly capable coprocessors and platforms for complete SoCs. They are often used for typical DSP applications like filter and transformation algorithms as well as for security applications. In order to improve speedup and power efficiency with FPGAs, one need to take advantage of the large number of computing resources by performing highly parallel executions. This becomes even more necessary due to the fact that the clock frequency of a typical FPGA is significantly lower than the frequency of an optimized general-purpose processor or ASIC. Nevertheless, many publications in the past have proven the viability of FPGA-based program acceleration [106, 99]. Another application domain is its usage as a peripheral unit in order to perform specialized IO operations. The combination of a large number of parallel IO blocks and the deterministic behavior of hardware designs makes FPGAs an ideal platform for

real-time IO with the potential to reduce the computational load of the main processor.

3.2 Soft-Core Processors

Microprocessors are an integral part of many digital systems. If used in conjunction with FPGAs it is perfectly logical to consider their implementation using configurable logic resources as well. In this case, the processor itself is implemented as a part of the FPGA design. Such so called soft-core processors are available from most FPGA vendors. Popular examples are the Altera NIOS II processor [89, 108] and the Xilinx Microblaze [119]. Typically, they are optimized for the underlying configurable fabric of the respective FPGA manufacturer. In addition, also open source and commercial soft-core processors for cross-platform usage exist. For instance, the LEON processor family [87].

The tight coupling of custom hardware and soft-core processor on the same chip allows the usage of low-latency interfaces for data transfers. For instance, it enables direct and symmetric access to shared memory without the need for specialized bus interfaces. This works especially well on FPGAs as they provide dual-ported memories (BRAMs) by default. Such memories can be used either as shared cache for processor and accelerator or directly as data memory. The latter makes sense especially for small embedded systems. Another feature of soft-core processors is the ability to adapt the processor core to the requirements of custom hardware. For example, this is done in the Microblaze pipeline with so called Fast Simplex Links [121] that can be used for special tasks like instruction set extensions or multi-core architectures.

3.2.1 SpartanMC

SpartanMC [139] is a soft-core processor that was developed to optimally fit the typical hardware structure of FPGAs. As FPGAs provide a processing width of 18 bit for several internal structures, e.g. DSP-blocks and BRAMs, the SpartanMC soft-core adapts its internal processing width to these circumstances. By using 18 bit wide instructions and data[4], the core ranges between 8 bit and

[4]The SpartanMC soft-core uses a word width of 9 bit and an integer width of 18 bit.

32 bit cores, thus, aiming at applications of medium complexity derived from the typical 16 bit domain of classical embedded microprocessors. The resulting processor (including additional modules for Timer and UART) provides a reasonable performance of 1.05 DMIPS/MHz with a minimal resource footprint of 852 LUTs and four BRAMs. Providing an elaborated toolchain and customized peripherals, the SpartanMC can be regarded as a viable platform for signal processing and controlling tasks.

Instruction Set

The SpartanMC soft-core processor implements an instruction set with a fixed width of 18 bit. In contrast to typical 16-bit architectures, it turns out that the additional 2 bits per instruction are essential in order to design a small reduced instruction set computer (RISC) architecture without the need for extension words. This enables the integration of two addresses or immediate values into a single instruction. The complete op-code matrix and its sub-matrices are shown in Tables B.1, B.2, and B.3. The available instructions can be grouped into four types, as shown in Figure 3.2.

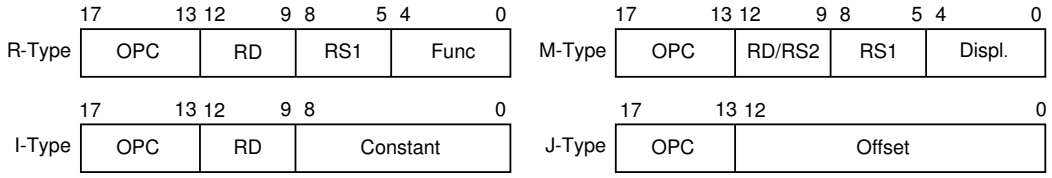

Figure 3.2: SpartanMC instruction types [139]

- The R-Type is used for arithmetic operations that take two 4 bit register addresses as operands. The result of the operation is stored in the first register operand. The bits 4 to 0 at the end of the instruction are reserved for the opcodes of the two sub-matrices.
- The I-Type includes all operations using constant values for calculations. Therefore, the constant is stored in the least significant 9 bits of the instruction. Again, the destination of the operation is the first register operand.

- The M-Type is used for load and store operations. The SpartanMC provides op-codes for full words (18 bit) and half words (9 bit). The unsigned displacement can be used as an additional offset for chained memory accesses.
- The J-Type is typically used for jump and branch instructions. The latter take their condition from a special condition code register.

Core Architecture

As shown in Figure 3.2, the SpartanMC instructions provide 4 bit to address internal registers, which implies that the core can effectively use a maximum of 16 registers. The 18-bit registers are implemented in a BRAM with 1024 cells. This decision was taken in order to utilize a sliding register window technique, as outlined in Figure 3.3. Hence, each new function or interrupt invocation provides eight new registers to the core. Four of the 16 registers are defined as globals (R_0–R_3), another four registers overlap in order to pass parameters between call levels (R_{12}–R_{15} or R_4–$R_7$of the calling function), and the remaining four registers (R_8–R_{11}) are considered to be local with respect to the current function. Sliding the window by eight memory cells for each call results in up to 127 nested call levels. This technique enables very fast function and interrupt invocations on the architectural level. Furthermore, the SpartanMC requires no dedicated stack for subroutine calls as long as the available registers are sufficient. If additional stack is required the necessary memory is allocated at the end of the main memory. The current stack-pointer is always stored in R_0.

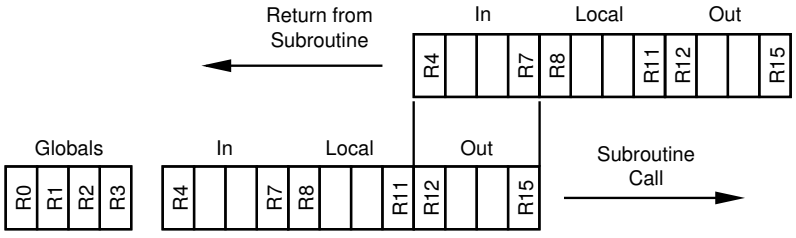

Figure 3.3: SpartanMC sliding register window [139]

The SpartanMC is designed as an RISC core executing one instruction per cycle. Therefore, it provides a three-stage pipeline with bypasses, e.g. between operand fetch and write-back or between write-back and the execution stage.

The timing of the pipeline was first developed by using a phase-splitting technique that allows the usage of two clock edges per cycle (cf. [139]). On the one hand, this eliminates some of the bypasses and minimizes the resource footprint of the core. On the other hand, it requires tedious clock management within the pipeline, which finally leads to a poor overall clock frequency. In context of this thesis, phase splitting is not used at the cost of a slightly higher resource consumption. A schematic of the actual used pipeline is given in B.2.

Memory Interface

The default implementation of the SpartanMC uses BRAMs (cf. Section 3.1.1) as memory for the program and data section of the application binary. A simplified structure of the SpartanMC memory interface is presented in Figure 3.4. The memory architecture is designed as a modified Harvard architecture. This means that the core has parallel access to a combined data and program memory by using separate interfaces for both, data and instructions. This approach is well supported by the underlying FPGA architecture as the used BRAMs always provide two independent ports. The actual size of the main memory is customizable by the granularity of BRAM pairs each providing a width of 9 bit and a depth of 2048 lines. The usage of two BRAMs at a time is necessary in order to grant 18-bit access for the instruction interface and native 9-bit access for the data port.

Besides the connection to the main memory, the SpartanMC provides two lightweight interfaces to integrate peripherals into the 18-bit address space.

First, a memory-mapped register interface is used for peripherals working on small amounts of data, using registers that are synchronized to the processor clock. As shown in Figure 3.4, such registers are integrated in a sub-address space while the remaining address bits are used to define the base address of the peripheral. For data write operations, single registers are selected by an address decoder.

Second, a direct memory access (DMA) interface is implemented that uses dual-ported BRAMs to exchange large data sets between processor and peripheral. The integration of the DMA interface is similar to that of the regular main memory, but instead of connecting the second memory port to the instruction fetch of the processor, it is connected as the data port of the peripheral.

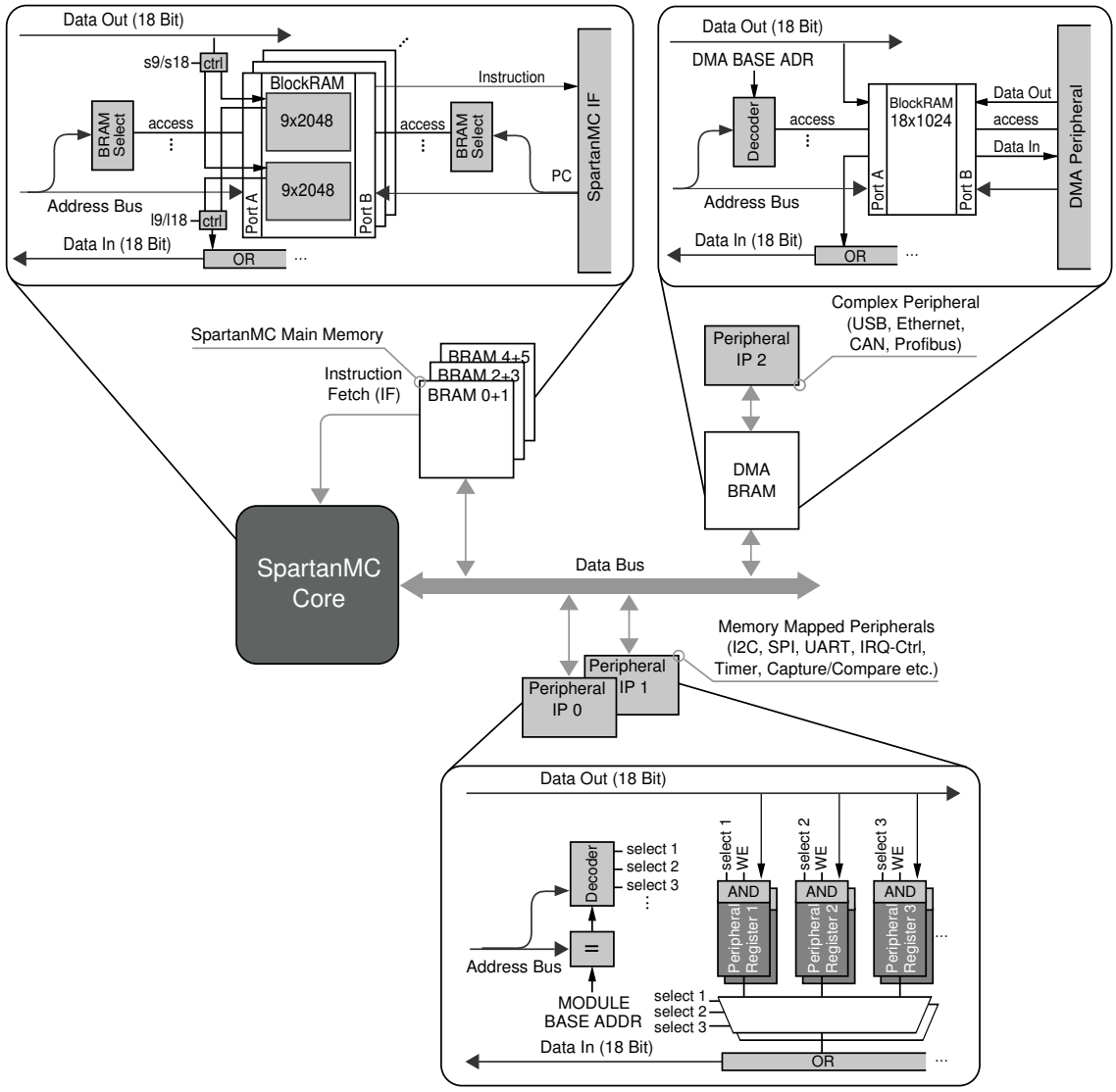

Figure 3.4: SpartanMC memory Architecture [139]

In order to avoid a distributed multiplexer for read operations from different peripherals the outputs of all memories and register sets are combined with an *OR* gate. This presumes that peripherals that are not selected always produce a zero on their outputs.

3.2.2 Toolchain

The major benefit of the SpartanMC environment lies in its customizability to application-specific requirements. Along with the SpartanMC processor soft-core exists a rich set of peripheral IP cores for standard functions, e.g. capture and compare and interrupt handling as well as for complex communication protocols like USB or CAN. The SpartanMC SoC-kit [140] allows the combination of such IP cores in any desired composition and quantity in order to generate a tailored SoC. Maintaining such a complex system and granting its accessibility to a broad community of end users requires a lot of effort in the area of tool development.

The SpartanMC SoC-kit is distributed as an open source project bundled with a GNU Automake [96] toolchain. Furthermore, it provides a cycle accurate simulator, a system-builder application, and a C compiler. An overview of the tool-flow for the SpartanMC SoC-kit is given in Figure 3.5.

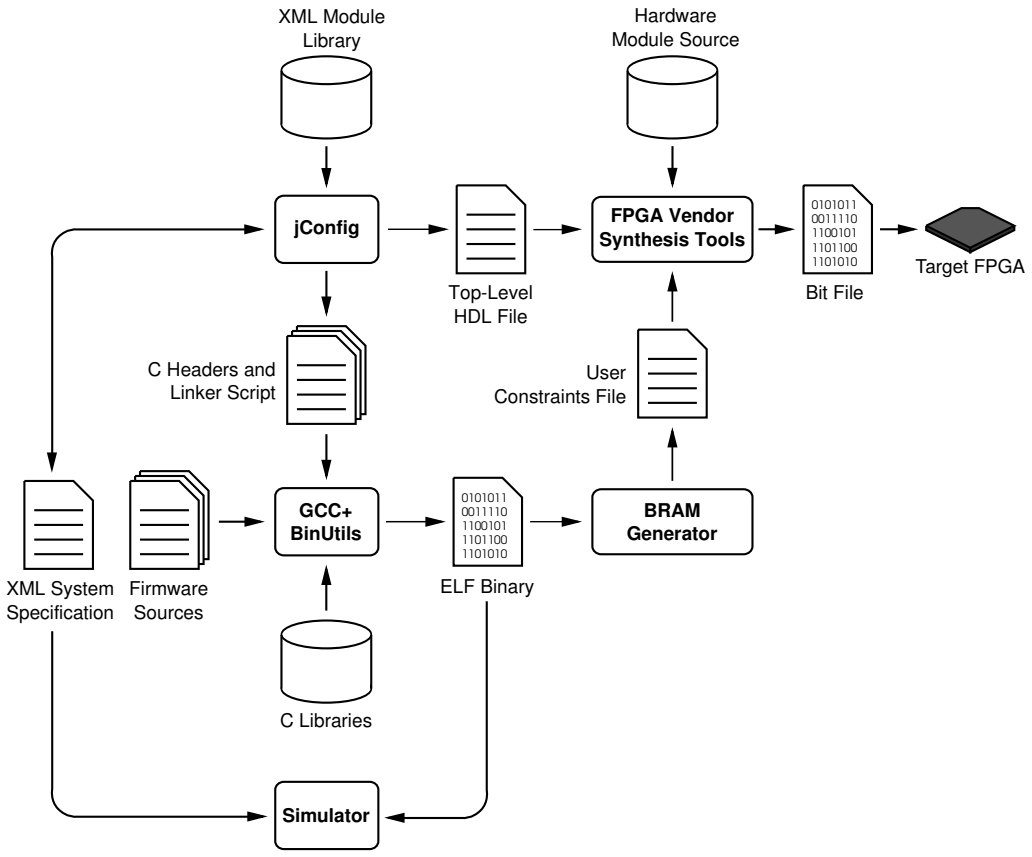

Figure 3.5: SpartanMC tool-flow [139]

The system-builder application in connection with the software compilation process, is of particular interest for this thesis. The system builder provides a graphical user interface to instantiate and connect IP cores in an easy and straightforward manner. In order to register hardware units for use in the SpartanMC system builder, hereinafter referred to as *jConfig*, one requires an eXtensible Markup Language (XML) description of the IO ports and the parameters of the respective hardware unit.

Another major tool in the context of this thesis is the C compiler. The SpartanMC SoC-kit uses the GCC framework for software compilation. As the GCC is known to generate an excellent code quality for two address machines, it was the natural choice for the SpartanMC processor. PIRANHA was first designed to generate application specific accelerators exclusively for the SpartanMC architecture. Hence, it was an obvious choice to use GCC as the underlying compiler framework for the plugin as well.

3.3 FPGA SoC

According to [107], more than the half of all FPGA designs already use at least one general-purpose processor besides their custom logic implementation. Thus, it was the logical next step for FPGA manufactures to offer processors as hard IP cores for their platforms. Although such integrated processors cannot provide the same flexibility as soft-core processors do, they compensate this with better performance and energy efficiency. As long as FPGA vendors use mainstream architectures like ARM, such hard IP processors are complemented by a highly optimized toolchain. With the use of hard IP processors on FPGAs, they have become increasingly relevant as a central component of an SoC. This approach is reflected in the name FPGA SoC.

3.3.1 Xilinx Zynq

The most recent FPGA SoC platforms from Xilinx are Zynq and Zynq Ultrascale. As shown in Figure 3.6, the platform used for this thesis is the Xilinx Zynq-7000 [129] on an Avnet ZedBoard [91]. Zynq-7000 contains an Artix-7 FPGA fabric in combination with a complete ARM subsystem consisting of a Cortex-A9

dual-core processor featuring two *NEON* floating-point units (FPUs) and a comprehensive set of standard peripherals. The integrated processor core is based on the ARMv7 instruction set and provides a Dhrystone performance of 2.5 DMIPS/MHz for each of the cores. The used clock frequency for the Zynq on the Zedboard is 667 MHz which makes the processor approximately seven times faster than a typical design for the associated FPGA fabric.

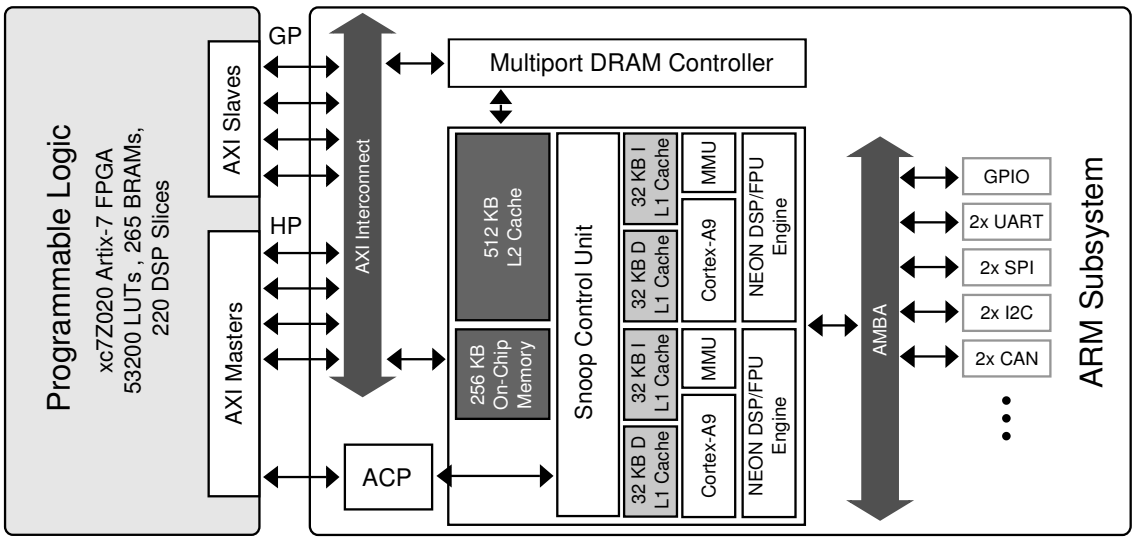

Figure 3.6: Zynq architectural overview [145]

For the communication between processor core and generated accelerators the memory architecture of the platform is of great importance. The main memory of the Zynq is connected to the ARM subsystem via two caches: a 32 KB first-level cache for each core and a combined 512 KB second-level cache for both cores. Moreover, the ARM core is connected to a memory management unit (MMU) providing hardware support for the virtual address management, which is required by operating systems. Both features, caching as well as virtual addressing, introduce new challenges for generating a processor-accelerator system. First, the virtual addressing of an operating system hinders direct memory access from within the accelerators, as the required data addresses are not known during compilation time. Second, the integrated cache architecture requires additional effort to keep the data between core and accelerator consistent. The latter problem is addressed by a special port for custom hardware units, the

accelerator coherency port (ACP). The ACP is internally connected to the Snoop Control Unit of the ARM and can be used for cache-coherent access to the ARM. In addition, the ARM subsystem provides a DMA unit to initiate burst transfers from the operating system that will be of further interest for the implementation of the accelerator interface.

Advanced eXtensible Interface

A challenging aspect of the integration of a standard processor on an FPGA is the implementation of the interface between the hard IPs and the FPGA fabric. Therefore, the ARM processor on the Zynq platform provides several Advanced eXtensible Interfaces (AXIs). AXI is a bus system derived from the ARM Advanced Microcontroller Bus Architecture standard (AMBA) [122] and is currently available in the fourth version (AXI4). The standard incorporates a full-featured interface that allows for full-duplex single-word transfers as well as memory bursts and a light-weight interface (AXI4-Lite) that is limited to word transfers only. Furthermore, AXI provides special features like unaligned data transfers by byte strobes. Generally, AXI is designed as a master/slave protocol where only masters are able to initiate the communication.

On Zynq devices, one distinguishes between full-duplex 64 bit high-performance and 32 bit general-purpose AXI interfaces. The ARM subsystem provides four general-purpose interfaces and four high-performance interfaces. However, the actual capabilities of the channel depend on the provided interface for both peers. For the communication with the FPGA fabric, the interface is always implemented as soft-core providing an AXI bus "uplink" and a non-standard interface to the custom logic.

4 GCC FRAMEWORK

The GNU Compiler Collection (GCC) is one of the most widely used compiler frameworks for software development in industrial and scientific areas. It is established for embedded systems as well as for consumer- and high performance computers. Its development started in 1987 with a free C-only compiler for the GNU operating system. Hence, the acronym stood for the GNU C compiler at first. Later, the compiler front-end grew to support additional programming languages like C++, Java, Fortran and Ada. Consequently, the name was changed to GNU Compiler Collection.

Today, the GCC can be regarded as a compiler generation framework that generates reliable and highly optimized compilers from descriptions of target platforms. It owes much of its popularity to its support for more than 30 machine architectures. Due to its wide variety of source languages and target machines, the GCC has become one of the most complex free software projects with a considerable code base of over two million lines[1]. Currently, the GCC is mostly written in C. Only some of the core functionalities of the API, e.g. `vec.h` were ported to C++ classes.

Using such a mature compiler helps to provide a fully transparent work-flow for software developers and introduces a rich set of code analysis and optimization strategies. Most of these strategies have proven their usefulness independent of the target machine architecture. Hence, the availability of a transparent work-flow and the huge amount of language- and architecture-agnostic optimizations make the GCC a promising choice for generating custom hardware from a high-level language.

In this thesis, many GCC data structures and terms will be used. Hence, the following chapter will introduce assorted mechanisms and structures of the compiler generation framework.

[1] Without target specific code and external optimization frameworks.

4.1 Compilation Flow

The GCC processes each source file in a single compilation run. In the case of C as input language a compilation run comprises a C source file and its inclusions (usually header files). In the following, the combination of an application source file and its inclusions will be referred to as a compilation unit. A high-level overview of the compilation process for one compilation unit, as described in [80], is given in Figure 4.1.

The compilation workflow can be divided into three stages: the front-end, the middle-end, and the back-end. The front-end is responsible for validating the syntactical structure of the given input. It creates internal structures for data types and variables and builds an initial abstract syntax tree (AST). The middle-end analyzes and transforms the program by following two design goals: make the resulting object code run as fast as possible and make it as space-efficient as possible. Finally, the back-end translates the intermediate code of the middle-end to the machine code of the target architecture. While the front-end depends on the input language and the back-end depends on the target architecture, the middle-end carries out all optimizations using a language-agnostic intermediate representation (IR). Hence, the middle-end can be regarded as language and target-independent.

Besides these compilation stages the GCC framework enables general customizations. Therefore, it is possible to modify the compilation process by command-line parameters. This can be used to generate debug information and to influence how GCC optimizes the compilation unit.

4.1.1 Front-End

After processing the command line parameters the parsing of the application starts in the front-end. Naturally, this process is language-specific and requires a special parser for each language.

The resulting tree representations of the parsed input code still depend on the input languages. In order to be processable by the middle-end, the tree must be converted from the language-specific tree representation to GENERIC (cf. Section 4.2). Note that this step is not necessary for C and C++, as they are directly

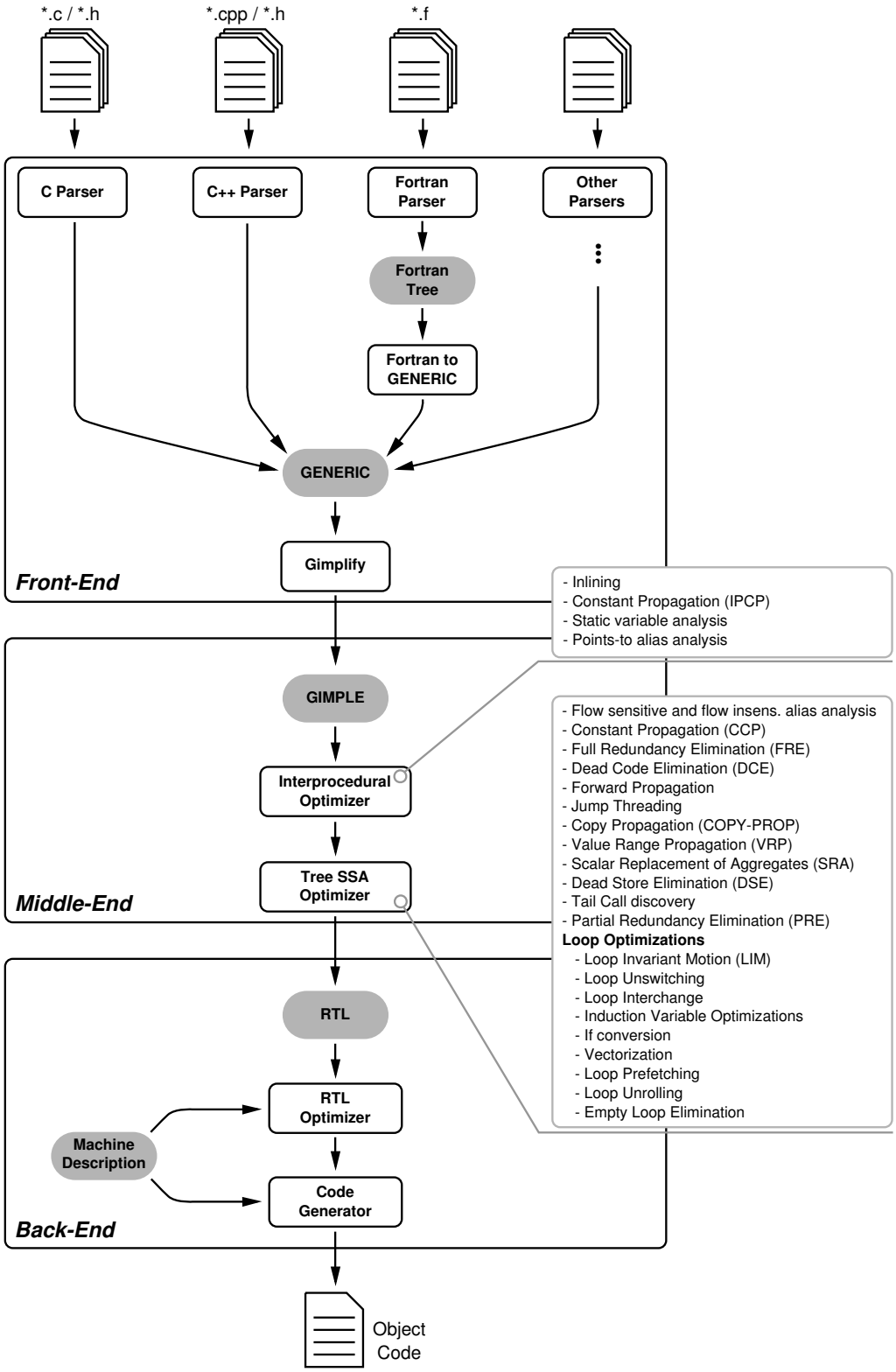

Figure 4.1: GCC compilation flow with common optimizations, adapted from [80].

converted to a subset of GENERIC. The special role of C can be explained by the historic evolution of the compiler. When the development of the GCC started, the IR was tailored to the specific characteristics of C, as it was the only supported input language at the time.

4.1.2 Middle-End

At this point of the compilation process, the complete compilation unit is converted (gimplified) to GIMPLE in Tree SSA (static single assignment) form (cf. [72, 78]). The details of GIMPLE and Tree SSA are described in the next Section 4.2.

This IR has no references to the source language or the target architecture. Therefore, it is suitable for carrying out all general code optimizations that have beneficial effects on most input languages and target platforms. This includes fine-grained optimizations – like common subexpression elimination – as well as coarse-grained optimization, e.g. function inlining. The optimization tasks are divided into so-called passes. A detailed explanation of the pass structure of the middle-end is given in Section 5.6.4.

4.1.3 Back-End

The back-end converts the optimized GIMPLE tree to the register transfer level (RTL). This Lisp-like[2] structure is suitable for optimizations close to the target machine (e.g. instruction scheduling or register allocation). It was the default intermediate representation before the introduction of GIMPLE in GCC 4 [82]. Finally, this representation is converted to assembly by using the machine definition of the target architecture. Afterwards, the integrated GNU Binutils [73] are used to convert the assembly to target byte code, which is emitted as an output file.

[2] Lisp is a functional, adaptable high-level programming language.

4.2 Intermediate Representation

The following section will describe GIMPLE and GENERIC. These representations are used by the middle-end optimization passes and represent the most important structures in context of this thesis.

In the early years of GCC development the internal representation for optimizations was RTL, which was directly derived from AST – the data structures used by the front-ends to represent the parse-trees. In the past, each front-end converted the input language to its specific flavor of parse-tree; hence, optimizations had to be adapted to the single front-end. This process was very error-prone and hard to maintain. In order to overcome these limitations in GCC version 4, the compiler framework was reworked, and GENERIC and GIMPLE were introduced. Both were implemented on top of a tree data structure using a formalism called static single assignment.

4.2.1 Tree SSA

The GCC internal Tree SSA structure [79, 72] is the foundation of the IR. It presents all information of the code structure concerning function-nodes, statements, operands, and operations. Single function-nodes and statements are structured as trees. A sequence of statements is structured as a tree chain. An example Tree SSA structure for a `PLUS_EXPR` is presented in Figure 4.2. Access to the sub-nodes is ensured with the macro interface provided by `tree.h`.

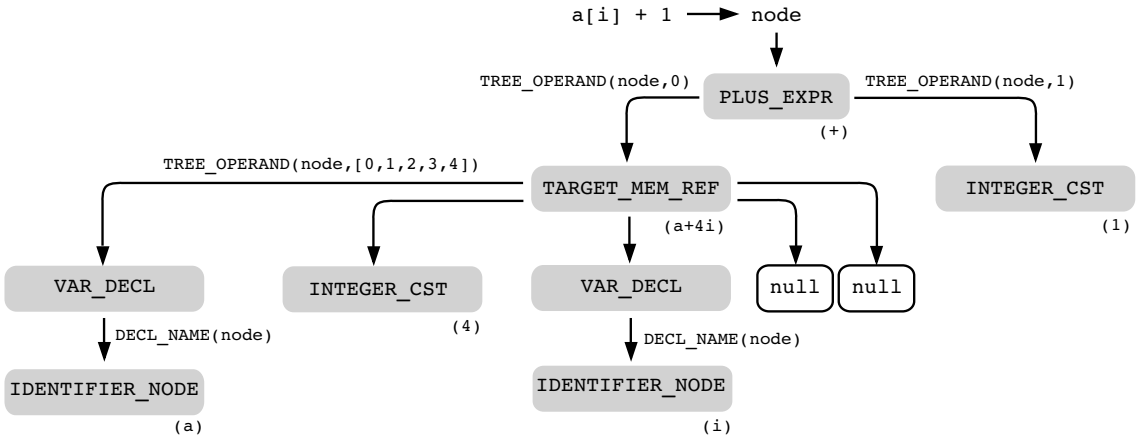

Figure 4.2: Tree structure of an `PLUS_EXPR` with `TARGET_MEM_REF`

Besides several variable and constant types, the given expression contains a memory reference (`TARGET_MEM_REF`) that is of particular interest within this thesis. Such references require special treatment during hardware generation, as they trigger the communication to external memories. The memory reference itself describes an access pattern to resolve the address and returns a type that describes the data gathered by the access.

The following two memory access patterns are used during the hardware generation process:

`target_mem_ref` A memory reference using a base b, a constant offset o, an index i_1 with a step size of s and a secondary index i_2 (operating on byte level). All values except the offset o and i_2 could be expressions. The address A for the access is calculated as follows:

$$A = b + i_1 \cdot s + i_2 + o$$

Not all variations of this formula are allowed or make sense for a given target architecture. For instance, both targets (ARM and SpartanMC) that are used in the context of this thesis waive the usage of i_2.

`mem_ref` A memory reference using only a base address b, which can be variable or constant, and a constant offset o. The address A is the sum of base and offset:

$$A = b + o$$

Listing 4.3 shows a sample loop that contains two array accesses. Both accesses are represented as `target_mem_ref` (`MEM[...]`) in the GIMPLE transcript of the loop body in Listing 4.4. The first memory reference accesses element `a` of the structure `foo`. It uses `ptr` as base, the variable `ivtmp` as index and step size, and an offset of zero. The second access reuses the base and the index of the first one but sets the offset to eight in order to read element `b`. The example gives an impression of the representation of memory references within GIMPLE. Furthermore, it shows clearly that the actual access parameters depend on the surrounding GIMPLE statements and could only be vaguely predicted from the given C code. In order to guarantee a correct handling of memory accesses, the

```
struct foo {
    int a;
    int b;
};
int main(struct foo *ptr) {
    int i, a;
    for (i=0; i<100; i++) {
        a += ptr[i].a + ptr[i].b;
    }
    return a;
}
```

Listing 4.3: Example loop with memory accesses

```
...
ivtmp.20 = PHI <0, ivtmp.19>
_9   = MEM[base:ptr.6, index:
    ↪ivtmp.20, offset:0B];
_16  = ptr.6 + ivtmp.20;
_10  = MEM[base:_16, offset:4B];
_11  = _9 + _10;
a_12 = _11 + a.18;
ivtmp.19 = ivtmp.20 + 8;
...
```

Listing 4.4: GIMPLE transcript of the loop body showing memory references

plugin performs an exact mapping of the GIMPLE representation to hardware, even if the representation provides room for improvements.

It should be noted that Tree SSA contains many more different memory access patterns, e.g. `BIT_FIELD_REF`, `COMPONENT_REF`, and `ARRAY_REF`. For the purpose of hardware generation, those patterns must be lowered by the plugin code or by previous optimization passes to a pattern or a combination of the patterns described above.

The handling of memory accesses in the plugin is based on `target_mem_ref`s, which represent the most generic description of a memory reference in the context of the given plugin implementation.

The GCC framework provides various convenience functions to traverse the Tree SSA structures. This helps to determine types of variables and constant values. Additionally, it allows the tracking of variable scopes and, finally, enables a basic alias analysis.

Another feature of the internal tree structure is the SSA property of variables. The central idea is the versioning of variables. Every time a new value is assigned to a variable, the compiler generates a new version of that variable. This is advantageous for many optimizations applied in the middle-end, for instance, *constant propagation*, which tries to compute expressions at compile time in order to avoid unnecessary computations at runtime. The SSA form supports this optimization as it helps the compiler to keep track of constant values across

different statements.

Naturally, programs are seldom written as a straight sequence of statements. As shown in code sequence 4.5, they contain control flow, like conditionals or loops, that leads to ambiguous values for SSA variables (cf. Listing 4.6).

```
x = get_x();
if (x > 0) {
  a = 0;
}
else {
  a = 42;
}

return a;
```

Listing 4.5: C code with conditional

```
x_1 = get_x();
if (x_1 > 0) {
  a_2 = 0;
}
else {
  a_3 = 42;
}
a_4 = PHI <a_2, a_3>
return a_4
```

Listing 4.6: SSA values with *PHI function*

In the example, it is impossible to compute the return value for `a` at compile time. The merge of such conflicting versions of SSA variables is called the *PHI function*. It indicates to the optimizers that `a_4` is not resolvable at compile time but can be `a_2` or `a_3` at run time.

4.2.2 GENERIC and GIMPLE

GENERIC is the interface between the language-specific front-ends and the middle-end optimizations. It was implemented as the superset of all tree representations in GCC. Hence, the conversion to GENERIC removes all language dependencies from the program but still preserves its structural complexity. The example in Listings 4.7 and 4.8 illustrates the lowering of C code (4.7) to GENERIC (4.8)[3].

In order to perform code optimizations, GENERIC is further lowered to GIMPLE, which provides a more abstract representation of the program. GIMPLE is a fusion of two IRs. On the one hand, there is SIMPLE, which was used by McGill University's McCAT compiler [74]. Its contribution to GIMPLE is the three-address representation of statements. On the other hand, there is GENERIC,

[3]It should be noted, that IRs exists in only internal compiler structures. The transcripts used in this thesis cannot show the complete expressiveness provided by the IR.

C Code

```
#define LEN 100

int foo_sum (int a[LEN]) {
  int sum = 0;
  for (int i=0; i<LEN; i++) {
    if (a[i] < 0) {
      return error();
    }
    sum += a[i];
  }
  return sum;
}
```

Listing 4.7: C code example

GENERIC

```
foo_sum (int * a) {
  int sum; {
    int i;
    i = 0; sum = 0;
    goto <D.6488>;
    <D.6487>:;
    if (*(a + (i * 4)) < 0) {
      return error ();
    }
    sum = *(a + (i * 4)) + sum;
    i++ ;
    <D.6488>:;
    if (i <= 99) goto <D.6487>;
    else goto <D.6489>;
    <D.6489>:;
  }
  return sum;
}
```

Listing 4.8: GENERIC code transcript

which imposes several structural restrictions (s. [78]). The fusion of both names, GENERIC and SIMPLE, results in GIMPLE. As GIMPLE allows only the use of three-address statements, complex expressions must be partitioned with temporaries. GIMPLE itself can be divided into High GIMPLE, which still contains lexical scopes and control structures, and Low GIMPLE, only using labels and `goto` instructions. Both representations are shown in A.2 and A.3 for the code given in Listing 4.7.

4.2.3 Control Flow Graph

Besides GIMPLE and Tree SSA, the GCC requires a third type of IR, the control flow graph (CFG) [82, p. 285 ff.]. As the name implies, this structure contains information about the control flow of an application. The representation of the CFG is built on top of GIMPLE and consists of so called *basic blocks* and *edges*. Each node of the CFG is a basic block. A basic block contains a sequence of non-branching GIMPLE statements. The transfer of control is represented by edges

between the blocks.

For instance, an `if` statement occurring in a basic block splits the control flow and creates two conditional edges. The following two blocks contain the statements for the *then* and *else* branches, respectively. The block behind such a conditional contains the PHI functions of all variables set in the conditional basic blocks.

Figure 4.9 illustrates the GIMPLE structure and the CFG that is generated for the function in Listing 4.7. This level of abstraction is the basis for the generation of hardware accelerators presented in this work.

Dominance Relation

The control flow structures of the CFG can be ordered according to their dominance relation. A node *i (inferior)* is dominated by another node *s (superior)* if each path from the entry of *i* must pass *s* in order to reach *i*. In this context, each basic block has exactly one immediate dominator. This block *s* dominates its subsequent basic block *i* directly without dominating any other dominator of *i*. The immediate dominator can be used to construct a dominator tree by defining an edge from each block to its immediate dominator. This tree is provided by the GCC and can be traversed. Among others, PIRANHA uses the domination tree to derive domination levels in order to identify parallelizable basic blocks.

Interface for Application Profiling

Naturally, the control flow of an application has a strong impact on its execution time. Thus, the GCC framework provides many optimization options for CFGs. Besides an integrated branch prediction, each basic block provides estimations about its *frequency* and its actual *count*. The former represents an estimation generated during compile time by the usage of *static profiling*. It provides information about how often a basic block may be executed within a function. The latter contains the hard-counted numbers of executions measured during a program run. This way of data acquisition is called *dynamic profiling*.

For dynamic profiling, GCC provides the compiler flag `-pg`, which equips the program to produce profiling data ([82] p. 290 ff.). After a test run of the application, the profiling data is written to an extra output file and can be used in an external

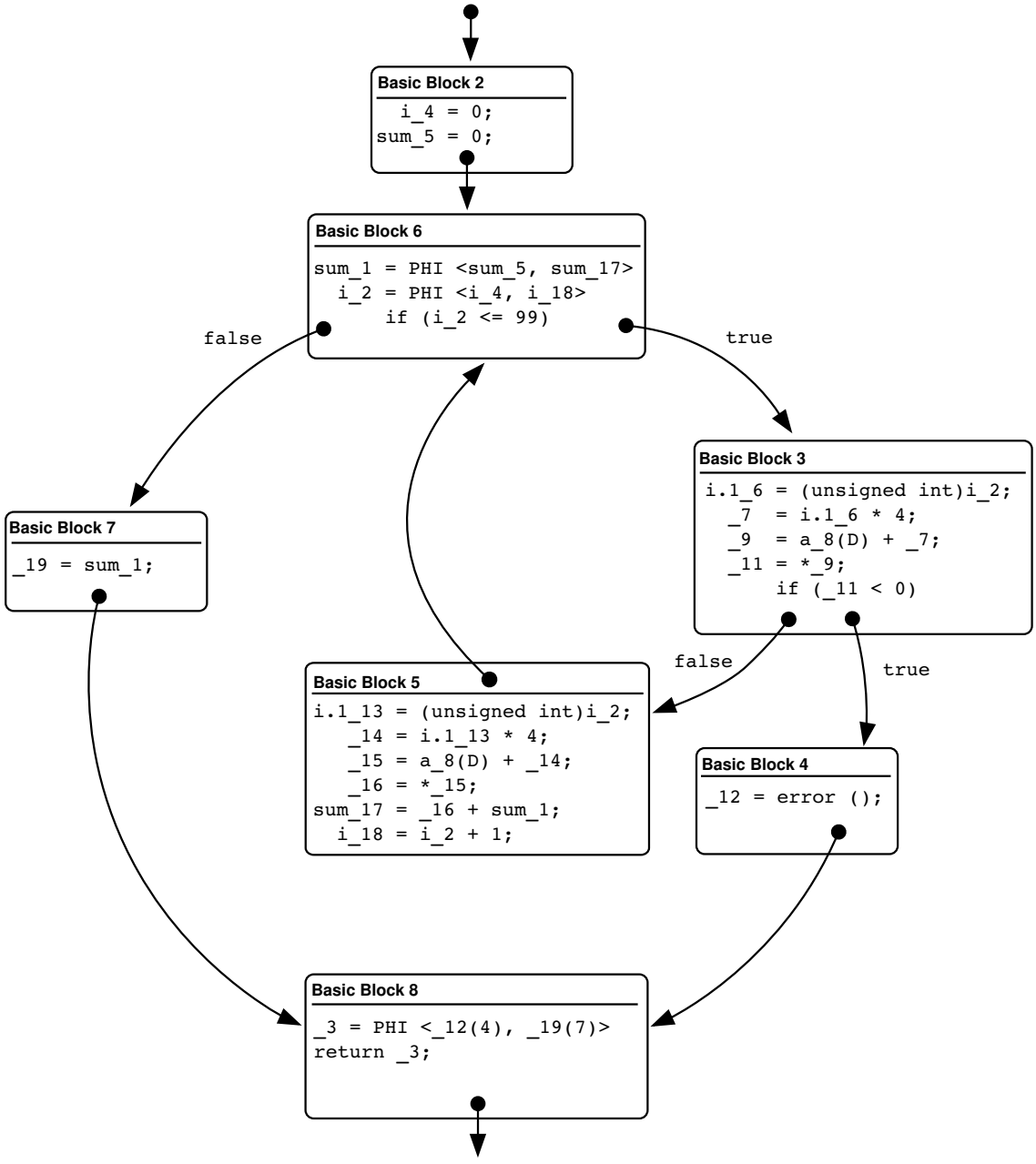

Figure 4.9: Example transcript of a CFG with GIMPLE basic blocks

tool or for a second GCC run. This method provides accurate information about the program but depends on the input data used for the test.

The profiling data used in this thesis is gained by static profiling only. Embedded systems typically use a cross-compiler toolchain for program translation. This implies that the architecture on which the compiler runs does not correspond to the architecture of the compilation target. Hence, the execution of the test for dynamic profiling would require additional effort. Particularly for small embedded systems that run on bare-metal code with a very limited set of libraries, gaining a dynamic application profile would not be trivial.

4.2.4 Register Transfer Language

The final optimizations of the compiler are done on a low-level IR called register transfer language (RTL) [82, p. 225 ff.]. This IR is a hardware-based representation for an abstract target architecture with an infinite number of registers. RTL defines and optimizes low-level features, e.g. memory-addressing modes, word sizes, types, and compare and branch instructions.

After the final GIMPLE passes, the IR is lowered to RTL which is still represented by internal structures and pointers. The GCC provides several interesting optimization passes for RTL structures, e.g. modulo scheduling. However, such optimizations are not strictly target-independent, as they are influenced by the machine description. This description provides target-specific RTL templates in a lisp-like structure that will further generate target-specific compiler code. Thus, the existing RTL optimizations are not applied for hardware generation in order to remain independent from the underlying architecture.

4.3 Optimization Passes

Optimizations during the compilation process are carried out in passes. A pass modifies the structure of the IR or converts the current representation into another IR. The sequence of their execution is specified in `passes.c`, which contains a fixed schedule for all optimizations. It is not recommended to rearrange this sequence, as the resulting IR of a pass does not necessarily fit the IR expected by an inserted one (even if both passes use GIMPLE). To attenuate this

problem, the C structure of a pass provides fields for its prerequisites and results. Such conditions may be considered by other passes, unfortunately this is not obligatory in GCC 4.8.3. The C structure of a pass defines a hook for a gate function and an execute function. The former is the entry point for the actual implementation and can be used to verify user-defined conditions before executing the pass, for example, the value of a compilation flag. Passes are grouped hierarchically according to their underlying IR and their scope of optimization, e.g. functions, loops, etc.. Optimizations that transform one IR into another (e.g. GENERIC to GIMPLE, AST to Tree SSA) are called lowering passes.

The GCC provides over 180 passes for optimizations based on GIMPLE, Tree SSA, and RTL. This includes various loop optimizations as well as common scalar optimizations, e.g. dead code elimination (DCE), conditional constant propagation (CCP), and dead store elimination (DSE). A selection of the most important optimizations is given in Figure 4.1.

Generally, the GCC distinguishes between three classes of optimization passes:

Interprocedural (IPA) passes work on the inter-function level and perform optimizations for the complete compilation unit. These passes traverse the internal call graph structure of the GCC, which contains every function of the compilation unit. The call graph is used for the application analysis described in Chapter 5.

Intraprocedural Passes work on loops and sequences of statements. These passes perform high-level optimizations on the granularity of GIMPLE statements, Tree SSA, and the CFG. Consequently, they are also called *GIMPLE* passes. The HLS described in Chapter 7 is implemented as such a GIMPLE pass.

RTL Passes are used to prepare and optimize the program code for the target architecture. RTL can be regarded as an assembler language for an abstract machine. Nevertheless, it uses several target-specific features.

The GCC provides a vast number of parameters to specify which passes are finally run within the compilation process. Almost every optimization pass can be influenced, activated or deactivated by such parameters. In order to simplify the usage of the compiler, sets of optimization passes can be selected by

the parameters `-O0`, `-O1`, `-O2`, `-O3`, `-Os` and `-Ofast`. While `-O0` guarantees the shortest compilation time and a comprehensible translation result, `-O1` through `-O3` optimize the code size as well as the execution time. The option `-Os` optimizes the code for size only, while `-Ofast` tries to speed up the code by using mathematical calculations with less accuracy. This option may lead to inaccurate results for programs that require an exact implementation of IEEE or ISO specifications.

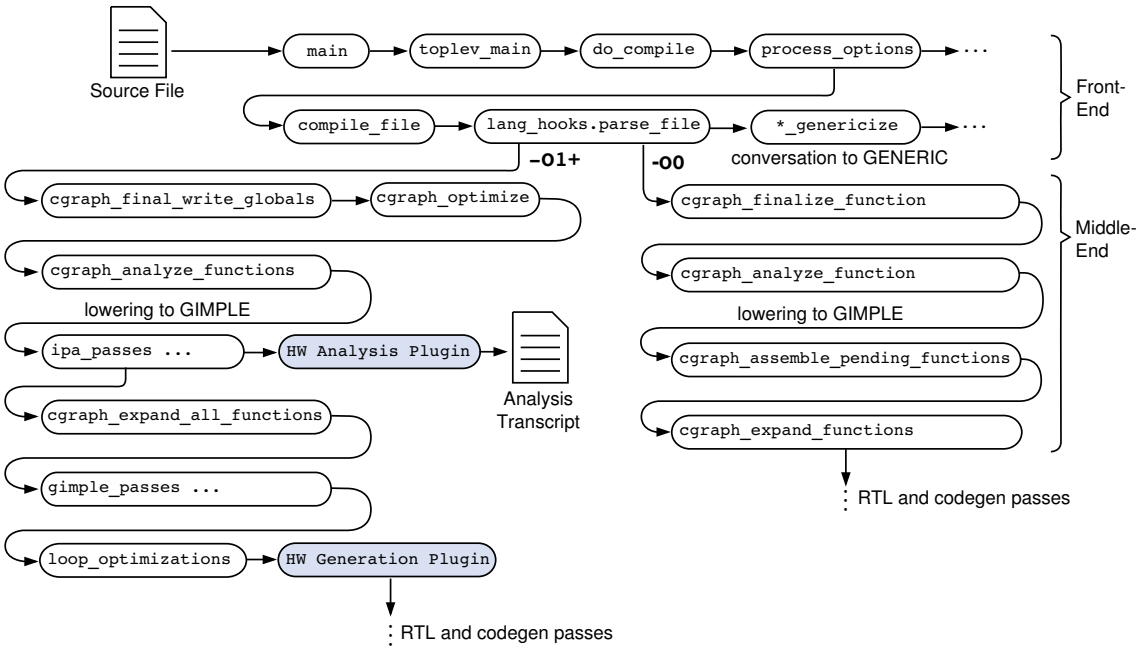

Figure 4.10: Simplified call graph for `-O0` and `-O1` – `-O3`

Figure 4.10 shows a simplified call graph for the optimization level `-O0` and the common optimization levels greater than or equal to `-O1`. The plugin for hardware generation is treated by the GCC as an additional optimization pass (blue boxes in Figure 4.10). For that reason, the hardware generation pass is only called for optimizations greater than `-O0`.

4.3.1 Link Time Optimization

The traditional implementation of the GCC only processes a single compilation unit per compilation run. Hence, the scope of interprocedural optimizations is limited to more or less one source file at a time. For programs that are scattered

across several files, this hinders the optimization passes from using their full potential.

In order to solve this problem, the GCC provides the link time optimization (LTO) framework. Since GCC version 4.5, the LTO framework can be used to expand the scope for optimizations to the whole program or, at least, to the part of the program that is visible at link time. Therefore, a bytecode representation of GIMPLE is integrated as a special section into the emitted object files (`*.o`). These, so-called "fat" objects[4] enable LTO to compose a whole program representation on the GIMPLE-level from the linked application. As LTO triggers the regular GCC optimizations twice, it almost doubles the compilation time of a program.

LTO is activated by using the GCC parameter `-flto` during compilation. Given this parameter the GCC calls two IPA passes `pass_ipa_lto_gimple_out` and `pass_ipa_lto_finish_out` which generate the GIMPLE section of the object file. The actual optimization is triggered when `collect2`[5] detects a linked set of `*.o/*.a` files containing LTO information. This information is aggregated into a single compilation unit that is further processed by regular IPA passes.

Similar to LTO, the generation of hardware accelerators also requires a global view of the application. Although, at first glance, the LTO work-flow seems feasible for the generation of accelerators, the currently available LTO passes do not provide suitable hooks for the required hardware generation passes. Nonetheless, the current hardware generation process is inspired by the LTO work-flow. Both work-flows run the optimization passes twice; the first run is used to gather information while the actual work is done in the second run.

4.4 GCC Plugin Interface

Since version 4.5.0, the GCC framework has provided support for plugins [75, 77]. Such modules are translated to shared objects and can be linked dynamically during the compilation run. The usage of plugins provides several benefits for the development of GCC extensions.

It allows the implementation and testing of new features without bootstrapping

[4] The resulting object could be up to five times the size of the original object.
[5] A tool that, among others, could be used to check the linker output.

the compiler. In particular for cross-compiler toolchains this process is error-prone and very time-consuming. Additionally, it provides a clear separation between the compiler's internal functions and the extension. Hence, it effectively keeps developers from modifying internal functions according to their personal requirements. Finally, it provides flexible and powerful access to the compiler's internal data structures through a well-defined interface.

As described in Section 5.6.4, the GCC optimizations are organized in passes that are executed according to a fixed schedule. In order to execute arbitrary plugin code between these optimization steps, the plugin interface allows the registration of callback functions on several predefined events within the optimization schedule for instance, after execution of all loop-optimization passes or before parsing a function body. In addition, the plugin interface provides a possibility to register regular GCC passes (according to the definition in `tree-pass.h`) instead of arbitrary callback functions. The registration of regular passes can be done dynamically using the *plugin pass manager* which provides a hook for almost every internal pass. Using these hooks allows the integration of own GCC passes between existing ones. The parameters, defining the new pass, are handed to the *plugin pass manager* via the `register_pass_info` structure. Listing 4.11 shows the declaration of this structure for the hardware synthesis pass (`pass_hw_synthesis`). This pass is called directly before finishing all loop optimizations (Lines 2 and 5). The value of `ref_pass_instance_number` is used to insert the pass in a specific instance of the reference pass (`loopdone`). A value of zero activates the pass for all instances.

```
static struct register_pass_info hw_synthesis = {
    .pass = &pass_hw_synthesis.pass,
    .reference_pass_name = "loopdone",
    .ref_pass_instance_number = 0,
    .pos_op = PASS_POS_INSERT_BEFORE
};
```

Listing 4.11: Register pass-info for plugin

4.4.1 Using GCC Garbage Collector with Plugins

With the growing number of optimization passes, the utilization of memory became a severe challenge for GCC developers. This was tackled with the introduction of the GCC Garbage Collector (*GGC*). Hence, most of the objects and structures that are used within the GCC are marked with `GTY((...))` macros. This prepares them for use with the maintenance mechanisms provided by the GGC. The declaration of such types obeys strict rules, e.g. they must be declared in a global scope and must not be static.

Even though the *GGC* provides many benefits for the GCC framework, its handling can be very tedious for the developer. Further information on the *GGC*, its features, and its usage in plugins can be found under [82, p. 608 ff.].

If a plugin only analyzes data, it could use its own memory management. As soon as a plugin modifies the IR or generates data that should be handed to the following passes, it is inevitable to allocate memory under the *GGC* administration. For each *GGC*-controlled structure, the allocation of *GGC* memory is encapsulated by a unique generated function. For example, the `struct GTY(()) dfg {...}` must be allocated with a generated function called `ggc_alloc_dfg()`. The header file `gt.h` containing this function is generated within the plugin build process by a GCC tool called `gengtypes`.

A common data type of the GCC, that is frequently used in the plugin is the *vector*. Vectors are typically used to aggregate data of the same type. In this context, the GCC template class `vec` provides the functionality to add or remove elements as well as to iterate or resize vectors. From the user's point of view, it is necessary to distinguish between two types of vectors: vectors that are allocated on the heap (which is the default creation strategy) and vectors that are allocated in the garbage collector memory. While the former is declared by `vec<data_type> name`, the latter requires the flag `va_gc` (`vec<data_type, va_gc> name`) in order to generate a vector that interoperates with the *GGC* machinery.

4.4.2 Plugin Call

Plugins are translated to a shared object (`*.so`). Hence, the target platform has to support dynamic linking using `dlopen()`. The location of a plugin is given to the GCC via the `-fplugin` flag. Parameters are passed using key-value pairs, e.g.

```
-fplugin=/path/to/<plugin>.so -fplugin-arg-<plugin>-<key1>[=value1].
```

The compiler extension presented in this thesis was first developed for GCC 4.4.5. Thus, it was integrated in the GCC as a set of optimization passes. With an adaption to GCC version 4.8.3, the code was reworked and implemented as a plugin.

Part III

Application Acceleration

5 APPLICATION ANALYSIS

The application analysis is divided into two parts. The first part is carried out in the first of two GCC runs by using the *collector plugin*. It identifies loops within C functions and collects the required loop data for the whole application[1]. This part of the analysis is shown in Figure 5.1 and consists of two steps. Both steps, *collect functions* and *collect loops* (gray boxes in Figure 5.1), are implemented as GCC passes that are integrated by using the plugin hook of the GCC pass manager. *Collect functions* is implemented as an additional simple IPA pass and is scheduled after all GCC internal IPA passes. Its task is to analyze the call graph of all functions. *Collect loops* is implemented as a GIMPLE pass. Its task is to identify loops within functions and extract all relevant loop parameters. The result of the first compilation run is a transcript file containing the static analysis data for each loop of the application.

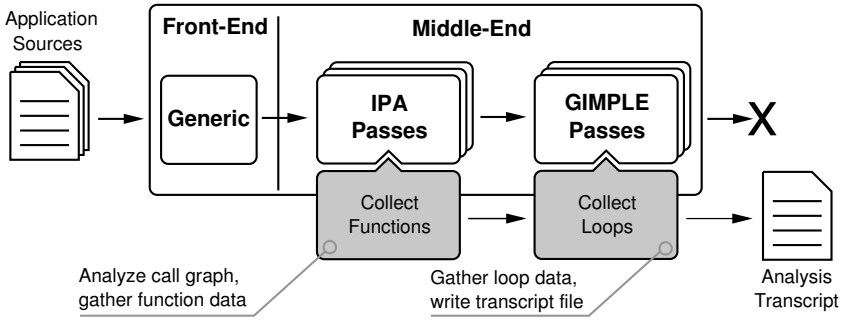

Figure 5.1: First GCC compilation run collecting loop-data

Even though the loop data collection is carried out in the first GCC run, the actual analysis – especially for parameters required for hardware generation – is carried out in the beginning of the second GCC run using the *synthesis plugin*. This part includes the analysis of memory accesses and the evaluation of possible speedups with respect to the desired target architecture. The analysis steps of the second GCC run are discussed in Sections 5.1 to 5.4.1 of this chapter.

[1] This includes every function within all compilation units.

5.1 Function Data Collection

In order to perform inter-procedural optimizations, the GCC generates an internal *call graph* [76] that is accessible in all IPA passes. The inserted pass, `collect_functions`, which gathers the function data, is therefore implemented as an IPA pass.

The call graph consists of unique nodes, each representing a function, and directed edges pointing from the caller function to the callee. Each edge provides a weight that represents the number of function calls out of loops of the caller. For instance, function `foo` in the example call graph 5.2 *(A)* contains a loop that calls `bar` six times.

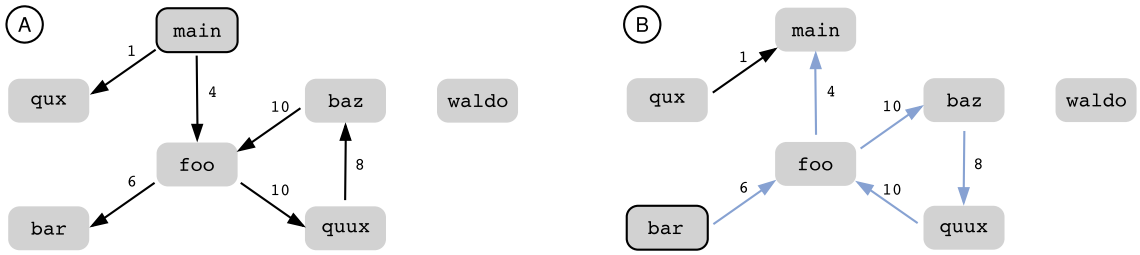

Figure 5.2: Example call graph *(A)* and inverted call graph for `bar` *(B)*

In the work of [133], the call graph is utilized for the *collect_functions* pass. Its only task is to mirror the graph for each compilation unit to local `hw_node` structures. The `hw_node` structures finally contain the number of calls for each function derived from the call graph and the `loop_node` structure containing the loops of each function. The latter is supplemented by the following *collect_loops* pass.

5.2 Loop Data Collection

After collecting all function data, the analysis of the loops for each function is carried out with an additional GIMPLE pass. The lifetime of the required loop data is limited to the loop optimization passes within the GCC. The gate to the sub-tree of these GIMPLE passes is implemented in `pass_tree_loop_init`. The actual loop collection pass "*collect_loops*" is inserted at the end of all loop

optimization passes to benefit from GCC internal optimizations for later hardware generation.

The implemented function to gather the loop data of a given function from the applications source code is `analyze_loop(...)`. This function is used to determine the number of iterations of a loop or loop nest by analyzing the tree of loops recursively. The tree and the required loop data is integrated in the CFG representation of GIMPLE (cf. Section 4.2.3). Further analysis is done by the two additional functions `compute_deps(...)` and `find_funcalls_loop(...)`. The former is used to determine memory references by utilizing the GCC internal function `compute_data_-dependences_for_loop(...)`. Such dependencies are useful in order to estimate the costs of hardware generation. The latter is used to gather function calls within a loop by searching for `gimple_call` statements in the loop body. It is necessary to find such function calls within loops as they represent a blocker for the hardware generation.

Finally, the resulting collection of loop and function data is preserved in an intermediate transcript file for later analysis in the second GCC run. This additional processing step is necessary in order to get a whole application view gathered from multiple compilation units. For example, the call graph analysis of another compilation unit could reveal that the expected number of iterations for an already analyzed loop is increased by another function call. For this reason, the actual analysis of the gathered data is moved to the second GCC run.

5.2.1 Transcript File

The first GCC run is used to analyze all compilation units and appends the analysis data of each unit to a transcript file[2]. At the end of the first compilation run, the transcript file contains the data of all loops regardless of whether the loop is suitable for hardware generation or not.

The data in the transcript file mirrors the hierarchical order of the source program. The analyzed compilation unit represents the top level, containing a sequence of included functions that further contain the subordinated loops. Listing 5.5 shows a simplified version of the transcript file for two compilation units 5.3 and 5.4.

The property well_nested is of particular interest for the later hardware generation.

[2]It should be noted that the data collection does not check for semantic errors like duplicate function names.

```
int fun3(int a, int b);
int fun1(int a, int b) {
    int c;
    for (int i=0; i<10; i++)
        c += fun2(b + a, a - b);
    return c;
}

int fun2(int a, int b) {
    for (int i=0; i<30; i++) {
        a += fun3(a, b);
        b -= a;
    }
    return a+b;
}
```

Listing 5.3: Source code of unit1.c [145]

```
int fun3(int a, int b) {
    for (int i=0; i<100; i++) {
        a += b;
        if (a > 200 )
            break;
        b--;
    }
    return a;
}
```

Listing 5.4: Source code of unit2.c [145]

```
unit1.c
  function=fun2
    loop1
    countall.low=29
    countall.high=0
    deps_computed=0
    call=1
    well_nested=0
    -fun3
  function=fun1
    loop2
    countall.low=9
    countall.high=0
    deps_computed=0
    call=1
    well_nested=0
    -fun2

unit2.c
  function=fun3
    loop3
    count.low=99
    count.high=0
    deps_computed=1
    call=0
    well_nested=1
```

Listing 5.5: Analysis transcript file after compiling unit1.c and unit2.c [145]

It indicates whether a loop or a loop nest is synthesizable at all. As in the example, `loop1` and `loop2` contain function calls, `fun2` and `fun3`, respectively; hence, they are excluded from hardware generation. According to the transcript file in Listing 5.5, only `loop3` with the property `well_nested=1` will be considered for hardware generation in the second GCC run. The GCC uses `double_int` types to store potentially large numbers, e.g the loop count. This type consists of an unsigned `host_wide_int` for the lower part of the number and a signed `host_wide_int` for the upper part. The complete transcript file of the code examples 5.3 and 5.4 can be found in A.4.

5.3 Processing the Transcript File

As shown in Figure 5.6, the parsing of the transcript file is carried out at the beginning of the second GCC run. The process is implemented as an additional GIMPLE pass `collect_data`, which sorts the analyzed loops by the given criteria. Later, the resulting list of accelerator candidates is used by the hardware generation pass.

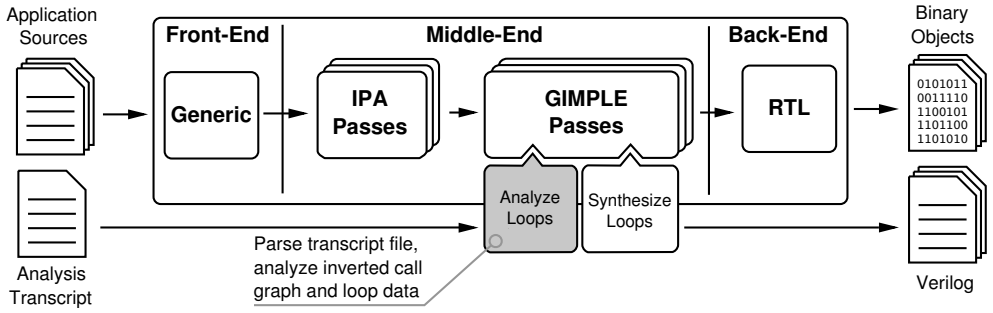

Figure 5.6: Second GCC compilation run analyzing loop-data

Besides the parsing of the transcript file, the key functionality of the pass is implemented in the `update_graph` function. It is used to search for the function that is called most often within the application. This optimization task can be boiled down to the question: Which is the longest Eulerian path[3] in the call graph with the highest product of all involved edge-weights?

5.3.1 Call Graph Analysis

The Eulerian path problem for a graph consisting of vertices V and edges E; $G = (V, E)$ is solvable with the complexity of $\mathcal{O}(|E^2|)$. The algorithmic solution in [133] is based on a depth-first search (DFS) for each node. As regular call graphs can contain cycles, it is mandatory to determine a termination condition for the DFS by marking visited edges. Intermediate results are stored in the leaves of the graph. The implementation of the search algorithm was simplified by inverting the edges and setting the current function as the root of the graph (Figure 5.2 *(B)*). For each node $v_i \in V$, the algorithm generates a tree of accessible nodes. In order to find the longest path (maximum of the product of all path weights),

[3] The longest path of the graph by using each edge only once.

the DFS has to investigate all possible paths in the call graph. This results in an exhaustive search with the complexity of $\mathcal{O}(2^{|E|})$. As call graphs of nearly all programs are sparsely populated with edges, additional optimizations of the search algorithm are not reasonable. An inverted graph is shown in Figure 5.2 *(B)* for the function `bar`, which also represents the leaf with the longest path (blue arrows) with respect to the example in 5.2 *(A)*.

5.3.2 Loop Analysis Functions

In order to sort the analyzed loops in a candidate list for hardware generation, the analysis of the call graph and loops utilizes several functions that implement different search criteria. The available compare functions are parametrizable via compilation parameters (cf. C.2) or could be implemented according to the special requirements of the user. Custom weighting functions must be implemented with respect to the following structure definition:

```
static struct {
    char name[32];
    loop_cmp cmp_fun;
    loop_cnt cnt_fun;
} lcmp_funs[]
```

Listing 5.7: Interface for compare functions [133]

The 31 characters for the name (cf. Listing 5.7) can be used to specify an arbitrary compare function via a compiler parameter. The `loop_cmp` type specifies the actual compare function. The second type `loop_cnt` allows the specification of a metric to count loops.

Available implementations of compare functions are `lcmp_iterations(...)` or `lcmp_-iterations_strict(...)`. Both use `lcnt_iterations(...)` as a metric to sort functions by their number of calls from wrapping loops. The compare function distinguishes between countable and uncountable loops. If the compared loops are countable, the actual number of iterations is evaluated in `lcnt_iterations(...)` and is used to rank the loop. If the loop is uncountable the resulting number of iterations is one or infinite.

The function `lcmp_iterations(...)` always gives uncountable loops a better

rating compared to countable. In this case, uncountable loops are considered to have infinite iterations. The function `lcmp_iterations_strict(...)` makes no difference between countable and uncountable loops. In this case, the actual rating of uncountable loops is carried out by `lcnt_iter-ations(...)`, resulting in an iteration count of one.

It is also possible to use a compare function that is based on the actual number of instructions or on a metric that considers the estimated hardware effort of a loop. Additional compare functions can be easily implemented by using the interface in Listing 5.7.

Besides the configurable compare functions the second GCC plugin requires two additional parameters. First, the `hw-nloops` parameter is used to define the maximum number of loops in the candidate list. Second, the `hw-ninstr` parameter is used to determine an upper bound of the software instructions per loop. The latter may be useful to limit the logic resources used for a single hardware accelerator. A comprehensive overview on plugin parameters and their function can be found in C.2.

5.4 Performance Estimation

In the previous Sections 5.3 – 5.3.2, the analysis of loops was the subject of discussion. The determined number of instructions or iterations was used to generate an ordered candidate list of potential application hotspots.

If necessary, this list can be further filtered by an analysis of the expected runtime. For this purpose, the presumed speedup of individual candidates is determined and, in addition, an efficiency metric is introduced. In order to allow a pre-sorting of accelerator candidates, the required data must be collected and evaluated at the beginning of the *synthesis pass* in the second GCC run (cf. Figure 5.8). The basic structure of the performance analysis described below have been developed in [132].

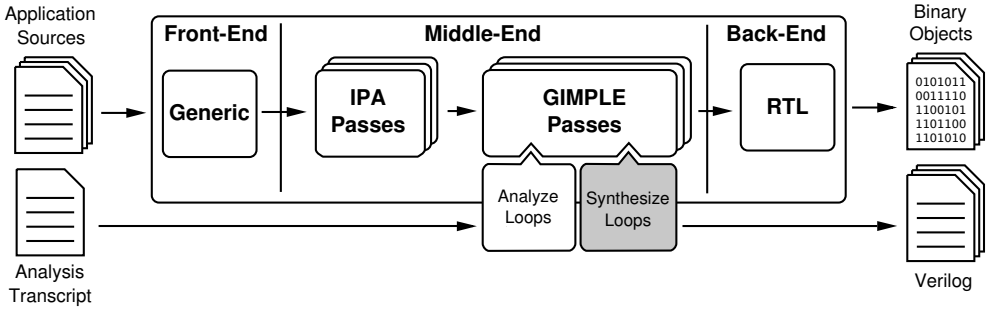

Figure 5.8: Second compilation run with *synthesis pass*

5.4.1 Performance Metric

The efficiency E of a hardware accelerator is defined by the ratio of gained speedup S to the complexity of the hardware C. The speedup itself is defined as the quotient of the software cycle count T_{SW} and the hardware cycle count T_{HW}:

$$E = \frac{S}{C} \quad , \qquad S = \frac{T_{SW}}{T_{HW}}.$$

Obviously, the speedup is of particular interest for the accelerator generation. To calculate the estimated speedup of an accelerator, one requires the maximum number of software instructions of the loop body I_{max} as well as the number of clock cycles for the longest path CP in hardware.

The number of software instructions is determined by GCC internal functions evaluating the longest path of GIMPLE statements in a loop. Therefore, each GIMPLE statement in the path is weighted in order to estimate its impact on the final assembler code. The used instruction counter is derived from the function inlining pass and uses its heuristic to estimate the weight of each statement. Due to architecture-specific back-end optimizations, this method may lead to slightly inaccurate results.

The value of CP is determined from the generated states of the hardware implementation. As memory accesses in hardware introduce architecture-specific latency-cycles (T_{mem}), they must be taken into account when calculating the value for CP. Accordingly, CP can be calculated as sum of states respectively cycles of the loop body T_{body} and the latency for all memory accesses of the loop ($N_{mem} \cdot T_{mem}$). Assuming a constant cycle count T_{mem} for each access. In addition, a static overhead OV is required that arises from the number of data transfers IO and the latency T_{mem} at the beginning and end of the accelerator call as well as the overhead T_{call} of the accelerator function itself. Consequently, CP and OV are defined as follows[4]:

$$CP = T_{body} + N_{mem} \cdot (T_{mem} - 1)$$
$$OV = IO \cdot T_{mem} + T_{call}.$$

As each architecture uses it own memory interface with dedicated latencies, the values for T_{mem} and T_{call} are defined via architecture-specific parameters. The resulting equation for the speedup of a given loop with the number of iterations $n \in \mathbb{Z}$ is defined as

$$S = \frac{n \cdot I_{max}}{n \cdot CP + OV} \qquad S \approx \frac{I_{max}}{CP} \quad | n \to \infty.$$

For a large iteration count n, the influence of the overhead OV is negligible. This is the case for uncountable loops analyzed with `lcmp_iterations(...)` as compare function (cf. Section 5.3.2).

Obviously, the speedup is further influenced by the difference in the operating frequencies between accelerator and host processor. For this reason, the resulting

[4] $T_{mem} - 1$ is required to subtract the one state-change cycle from T_{mem} that is already included in T_{body}

speedup must be scaled by the ratio of the accelerator clock frequency F_{Acc} to the host processor frequency F_{CPU}. The resulting equation for the speedup is

$$S = \frac{F_{Acc}}{F_{CPU}} \cdot \frac{n \cdot I_{max}}{n \cdot CP + OV} \quad .$$

In order to make accelerators on the candidate list comparable to each other, a coarse efficiency metric is introduced. It uses an artificial value for the hardware complexity that is loosely coupled with the resource footprint of the resulting hardware design. The implemented equation for the complexity is influenced by the number of nodes in the data-flow graph (DFG) N_{DFG} and memory accesses N_{mem} of the accelerator. In [132] it was decided to regard both components as orthogonal vectors. Accordingly the scalar value of the complexity can be evaluated as follows

$$C = \sqrt{N_{DFG}^2 + N_{mem}^2} \quad .$$

Finally, the overall efficiency is defined by the following equations for a small (5.1) and large (5.2) amount of iterations:

$$E = \frac{F_{Acc}}{F_{CPU}} \cdot \frac{n \cdot I_{max}}{(n \cdot CP + OV) \cdot \sqrt{N_{DFG}^2 + N_{Mem}^2}} \quad , \tag{5.1}$$

$$E \approx \frac{F_{Acc}}{F_{CPU}} \cdot \frac{I_{max}}{CP \cdot \sqrt{N_{DFG}^2 + N_{mem}^2}} \quad | n \to \infty. \tag{5.2}$$

The described metric is implemented in the function `reorganize_vec`, which is used to mark candidates that do not meet the given objectives in terms of speedup or efficiency.

5.4.2 Pseudo-Scheduling

To determine the longest execution path, N_{DFG}, and N_{mem}, the analysis functions rely on HLS algorithms that are normally used for the regular hardware generation. The "pseudo-scheduling" is the last analysis step which is performed at the

beginning of the actual hardware synthesis pass. It carries out all scheduling and optimization steps according to the parameter configuration of the plugin, except for the GIMPLE modifications and the actual HDL generation, which are excluded to avoid non-reversible modification of the application code. The resulting DFG is analyzed in the function `compute_complexity` in order to obtain the required parameters for the performance metric.

As shown in Figure 5.9, the pseudo-scheduling is carried out for each accelerator candidate in the corresponding candidate list. After applying the pseudo-scheduling, the vector incorporating the loop candidates holds additional performance information for each loop. This data is used by the performance analysis afterwards. Finally, the vector is sorted (Reorganize Candidates in Figure 5.9) according to the desired performance metric specified by the following plugin parameters:

`hw-synth-strategy=<0|1|2>` The strategy specifies the behavior of the analysis. The current implementation supports three modes.

- `0` This is the default value. The given efficiency must be achieved for the synthesis of accelerators. If the efficiency is not predictable due to an unknown number of iterations, $n \to \infty$ is assumed and the simplified formula (5.2) is used.
- `1` The given efficiency must be achieved, but candidates with an unpredictable efficiency are excluded.
- `2` The efficiency estimation is ignored. All available candidates will be synthesized.

`hw-synth-min-efficiency=<value>` This is the minimal value for the required efficiency, which is evaluated if the strategy is `0` or `1`. If the strategy is set to `2`, the parameter is ignored. The efficiency is set to `0` by default which automatically includes all candidates with a potential speedup greater than one.

After the pseudo-scheduling has been applied, the HLS is carried out, which will be discussed in the following chapters.

It should be noted that the metric described here contains some daring assumptions. For example, the value used for N_{DFG} relies on an identical resource

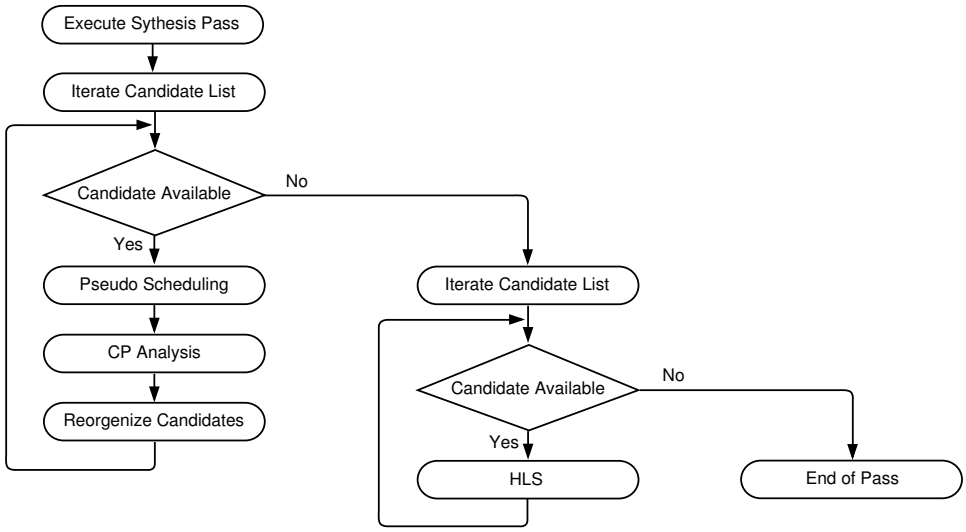

Figure 5.9: Flow-chart of the pseudo-scheduling

consumptions for different operators in the DFG, which may be not the case in a real hardware design. Further the value for I_{max} is based on a internal heuristic of the GCC that may be too inaccurate for this particular use case.

Later experiments indicated that the described efficiency estimation was too coarse for practical application. Therefore, this plays only a minor role in the further sections of this thesis. However, PIRANHA provides the infrastructure for later implementations of a more accurate estimation.

5.5 Theory of Memory Access Analysis

After determining application hotspots on the granularity of loops and loop nests, the memory access analysis aims to optimize the generated hardware. First attempts at hardware generation have shown that the number of states in the critical path strongly depends on the number of memory accesses within the loop. Due to the characteristics of C, different memory accesses must be considered to alias each other as long as nothing is known about the semantic of the accesses. In this context, aliasing means that they potentially reference and modify the same memory location. Without further analysis, memory accesses must be scheduled consecutively in order to preserve the correctness of the hardware kernel. This hinders a prefetching of data for multiple loop iterations and a usage

of parallel memory ports. It also introduces an upper bound for the possible maximum ILP in a loop. Even after applying all optimizations on the C code, there will always remain an execution path that is determined by the shortest sequence of memory read and write operations. Without memory access analysis, this path cannot be further optimized.

On these grounds an alias analysis is applied for all memory accesses in the GIMPLE structure considered for hardware acceleration. Besides information about the potential aliasing of memory accesses, this analysis provides detailed data about the used memory access patterns. Such data is highly useful for further streaming or caching mechanisms. Naturally, the results of a static alias analysis are not complete. For this purpose the analysis during compile time is supplemented with a light-weight online analysis during runtime.

The GCC contains a powerful loop optimization framework [71] providing two functionalities that help with analyzing memory accesses. First, the *data dependency* interface, which provides a list of all memory accesses within a loop. Second, the *scalar evolution* interface, which allows analyzing the change of values of variables during each iteration of a loop.

If scalar evolution can produce proper results during static code analysis, the provided data can be used to determine the exact stride and step width of a memory access with respect to its starting address.

5.5.1 Analysis Example

Assuming a simple C function, as given in Listing 5.10, it is possible that the parameters `in` and `out` point to overlapping or identical regions in memory. If this is the case, then the two parameters would *alias*. In most cases, it is impossible to determine at compile time whether or not two pointers reference the same memory location. The compiler optimizations must assume that each modification of "`out`" potentially affects the memory that is referenced by "`in`".

For the given example, this implies that the generated code must always wait for the result of the multiplication and store it to the memory before the load of the next value can be carried out. Hence, it is impossible to apply any parallel loop execution, as the iterations are assumed to depend on each other. Moreover, it

```
void mul(int* in, int* out, int a) {
    int i;
    for (i = 0; i < 64; i++) {
        out[i] = in[i] * a;
    }
}
```

Figure 5.10: Potential pointer aliasing

prevents the accurate prefetching of data and the utilization of a streaming interface as a fast way to transfer data to custom hardware. Ironically, this behaviour is often not intended by the developer. Nonetheless, the provided C syntax that informs the compiler about non-aliasing pointers, e.g. "`int* restrict out`" is used very rarely in legacy code.

This problem is addressed by reversing the typical causal chain of alias analysis. Instead of assuming that a given code aliases until the opposite can be proven by the GCC, the implemented approach always presumes non-aliasing accesses. This is possible, as the generated accelerators provide a software fall-back mechanism (cf. Section 6.3). The final decision on the code execution path that should be used is taken at application runtime.

The aim of the memory access analysis is to identify the used symbol, the number of iterations, the stride, and the step width of an access. For the given example in Listing 5.10, both symbols `in` and `out` are used for one access in 64 iterations. Furthermore, scalar evolution would identify a stride of four (assuming an integer size of 4 byte) and a step width of one. A possible aliasing of both accesses can be excluded if the distance between `*in` and `*out` is more than 256 byte. Note that the latter can only be evaluated at application runtime after determining the actual value of the function parameters.

5.5.2 Chains of Recurrences

Before going into the details of scalar evolution, the *chain of recurrence* (*chrec*[5]) formalism has to be introduced. It forms the basis of the approach used in the GCC interface and the returned data relies on its mathematical structures.

[5]This abbreviation is used to keep conformance with the terminology used in GCC

Chains of recurrences were first described by Bachmann, Wang, and Zime [70]. The idea behind the formalism was to unify the construction of recurrence relations of functions for a given interval. This allows a fast and algorithmic evaluation of a function at a given number of points. Being more formalistic, it computes the value of a function $G(x)$ where $x \in \{x_0 + ih \mid i \in \mathbb{N}\}$ with constant x_0 and h. The aim is to generate a *chrec* Φ with $\Phi(i) = G(x_0 + ih)$ that can be evaluated potentially faster than the direct evaluation of G at a given point. Therefore, the actual function value $G(x_0 + ih)$ is calculated by using the value for $G(x_0 + ih - h)$.

An example taken from [70] elucidates this basic concept. Given a function $G(x) = 3x + 1$, the sequence $G(x_0 + ih)$ with $i \in \mathbb{N}$ could be naïvely calculated by re-evaluating the function for each i. The following calculations show the classical approach *(left side)* and the approach using a recurrence relation *(right side)*:

$$\begin{aligned} G(x_0) &= 3x_0 + 1 \\ G(x_0 + h) &= 3(x_0 + h) + 1 \\ G(x_0 + 2h) &= 3(x_0 + 2h) + 1 \end{aligned} \qquad \begin{aligned} G(x_0) &= 3x_0 + 1 \\ G(x_0 + h) &= G(x_0) + 3h \\ G(x_0 + 2h) &= G(x_0 + h) + 3h. \end{aligned}$$

As $3x_0 + 1$ and $3h$ can be pre-calculated, the approach using the recurrence relation takes one addition instead of one multiplication and one addition for each value of G.

The formalism for a simple recurrence relation is called *basic recurrence*. It is represented by a tuple containing the constant φ_0, the operator $\odot$ which can be either "$+$" or "$\cdot$" and the function f_1 defined over the natural numbers $\mathbb{N}$.

$$f = \{\varphi_0, \odot, f_1\}.$$

For instance, the function $G(x_0) = 3x_0 + 1$ would be represented by the following basic recurrence:

$$G' = \{3x_0 + 1, +, 3h\},$$

with $f_1(h) = 3h$ describing the evolution of the function value of G with respect to the starting value $3x_0 + 1$.

The basic recurrence is defined as function $f(i)$ over $\mathbb{N}$ by

$$f(i) = \{\varphi_0, +, f_1\}(i) = \varphi_0 + \sum_{k=0}^{i-1} f_1(k) \text{ or}$$
$$f(i) = \{\varphi_0, \cdot, f_1\}(i) = \varphi_0 \prod_{k=0}^{i-1} f_1(k),$$

depending on the operator used for $\odot$. These basic recurrences are special cases for first-order linear recurrences. Functions that are more complicated will be represented by chrecs containing several constants. The generalized tuple of a chrec Φ consists of the constants φ_i, $i \in \{0, \ldots, k-1\}$, the operators $\odot_i$, $i \in \{1, \ldots, k\}$ with $\odot \in \{+, \cdot\}$, and a function f_k

$$\Phi = \{\varphi_0, \odot_1, \varphi_1, \odot_2, \varphi_2, \ldots, \odot_k, f_k\}$$

or recursively as a function over $\mathbb{N}$

$$\Phi(i) := \{\varphi_0, \odot_1, \{\varphi_1, \odot_2, \varphi_2, \ldots, \odot_k, f_k\}\}(i).$$

This section has given only a brief introduction of chrecs in order to provide all information to understand the methods used in the thesis. More detailed information about the generation of the chrecs as well as several additional rules and applications for chrecs are presented in [70].

5.5.3 Scalar Evolution

In this thesis, the formalism provided by chrecs is of great importance. It is used to extract the subscript of memory accesses for regular arrays and structures. Such subscripts contain incremental changes of the accessed address locations relative to their base address. This is also referred to as the *scalar evolution* of an access. An algorithmic implementation to extract the scalar evolution from the Tree SSA structure of the GCC has been described in the work of D. Berlin et. al. [71]. For the development of the analysis within this thesis, this boils down to the usage of two functions `analyze_scalar_evolution(...)` and `instanciate_parameters(...)`. The first function is used to generate a chrec that still contains references to other variables inside the loop. The second

function tries to resolve the variables' values until the chrec only contains values that are constant in the context of the loop. The resulting function for an sample pointer subscript `p[i*4+j]` accessing integers with the length of 4 byte will be represented as follows:

$$G(i,j) = \mathrm{p} + 16i + 4j = \{\mathrm{p}, +, 16\}\,(i) + 4j = \{\{\mathrm{p}, +, 16\}\,(i), +, 4\}\,(j).$$

The base address, given here with the pointer *p*, and all evolutions of the access have been reduced to constants (albeit this is a trivial transformation in this example). It should be noted that the inner chrec $\{p, +, 16\}(i)$ can be regarded as a constant φ_0 with respect to *j* in the outer chrec $\{\varphi_0, +, 4\}(j)$.

The tree node type `POLYNOMIAL_CHREC` is used for the GCC-internal representation of chrecs. It contains the tree operands *b*, *loop*, and *s* which refer to a chrec $\{b, +, s\}$ with respect to the loop *loop*, while *b* may be a chrec describing another loop. Consequently, nested loops are expressed as nested tree nodes of `POLYNOMIAL_CHREC`s.

5.5.4 Extraction of Access Patterns

A sequence of nested slices, as introduced in [135], can be used for further processing of the information gained by scalar evolution. A slice is defined as a tuple (*start, end, stride*) that aggregates all information of a memory access pattern. The value for *start* depends on the base address of the access or is defined by a constant offset. Both cases are handled separately. For this reason, the *start* value is set to zero by default. Accordingly, a slice is defined as $S = (n, s)$, where *n* is the number of subsequent accesses made and *s* is the number of bytes between the starting points of consecutive accesses. For instance, the stride (4,3) specifies the address offsets $o = \{0, 3, 6, 9\}$.

The generation of slices is demonstrated for a code example taken from [135], calculating a four-dimensional matrix-vector product (cf. Listing 5.11). Due to array subscripts influenced by both loop variables (*i, j*), the determined set of slices must be nested as well. Thus, at least two of three array accesses are described by multiple nested slices.

An example for nested slices is given in Figure 5.12, which shows two slices $S_0 = (5, 5)$ and $S_1 = (2, 2)$. In this example, the nesting means that the inner

```
void matrix4_mul_vec4(int *mat, int *vec_in, int *vec_out) {
    int i, j;
    for (i = 0; i < 4; ++i) {
        int accum = 0;
        for (j = 0; j < 4; ++j) {
            accum += mat[i*4+j] * vec_in[j];
        }
        vec_out[i] = accum;
    }
}
```

Figure 5.11: Four-dimensional matrix-vector product [135]

slice S_0 "runs" entirely for each iteration of the outer slice S_1. Such nested slices can also be written as a finite sequence of slices, starting with the outermost slice:

$$\mathcal{S} = (S_0, S_1, \ldots, S_{|\mathcal{S}|-1}).$$

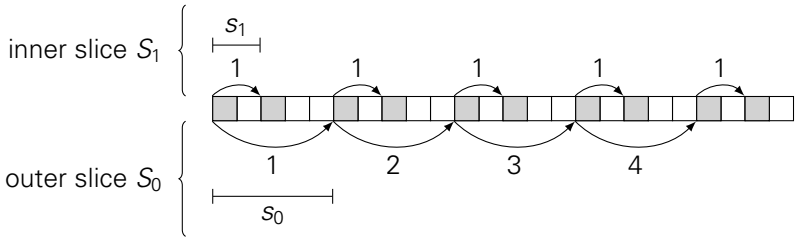

Figure 5.12: Memory access pattern produced by two nested slices S_0 and S_1 [135]

Note that each access in Example 5.11 was extended by an additional slice with a stride of one and an iteration count of four. This stride is used to model the multiple-byte access caused by the 4-byte integer types used in the code. The three memory accesses of the four-dimensional matrix-vector product can be described by the following set of strides:

Output Vector `vec_out[i]` (write access): The write access to the output vector can be determined in a straight forward manner. The subscript is determined by the iteration of the outer loop described by the stride $S_{V_{out};0} = (4, 4)$, which results in the following set:

$$S_{V_{out};0} = (4, 4), \quad S_{V_{out};1} = (4, 1), \quad \mathcal{S}_{V_{out}} = (S_{V_{out};0}, S_{V_{out};1}).$$

Matrix `mat[i*4+j]` (read access): The matrix access requires a nested slice, as the iteration variables of both loops (i,j) are part of the subscript $o = 4i + j$. The outer slice $S_{M;0}$ has an upper bound of four and a stride defined by $o_1 = 4i$, which results in $s_1 = 16$. The inner slice $S_{M;1}$ again has an upper bound of four and a stride defined by $o_2 = j$, which resolves to $s_2 = 4$. Consequently, the slices $S_{M;0} = (4, 16)$ and $S_{M;1} = (4, 4)$ are added to the sequence:

$$S_{M;0} = (4, 16), \quad S_{M;1} = (4, 4), \quad S_{M;2} = (4, 1), \quad \mathcal{S}_M = (S_{M;0}, S_{M;1}, S_{M;2}).$$

Input Vector `vec_in[j]` (read access): The input vector is an interesting case. It is part of the loop nest, but requires only the loop variable of the inner loop for its subscript, which is presented by a slice $S_{V_{in};1} = (4, 4)$. The fact that the outer loop has no effect on the subscript implies that the inner loop has to wait four iterations of j before carrying out its next access. This is represented by a dummy slice for $S_{V_{in};0}$. The dummy slice use the iteration number of the outer loop (four) and a stride of zero. The resulting set of slices supplemented by $S_{V_{in};0} = (4, 0)$ and $S_{V_{in};1} = (4, 4)$ is:

$$S_{V_{in};0} = (4, 0), \quad S_{V_{in};1} = (4, 4), \quad S_{V_{in};2} = (4, 1), \quad \mathcal{S}_{V_{in}} = (S_{V_{in};0}, S_{V_{in};1}, S_{V_{in};2}).$$

5.5.5 Aliasing of Access Patterns

After the sequences of slices have been successfully generated the actual alias analysis can be carried out. In case of array accesses this requires finding potential overlaps between read and write accesses. Later, such information can be used to generate hardware structures enabling concurrent memory accesses.

Finding shared points between two access subscripts is a common task in the area of compiler construction. This problem can be expressed by a system of homogeneous linear Diophantine equations [68]. Such equations are of the form

$$a_0 + a_1 x_1 + \ldots + a_n x_n = 0$$

with the coefficients $a_i \in \mathbb{Z}$ and the variables $x_i \in \mathbb{Z}$, $1 \leq i \leq n$ or, alternatively, of the form

$$a_0 + a_1 x_1 + \ldots + a_n x_n \geq 0$$

which is hereinafter called a linear Diophantine inequality. In order to compare two sequences of slices, a system of such equations can be specified by introducing values and bounds for x_i and a_i derived from the current slices.

System of Diophantine Equations

Assume that there are two sequences $n := |\mathcal{S}_A|$ and $m := |\mathcal{S}_B|$, containing nested slices each with the stride s_i and the iteration n_i. Let there also be α_i, $1 \leq i \leq n$, and β_j, $1 \leq j \leq m$ values over $\mathbb{Z}$, associated with the iteration variables of the loop. As all iterations are constrained to be greater than zero, the first two inequalities would be:

$$0 \leq \alpha_i \qquad \text{for all } i \tag{5.3}$$

$$0 \leq \beta_j \qquad \text{for all } j. \tag{5.4}$$

Furthermore, the iteration variables are bound to the number of iterations:

$$\alpha_i \leq n_{A;i} - 1 \qquad \text{for all } i \tag{5.5}$$

$$\beta_j \leq n_{B;j} - 1 \qquad \text{for all } j \tag{5.6}$$

and the coefficients $a_i \in \mathbb{Z}$, $1 \leq i \leq n$ and $b_j \in \mathbb{Z}$, $1 \leq j \leq m$ are defined by the strides of the given slices:

$$a_i := s_{A;i} \qquad \text{for all } i \tag{5.7}$$

$$b_j := s_{B;j} \qquad \text{for all } j. \tag{5.8}$$

Finally, the equation used to determine if both sequences of slices touch the same offset is defined as follows:

$$c = \left(\sum_{i=1}^{n} a_i \alpha_i \right) - \left(\sum_{j=1}^{m} b_j \beta_j \right) + d. \tag{5.9}$$

In case the sequences overlap each other this equation is true. The offset to the single slices is calculated by the product of α_i and the stride a_i or b_i and β_j, respectively . These both offsets are subtracted and the base offset d is added. The constants c and d ($c, d \in \mathbb{Z}$) are introduced to take into account that the slices may not start at an offset of zero. For instance, slices accessing different `struct` members introducing the offsets c_1 and c_2 are presented by $c := c_2 - c_1$ and $d := d_2 - d_1$, respectively.

Adapting this to the example of the code in Listing 5.11, the alias detection has to be carried out on the sequences determined in Section 5.5.4. The aliasing is tested pair-wise using at least one write access with another memory access. For this reason, the two combinations (`vec_in[j], vec_out[i]`) and (`mat[i*4+j],vec_out[i]`) will be relevant for possible aliasing:

1. **Input vector vs. output vector**
   Considering the sequences $\mathcal{S}_{V_{in}}$ and $\mathcal{S}_{V_{out}}$, the test for aliasing can be formulated by using Equation 5.9:

   $$0 = 0 \cdot \alpha_1 + 4 \cdot \alpha_2 + 1 \cdot \alpha_3 - 4 \cdot \beta_1 - 1 \cdot \beta_2 + d$$
   $$0 = 4\alpha_2 + \alpha_3 - 4\beta_1 - \beta_2 + d.$$

   Note that the offset between the base addresses `vec_in` and `vec_out` is only known at runtime. Thus, the value for d is substituted with both variables:

   $$0 = 4\alpha_2 + \alpha_3 + \texttt{vec\_in} - 4\beta_1 - \beta_2 - \texttt{vec\_out}.$$

   The remaining loop constraints result in the following inequalities:

   $$0 \leq \alpha_2 \leq 3$$
   $$0 \leq \alpha_3 \leq 3$$
   $$0 \leq \beta_1 \leq 3$$
   $$0 \leq \beta_2 \leq 3.$$

2. **Input matrix vs. output vector**
   Considering $\mathcal{S}_M$, $\mathcal{S}_{V_{out}}$, the usage of Equation 5.9 results in:

   $$0 = 16 \cdot \alpha_1 + 4 \cdot \alpha_2 + 1 \cdot \alpha_3 - 4 \cdot \beta_1 - 1 \cdot \beta_2 + d$$
   $$0 = 16\alpha_1 + 4\alpha_2 + \alpha_3 - 4\beta_1 - \beta_2 + d$$
   $$0 = 16\alpha_1 + 4\alpha_2 + \alpha_3 + \texttt{mat} - 4\beta_1 - \beta_2 - \texttt{vec\_out}.$$

   Placing the loop constraints in equation form leads to the following inequalities

that are added to the system:

$$0 \leq \alpha_1 \leq 3$$
$$0 \leq \alpha_2 \leq 3$$
$$0 \leq \alpha_3 \leq 3$$
$$0 \leq \beta_1 \leq 3$$
$$0 \leq \beta_2 \leq 3.$$

As the loop boundaries are the same for all iterations and the integer size matches the matrix dimension, the derived inequalities all look similar.

As mentioned already, in this example, the offset *d* is only known at runtime. Consequently, the system can only be solved by containing *d* as a remaining parameter. The result is a set of possible offsets *d* for which aliasing occurs . Subsequently, the alias analysis can be broken down to a runtime test for a set membership. As shown in the following Section 5.6, this is used to enable an elementary alias analysis at runtime.

Solving a system of Diophantine equations is NP-complete and a solution typically utilizes integer-programming techniques [69]. In context of this thesis, a library will be used for that purpose. A detailed description of the library used is given in Section 5.6.1.

5.6 Implementation of Memory-Access Analysis

After explaining the theoretical principles of the alias analysis, this section will describe its implementation. In order to provide a full access analysis, the GCC plugin must provide a comprehensive specification of all involved memory accesses and their relative position to each other. Unfortunately, such information (especially the base address of an access) is only rarely available during compilation time. If at all, the required data can be derived after linking time, but usually it is available only during the execution time of the program. Due to this lack of information, the alias analysis must follow a two-layered approach. The implementation of the plugin must provide internal structures for runtime analysis but

also compilable C code for runtime use. According to these premises, the tasks of the actual implementation are:

- The detection and resolution of memory references inside a loop.
- The generation of an *alias matrix* that stores information about aliasing between all involved symbols[6]. This matrix must further contain the set of base offsets for which aliasing occurs.
- The generation of C code that performs the runtime analysis for the given alias matrix.

All three tasks have been implemented in [135] as a C++ class, the `MemrefAnalyzer`. However, the actual implementation uses a helper library to solve the system of equations describing the potential alias pattern between two symbols.

5.6.1 Integer Set Library

As already mentioned in Section 5.5, the computational complexity for the solution of a system of Diophantine equations (cf. 5.3 to 5.9) is NP-complete [69]. Furthermore, a universal algorithmic solution is not trivial. Fortunately, there is a library for that work, the Integer Set Library (libISL) [83]. It provides operations on sets of integer points, e.g. for set difference and union, and beyond that, it contains tools for the analysis of conditions derived from Diophantine equations. The libISL is a integral part of the toolset used for polyhedral optimizations.

Polyhedral Optimizations

The concept of polyhedral optimizations became popular as the classical target specific optimizations (applied in the compiler back-end) turned out to be no longer sufficient for current architectures. This was fueled by emerging architectures that typically provide the potential for parallel computing on different cores; others may contain micro optimizations that allow for instruction-level parallelism. It turned out that the sheer number of such architectures necessitates a more holistic approach, which uses a higher level of abstraction. Polyhedral

[6] A symbol describes one distinct memory reference in the loop

optimizations get their name from the underlying mathematical structure. Used in integer space, a polyhedron can be represented as a set of linear Diophantine inequalities.

In the GCC framework, polyhedral optimizations were introduced with GRAPHITE (GIMPLE Represented as Polyhedra with Interchangeable Envelope) [81]. The framework focuses on optimization of whole loop nests and loop sequences. In order to enable parallelization, it performs loop re-nesting but also loop fusion or fission. For the tool-flow in this thesis, most of the outcomes of GRAPHITE are not directly usable without further refinements. Nevertheless, some concepts of the polyhedral community are very interesting in the context of HLS and might be worth investigating in future.

Adoption of Access Pattern to libISL

The libISL is designed for working with sets and relations of integers. It uses generic sets that consist of so-called basic sets. Basic sets are always defined for a numerical range and a set of constraints restricting its possible members. In order to use libISL for alias analysis, the given sequences of slices must be transformed into such sets. First, the so-called universe U, has to be specified according to the values of the Diophantine equation system defined in Equations 5.3 to 5.9:

$$U = \left\{(\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_m, d) | (\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_m, d) \in \mathbb{Z}^{n+m+1}\right\}.$$

Second, the set of constraints is specified:

$$P = \left\{(\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_m, d) \in \mathbb{Z}^{n+m+1} | \textit{equation 5.3 to 5.9}\right\}.$$

Starting from these sets libISL can be used to compute an alias set A containing all conditions of d[7] (the base address) for which aliasing occurs

$$A = \{d \in \mathbb{Z} | \textit{condition}\}.$$

The condition can be either a range of values or a rather complex expression including existentially quantified variables. The following example illustrates the

[7] The other variables α_i, and β_j were removed from the results as the specific iteration that aliases is not of interest. This operation can be carried out by LibISL and is called projection.

results of the alias analysis generated by libISL for the code of the matrix vector multiplication in Listing 5.11 that was already discussed in Section 5.5.5.

1. **Input vector vs. output vector**
   The universe and constraints sets are:

$$\begin{aligned} U = \{\ & (\alpha_1, \alpha_2, \alpha_3, \beta_1, \beta_2, d) \in \mathbb{Z}^6\ \}, \\ P = \{\ & (\alpha_1, \alpha_2, \alpha_3, \beta_1, \beta_2, d) \in \mathbb{Z}^6 \mid \\ & 0 \leq \alpha_1 \leq 3 \wedge 0 \leq \alpha_2 \leq 3 \wedge 0 \leq \alpha_3 \leq 3 \wedge \\ & 0 \leq \beta_1 \leq 3 \wedge 0 \leq \beta_2 \leq 3 \wedge \\ & 0 = 4\alpha_2 + \alpha_3 - 4\beta_1 - \beta_2 + d\ \}. \end{aligned}$$

   The resulting alias set A, after eliminating the other variables α_i and β_j, is:

$$A = \{d \in \mathbb{Z} \mid -15 \leq d \leq 15\}\,. \tag{5.10}$$

2. **Input matrix vs. output vector**
   The universe and constraints sets are:

$$\begin{aligned} U = \{\ & (\alpha_1, \alpha_2, \alpha_3, \beta_1, \beta_2, d) \in \mathbb{Z}^6\ \}, \\ P = \{\ & (\alpha_1, \alpha_2, \alpha_3, \beta_1, \beta_2, d) \in \mathbb{Z}^6 \mid \\ & 0 \leq \alpha_1 \leq 3 \wedge 0 \leq \alpha_2 \leq 3 \wedge 0 \leq \alpha_3 \leq 3 \wedge \\ & 0 \leq \beta_1 \leq 3 \wedge 0 \leq \beta_2 \leq 3 \wedge \\ & 0 = 16\alpha_1 + 4\alpha_2 + \alpha_3 - 4\beta_1 - \beta_2 + d\ \}. \end{aligned}$$

   The resulting alias set A is:

$$A = \{d \in \mathbb{Z} | -63 \leq d \leq 15\}\,. \tag{5.11}$$

   As expected, in this example, the alias interval increases, as the memory region touched by the access to `mat` is larger than the memory region touched by `vec_in`.

For the sake of clarity, the example above generates only two simple alias sets. This makes it handy enough to understand the principle. In order to show the

capabilities of libISL, a more complex example is provided in Appendix A.1. The underlying C code in A.1 generates interleaved accesses by using structures. As a consequence, the resulting alias sets contain rather complex conditions that cannot be represented as simple interval expressions.

5.6.2 Resolution of Accesses

As mentioned at the beginning of Section 5.6, the first tasks of the implementation are the detection of memory references and the generation of corresponding memory access structures. For this purpose, the resulting structures from the GCC internal function `compute_data_depen-dences(...)` are used. This function provides a sequence of data references for the current loop as well as structures describing how those data references relate to each other. These structures are called data dependence relations (DDR).

By analyzing the DDR structure, it is sometimes possible to obtain alias information about the memory access without performing any elaborate analysis. For instance, if the keyword `restrict` was used in the program code or a re-traceable `malloc` was called for generating the pointer in question, the GCC marks the memory references as independent. In that case, the following analysis reuses this data and provides direct results without any further analysis.

The DDR structures contain, among others, a direct reference to the GIMPLE statement, the GENERIC tree of the memory reference, and a Boolean, indicating the direction (read or write) of the access. In order to get information about all memory accesses within the loop, the provided sequence of accesses is iterated and each access is analyzed using the functions of `MemrefAnalyzer`.

As described in Section 4.2.1, the general case of a memory reference tree is `target_mem_ref`. The memory access of a `target_mem_ref` can be expressed by the equation $A = b + i_1 \cdot s + i_2 + o$. Here, the type of A defines the width of the access. In order to gather information about such memory references the *chrec* describing the evolution of the base b is generated and analyzed. If this is successful, the further expressions, e.g. index i_1 with stride s are analyzed in the same way. However, the current implementation assumes that i_1 and s are constants within the loop nest. After their analysis, the resulting slice is added to the results of the base analysis while the constant o is added to the

static offset field of the `memory_access`.

The function `analyze_data_access_base(...)` is the entry point for the analysis process. This function triggers all analysis steps that are required to generate the `memory_access` structure which is the result of the access resolution. This structure contains the following fields:

`min_offset` Describes a constant offset that is applied to the base before any other field is evaluated.

`slices` A list of slices that describes the access to the base of the reference. The slices are ordered from the outermost to the innermost loop.

`word_size` The width of a single memory access.

`known` A value of *true* indicates that this access was successfully analyzed. If the analysis returns `unknown access`, it means this value is set to false. Otherwise, it is always true.

`mode` This value indicates whether the access is a read access (`AM_LOAD`) or a write access (`AM_STORE`).

`dr` Data reference of this access as returned by the generating GCC function

The analysis is carried out in `analyze_data_access_base(...)` for each memory reference provided by the DDR structures. A survey of the single processing steps within this function is given in Algorithm 1. In general the analyzed tree `base` could contain five possible values that imply a different handling by the algorithms described in this section:

1. `base` **is an SSA name**: In this case the defining GIMPLE statement is analyzed. If the SSA name is defined in a super loop or somewhere else beyond the scope of the loop nest, the referred value is obviously constant within the loop. If the SSA name is defined within the loop, the defining statement must be analyzed recursively.
2. `base` **is a constant**: In this trivial case the `base` can be used directly as the constant value within every associated loop.

3. `base` **is a binary expression**: The result of such an expression is constant if both operators are constant, which resolves to a simple constant base. If the expression contains a non-constant value (e.g. an SSA name or another expression) and a constant value, the resulting analysis uses the further analyzed non-constant value as a parameter for the later runtime analysis. The constant part is interpreted as an offset. Expression containing two non-constant operands are not supported by the current implementation.

4. `base` **is a** `NOP_EXPR`: If this occurs, the expression is ignored and the single operator of the `NOP_EXPR` is analyzed further.

5. `base` **is a chrec**: If the base is described by a *chrec* ($\{b, +, s\}_{loop}$), it is considered to be non-constant with respect to the current loop. The stride s is extracted and the included base b is analyzed further.

First, the *chrec* structure (cf. Section 5.5.2) `scev` for the current base with respect to the loop is extracted by using the functions from scalar evolution. These GCC functions return an *invalid* scalar evolution structure if the base is constant within the current loop. However, they do not check for changes in the remaining loop nest. For this reason, it is necessary to do this with the function `is_constant_in_loop(...)`. This function checks whether the *chrec* does change within the given outer loop[8]. If this is the case, the base expression b is further analyzed in the function `analyze_data_access_base_chrec(...)`. If the base turns out to be constant, the related value is directly used as the base for the current access. Finally, the value of the base is *normalized* in the function `normalize_base(...)` by extracting the actual components of the base. The last two functions do the major work of the analysis and will be described in detail in the next sections.

Base Analysis

The function `analyze_data_access_base_chrec(...)`, as shown in Algorithm 2, is used to recursively resolve the previously found *chrec*. The function terminates if the base turns out to be constant within the loop nest. The result of the function is a filled in `memory_access` structure with at least one slice for each iteration level.

[8] A non-constant *chrec* can be assumed, if the contained base b is a *chrec* again.

Algorithm 1: Function `analyze_data_access_base` [135]

```
   Input: base loop outermost_loop
   Output: result of type memory_access
1  scev ⟵ analyze_scalar_evolution(base, loop)
2  scev ⟵ analyze_scalar_evolution(outermost_loop, scev)
3  if not is_constant_in_loop(outermost_loop, scev) then
4      result ⟵ analyze_data_access_base_chrec(scev)
5  else
6      result ⟵ scev
7  result ⟵ normalize_base(result)
8  if result == NULL then
       /* Analysis failed                                          */
9      return unknown access
10 return result
```

The function returns the special value *unknown access* if the input is not a *chrec*. This return value indicates that the GCC plugin cannot resolve the current access; hence the `known` field in the `memory_access` structure is set to *false*. In Lines 3 to 5, the base *b*, the stride *s*, and the corresponding loop of the *chrec* (`slice_loop`) are extracted from the *chrec* `base`. As non-constant strides are not supported, the following lines (6–8) are used to check if the stride is a constant value. If the base *b* is not constant within the loop nest, the function calls itself recursively to further evaluate the base of the next nesting level. If the base *b* finally turns out to be constant, the `memory_access` structure is initialized as a new *known* access (Lines 12–16). After the base has been determined, the algorithm has to insert dummy slices (Line 20) for each loop that did not produce a *chrec* (cf. Section 5.5.4). Finally, the new slice as well as the base and the stride are added to complete the `memory_access` structure.

Base Normalization

Another major function of the access resolution is `normalize_base(...)`. This function is called after a base has been evaluated and turns out to be constant with respect to the loop. According to Algorithm 3 the function analyzes the (constant) base in order to generate a tuple {*base_addr, offset*} describing

Algorithm 2: Function `analyze_data_access_base_chrec` [135]

```
   Input: base of type chrec
   Output: result of type memory_access
 1 if not base is chrec then
 2     return unknown access
 3 b ⟵ get_base(base)
 4 loop ⟵ get_loop(base)
 5 s ⟵ get_stride(base)
 6 if not s is constant then
 7     return unknown access
 8 if not is_constant_in_loop(outermost_loop, b) then
 9     result ⟵ analyze_data_access_base_chrec(b)
       /* Exit recursion if result turns out to be an unknown access
          */
10     if result == unknown access then
11         return result
12 else
13     result ⟵ init_memory_access(b)
14 result ⟵ fill_dummy_slices(loop)
15 result ⟵ add_slice(s)
16 return result
```

the base access. This tuple consists of either a single constant or a combination of a variable (the dynamic base address) and a constant offset.

By implementing the conditions one through four for the possible values of the base, as mentioned in the enumeration above, the function can generate the corresponding tuple for each base. Note, that the implementation does not care about `chrec` trees, as these values were already analyzed in `analyze_data_access_base_chrec(...)`.

If the base is not an SSA name, a `NOP_EXPR` or a constant value, the evaluation of binary expressions must be carried out. Therefore, the base is divided into a left-hand-side operand and a right-hand-side operand. Due to the characteristic code translation of the GCC the possible operator can be limited to `PLUS_EXPR`. On the one hand, the GCC does not support multiplicative offsets. On the other hand, potentially negative offsets are transformed by the compiler to an addition

Algorithm 3: Function `normalize_base` [135]

```
   Input: base
   Output: tuple of base and constant {b, c}
1  switch type of base do
2      case SSA_NAME do
3          return {base, 0}
4      case INTEGER_CST do
5          return {base, 0}
6      case PLUS_EXPR do
7          lhs ⟵ base_operand1
8          rhs ⟵ base_operand1
9          lhs_const ⟵ is_constant(lhs)
10         rhs_const ⟵ is_constant(rhs)
11         if lhs_const and rhs_const then
               /* Unsupported, typically not occurs                    */
               /* due to prior constant folding                        */
12             return {NULL, 0}
13         else if lhs_const then
14             return {lhs, $normalize_base(rhs)}
15         else if rhs_const then
16             return {$normalize_base(lhs), rhs}
17         else
               /* Currently not supported                              */
18             return {NULL, 0}
19     otherwise do
20         return {NULL, 0}
```

modulo $2^n - 1$. If both sides of the expression contain a constant value, the tuple $\{\texttt{NULL}, 0\}$ is returned, indicating an unsupported statement. Nevertheless, this is very unlikely to occur, as there is no reason for the GCC to generate this without optimizing it. As already mentioned, combinations of constant and none-constant values are valid whereas the combination of two non-constant values leads to an error. Although very unlikely, this case seem to be possible during compilation and results in an irresolvable memory reference with respect to the current implementation.

5.6.3 Alias Matrix Generation

If all accesses have been analyzed and the source code contains no access that has returned the *unknown access* value, the alias matrix can be constructed. This structure is required to determine which access must be resolved during runtime and which accesses can be regarded as safe. The matrix describes the relationships between all of the memory references of the underlying code. Due to the symmetry of the alias relations[9] only half of the matrix is actually presented in memory. Each cell of the matrix represents a `memory_alias_matrix_entry`. Among others, these entries contain a summary of the alias relation that could be either `MEMREF_NO_ALIAS`, `MEMREF_MAY_ALIAS`, or `MEMREF_ALIASES`. Additionally, an entry comprises the libISL basic sets for each access pair, and an *info* field that indicates the reason for the alias relation found. The reason could be either "read/read" which can be considered safe per se, as two read accesses never alias, "gcc" which indicates a relation found by the GCC-internal analysis or "none" for relations that were not analyzable at all.

Table 5.1: Alias matrix for Listing 5.11

Symbol	`mat`	`vec_in`	`vec_out`
`mat`	`MEMREF_NO_ALIAS`	—	—
`vec_in`	`MEMREF_NO_ALIAS`	`MEMREF_NO_ALIAS`	—
`vec_out`	`MEMREF_MAY_ALIAS`	`MEMREF_MAY_ALIAS`	`MEMREF_NO_ALIAS`

The alias matrix for the loop from Listing 5.11 in Section 5.5.4 is shown in Table 5.1. The example only shows the summary field of the `memory_alias_matrix_entry`. The relations between `vec_in` and `mat` are trivial, as both describe *load* accesses. Also, trivially, the involved accesses cannot alias with themselves.

More interesting is the relationship between `vec_out` and the other symbols. Therefore, Equations 5.10 and 5.11 must be tested at runtime in order to exclude possible aliasing for these accesses.

[9] If symbol *A* aliases with symbol *B*, symbol *B* must also alias with symbol *A*

5.6.4 Runtime Alias Analysis

The basic sets for the runtime alias analysis are generated from the sets of each alias matrix entry. The internal representation of the alias sets uses extensive structures from libISL. Hence, they need to be optimized with regard to the performance of the runtime code. This is accomplished by several libISL functions that minimize the existing alias sets:

- `isl_set_remove_redundancies(...)`: Removes redundant constraints from all basic sets contained in the entry. This function is called after each major modification of the basic sets.
- `isl_set_compute_divs(...)`: Transforms existentially represented conditions[10] into a set of explicit conditions[11] if possible.
- `isl_set_coalesce(...)`: Merges basic sets and removes identical basic sets after merging.

At runtime, the optimized basic sets are evaluated in ascending order defined by their complexity. This is accomplished by storing them in the sorted list `alias_basic_sets`. The complexity of a basic set is determined by the number of existentially represented conditions as the first criterion. As a second criterion the number of explicit conditions is considered.

5.6.5 C Code Generation

The generated C code for the runtime analysis comprises a simplified version of the access matrix with all alias sets. If data prefetching is enabled, a runtime analysis of these alias sets is essential, since the final decision about overlapping memory accesses usually depends on runtime data. The alias sets have to be evaluated right before the accelerator execution starts.

For this reason two types of C representation for the given sets from `alias_basic_sets` are generated. On the one hand, rather simple sets, consisting of a direct comparisons between input values and constants, are transformed into conditional statements. On the other hand, *complex* alias sets are transformed into

[10] Conditions that require an existential quantifier for their definition.
[11] Conditions that are defined by a contiguous interval.

a libISL call with the representation of the alias set as parameter. In these cases, a runtime version of libISL has to perform the alias analysis (cf. Section 6.3).

For each loop nest the generated C code is stored in a file with the name of the source file, the number of the processed loop, and the name of the function formatted into the filename (`<name>_<no>_<fun>_memref_info.h`). In addition, a similar safe naming scheme is applied to variables within the generated code. Therefore, the symbol name and the unique index of the respective access are formatted into the name. Besides the direct integration into the header file, the generated functions are declared as `static inline` in order to minimize the overhead of the runtime code. The required structures are provided by the static file `memref_info_generic.h`.

The accelerator wrapper function is described in detail in Section 6.2.2. A comprehensive overview of the provided structures for memory accesses is given in Table C.3.

6 APPLICATION MODIFICATION

In order to use a generated hardware accelerator from the given C application, the existing code must be extended at compile time by additional statements that implement the software interface of the hardware accelerator. Several existing HLS approaches, e.g. Vivado HLS [60] consider this processing step as irrelevant because they focus on the generation of single IP cores. If integration into a host processor system is required in such an HLS tool, adaptation of the host software has to be carried out manually. Other approaches like LegUp [24] or Nymble [34] use source-to-source compilation to integrate accelerator call functions in their application code. The technique presented in this thesis is slightly different from the latter approach. The patching of the application code is carried out on the GIMPLE-level at the end of the second compilation run. As a consequence, the software interface is easily adaptable to the actual generated accelerator. The code-patching process can even be aborted if the accelerator does not turns out to be beneficial. For this reason, the direct modification of the GIMPLE-IR can be regarded as a seamless and flexible way to implement the software modification.

Although the application patching is carried out at the very end of the plugin execution, its implementation should be discussed together with the application analysis. Both processing steps work on the internal GIMPLE representation while the remaining HLS steps use their own internal representation. Hence, besides the application analysis, the application modification (cf. Figure 2.11, *step 1,2, and 4*) is the second processing step of the plugin that directly depends on the underlying GCC version.

6.1 Modifying the GIMPLE Structure

The synchronization between software application and hardware accelerator is implemented via peripheral registers. Such registers are mapped into the address space of the host processor system. The registers are used for initial parameters, either representing loop input and exit variables that are derived from the GIMPLE-IR or control values of the accelerator hardware. The overall software-flow of an accelerator call is sketched in Figure 6.1.

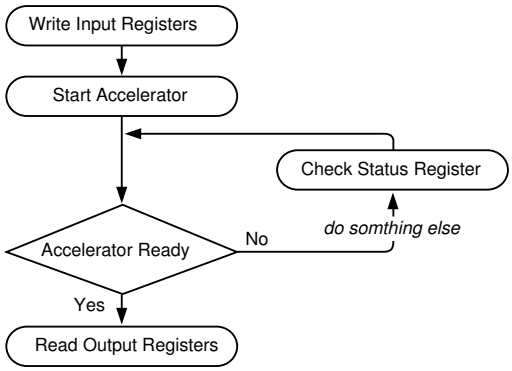

Figure 6.1: Overall software-flow of an accelerator call

The first task of the application patch is to transfer the individual parameter set from GIMPLE variables to accelerator registers. Later, the hardware is started by a control signal. While the accelerator is running, the software waits in a loop for the change of a status register that is set by the accelerator once it is finished. This polling loop can be used for further purposes, e.g. running another software thread. Finally the results are read from the accelerator.

This desired behavior could be achieved by two different implementations. Both modify the GIMPLE representation but with different degrees of invasiveness. The first variant generates completely new GIMPLE statements to write and read peripheral registers, and replaces the original loop with a new polling loop. The second approach generates only an additional function call in the GIMPLE-CDFG. Later, this function contains generated C code for the required accelerator control flow. The original loop remains untouched and is only bypassed by a generated CDFG edge.

A detailed discussion of the implementations is given in the following sections. Regardless of the actual used method, both approaches require an input/output-analysis of loop variables.

6.1.1 Loop Input/Output-Analysis

Before the actual patching of the GIMPLE tree can be carried out, it is necessary to identify all input and output parameters of the chosen loop. In contrast to memory access operations, the set of input and output variables is transferred

in advance and after completing the accelerator operation, respectively. Consequently, these parameters directly influence the register interface of the generated hardware and must be taken into account when modifying the application code.

The set of input and output parameters is gathered in the vectors `in_vars` and `out_vars`. The generation of both vectors is carried out during the GIMPLE analysis in order to provide data for the code patching and generation of the hardware interface.

As data-flow into a loop always causes PHI-nodes within the loop, the vector of input variables can be created by iterating all PHI-nodes within the loop (cf. Algorithm 4). The right-hand-side (RHS) operands of the PHI-statement are treated as the potential input parameters, while the left-hand-side (LHS) is regarded as a local variable. If the RHS turns out to be a constant, the value is directly assigned within the accelerator. As a consequence, the RHS is added to the list of inputs if the variable is a function parameter (GCC macro `SSA_NAME_IS_DEFAULT_DEF`) or the variable is set in a basic block that does not belong to the current loop. The defining statement of a variable could be determined by the GCC internal macro `SSA_NAME_DEF_STATEMENT`.

Algorithm 4: Identify input variables

Input: *CDFG*(*V, E*) *of loop body*
Output: *set of input variables* (*in_vars*)

```
1 foreach phi_node ∈ V do
2     foreach RHS ∈ phi_node do
3         tree ⟵ RHS
          /* RHS operand is a function parameter                    */
4         if SSA_NAME_IS_DEFAULT_DEF(tree) then
5             in_vars ⟵ add(tree)
6         else
7             defining_statement ⟵ SSA_NAME_DEF_STATEMENT(tree)
              /* RHS operand is not part of the loop body           */
8             if defining_statement ∉ V then
9                 in_vars ⟵ add(tree)
```

In order to determine the output variables of a loop, all CDFG-edges pointing to

PHI-nodes contained in basic blocks outside the loop must be examined. Algorithm 5 shows the actual method to determine these so called exit variables.

Algorithm 5: Identify loop exit variables

Input: *CDFG(V, E) of loop body*
Output: *set of output varaibles* (*out_vars*)

```
1 foreach edge ∉ E do
2     exit_node ⟵ get_phi_node(edge)
3     foreach RHS ∈ phi_node do
4         tree ⟵ RHS
5         defining_statement ⟵ SSA_NAME_DEF_STATEMENT(tree)
          /* RHS operand is part of the loop body                    */
6         if defining_statement ∈ V then
7             out_vars ⟵ add(tree)
```

6.1.2 Accelerator Invocation Using Invasive GIMPLE Modifications

The *invasive* method for integrating the accelerator into the application was implemented in [130]. Figure 6.2 *(A)* represents an example of a basic block (BB) structure for an unmodified loop. The *Header* and *Latch* blocks form the outermost loop. The loop body (dashed box) contains the basic blocks of the internal control flow. Possible loop exits could occur in the loop body or directly in the header block. The example in Figure 6.2 *(A)* shows a loop with two exits from the loop body and a bypass from BB 2 to BB 4. For each loop exit, the GCC generates a PHI-node, even if it has one input edge only (cf. BB 3 in Figure 6.2 *(A)*). Such unary PHI-nodes change the SSA-name of a variable, which is typically necessary at the exit of loops. Finally, the control-flow is joined in BB 5, which represents the last block of the loop.

In order to prepare the software loop for removal (Figure 6.2 *(B)*), the control-flow is extended by several new basic blocks (grey blocks). Adding new basic blocks to the current control flow can be done by the GCC internal function `split_edge(...)`. This function splits a given edge and inserts an empty basic block between two existing basic blocks. The *new Header* and the *new Latch* block are required to form the new polling loop. Consequently, they must be prepared with the required GIMPLE statements. The new initialization block *(Init)*

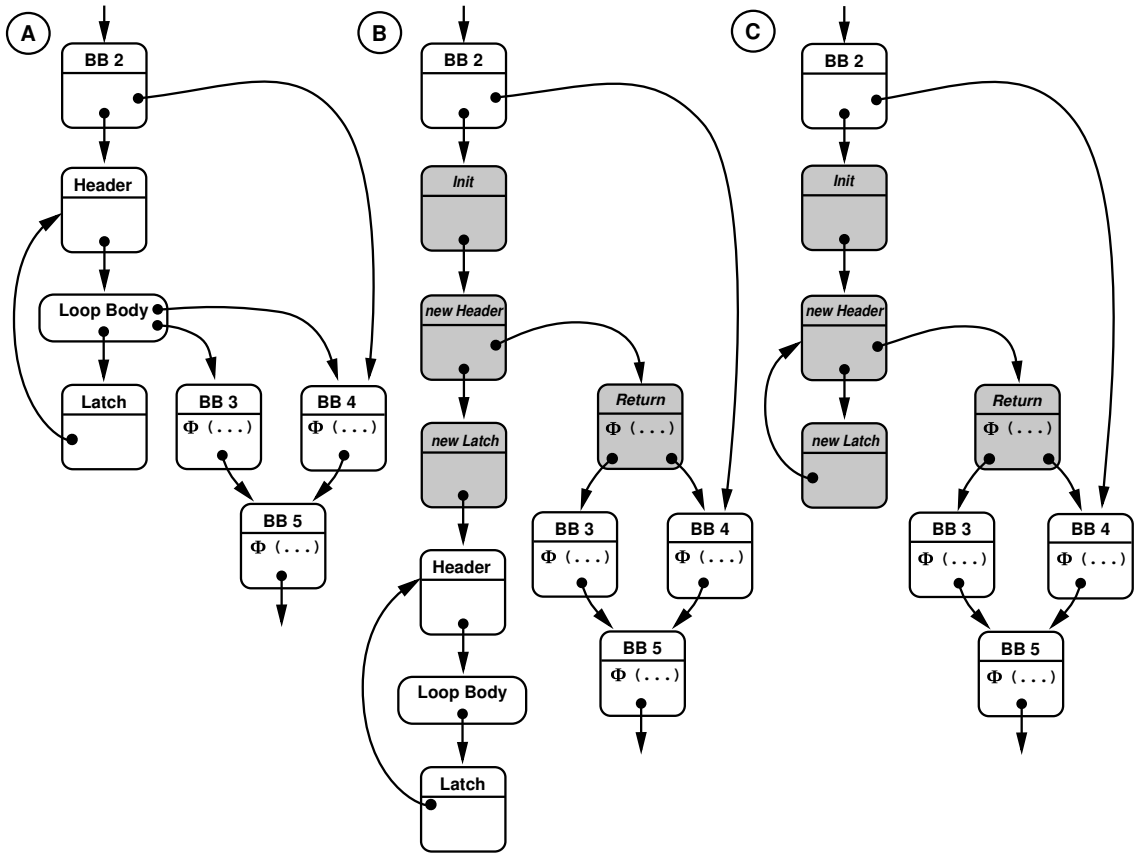

Figure 6.2: Remove loop and CFG modifications

is used to set the parameters and the start signal of the accelerator. In order to enable the return block *(Return)* to control the further data-flow it requires one output edge for each exit of the original loop.

After cutting all exit-edges of the original loop the remaining unreachable blocks are removed using a GCC internal function for elimination of dead paths. Finally, the new polling loop is generated by adding the back edge from the *new Latch* block to the *new Header* block (cf. Figure 6.2 *(C)*).

Data Transfer and Accelerator Control

The data transfer to the hardware is shown in the excerpt of the modified GIMPLE transcript (Listing 6.3) for the control flow from Figure 6.2. The loop bypass in BB 2 is generated by the if statement using the variable `con_1` in this example. The initialization of accelerator registers for the parameters `a_6` and `b_3` is

implemented in BB *Init* by GIMPLE assign statements. The target operand is the direct hardware address of the corresponding register. After all parameters are transferred, the accelerator is started by setting the control register at address 212992*d* (0x34000). Later, the host processor polls the status register 212994*d* (0x34002) until the hardware accelerator finishes its operation. Analogously to the initial parameter transfer, the results of the accelerator (`res_1, res_2`) are read in the *Return* block directly from the corresponding register addresses. The condition for the contained PHI-node is derived from the index of the exit basic block of the original software loop (cf. `bb_idx` in Listing 6.3), which is returned by the hardware accelerator. In order to assign the correct return value to the final GIMPLE variable `a_14`, the results of the loop must be handled by adapted PHI-nodes in BB 3 and 4.

The assembler code of a patched application for the SpartanMC soft-core processor is shown in Listing 6.4. Lines 6 to 13 show the initialization of the registers. The instructions of the polling loop that checks the control register (18-bit address 0x34002) for the ready bit are shown from Lines 16 to 19. Finally, the results are written to `res_1` or `res_2` from Lines 22 to 27. For the sake of clarity, BB 3,4, and 5 are not shown in Listing 6.3.

One of the big advantages of the *invasive* application patching is the direct generation of compact and low-latency machine code for accelerator calls. On the one hand, the resulting solution can be regarded as nearly optimal with respect to the given parameter set. On the other hand, the extensive modifications of the GIMPLE representation at this stage of the compilation process turns out to be very complex and error-prone[1]. Furthermore, the *invasive* modification process creates a strong dependency on the current GIMPLE representation. This may cause unnecessary problems when porting the plugin to a newer GCC version. Finally, the *invasive* modification aggravates enhancements of the polling loop (e.g. adding code to yield the current process) and the debugging of the modified code.

Due to these reasons, the *invasive* code patching was reimplemented with the goal of causing the least possible disturbance to the existing GIMPLE representation.

[1] Adding several new GIMPLE statements and removing an entire loop seems to be not intended at the end of loop-optimization passes.

```
  ...
<bb 2>:
  ...
  if (con_1) goto <bb 4>
  else goto <bb I>
<bb I>
  *212996 = a_6;
  *212998 = b_3;
  *212992 = 1;
<bb H>:
  if (*212994B != 1) goto <bb L>;
  else goto <bb R>;
<bb L>:
  goto <bb H>;
<bb R>:
  res_1 = *213000;
  res_2 = *213002;
  if (bb_idx != 3) goto <bb 4>;
  else goto <bb 3>;
<bb 3>:
  a_12 = PHI <res_1>;
  goto <bb 5>;
<bb 4>:
  a_13 = PHI <a_6, res_2>;
<bb 5>:
  a_14 = PHI <a12, a_14>;
  ...
```

Listing 6.3: GIMPLE transcript of an accelerator call

```
1  ; loop bypass
2       BNEZ   r4, L1
3       OR     r0, r0
4
5  ; transfer parameters
6       LHI    r12, 416
7       ORI    r12, 4
8       S18    0(r12), r5
9       S18    2(r12), r4
10      LHI    r4, 416
11      ORI    r4, 0
12      MOVI   r5,1
13      S18    0(r4),r5
14
15 ; polling loop
16 L2: L18    r12, 2(r4)
17      SNEI   r12, 1
18      BNEZC L2
19      OR     r0, r0
20
21 ; transfer results
22      LHI    r4,416
23      ORI    r4,8
24      L18    r5,0(r4)
25 L1: LHI    r4,416
26      ORI    r4,10
27      L18    r6,0(r4)
```

Listing 6.4: SpartanMC assembler code with polling loop

6.1.3 Accelerator Invocation Using a Function Call

The second implementation of the GIMPLE code patching was presented in [145]. It reuses several methods of the former implementation. Nevertheless, the second approach focuses on minimizing modifications of the existing GIMPLE structure in order to maintain a good portability of the plugin and increase the flexibility of the inserted code. The basic idea was to decouple the GIMPLE patching from the communication protocol with the accelerator. Therefore, the GIMPLE representation is only patched with a function call that implements the actual accelerator invocation. The implementation of the function itself can be

provided as C code. Hence, this method minimizes the required GIMPLE modifications and, furthermore, allows a flexible C-based implementation of the communication protocol. The C code is generated into an extra compilation unit that is translated to a separated object. Both are linked together after the compilation process.

Adding a function call statement on GIMPLE-level requires an appropriate function definition in GIMPLE. This includes a function name and a list of parameter types, including the return value. The name of the new function is composed of the wrapping functions name containing the loop candidate, the corresponding compilation unit, and a consecutive number. The functions parameter list is used for inputs as well as for outputs; therefore, all passed parameters are treated as pointer types.

After creating the function definition, the invocation of the function is added as a `GIMPLE_CALL` statement in a novel accelerator basic block (*BB Acc* in Figure 6.5 *(B)*). The `GIMPLE_CALL` is defined by the function definition and requires a list of SSA trees that represent the actual parameters set for the function invocation. The list of trees is derived from the `in_var` and `out_var` vectors of the loop (cf. Section 6.1.1).

Later, the newly created basic block is inserted preceding the header block of the loop. The structure of the patched CFG is shown in Figure 6.5 *(B)*, which originates from the sample CFG structure in Figure 6.5 *(A)*. In contrast to the former invasive approach, the software loop and the corresponding exit edges are preserved and remain functional. In order to retain the correct control-flow after finishing the accelerator function, the inserted basic block is supplemented by branches that provide exit edges to bypass the software loop. In addition, each PHI-node within the exit blocks has to be extended by a new variable referring to the respective output variable of the accelerator function. If the accelerator finishes its execution successfully, the control flow continues with the corresponding exit node behind the loop. Analogous to the invasive approach, the condition used to select the correct exit node is derived from the returned index of the exit block (cf. `bb_idx` in Figure 6.5 *(B)*) . The index is read from an output register of the hardware accelerator. Its value is always positive after the hardware accelerator was executed. Consequently, negative return values can be used to indicate an exception that results in a fall-through to software execution.

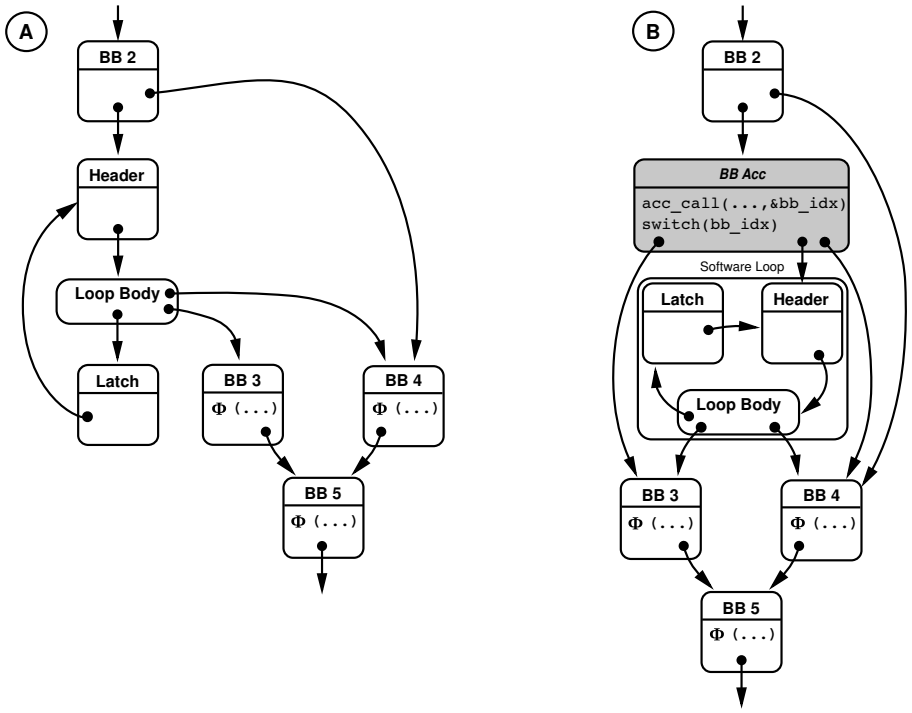

Figure 6.5: Add function call to CFG

The GIMPLE transcript next to the corresponding SpartanMC assembler output of a sample accelerator function call is shown in Listings 6.6 and 6.7. The used hypothetical input/output data of the accelerator is congruent to the example from Listing 6.3. The parameters of the function call (`jalrs` Line 8) are transferred via Registers 12, 13, 14, and 15, which are intended for parameter transfers to subroutines (cf. SpartanMC architecture Section 3.2.1) and one value on the stack (Lines 2 to 5 and Line 12). Note that `r13` is written within the delay slot after the function call. As the SpartanMC pipeline requires one clock cycle to calculate the jump target, this empty slot can be filled by the compiler with a suitable instruction. After the return from the accelerator wrapper function (address `acc_fun`), the results are read from the stack (Lines 16 to 18) by using `r0` as stack pointer.

The temporaries `tmp_9` and `tmp_8` in the GIMPLE listing are generated return variables that provide the return value from hardware. They correspond to the original loop exit variables `res.1` and `res.2` that are still used by the software implementation of the loop. The hardware return value and the software return value are merged into the extended PHI-nodes of BB 3 and 4. Note that the handling of *start* and *ready*, as well as the data transfer to registers and the polling

```
<bb 2>:
  ...
<bb Acc>
  acc_call(&res_1, &res_2,
           &a_6, &b3, &bb_idx);
  if (bb_idx == 3) goto <bb 3>
  if (bb_idx == 4) goto <bb 4>
  else goto <bb H>
<bb H>:
  /* software loop */
  ...
<bb 3>:
  a_12 = PHI <res_1, tmp_8>;
  goto <bb 5>;
<bb 4>:
  a_13 = PHI <a_6, res_2, tmp_9>;
<bb 5>:
  a_14 = PHI <a12, a_14>;
```

Listing 6.6: GIMPLE transcript of an accelerator function call

```
1  ; transfer parameter one
2      S18   2(r0),r5
3      MOV   r12,r6
4      MOV   r13,r7
5      MOV   r14,r8
6
7  ; accelerator call
8      LHI   r10, %hi(acc_fun)
9      ORI   r10, %lo(acc_fun)
10     JALRS r10  ;function call
11 ; transfer parameter two
12     MOV   r15,r9
13
14 ; transfer results
15     ...
16     L18   r5,2(r0)
17     L18   r6,4(r0)
18     L18   r7,6(r0)
```

Listing 6.7: SpartanMC assembler code for accelerator function call

loop, is implemented in C within the accelerator function. It is not shown in Listings 6.3 and 6.6, which focus only on modifications of the GIMPLE structure.

Software Delegation

The delegation to software execution is used on critical errors and resource conflicts or to prevent the accelerator execution by user command. The decision to execute a loop in software has to be taken *before* the actual execution of the accelerator starts. Though it would be technically possible to abort the accelerator during execution of the accelerator wrapper function, it is not recommended in the current implementation as it could lead to an inconsistent memory state. Due to the direct data access of the accelerator hardware, the data memory may already have been modified at the point of interruption, which may cause the following software run to produce wrong results.

Nevertheless, the possibility to fall back to software execution is essential when targeting platforms running a full-featured operating system (OS), such as Linux on Xilinx Zynq. As these systems typically run multiple applications at the same

time, the applications must share the underlying hardware resources. For this reason, management of hardware accelerators and the possibility of a seamless fall back to software execution are vital for providing the required flexibility for such systems.

Moreover, the option to evaluate various error codes from the hardware accelerator introduces a light-weight debugging interface. It can be used to signal warnings or errors when unexpected input parameters occur (cf. Section 6.3).

6.2 Accelerator Function

The purpose of the accelerator wrapper function (hereinafter referred to as *accelerator function*) is to perform the communication with the hardware accelerator. It encapsulates the transfer of initial parameters as well as the management of the accelerator FSM (e.g. *start* and *ready* signal). Furthermore, it enables the implementation of host-triggered data-transfer schemes that will be further used for complex memory architectures.

The actual implementation of the generated accelerator function depends on the software environment of the target architecture. Currently, the used architecture is distinguished by the *target* option of the *configure* script of the plugin. If the target is set to `target=spartanmc`, the plugin is built for the SpartanMC softcore. On the one hand, this implies the generation of a SpartanMC peripheral interface; on the other hand, it presumes a bare-metal software architecture. The second target (`target=arm`) prepares the plugin for generation of an AXI interface that is used on the Xilinx Zynq platform. In contrast to the first target, it presumes a full-featured Linux OS.

From the accelerator's viewpoint, the major difference between a bare-metal and an OS-based architecture is the management of memory addresses. For ARM platforms, this problem is tackled by the accelerator function. The following sections introduce possible hardware solutions for virtual address management.

The classical embedded system is tailored for *one* task that is covered in exactly *one* application. Usually, such applications run on *bare-metal*; in other words, they run on the target machine without a further abstraction layer and with direct access to all system resources. The communication with attached peripherals

and memory is performed by physical addresses. For accelerators directly extracted from the C code, this implies that both parts – software and hardware – run in the same memory domain and can use the same addresses. On these grounds, consecutive access patterns in software imply a corresponding, consecutive, physical memory allocation. Such memory accesses will always succeed as long as the calculated address refers to a valid physical address. Although this type of memory management is handy for a small application it is hard to control for multiple applications running in separate processes.

As a consequence, an arbitrary application running on an OS works on a virtual address space. However, for data exchange with peripherals or external memories, the physical address of the device is still required. Therefore, the translation to a physical address is carried out on a special hardware unit called the memory management unit (MMU). In case of automatically generated accelerators, this implies that all memory addresses that could be determined at compile time must be considered to be virtual and, therefore, become invalid for direct utilization in hardware. The actual physical address is determined by the MMU at the application's runtime. Due to this fact, a direct physical access to the data memory of the host processor is not possible when using generated hardware on an OS-based software architecture. This issue is solved on recent desktop platforms that provide a special input/output MMU (IOMMU) to translate addresses for DMA operations. Unfortunately, such IOMMUs are not available for current embedded platforms. As a consequence, the address translation is delegated to the software layer of the system or is circumvented by other mechanisms.

6.2.1 Address Translation in Related HLS Approaches

The generation of accelerators for applications running on virtual memory is a vital piece of work in order to make automatic HLS usable for a large user community. This section gives a brief overview of possible solutions for this issue. However, there are only a few examples of other approaches that cope with this particular problem. This might be because most of the HLS approaches discussed in Section 2.3 are based on a semi-automatic HW/SW co-design that requires manual interaction to integrate the accelerator in the software application. In that case, the data that is required within the corresponding hardware can manually be al-

located to physically contiguous memory or copied to suitable memory sections beforehand.

One sample implementation of the latter methodology is used in Nymble [34] and Comrade [39]. Both HLS tools can rely on the Accelerator-Integrating Shared Layout for Executables (AISLE) [104]. This is a special memory layout that copes with virtual memory management in order to provide an accelerator-friendly software structure. The memory layout of AISLE defines each data section, `.bss`, `.data`, stack, and heap in a single, contiguous region of physical memory. Additionally, this region is placed in the DMA buffer which is accessible from the host processor and the accelerator as well. With the given layout, the accelerator requires only the offset between physical and virtual memory as a parameter to access valid physical addresses. Even though this solution is effective and easy to implement, it has poor scalability and thwarts the intended purpose of virtual address management due to the following reasons: First, the DMA buffers used are suitable for small amounts of data only. Second, the approach requires pinned[2] memory pages, which re-introduces the problem of memory fragmentation that was intended to be solved with virtual address management.

Instead of circumventing the problem, as shown above, some approaches directly cope with the challenge of a dynamic address translation. The most obvious idea to solve this issue is the implementation of an embedded IOMMU. The hardware design of a full-featured IOMMU was presented in [101]. The proposed implementation was tailored for a Xilinx FPGA architecture and consumes a reasonable amount of resources (11k slices $\approx 75\%$ of an Artix-7 FPGA). The implementation supports full virtualization of accelerators and fast context switches. A similar approach using a shared translation lookaside buffer (TLB)[3] between accelerator and host processor was presented in [103]. The implemented architecture, called Processor-Hardware Accelerator Shared Environment with Virtual addressing (PHASE/V), was intended as improvement to the AISLE concept. Unfortunately, both approaches are not trivially adaptable for the Xilinx Zynq architecture, as it requires access to a shared TLB from both parts of the system. This can be achieved by instrumenting the kernel functions for writing the MMU/IOMMU. However, the required engineering work to solve this issue could be tremendous

[2] Once allocated, memory pages cannot be moved or reallocated.

[3] The TLB is a hardware cache which provides the mapping from virtual addresses to physical addresses. It can be regarded as the key component of an MMU.

and was not carried out in the context of this thesis. Moreover, such an implementation could turn out to be a bottleneck for the ARM subsystem due to the unbalanced clock frequencies between both parts of the system.

Since the pure hardware implementation of an IOMMU for the Xilinx Zynq platform is not easy to achieve, it seems appropriate to use a software backed approach. A sample implementation of this idea was presented in [117]. The authors propose a soft-core address buffer called Remapping Address Block (RAB) that is used as an additional input/output TLB. In order to minimize the hardware effort, the handling of TLB misses or the prefetching of upcoming transactions is carried out in a kernel-level driver on the host processor. For this reason, a flawless operation of the RAB is ensured by providing the complete access pattern of the accelerator at initialization time. In addition, the host processor requires a runtime environment that keeps the RAB consistent with the current virtual memory layout. The actual memory access is carried out with a DMA soft-core implemented on the FPGA fabric.

Another approach was proposed in [144]. It follows the same philosophy of a software-backed address translation, but instead of adding an additional TLB, the presented idea introduces a cache architecture. The required management operation of the cache is implemented in software on the host processor while the data cache itself is implemented on the FPGA fabric. Due to the large difference in the clock frequencies between ARM processor and FPGA fabric it is possible to use the ARM to transfer data and handle cache misses in software. One design goal of the cache architecture presented in [144] was the usage of burst data transfers for nearly all operations. The reason for that is twofold: On the one hand, it avoids execution of the whole software stack for each single value. On the other hand, it enables the usage of an integrated DMA controller of the Zynq. After initialization, it allows transferring one value per clock cycle.

For an effective implementation of the cache architecture, one needs to guess the virtual addresses of upcoming transactions. This is supported by the memory access analysis that was described in Section 5.6. The actual implementation of the address translation uses the idea of a software-backed function, as presented in [144]. But, instead of a hardware cache, the current implementation uses a rather simple data prefetching mechanism that is tailored for FIFO memories. Nonetheless, this approach uses burst transfers to achieve a reasonable

data-transfer rate. Instead of a cache architecture, the FIFO interface requires a precise prediction of the upcoming data access. The structure of the used hardware interface is discussed further in Chapter 8.

A discussion about the usage of virtual memory from the software point of view can be found in Appendix D.

6.2.2 Implementation of the Accelerator Function

The type of the generated accelerator function is determined by the `target` parameter during the plugin build process. It defines the host machine and the hardware interface for which the compiler plugin will be built. Additionally, a runtime parameter (`-fplugin-arg-hw_generation-fifos`) is used to allow the generation of FIFO interfaces for the hardware accelerator. A design objective of the generated function code was to introduce the lowest possible overhead to the accelerator execution. Therefore, the implementation does not contain any function calls. Besides the generated accelerator function, PIRANHA provides a static library that includes some convenience functions to handle data transfers, and to activate or deactivate accelerators at runtime. If this library is included in the user's application, the functions can be used to control the behavior of the accelerators.

However, such a FIFO interface is not useful when building accelerators for the SpartanMC soft-core. The use of a FIFO implies that the filling of its buffer is carried out by the host processor or an additional hardware unit, for instance a DMA controller. The benefit would be a memory access within one clock cycle for the accelerator. For the current implementation of the SpartanMC architecture, the latency for a direct master-mode memory access is already just one clock cycle. Consequently, it would lead to slowdowns if the access is back-delegated to the host processor. Nonetheless, the FIFO interface could enable the prefetching of data in parallel to computational parts during accelerator execution. This may improve the overall performance even on SpartanMC based systems. Unfortunately, the current FIFO implementation still requires a runtime alias analysis in software that would introduce a massive loss of performance for a SpartanMC accelerator as both system parts nearly run at the same clock frequency. An efficient implementation of prefetching for the SpartanMC should be part of future

research for this platform.

The current implementation of the plugin supports three possible implementation schemes of the accelerator function.

Scheme I: Bare Metal Implementation

This scheme is triggered if the target architecture is set to SpartanMC. The runtime parameter that enables the use of FIFOs is considered to be meaningless for this target.

The resulting control flow of the generated accelerator function is shown in Figure 6.8 *(Scheme I)*. First, the control flags of the accelerator library[4] are checked. If the accelerator is enabled, the following code triggers the initial register transfer and starts a simple polling loop that constantly checks the accelerator status register until the "`ready`" flag is set. As the SpartanMC architecture allows direct access to its data memory, no further software assistance is required to run the accelerator. All memory modifications are directly executed from the accelerator hardware. After finishing the hardware execution, the output registers of the accelerator are read and the index of the following basic block is returned in order to continue the software execution.

Scheme II: Random Access with Address Translation

The second access scheme, as shown in Figure 6.8 *(Scheme II)*, is used if the target architecture is set to ARM but the application code requires random access operations. The underlying concept of this implementation is the back-delegation of the address translation to the host processor. Thus, the control flow of the polling loop is extended by an additional loop that evaluates a memory request register. This register provides a valid virtual address to signal a read or write request. As the data transfer is carried out by software on the host processor, the address translation is performed implicitly. The direction of the transfer is indicated by an additional status bit. A major drawback of this implementation is its latency. Each request triggers the transfer of a single value on the AXI interface. The required arbitration of the AXI bus introduces an overhead of more

[4]The accelerator library `libacc` is a predefined C library that is used to control the accelerator interface. The library can be further used to bypass the accelerator from the user's application.

than 10 accelerator clock cycles. This easily sums up to a latency of several thousand clock cycles for the whole hardware loop in case of extensive memory accesses.

Scheme I ***Scheme II***

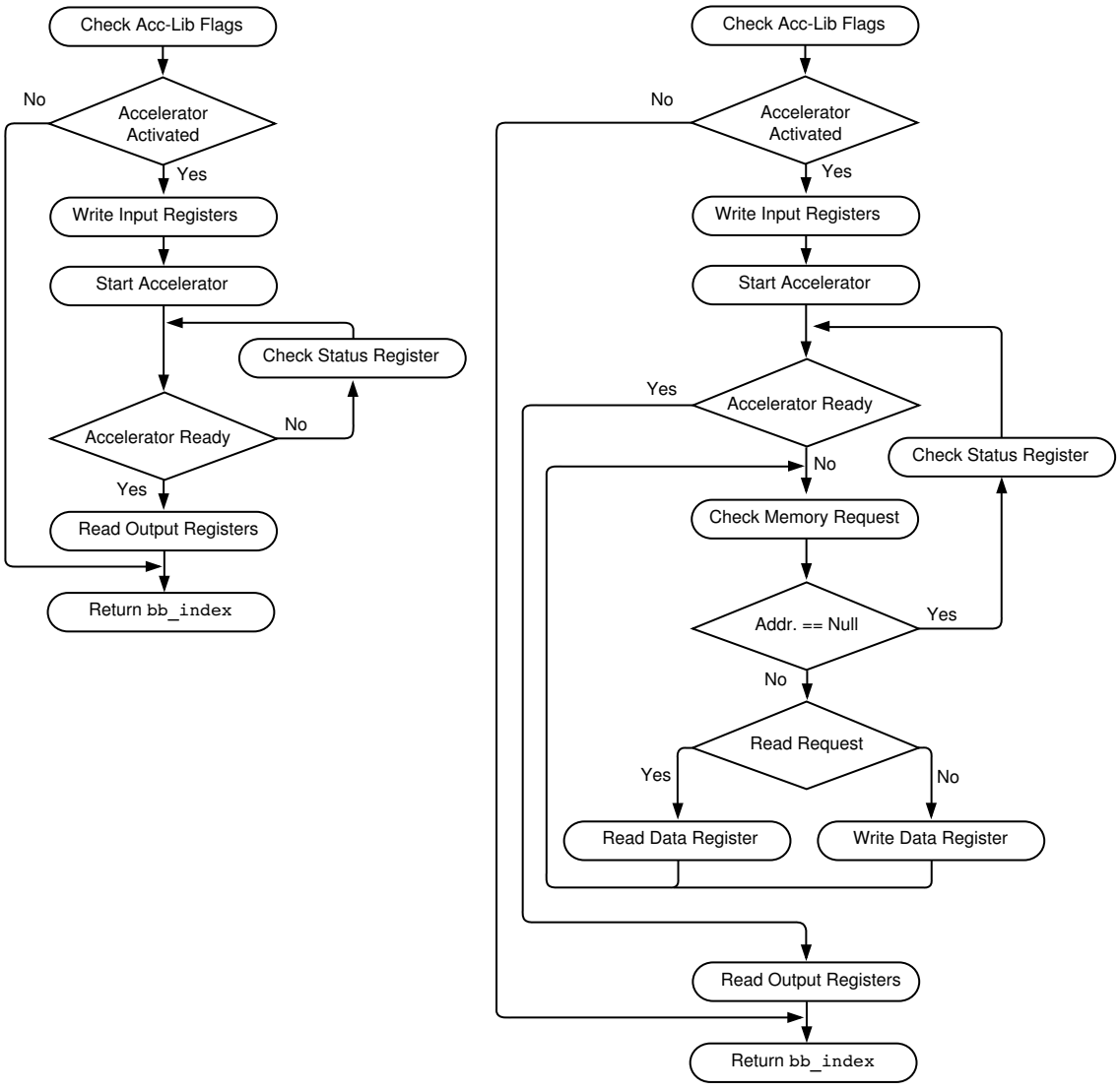

Figure 6.8: Control-flow of the accelerator function for Scheme I/II

Scheme III: Prefetching with Address Translation

Due to the large latency of random memory accesses, the third access scheme is preferred for the ARM architecture. This scheme is automatically activated if the

plugin is parametrized to use hardware FIFOs. The FIFO interface was designed to allow a streaming-like data transfer. Similar to Scheme II, it requires the back-delegation of memory accesses to the software layer of the host processor. The simplified control flow of the generated access scheme is shown in Figure 6.9. In contrast to Scheme II, it uses burst operations to transfer data during accelerator execution. Note that the use of burst transfers implies the need for a prefetching mechanism for the required data, which in turn relies on the memory access analysis introduced in Section 5.6.

One objective of the currently generated access function was the ability to run asynchronously to the hardware execution. Although the used FIFOs provide synchronization information, such as a *full-* and an *empty-*flag and the *filling-level*, the access to these registers is not used in order to avoid the additional latency. The current implementation of a FIFO write transfer blocks if the FIFO turns out to be full while the read access blocks if the FIFO is empty. In order to ensure non-blocking data transfer to the FIFO memories, the memory access analysis must provide the length of the used data array, the direction and stride of the index as well as the size of a single access. In the best case, this information is provided by the memory access analysis during the compilation process[5]; otherwise, a compiler warning is generated and the access is considered unsuitable for prefetching.

If the data specifying the memory accesses is completely available, it can be used to generate a copy loop that mimics the memory access of the original software loop. In this way, the required data is copied in predefined chunks to the DMA buffer and, later, to the FIFO buffer. The actual data transfer is carried out by the DMA controller of the Zynq.

As the data transfer runs asynchronously to the hardware, it is possible that an empty or full FIFO would block the current transmission. Such temporary stalls are not severe as long as the whole accelerator FSM does not block due to a dead lock situation. In order to avoid such dead locks, the data transfer is carried out in the chronological order of the original software loop. Therefore, the domination level of the initial GIMPLE statement is used to generate the correct sequence of access operations in the accelerator function. This mechanism to prevent dead

[5]The base address is not necessarily required for using FIFOs. The actual base address is typically transferred as a parameter on accelerator invocation.

locks is not feasible for conditionally executed memory accesses. For that reason, the control-flow of the questionable loop must be evaluated during compile time. If the loop contains an unsuitable CFG the current implementation generates a warning. A extension of this mechanism could modify the generated function to use synchronous FIFO transfers instead.

As shown in Figure 6.9, the first task of the generated function comprises the

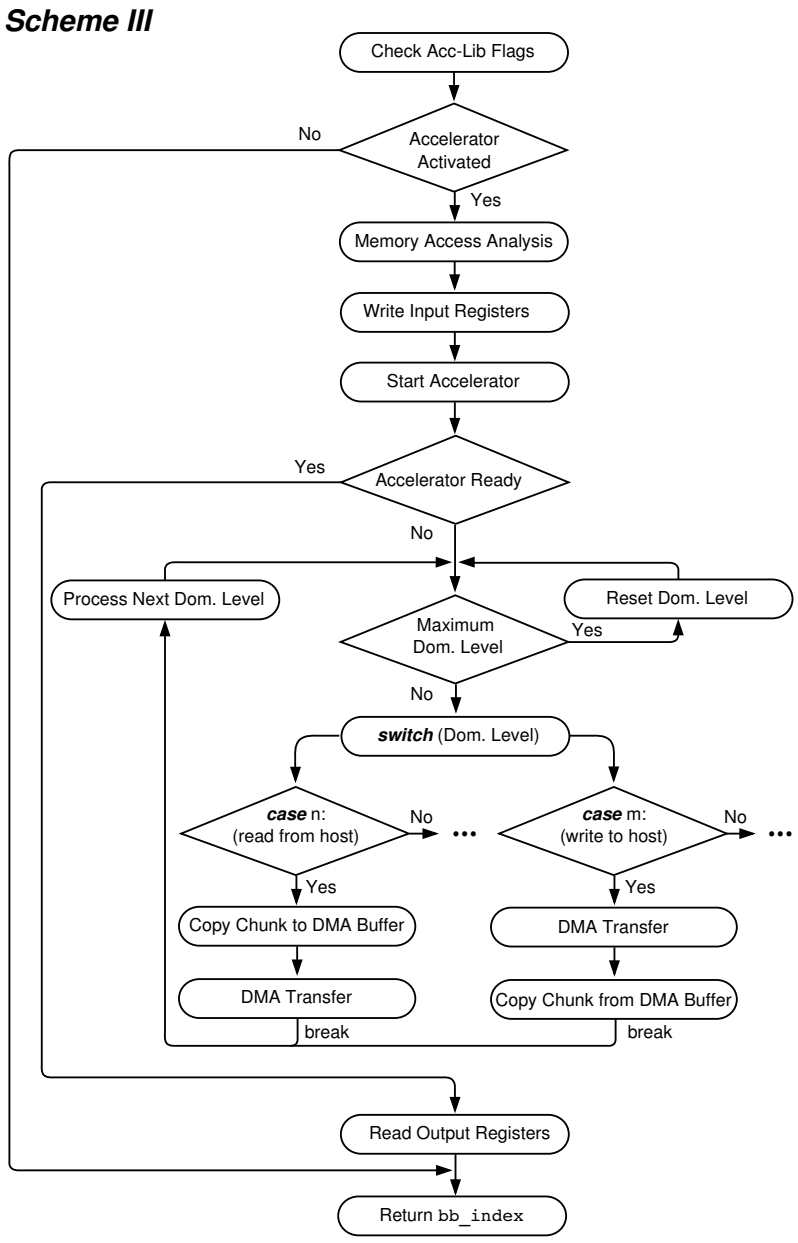

Figure 6.9: Control-flow of the accelerator function for Scheme III

runtime evaluation of memory accesses. The alias analysis usually requires the actual base address of concurrent memory accesses in order to identify possible overlapping. Typically, such base addresses are part of the parameter set of the hardware accelerator; thus, they are only available at runtime. The runtime alias analysis will be described in detail in the following Section 6.3. If an aliasing of the current write accesses is detected, the accelerator call is aborted and the software execution is started.

After the memory access analysis and the parameter transfer, the data prefetching loop is carried out. This loop iterates over all domination levels (*Dom. Level*) of the original software loop in chronological order. The corresponding loop body contains a *case*-block for each domination level that carries out the actual data transfer for read or write operations. Therefore, the data access of the original loop is replicated according to the determined access patterns derived from the memory analysis. Furthermore, the burst length of the transfer is parametrizable and must be taken into account during the data transfer. Listing 6.10 shows an example of a nested C loop that contains three memory references, two read operations, one write operation, and an absolute iteration count of 100. The arithmetic part between the memory accesses has been omitted for sake of clarity.

```
/* assuming a buffer containing *A, *B, *C */
int x, y, z;
for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 10; i++) {
        x = buffer->A[i][j];
        y = buffer->B[i][j];
        /* ... some arithmetic defining z */
        buffer->C[i][j] = z;
    }
}
```

Listing 6.10: Nested C loop with three memory references

The resulting accelerator function for this example contains three *case* blocks (two for memory-read and one for memory-write). Assuming a generated chunk size of 64 values, the resulting qualitative execution trace of the communication between host processor and accelerator is shown in Figure 6.11. The execution trace clearly shows two copy operations that lead to hardware stalls during the

read operations in the accelerator FSM. After the initial data transfer, the remaining memory accesses during accelerator execution can rely on the prefetched data. Despite the delay introduced by the hardware stalls, the overall burst transfer is still faster than the transfer of single values.

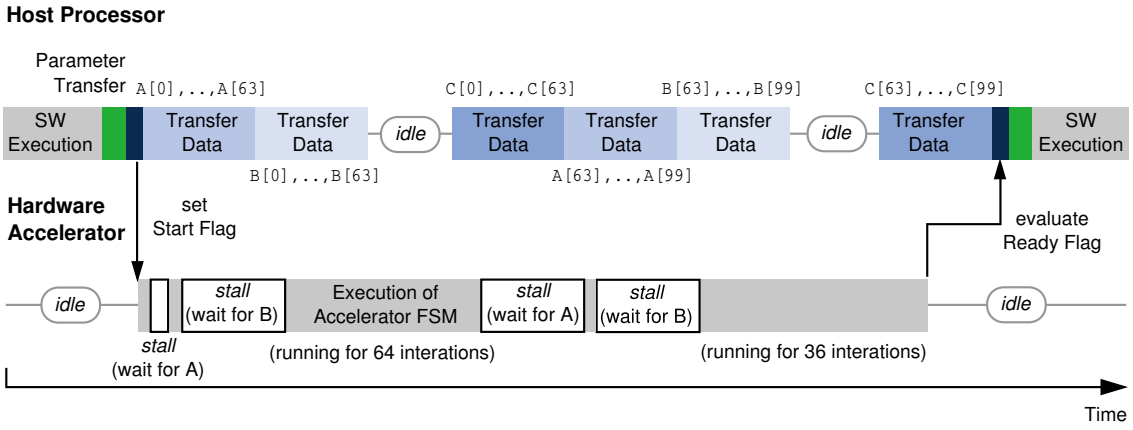

Figure 6.11: Qualitative execution trace for Listing 6.10

Similar to the former accelerator function schemes, the data transfer ends when the accelerator sets the ready flag in its control register. Finally, the resulting parameters are transferred and the exit block is returned to the software part of the application.

6.3 Runtime Alias Analysis

The typical results of the alias analysis during program compilation comprise the number of consecutive memory accesses (number of loop iterations), the stride, the word-size, the constant offset to the base address, and the mode of operation (either memory-read or memory-write). But even assuming all these values are given, as long as nothing is known about the relative position of such memory references to each other, it is not possible to determine whether they will overlap or not. Actually, the relative position is first defined during runtime by the base address of the memory reference. The vast majority of software loops derive this value from one of their input parameters. Nevertheless, in some cases it could be possible to discover the base address even during static program analysis, e.g. if the variable points to a constant memory address. Moreover, it is possible

to find indirect evidence that GIMPLE variables point to a disjointed memory location for instance, if the variable originates from a `malloc` operation within the current function. However, in most cases it is not possible to resolve the base address completely at compile time. Thus, the accelerator function must perform a runtime alias analysis in order to find aliasing between memory references.

The analysis data that is gathered during the compilation process is provided in a generated file for the runtime alias analysis. To ensure the uniqueness of the generated C-file for each accelerator, its name is composed by the name of the corresponding C file, the loop number, the name of the function containing the loop, and the static string "`memref_info`". It contains the structure `memref_context`, which, in turn, contains an array of `memref_symbol` structures. Each `memref_symbol` refers to a memory reference in the analyzed code and is initialized with the corresponding analysis data. In the later compilation process, the newly created compilation unit is linked with the patched accelerator binary. After providing the required data for the runtime alias analysis, the actual analysis is done in the function `ma_impl_analyze(...)` which is called from `memref_alias_analyze(...)` in the accelerator function. Note that all these functions are optimized and defined as *static inline* to reduce their impact on the runtime of the accelerator call.

The runtime analysis itself uses a light-weight version of the alias matrix that was introduced in Section 5.6.3. For all occurrences of `MEMREF_MAY_ALIAS` entries in this matrix, the corresponding access pairs must be evaluated at runtime. Therefore, an evaluation function for each entry is generated that solves the alias problem for the given pair of accesses. According to Section 5.6.1, the alias problem is defined by basic alias sets derived from the definitions of LibISL. These sets are transformed to Boolean conditions in C code, which could be easily checked during runtime. However, the crux of LibISL-generated alias sets is that the derived conditions sometimes lead to very complex C code, which would introduce an untenable overhead to the accelerator function. Thus, the additional effort of a heuristic optimization of the generated runtime analysis code was implemented in favor of a low-latency accelerator call.

6.3.1 Alias Set Optimization and ISL Queries

The plugin generation process distinguishes between two kinds of alias sets: sets that could be evaluated by simple Boolean conditions and sets that require an iterative test. The optimized test code for alias sets is generated by the `isl_set_compute_divs(...)` function. All resulting sets refer to an offset $d \in \mathbb{Z}$ that is derived from the offset between two base addresses. Each alias set covers another interval of $\mathbb{Z}$. If d is within the interval, it can be evaluated by the corresponding conditions. If the tests for a given d returns *true*, aliasing occurs between the base addresses; if it returns *false*, the memory references will never overlap.

In order to evaluate simple conditions first, all sets that only include d and a constant interval are translated into conditional statements. These simple membership tests are used to evaluate whether d is in the required range for a specific set. As long as there are no other conditions except these, such tests are sufficient to achieve clear results. For instance, the code describing the four-dimensional matrix-vector product in Listing 5.11 results in a simple membership test as shown in Listing 6.12.

```
static inline bool
ma_impl_aliases_0_0_vec_out_25_with_ivtem_33_16( ptrdiff_t offset) {
    if (offset >= -51LL && offset <= 15LL) {
        /* above condition sufficient */
        return true;
    }
    return false;
}
```

Listing 6.12: Generated and optimized alias test between output vector and matrix of the matrix-vector-product example 5.11 [135]

Unfortunately, not all alias tests can be reduced to a membership test with a constant interval. The more complex conditions are evaluated with the help of a direct LibISL call during application runtime. Therefore, the generated string representation of the basic alias sets is parsed and transformed into `isl_basic_sets` in order to be processable as a LibISL query. The generation of `isl_basic_sets` is carried out once at the startup of the application on the host machine. This is

accomplished by using a function with the `constructor` GCC function attribute. This attribute should not be confused with the constructor known from object-oriented programming languages. In fact, it is a special GCC attribute making a function execute right before the main function of a program is called. It is considered a convenient way to patch initialization into an application without touching its source code.

After generating an `isl_basic_set` for all complex alias sets, it is possible to evaluate such sets by calling a target version of LibISL at each accelerator call. However, sample evaluations (cf. Section 9.2.1) reveal that the use of LibISL at runtime will introduce a massive decline in performance, which should be avoided by all available means. The first and most obvious way to avoid the performance hit of the LibISL is to prevent its execution at runtime. Accordingly, all complex alias sets would return *true* and would be considered to overlap. Although not very elegant, this method is still effective for the majority of memory references. An alternative approach was implemented in [135]. It is based on a hash table that stores previous results of complex alias tests. Assuming that most accelerator calls are carried out on the same or at least a very similar memory layout. Therefore, each access pair generates its own hash table. The offset between the base addresses is reinterpreted as unsigned value and taken modulo 1024, which is further used as a key for the hash table. The entry contains a valid bit, an aliases bit, and the original offset in order to detect collisions. If a collision occurs, the entry is treated as invalid, which triggers the regular LibISL evaluation. As non-trivial alias sets are usually generated for accesses using arrays of *structs* while accessing different members, the keying is optimized for small offsets ($|d| < 512$). Such offsets are expected for *struct* accesses with $|d| < \texttt{sizeof}(T)$ for a structure T.

6.4 Generated Files

The largest set of input files and generated output files is involved for a plugin configuration using FIFOs in combination with the ARM target architecture. Figure 6.13 provides a comprehensive overview on all input and output files of the GCC plugin. It shows the compilation process for a single compilation unit that may consist of several C and header files containing a part of the user's application.

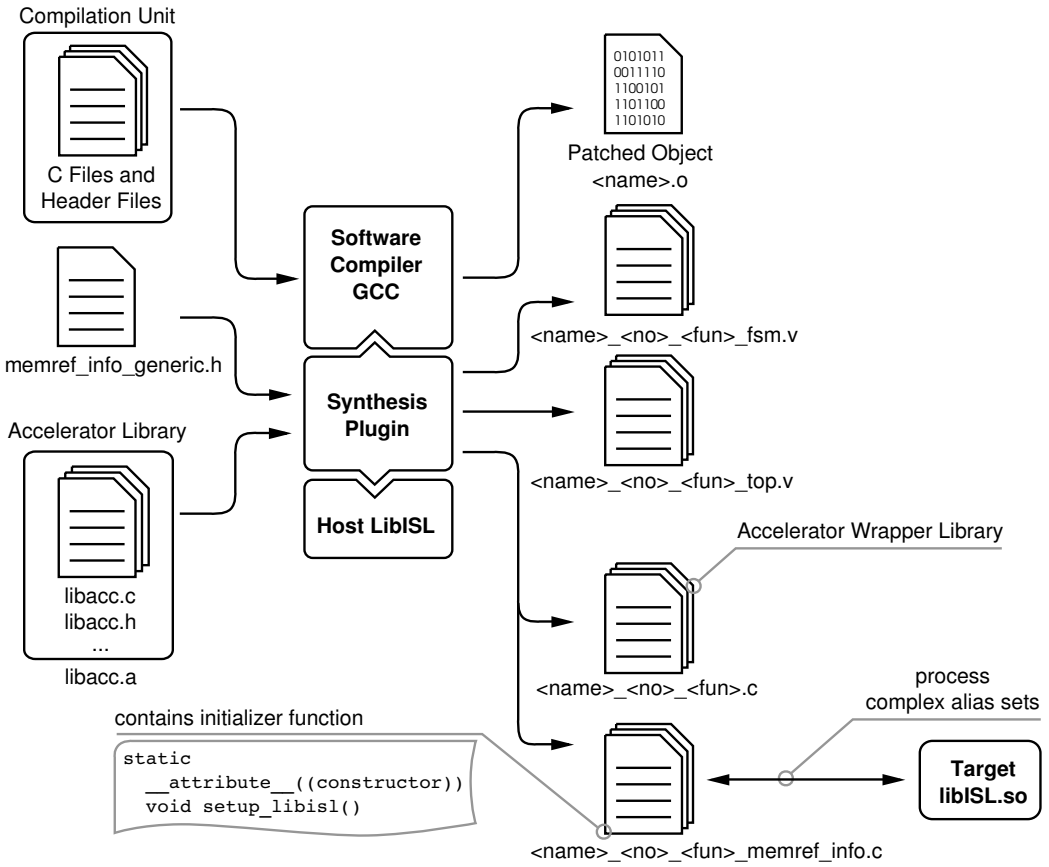

Figure 6.13: Input and output files of the GCC plugin

In order to allow seamless integration of accelerators in the resulting software/ hardware co-design, the plugin requires additional immutable input files. First, is the *libacc.c*, which represents the interface for several convenient functionalities, e.g. DMA write and read. These functions are later required for the generated accelerator wrappers. The *libacc.c* is compiled with architecture-specific C files to the accelerator library *libacc.a*.
Second, the header file `memref_info_generic.h` is used to define the structures for the memory analysis.

The output files comprise the patched object of the compilation unit, the HDL files of the accelerator(s), and the C files containing the generated accelerator wrapper functions. The generated C files and Verilog files (`*.v`) must be unique for each generated accelerator. Therefore, the file names as well as the names of generated functions and variables are derived from the name of the original

compilation unit, the number of the loop (from GIMPLE-IR), and the name of the corresponding function.

The generated Verilog files contain the FSM of the accelerator (`*_fsm.v`) and the interface to the host architecture (`*_top.v`). In general, the integration of accelerator hardware requires some additional adjustments on the hardware configuration files of the SoC, e.g. the project XML file of jConfig. These modifications are independent from the compilation tool-flow and, therefore, not shown in Figure 6.13.

The generated C files `<name>_<no>_<fun>.c` contain the accelerator function that implements the polling loop and the data transfer for memory references, as discussed in the previous sections. The second group of C files (`*_memref_info.c`) contains the initialization of the `isl_basic_sets` by using the `constructor` attribute. They further contain the alias checks and the data for the alias sets, created by the host version of LibISL during program compilation. Furthermore, the target version of LibISL can be used to carry out the alias analysis for complex basic sets at program runtime.

The generated C files are treated as additional compilation units that will be translated afterwards. The resulting object files are further combined within the accelerator wrapper library `libwrapper.a` (not shown in Figure 6.13), which is finally linked with the patched application binary.

6.5 OS Integration of Accelerators

Besides the address translation and the handling of memory access, the integration of custom hardware into modern operating systems also requires support for methods to manage parallel processes that access critical resources. Such mechanisms are necessary, for instance, in the case of two processes sharing one accelerator. For this reason, the required helper tools and a prototypic infrastructure to run accelerators on a Linux system have been developed in this thesis.

One of the first tasks was implementing a tool-flow that combines the FPGA configuration data with the application executable. Therefore, the Executable and Linking Format (ELF) was extended by an additional section. The new format

is called ZwoELF. The additional section includes the FPGA configuration and the device tree entries of the accelerators. The extended ELF file is generated by a newly developed tool that is also called *zwoelf*. In order to load the new section to the FPGA fabric, the kernel ELF loader was extended by an additional binary format handler that processes the ZwoELF section. The device tree entry within ZwoELF is integrated as device tree blob (DTB), which can be generated from a device tree source (DTS) file. The DTB is applied at application startup as a device tree overlay introducing the new hardware to the OS.

Besides the tool-flow to load custom hardware, a device driver prototype was implemented in order to provide a runtime interface for accelerators. The implemented functionalities are accessible in user-space via an accelerator library. These functions are utilized by the accelerator function described in Section 6.1.3.

Another task of the driver is the bookkeeping of multiple accelerators. Therefore, the runtime status of each registered accelerator is managed within the driver-internal structures. The accelerators are identified by their base address and the used register window. If a resource conflict (calling a busy accelerator) is detected the affected process must switch to software execution.

A detailed description of the implemented tools and mechanisms for OS integration is given in Appendix D.

6.6 Base Address Assignment

Even though the generation of accelerators and the corresponding C library have been automated as far as possible, one task must still be handled manually. This is the managing of base addresses for register windows. The correct base address of the accelerators IO memory is required twice in the presented tool-flow.

- In the accelerator function in order to resolve the register IO of an accelerator.
- In the DTS file, as starting position of the register window for the device driver.

Typically, the base address of a device depends on the actual configuration of the bus interface that is used for the accelerator. In case of both evaluated platforms,

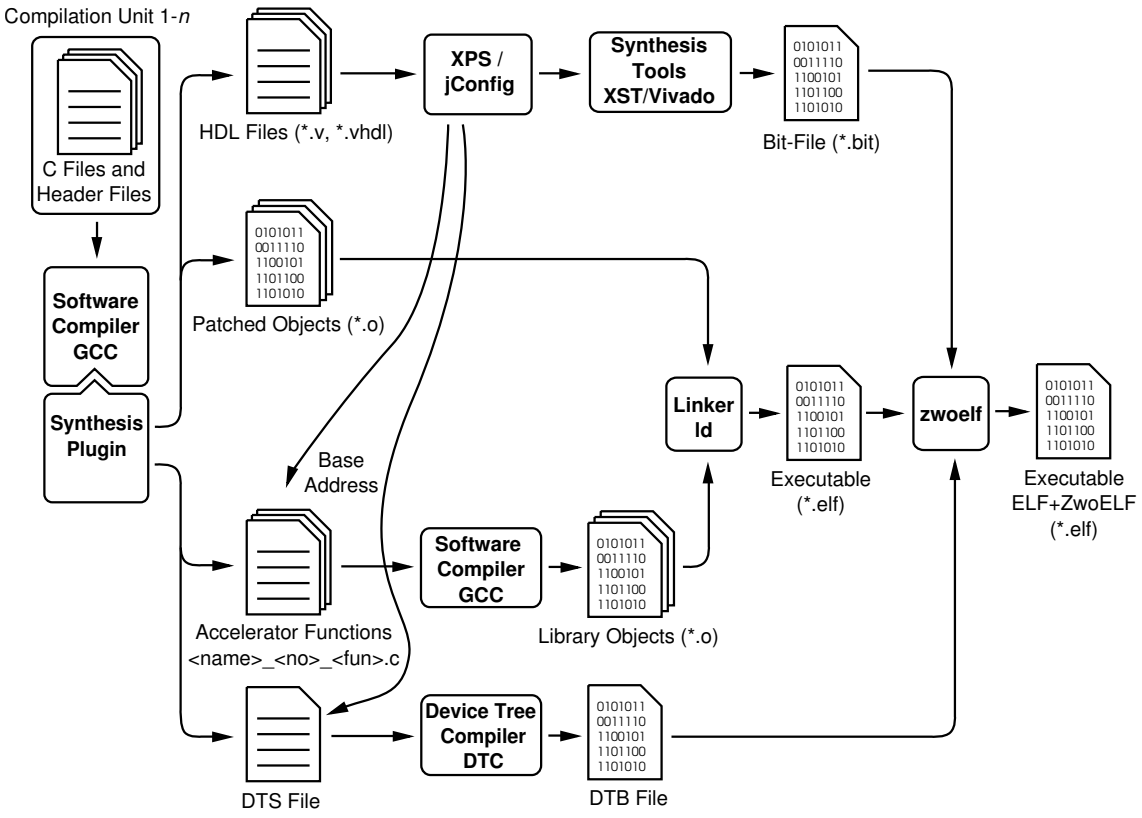

Figure 6.14: Decomposed compilation tool-flow and patching of base addresses

SpartanMC and Zynq SoC, the base address is determined in a system-builder application (jConfig or XPS) while adding the generated accelerators. This makes perfect sense, as these applications are being used to manage the complete peripheral setup of the SoC. They provide an overview of the currently used address space and allow fitting devices optimally. Unfortunately, registration of the accelerator hardware can take place only after all accelerators have been created. This necessitates a decomposition of the compilation process, as shown in Figure 6.14. First, all compilation units belonging to the application are translated in order to detect and generate the accelerators. Afterwards, the generated hardware units can be added to the system-builder application to gather the base addresses. These addresses will be further used to patch the generated C code or the DTS files (dashed arrows in Figure 6.14). Finally, the generated C files and the library are translated and linked with the application objects into an executable that is further processed by `zwoelf` in order to combine it with the bit-file and DTB.

It is obvious that the whole process can be automated by some light-weight scripts wrapping the current work-flow. However, this has been only partially implemented so far. On the one hand, the execution of the compilation tool-flow, the linking, and the synthesis work-flow is carried out by the make tool-chain. On the other hand, the patching of base addresses and the generation of the final executable (ELF+ZwoELF) is carried out manually. These parts of the tool-chain are only required for architectures with an OS. Thus, they are currently not integrated to keep the tool-chain compatible with the rather simple SpartanMC tool-flow.

7 HIGH-LEVEL HARDWARE SYNTHESIS

The following chapter describes the translation from *GIMPLE-IR* to the hardware description language *Verilog*. This process is the substantial part of the *synthesis pass* in the second GCC run (cf. Figure 5.8). The flow-chart in Figure 7.1 shows the processing steps of the synthesis pass. The HLS flow is executed after the candidate selection encompassing the critical path analysis, the pseudo-scheduling, and the memory access analysis (cf. Chapter 5). Subsequently, the HLS is carried out, generating an accelerator for each loop candidate in the candidate list.

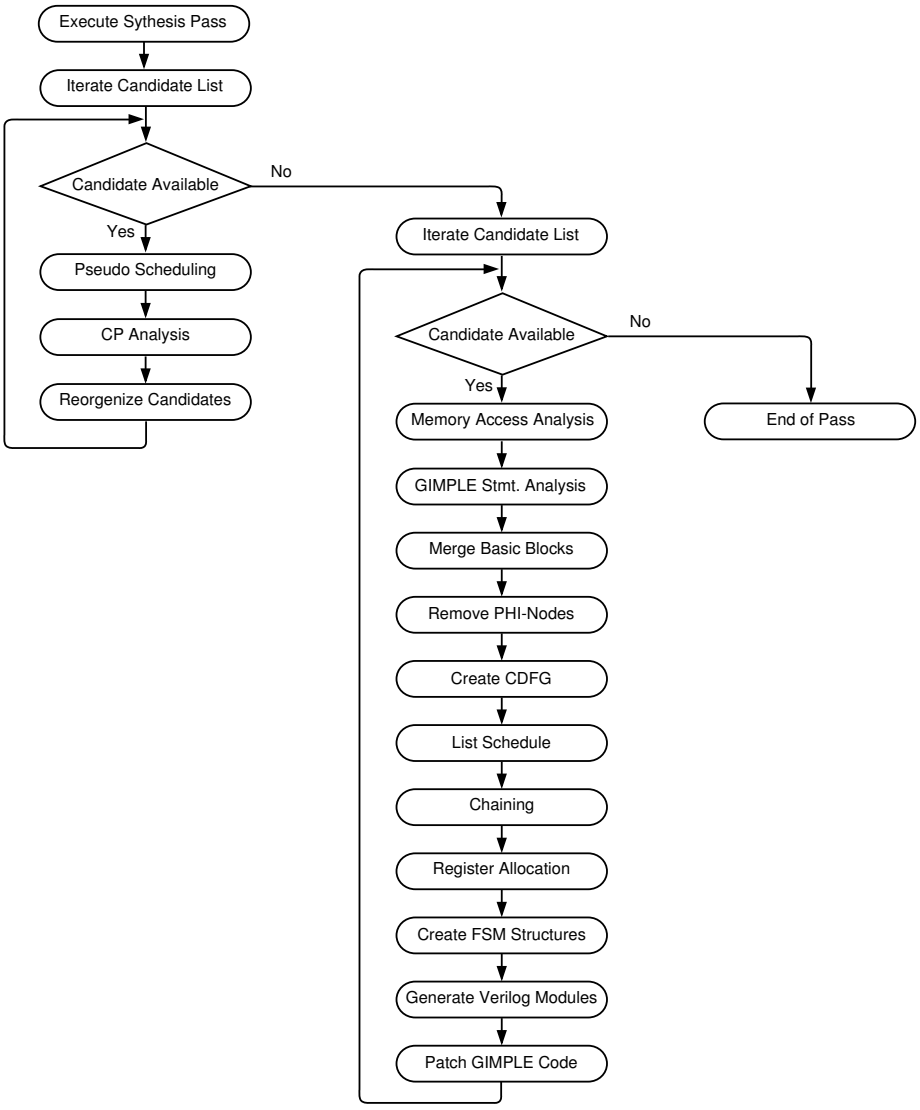

Figure 7.1: Flow-chart of the synthesis pass

7.1 Variants of Processor Customizations

Before implementing the HLS part of the synthesis pass, the general efficiency of different design approaches was evaluated in [141]. In this publication two case studies that compare different design approaches for processor customizations were implemented. Both approaches use the SpartanMC soft-core processor as underlying architecture.

First, an acceleration approach based on instruction set extensions was evaluated. Such extensions are already successful on other architectures, e.g. for digital signal processors using the Tensilica Xtensa technology [93]. In [141] they were implemented as special operations for a JPEG decoder and an AES cypher[1]. The result was an intermediate speedup but also a very small resource footprint for the accelerator. Second, both algorithms were accelerated by two rather complex peripheral units. Hence, the application was executed almost completely in hardware. Naturally, this results in better speedups but at the price of the exhaustive use of hardware resources. Surprisingly, the experiments have shown that the efficiency[2] of the dedicated peripheral was, by the order of a magnitude, higher than an implementation based on the instruction set extension. Furthermore, the peripheral extensions provide better scalability of the interface and require less engineering effort for integration in the host processor.

According to these experiments, the automatically generated hardware accelerators are considered to be implemented as peripheral extensions.

7.2 From GIMPLE to HDL

The first task for the HLS is the analysis of the existing GIMPLE structure in order to generate an internal CDFG. The generated CDFG forms the basis for all HLS optimizations that are carried out in the plugin. The transformation to a new internal representation is necessary, as the existing GCC structures are neither suitable for all optimizations and modifications nor contain all required information for hardware generation[3]. The created CDFG is derived from the DFG inside the

[1] Evaluated extensions were the inverse discrete cosine transformation (IDCT) operation in a JPEG decoder and the MixColumn operation of an Advanced Encryption Standard (AES) cipher.

[2] The efficiency was measured in speedup per LUT.

[3] A modification of GCC internal structures is not recommended or even possible when using plugins.

basic blocks and the CFG between the basic blocks.

The entry point for the transformation is the loop header basic block of the outermost loop. The transformation ends with the corresponding final block of the loop body – the latch block. Initially, the generated intermediate structure strictly follows the existing GIMPLE structure. The generated blocks use the same indices as the corresponding original blocks. In addition, each block is examined to identify loop exit blocks or memory accesses. Both are attached to the graph as attributes.

In the case of *if-else* or *switch* statements the existing DFG is mirrored to the newly generated DFG. Hence, the resulting internal structure contains the same basic block boundaries as the original GIMPLE representation.

7.2.1 GIMPLE Statement Analysis

In order to obtain the data-flow sequence within a basic block, each GIMPLE statement is analyzed and translated to an internal structure that preserves the given SSA structure of the original operands. Therefore, the GCC internal GIMPLE-statement-iterators (GSIs) are used. Each statement contain at least one operand for the LHS and up to two operands connected with an operator for the RHS. These Tree SSA operands are analyzed in the recursive `analyze_operand(...)` function. The Tree SSA structure itself is an acyclic tree that can be analyzed by using the `TREE_TYPE` macro. It returns the corresponding subtree type or the `NULL_TREE` (cf. Section 4.2.1) as the final node. The number of tree types occurring in the GIMPLE representation, on a specific optimization level, varies with the GCC version or the translated application. For this reason, the current implementation follows a white list approach. The used analysis function implements only the most widely used types. If the found type is not supported, e.g. floating-point related types like `REAL_CST`, the plugin throws a warning and the loop candidate is rejected. A list of currently supported tree types is given in C.8.

As already mentioned, the operands of the RHS are commonly connected with an operation. These operations are directly mapped to an equivalent Verilog operation. Not all possible GIMPLE operations, e.g. division or modulo, are currently supported for hardware generation. The implementation throws a warning if an unsupported operation occurs. The supported operations are listed in C.9.

Additional tree types as well as additional operations can be easily implemented by extending the corresponding plugin functions. Nevertheless, such extensions require considerable engineering effort, as they usually imply the design of special hardware modules, for instance, a floating-point ALU.

PHI-nodes typically occur in basic blocks to join the control flow behind `if-else` or `switch` structures. They were analyzed with a GSI especially for PHI-statements. In order to simplify the generated hardware of PHI-nodes, they were treated as unary assignments and, thus, moved upwards into the corresponding branches. As a consequence, the plugin broke up with the SSA structure for these nodes. If the speculative execution (cf. Section 7.4.1) is applied successfully for alternative execution paths, such PHI-nodes can be translated to multiplexers.

7.2.2 Transformation of Memory Accesses

Memory accesses on the GIMPLE-level can be lowered to `TARGET_MEM_REF`, or `MEM_REF`, or can be described as direct pointer arithmetic. While the latter contains a direct address that can be used for memory access in hardware, the former contains an implicit address calculation that must be expanded to dedicated arithmetic operations. For this purpose, the generated CDFG is extended with additional operations and temporaries. The generated arithmetic for a `TARGET_MEM_REF` is shown in Listings 7.2 and 7.3.

```
D1828.1 = D_MEM_REF [
    base:   D1829.2,
    index:  i_19,
    offset: 2]
```

Listing 7.2: Transcript of memory access in GIMPLE [131]

```
tmp1 = D1829.2 + i_19
tmp_addr = tmp1 + 2
mem_set_addr(tmp_addr)
D1828.1 = get_data()
```

Listing 7.3: Transformed memory access in CDFG transcript [131]

For memory read[4] operations, the setup of the memory address is implemented as a separate CDFG note. For write operations, a *fire-and-forget* technique is implemented. Consequently, the address setup and the write operations are

[4] Read and write operations are regarded from the accelerators point of view. *Read* means that the accelerator reads the host processor memory. Vice versa, *write* describes a write operation from the accelerator to the host processor memory.

unified in one CDFG node. If the memory access analysis finds any interdependencies for the current memory access, the corresponding accesses must retain the domination order given by the GIMPLE representation. Otherwise, the accesses could be parallelized. However, such parallel accesses are usable only if they are supported by the underlying memory interface. This is only the case for the ARM FIFO interface (cf. Section 8.3). The domination order of memory accesses is stored into a separate linked list that is used also by later optimization algorithms.

7.3 Generation of the State Machine

After generation of an internal CDFG from the GIMPLE representation, it is now possible to generate a state machine structure that is based on the CDFG representation. The state machine structure is an internal representation of the loop body that contains almost all structural properties of the later hardware design. Hence, this structure represents the last stage before the actual code generation can be carried out.

The entry point of the state machine generation is the function `generate_statemachine(...)`, which creates the main structure `hw_statemachine`. This structure, as shown in Listing 7.4, contains among others a vector of generic states as well as references to special states that define the state machine, e.g. the IDLE state or the exit states.

Each state is defined by a list of instructions, and a list of predecessor *states* and successor *branch-states* (cf. Listing 7.5). Hence, each state node is followed by a conditional state transition that is specified in a branch-state (cf. Listing 7.6).

This structure contains a reference to its next state and a condition for the transition. If the state transition is unconditional the condition reference is `NULL`.

```
struct GTY(()) hw_statemachine {
    ...
    hw_state_p idle_state;
    vec<hw_state_p,va_gc> *exit_states;
    vec<hw_state_p,va_gc> *states;
};
```

Listing 7.4: State machine structure (cf. [131])

```
struct GTY(()) hw_state {
    ...
    bool exit_state;
    vec<hw_state_p,va_gc> *predecessors;
    vec<hw_state_branch_p,va_gc> *sucessors;
    vec<hw_instruction_p,va_gc> *instructions;
    ...
};
```

Listing 7.5: State structure (cf. [131])

```
struct GTY(()) hw_state_branch {
    hw_cond_p condition;
    hw_state_p next_state;
    ...
};
```

Listing 7.6: State branch structure (cf. [131])

The actual generation of the state machine structure is carried out in the function `generate_-state_for_block(...)`. The general principle of operation is given in Algorithm 6. This function is initially called by `generate_state-machine(...)` with the *header* basic block of the loop as starting point. As shown in Algorithm 6, the function generates a single state for all DFG nodes sharing the same timeslot.

The timeslot describes the time segment for one execution step of the later datapath. For an *unoptimized* CDFG, each timeslot includes only a single assignment operation. The sequential order of timeslots is derived from the SSA order within

the current basic block.

After the iterative generation of states for each DFG node of the current basic block, the function traverses the graph of subsequent basic blocks. In the context of a loop body, such subsequent blocks can occur after *if-than-else* or *switch* statements. The resulting state machine representation is a bidirectional linked graph. This graph may also contain cycles that could be introduced by nested loops.

Algorithm 6: Function `generate_state_for_block`

```
   Input: BB ⊆ CDFG(V, E)
 1 if state ≠ ∅ then
 2     return
 3 foreach timeslot ∈ BB do
 4     state ⟵ createState()
 5     foreach v_i ∈ V | v_i ⊆ timeslot do
 6         state ⟵ add_instruction(v_i)
 7         if is_branch_node(v_i) then
 8             foreach branch_target(v_i) do
 9                 next_BB ⟵ branch_target(v_i)
10                 generate_state_for_block(next_BB)
11 return next_state ⟵ state
```

7.3.1 Generation of HDL Code

The final step, in order to generate an HDL description from GIMPLE, is the composition of HDL code statements. While the internal representation of the state machine is still platform-independent, this approach was not applicable for the HDL code generation. For instance, the memory interface of different platforms could introduce variations of the state sequence and the interface logic. Nevertheless, in context of this thesis, the generated FSMs were portable to two target platforms with very little effort.

The concept of the FSM follows a simple but effective approach. After reset, the generated FSM (cf. Figure 7.7) remains in the IDLE state. Before starting the accelerator at the first time, all parameter registers must be initialized. If this

is done, the first state change could be triggered by setting the *start* flag in the accelerators control register. After that, the accelerator starts the generated FSM for the loop body. The accelerator may contain multiple loop exits that always trigger a special exit state. These states are used to write the corresponding return values of the loop into parameter registers. Even if there is no available return value for a loop, there will be at least one exit state providing the value `bb_idx`. This value specifies the index of the next basic block in the program control-flow.

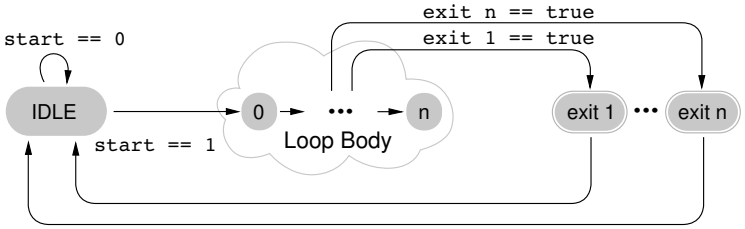

Figure 7.7: Structure of the FSM

Besides the integrated parameter registers, the FSM performs master mode memory accesses[5] during runtime. Depending on the used interface, such accesses may introduce an indeterministic delay to the FSM execution. This issue must be considered when generating states that access external memories. For this reason, each state containing a memory access is generated with a self-referencing edge as presented in Figure 7.8. The only way to leave such a *memory-access* state is an acknowledgement from the external memory interface. This technique is used for memory accesses on both target platforms. However, for a synchronous memory interface that provide a deterministic latency, the acknowledgement is internally controlled by the accelerator hardware. For instance, the memory interface for SpartanMC *always* provides its data within the next clock cycle, thus the acknowledgement is set automatically.

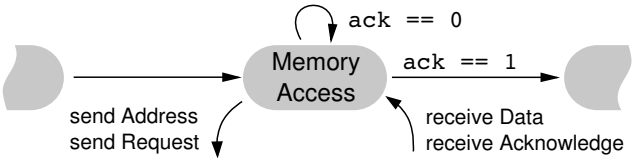

Figure 7.8: Memory access state

[5]For the Zynq platform, the FSM handles only slave mode memory *requests*.

The generated datapath for accelerators corresponds to the generic structure presented in Figure 7.9. It basically consists of IO registers, intermediate registers, and arithmetic units (AU). The data-flow between these components is driven by a set of multiplexers controlled by the current state of the FSM.

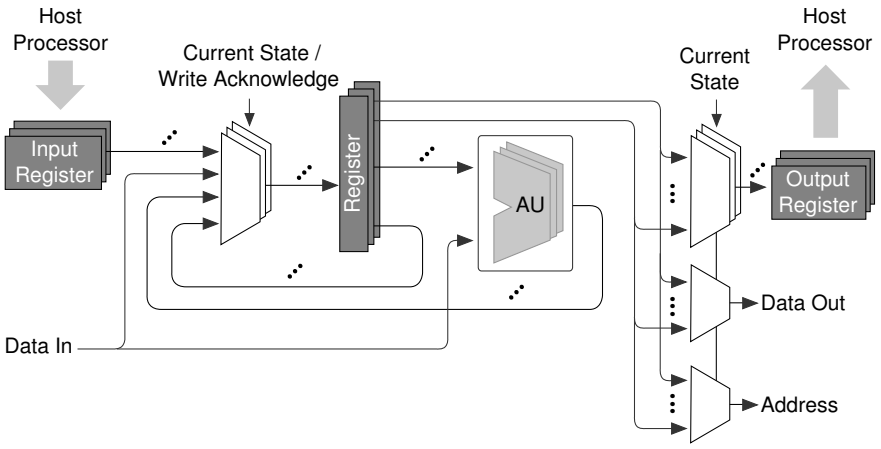

Figure 7.9: Generic datapath structure of the accelerator (cf. [131])

7.4 Optimization Strategies

The FSM description generated by the straight transformation process described above produces a technically correct hardware design that would already be functional when translated to Verilog code. However, it describes only a one to one mapping of the given GIMPLE sequence.

Nevertheless, the described process and the provided intermediate representation (the CDFG) can be regarded as a good basis for applying various optimization strategies. The following sections discuss three optimization strategies that were implemented for this thesis, namely speculation, scheduling, and chaining. Each presented optimization strategy can be enabled via a plugin parameter (cf. C.2).

7.4.1 Speculative Execution of Branches

The first optimization step is the speculative execution of branches that was implemented in [131]. Naturally, this step is accompanied by a modification of the basic blocks structure. Hence, the whole CFG of the loop candidate is potentially restructured during this optimization. Meanwhile, further optimizations – scheduling and chaining – are limited to the DFG structure inside the basic block. For this reason, the actual generation of the complete CDFG is scheduled after the steps required for the speculative execution, namely "*Merge Basic Blocks*" and "*Remove PHI-Nodes*" in Figure 7.1.

Speculative execution in the context of hardware generation means the merging of parallel branches or basic blocks. Therefore, the result of each branch is calculated regardless of the given condition. At the end of the speculative execution path, a multiplexer decides which results will be used for the following operations. The merging of basic blocks increases the number of nodes in a single basic block that, in turn, provides a better schedule of the DFG within the merged block.

Merging Basic Blocks

In order to find conditionals that can be merged, the CFG is searched for basic blocks containing `GIMPLE_BRANCH` or `GIMPLE_SWITCH` statements. If such

branch statements are found, the underlying basic block structure is tested for the specific branch type, e.g. *if-then-else* or *switch* (cf. Algorithm 7). The branches are further checked for subordinated branches. If an innermost branch is identified as suitable for speculative execution the merging of basic blocks is carried out.

Algorithm 7: Merging basic blocks

```
   Input: CFG_predicative
   Output: CFG_speculative
1  CFG ⟵ CFG_predicative
2  repeat
       /* find if-then-else-candidate (C) and merge blocks          */
3      foreach C ⊆ CFG | contain_if_then_else(C) do
4          find_innermost(C)
5          CFG ⟵ merge_if_then_else_blocks(C)
       /* find if-candidate and merge blocks                        */
6      foreach C ⊆ CFG | contain_if(C) do
7          find_innermost(C)
8          CFG ⟵ merge_if_blocks(C)
       /* find switch-candidate and merge blocks                    */
9      foreach C ⊆ CFG | contain_switch(C) do
10         find_innermost(C)
11         CFG ⟵ merge_switch_blocks(C)
       /* find candidate for linked blocks and merge them           */
12     foreach C ⊆ CFG | is_linked(C) do
13         CFG ⟵ merge_linked_blocks(C)
14 until no blocks merged
15 return CFG_speculative ⟵ CFG
```

The actual merging of conditional basic blocks (`merge_if_blocks(...)`, `merge_switch_blocks(...)` and `merge_if_then_else_blocks(...)` in Algorithm 7) is implemented for two kinds of branch structures: conditions containing empty branches and conditions without empty branches. The merging of structures without an empty path is outlined in Figure 7.10 *(A)*. First, the edges to the original BB 2 and 3 are cut and a new BB 5 is integrated into the CFG. The condition and the PHI-nodes of the original blocks are merged to a multiplexer structure that is integrated into the new basic block. Later, the remaining

statement lists of both branches are copied into a merged list for the new basic block. This is possible as the data-flow between arithmetic statements uses the SSA property. If the condition contains an empty branch (Figure 7.10 *(B)*), the merging process is quite similar, with the exception of the new basic block. This additional block is not required, because the unified statement list and the multiplexer (PHI-node and condition) can be integrated in the existing basic block (Block 1 in Figure 7.10 *(B)*). The resulting chain of consecutive basic blocks is merged afterwards in the function `merge_linked_blocks(...)` by a following iteration of the embracing loop in Algorithm 7.

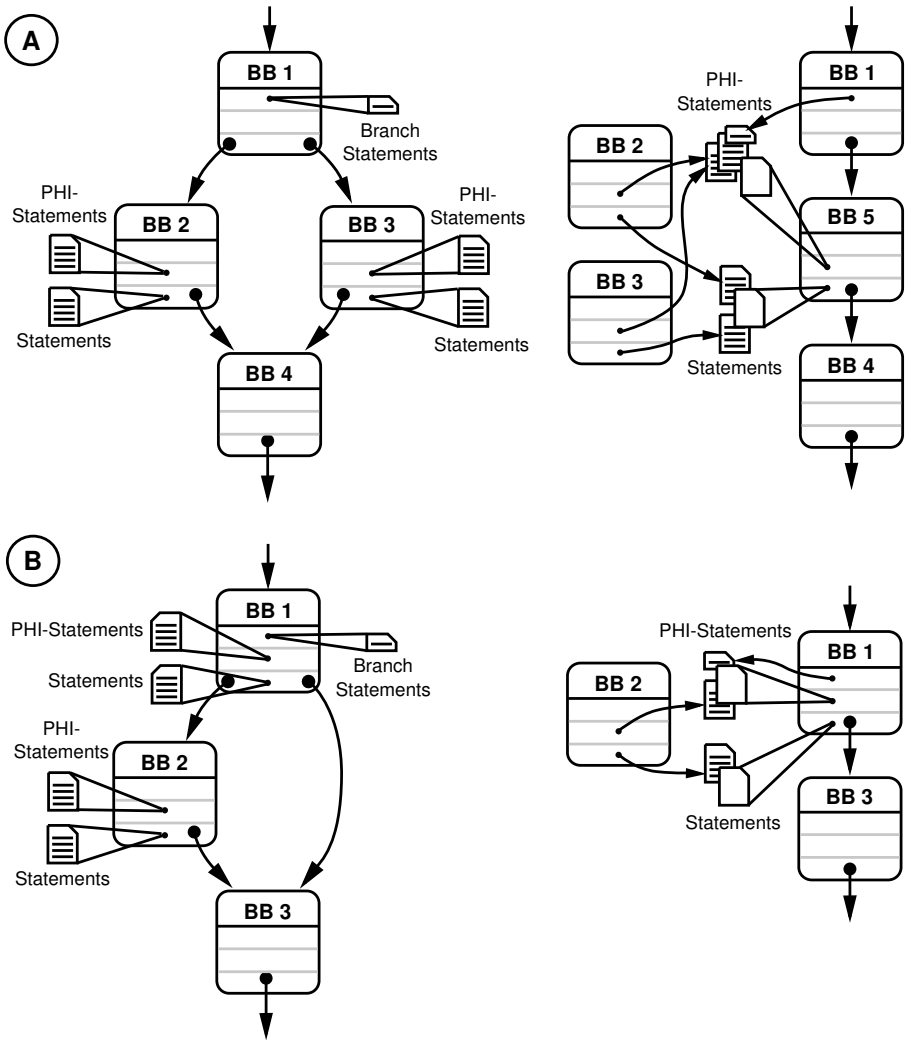

Figure 7.10: Merging of basic blocks for balanced *(A)* and unbalanced *(B)* branch structures [131]

Speculative or Predicative Execution

If the parallel branches contain memory-write operations or the branch itself is the loop body of a subordinated loop structure, the unmodified predicative execution is preserved. In all other cases, the choice between speculative or predicative execution depends on a metric that is based on the *block weights* of the parallel branches. The idea behind this metric is the avoidance of a speculative execution for very unbalanced branches that could introduce an unfavorable impact on the overall execution time of an accelerator.

The weights are determined for each branch by using the number of statements or the latency[6] of the execution path. The actual used metric is specified with a plugin parameter (cf. C.2). After determining all *weights*, the average weight

$$avg\_weight = \frac{\sum weights}{\#blocks}$$

is calculated. Furthermore, the minimum $weight_{min}$ and the maximum $weight_{max}$ of the given branches are used to determine the minimum and maximum deviations from the average:

$$dev_1 = |weight_{min} - avg\_weight|,$$
$$dev_2 = |weight_{max} - avg\_weight|.$$

If the deviation either dev_1 or dev_2 exceeds a predefined threshold, speculative execution is declined for the given branch.

Speculative Execution Example

Listing 7.11 shows a GIMPLE transcript containing a conditional branch. It contains the variable `sum` in three versions: Version 1 and 2 within the branches and Version 3 in BB 6. The value of `sum` is assigned in a PHI-node that decides which version will be used for further calculations.

This code is translated to the CDFG, as shown in Figure 7.12 *(A)*. The edges are labeled with the variable names derived from the GIMPLE representation. Note

[6] This is approximated with an as soon as possible (ASAP) schedule on the internal DFG.

```
<bb 3>:
  mul_7 = a_1 * b_6;
  if (mul_7 < 0)  goto <bb 4>;
  else  goto <bb 5>;
<bb 4>:
  tmp_8 = -mul_7;
  sum_1 = tmp_8 + c_9;
  goto <bb 6>;
<bb 5>:
  sum_2 = mul_7 + d_11;
<bb 6>:
  sum_3 = PHI <sum_1(4), sum_2(5)>;
```

Listing 7.11: Conditional branches – GIMPLE transcript

that the PHI-node in BB 6 is split and moved to the branches in BB 4 and 5 as described in Section 7.2.1. If the speculation algorithm is applied to the given CFG, the branches are merged to a new basic block (cf. Figure 7.12 *(B)*). The PHI-node for `sum_3` is implemented as a multiplexer in the resulting block. Even though the critical path of the new block (BB7) is shortened by one operation, the execution path of the former BB 5 is now dominated by the execution path of the longer BB 4. If the deviation of the block weights was set to one, the given code sequence would never have been a suitable candidate for speculation.

7.4.2 DFG Generation and Scheduling

After analysis of the GIMPLE representation and the generation of speculative execution paths, scheduling can be applied to each DFG[7]. Therefore, the internal representation of the DFG is generated from the list of statements, PHI-nodes, and branch operations in each basic block. Together with the CFG from the speculative execution, they form the CDFG.

The DFG(V, E) on which the scheduling is carried out is an acyclic graph of nodes $v_i \in V$. Each node represents an operation while the edges between these nodes are defined by a tuple $(v_i, v_j) \in E$ with respect to the data dependency or domination order (v_i dominates v_j). The entry points of the graph form a set of edges with the predecessor node $(v_i, v_S) \notin E$. The generation process of

[7] Of course, the existing DFG already provides a valid schedule which is derived from the order of GIMPLE statements, but this schedule neither exploits parallel execution nor considers any resource constraints.

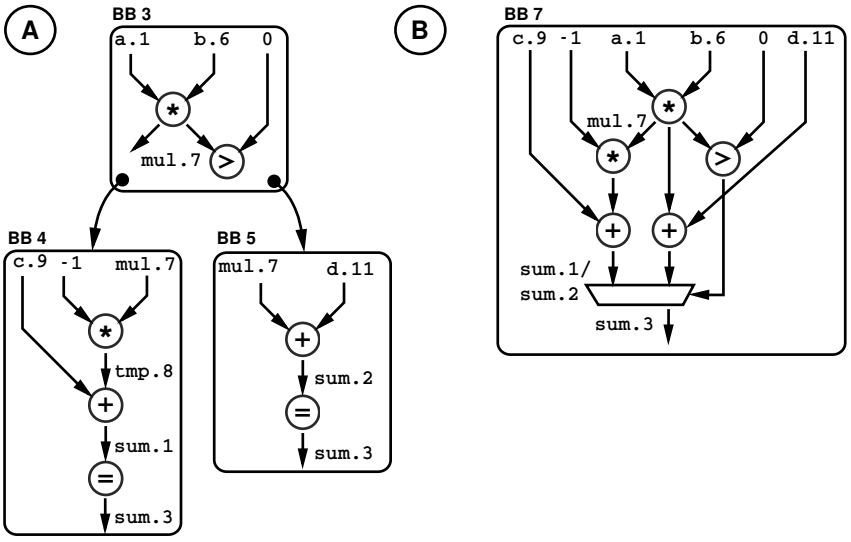

Figure 7.12: Original DFG *(A)* and speculated DFG *(B)* [131]

the DFG is shown in Algorithm 8. First, the statement list of each basic block is iterated over and a corresponding data-flow graph structure is generated. The operators within the statements are used to define the DFG-nodes of the graph. In addition, the memory operations are stored in a sorted list according to their domination order, which is later required for a correct scheduling of memory operations. Afterwards, the SSA property of the operands in the DFG-nodes is used to generate the data-flow edges of the graph. Finally, the resulting DFG is ready for scheduling.

7.4.3 List-Scheduling

The schedule is generated using a list scheduling algorithm [95, p. 207 ff.] that was implemented in [131]. The algorithm aims to assign the starting time t_i to all nodes in a DFG in order to minimize the overall execution latency. Naturally, scheduling algorithms are tailored for linear code sections. Thus, they cannot handle conditional branches and loop nests. In order to achieve a correct handling of the existing CDFG, the scheduling is executed multiple times, once for each basic block. During the scheduling process, the algorithm requires a heuristic priority criterion in order to choose the next operation node from a candidate list. The plugin provides two heuristics for this task that can be used in an arbitrary manner.

Algorithm 8: Generation of a DFG [131]

```
   Input: List of GIMPLE statements
   Output: Internal DFG(V, E)
1  foreach BB do
2      DFG(∅, ∅) ⟵ init_DFG_structure()
3      foreach Statement do
           /* Allocate DFG-node structures (V)                    */
4          DFG(V, ∅) ⟵ init_DFG_node()
5          Memory_Access_List ⟵ find_mem_operations()
6  foreach DFG(V, ∅) do
7      foreach Node do
           /* Generate DFG edges (E)                              */
8          DFG(V, E) ⟵ calculate_dependencies()
9  foreach DFG(V, E) do
       /* Execute optimizations, e.g. list scheduling             */
10     DFG(V, E) ⟵ do_scheduling()
```

First, is the mobility of an operation, which is provided in the `dfg_node` structure. It defines the difference between the latest start time $t_i{}^L$ and the earliest start time $t_i{}^S$ of an operation within the DFG, under the condition that the dependency on the predecessor and successor node is preserved. The earliest beginning of an operation is determined with an ASAP schedule. Therefore, the earliest start time ($t_i{}^S$) is defined by the maximum of the sum of the start time t_j and the delay time d_j for each predecessor node

$$t_i{}^S = max(t_j + d_j) \quad \forall v_i, v_j : (v_i, v_j) \in E.$$

Vice versa, the latest start time $t_i{}^L$ is calculated by an as late as possible (ALAP) schedule defining $t_i{}^L$ as follows:

$$t_i{}^L = min(t_j + d_j) \quad \forall v_i, v_j : (v_i, v_j) \in E.$$

Second, the weight of a node can be used to determine the next candidate for the list scheduling. The implemented algorithm defines the weight w_i as the number

of successor nodes within the path of the current node. The implementation uses the following recursive rule to define the weight for a node $vi \in V$:

$$w_i = \begin{cases} 1, & \forall v_i, v_j : (v_i, v_j) \notin E \\ max(w_i) + 1, & \forall v_i, v_j : (v_i, v_j) \in E \end{cases} .$$

If more than one node could be scheduled in case of identical weight or mobility, the implementation of the algorithm picks the first node in the candidate list.

One feature of list scheduling is the possibility to include resource constraints. Due to the large amount of hardware resources on current FPGAs, the required constraints are not too harsh. Nevertheless, they are implemented in the plugin. Typical resource types within the DFG namely, ALU, multiplier, special operation, and memory accesses are defined in an enumeration and can be allocated by the scheduling algorithm. The function τ is used to determine the resource type of a node $v_i \in V$. The quantity of each resource type k is provided by a list a_k with $k = \{1, 2, \ldots, n_{res}\}$. The implemented algorithm (cf. Algorithm 9) is derived from [95]. It is implemented in two nested loops. The outer loop is executed until all nodes are scheduled while the inner loop is used to iterate over resource types in order to find a suitable next candidate.

Algorithm 9: List scheduling [95]

Input: $DFG(V, E)$
Output: $DFG(V, E)_{scheduled}$

```
repeat
    foreach k do
        U_{l,k} ⟵ get_candidate_nodes()
        T_{l,k} ⟵ get_busy_nodes()
        S_k ⟵ select_next_nodes()
        DFG(V,E) ⟵ schedule_to_timeslot(v_k ∈ S_k)
until ∀v_i ∈ V,  v_i → scheduled
DFG(V,E)_scheduled ⟵ DFG(V,E)
```

The loop body of the list scheduling comprises four steps in order to calculate a start timeslot[8] for each node.

[8] Possible start times are assigned on the granularity of timeslots, which corresponds to clock cycles in hardware.

1. Determining the set of candidate nodes $U_{l,k}$ for the timeslot l and the resource k (implemented in the function `get_candidates(...)`). Possible candidates are nodes whose predecessors are already scheduled:

$$U_{l,k} = \{v_j \in V : \tau(v_j) = k, t_i + d_i \leq l, \quad \forall v_i : (v_i, v_j) \in E\}.$$

   Moreover, all memory nodes are gathered and stored in an extra list, sorted by their domination order. This is necessary, as memory accesses are considered to have side effects. Thus, the domination order must be preserved during the scheduling process.

2. In the next step, the set of busy resources $T_{l,k}$ for the current timeslot is determined:

$$T_{l,k} = \{v_i \in V : \tau(v_i) = k, t_i + d_i > l\}.$$

3. Subsequently, the set of scheduling candidates $S_K(S \subseteq U_{l,k})$ is determined under the condition that the sum of scheduling nodes and busy nodes of a specific resource type does not exceed the number of available resources:

$$|S_k| + |T_{l,k}| \leq a_k.$$

   Scheduling candidates are selected by the implemented heuristic (mobility or weight).

4. The candidates are scheduled in the active timeslot until the maximum of resources is reached. Memory nodes consisting of address-set and read operations are categorically scheduled consecutively.

Finally, the scheduling algorithm checks for possible branch nodes. Such nodes indicate the end of the basic block and must always be inserted in the final timeslot.

7.4.4 Chaining

One premise of the current scheduling algorithms is that each operation node in the execution path is scheduled in an exclusive timeslot. Hence, the operation

with the highest latency determines the resulting length of the timeslots, which further indicates the best achievable clock frequency for the whole design. For the remaining (faster) nodes, an unnecessary slack is introduced. The idea of chaining is to minimize this slack by combining consecutive nodes within one timeslot. The approach is illustrated in Figure 7.13.

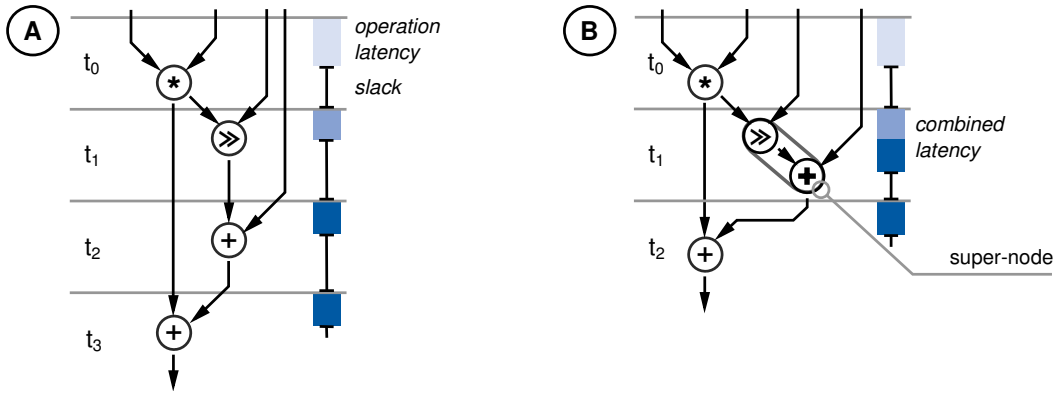

Figure 7.13: Original DFG *(A)* and chained DFG *(B)*

Part *(A)* of the figure shows the original DFG with the execution time and slack for each node. Figure 7.13 *(B) chains* the consecutive ALU nodes to a newly generated super-node. The nodes are now executed within one timeslot which shortens the DFG by one timeslot and reduces the slack of the resulting node.

Chaining is a post-scheduling optimization but it does not change the schedule at all. It just combines suitable nodes in an already scheduled DFG to new super-nodes. The resulting DFG is still intact and can be scheduled again or used for other optimizations. The general implementation of the chaining is outlined in Algorithm 10. It is implemented in four nested loops.

1. The outermost loop iterates over all timeslots of the acyclic $DFG(V, E)$.
2. The subordinate loop is executed until all available nodes are chained or no further optimizations can be applied.
3. In the next loop the candidate set $U_{l,k}$ and the set of free resources $R_{l,k}$ are determined for the current timeslot l and each resource k.
4. In the innermost loop, candidates from the candidate set are chained until the set is empty or the available resources are insufficient. Chaining is carried out if the delay of the current candidate is larger than or equal to the

Algorithm 10: Chaining [134]

Input: $DFG(V, E)$
Output: $DFG(V, E)_{chained}$

```
1  foreach Timeslot do
2      repeat
3          foreach Resource type k do
4              U_{l,k} ⟵ get_candidate_nodes()
5              R_{l,k} ⟵ get_free_resources()
6              repeat
7                  v_k ⟵ select_next_node()
8                  DFG(V, E) ⟵ chain_candidate(v_k ∈ U_{l,k})
                   /* Remove resource for chained operation node from list
                      of available resources.                              */
9                  |R_{l,k}| − −
10             until U_{l,k} == ∅ or |R_{l,k}| == 1
11     until ∀U_{l,k} ∈ V,   v_i →chained or optimized
12 DFG(V, E)_{chained} ⟵ DFG(V, E)
```

slack of the upstream operations. The priority criterion for picking the next candidate is the timeslot of its former position in the DFG. Candidates from an earlier timeslot are chained first. If this criterion is identical for several candidates the node priority from the previous list scheduling (weight, mobility) is used. If this second criterion is also identical the nodes are picked according to their position in the candidate list.

To ensure good utilization of timeslots, a heuristic model for the delay times of operations is applied. Therefore, the delay time of each operation is assigned a value between zero and 100. Accordingly, operations are clustered within a timeslot as long as the sum of all delays does not exceed 100. This allows the classification of operation delays without the need for absolute latencies that would require a non-trivial evaluation on the underlying architecture. Nevertheless, the used values in Table C.2 represent only a heuristic model, roughly adapted to the current FPGA architecture.

The current implementation excludes memory access nodes from chaining by specifying a delay time of 101. This is necessary because such nodes typically require strictly synchronous timing behavior.

7.4.5 Register Allocation

The generated FSM uses internal registers between each stage to store the result of an operation for the next stage. By default, such intermediate registers are exclusively used by the arithmetic units of the corresponding FSM state. Hence, each intermediate register can be regarded as a dead piece of hardware if the corresponding state is not active.

In order to minimize this obvious waste of hardware resources, several register-sharing strategies have been evaluated. The used strategies are based on the left-edge algorithm for register allocation, which was described in [102]. This algorithm derives lifetime intervals for variables from a scheduled DFG. Initially, the introduced algorithm was designed for sequential DFGs without branches or loops. In order to use it in the context of the generated CDFGs, it has to be adapted (cf. [131]) to the available representation. This was achieved by supplementing the classic lifetime intervals for variables with hierarchy levels for each basic block of the CFG. As a consequence, the resulting allocation mechanism was divided into local (inside a basic block) and global allocation steps (across multiple basic blocks).

This implemented algorithm was tested in order to assess its impact on the area consumption and speed of the resulting accelerator design. Surprisingly, the gathered results indicate that the implemented register-allocation strategy seems counterproductive when used together with the vendor-synthesis tools of the FPGA. According to the results presented in [143], the best register-allocation strategy is also the simplest one, which was described as the initial situation. The best results can be achieved by using a fresh register for each variable. A detailed description of the referred experiment and its outcomes will be given in the evaluation part (cf. Section 9.4) of this thesis.

8 GENERATED HOST PROCESSOR INTERFACE

This chapter describes the hardware that is generated to transfer data between host processor and accelerator. Even though the functionality of the interfaces is very similar for both supported targets (SpartanMC and ARM), the actual implementation differs due to the underlying bus architecture. Thus, both architectures require completely different HDL modules for their interface implementation. The interface to the host processor includes all HDL files except the FSM module which uses the same implementation for both platforms.

The following chapter describes the general functionality of the interfaces and points out differences between the supported platforms.

8.1 Parameter Interface

The parameter interface is used to transfer single values to registers of the accelerator. It is intended for the arguments of the accelerator function and some status flags. Either these parameters have to be transmitted before the actual accelerator FSM starts or they are gathered after finishing the calculation in hardware. The interface is implemented as a classical memory-mapped IO. The corresponding registers are embedded in the peripheral address space of the host processor. From the hardware point of view, this means that the accelerator must provide several registers that are connected, usually with some glue logic, to the peripheral bus of the host architecture. For such registers, the host processor always acts as the master that initializes the data transfer to the peripheral. Therefore, both peers must agree on a minimum communication protocol in order to transfer data in a bidirectional way. Furthermore, it would be a good idea to allow the accelerator and the host processor to operate within their own clock domain, which, in turn, requires a synchronization mechanism at the interface between both parts of the system.

In order to support these requirements, the implementation benefits from the special purpose of the interface. As already mentioned, the parameter transfer is required at the beginning and at the end of the accelerator execution. This allows the status flags, `start` and `ready`, to act as a synchronization signal

between both system parts. The resulting communication protocol for a regular accelerator run could be described as follows:

1. The host processor writes the input parameters provided by the accelerator function.
2. The host processor sets the `start` bit.
3. The accelerator FSM switches from *idle* to *running*. The required parameter registers are directly exposed to the accelerator FSM. Thus, inputs can be read and the results can be written during the accelerator execution.
4. After completing execution, the FSM enters the `IDLE` state and the `ready` bit is set.
5. The host processor recognizes the ready bit, reads back the result registers, and continues software execution.

Due to the handshake protocol implemented by the signals `start` and `ready` and a strict separation of input and output registers, an asynchronous access to the parameter interface is possible. Consequently, both parts of the system can use the peripheral registers in their own clock domain without interfering.

8.1.1 SpartanMC Parameter Interface

The parameter interface for the SpartanMC soft-core is implemented by using a static accelerator stub module. This stub is intended as a black-box peripheral module for the SpartanMC system builder `jConfig`. The corresponding Verilog code implements the interface between the peripheral bus of the SpartanMC and the generated modules.

Once the SoC is created by jConfig, the accelerator stub can be regarded as one container for all accelerators. Thus, accelerators can be added or removed without manual modifications of the SoC. The resulting work-flow is shown in Figure 8.1 and consists of four main tasks: Create an SoC design, implement the firmware, use the GCC plugin to compile the firmware with accelerators, and start the vendor synthesis tools.

The idea behind the stub module is to reserve a generic address space for potential hardware accelerators. As jConfig only works on the interface's description,

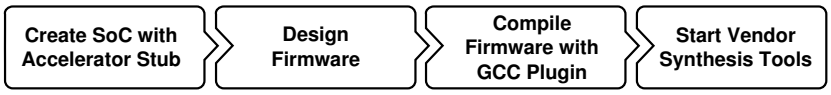

Figure 8.1: Straight work-flow with accelerator stub module

it would treat the black-box module as a regular peripheral. The design of such a module to mimic a SpartanMC peripheral can be accomplished by taking a part of the 18-bit address bus and assigning it to an address decoder within the peripheral module. SpartanMC peripherals are embedded in the peripheral address space typically starting at `0x1A000`. In case of the accelerator stub, seven address bits are reserved for the use of generated accelerators. Thus, the number of 18-bit-registers reserved for accelerators is 128. As accelerators typically need small amounts of input and output registers[1], the number of registers should be sufficient for more than 20 accelerators. Besides the absolute number of accelerators, there is still another reason to encourage small register sets.

It is characteristic for the SpartanMC peripheral interface to allow access to registers in a synchronous fashion within a single processor clock cycle. Besides the advantage of providing very fast peripheral access, this also introduces some minor issues for the development of peripherals. As a consequence of the fully synchronous peripheral access, the logic path that accesses the registers must be a part of the execute-stage of the pipeline. Therefore, the address of a distinct register is decoded by a three-level hierarchical tree of de-multiplexers or address decoders. The write-back port of all registers is combined with a wide *OR* gate. Even though these modules are implemented prudently by using hand-optimized code, it would not be wise to implement too large register sets for one accelerator, as this could influence the critical path of the pipeline. Currently, this is not a problem for accelerators, given the rather small parameter sets.

Figure 8.2 gives an overview on the structure of the peripheral interface. Besides the connection to the peripheral bus and the accelerator stub module, it also shows the module structure for the dynamic part of the interface. This part consists of a generated *top* module that instantiates all subordinate accelerator modules. Therefore, the file `accelerator_top` is composed after the generation of the accelerator FSM modules has taken place.

[1] Most of the accelerators require approximately five or six registers.

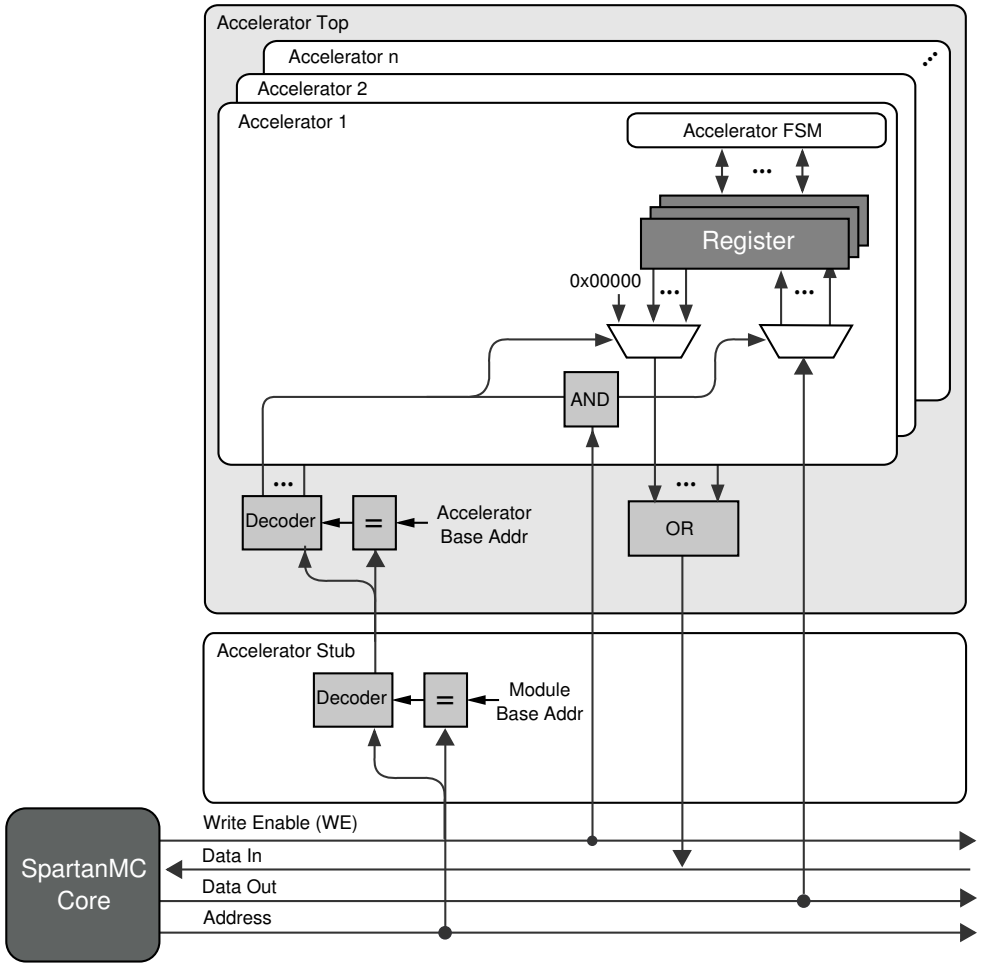

Figure 8.2: SpartanMC register interface

8.1.2 ARM Parameter Interface

The implemented tool-flow for the Xilinx Zynq platform uses Xilinx Platform Studio (XPS) [123] to generate the SoC setup. In contrast to the jConfig-based SoC design, the integration of accelerators in XPS is implemented in a more simplified way. Although technically possible, the ARM interface implementation waives the extra effort of a static accelerator stub module. This less-convenient approach was considered to be acceptable as the support for ARM accelerators only was implemented to evaluate the portability of the presented HLS approach. As a consequence, the overall tool-flow for the Xilinx Zynq platform (cf. Figure 8.3) requires an additional step in XPS (gray box in Figure 8.3) in order to integrate the generated hardware into the existing SoC.

Figure 8.3: Work-flow with manual accelerator integration

The Xilinx Zynq platform uses the AXI bus for communication with peripheral units. Hence, the top module used for interfacing the AXI bus is an AXI interface, integrated into the AXI4 IP Inter Connect (IPIC) [120] module provided by XPS. The address space used for peripherals is part of the general-purpose AXI address space (`0x40000000` – `0xBFFFFFFF`) [129, p. 113], which is provided by the ARM processor.

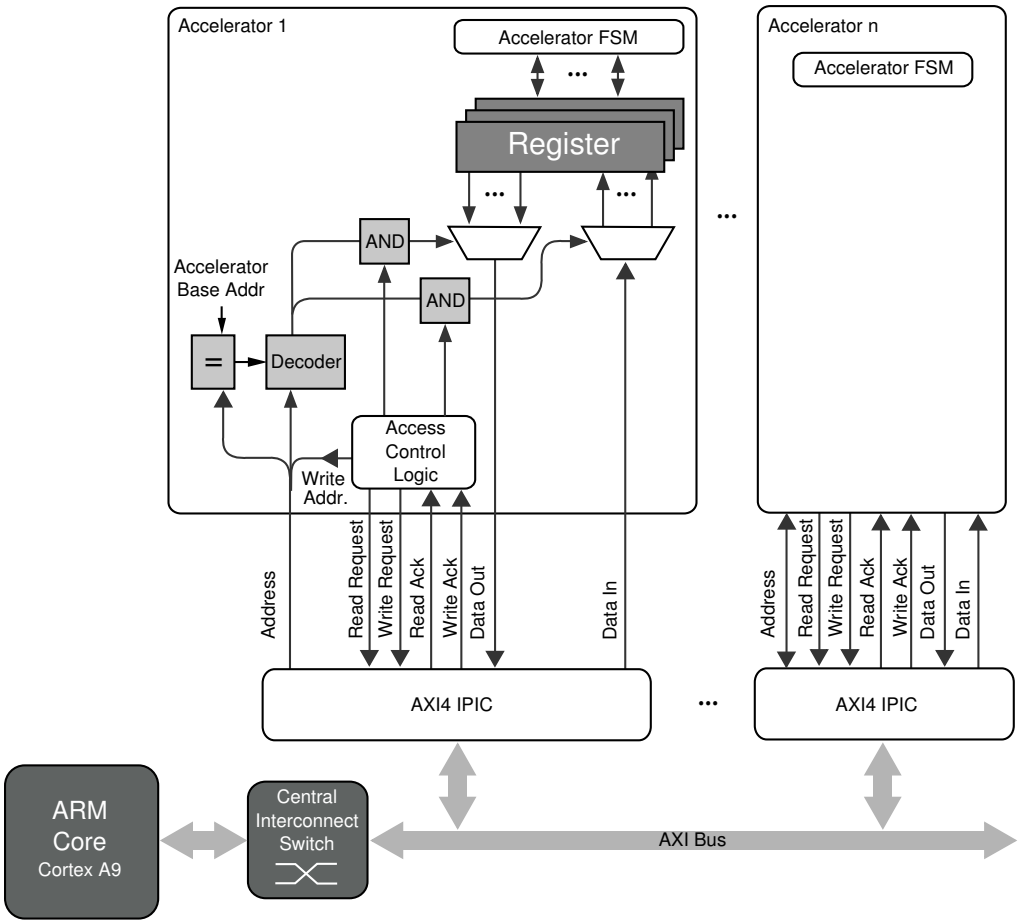

Figure 8.4: ARM register interface

As the accelerator stub interface is not implemented yet, each accelerator implements its own AXI interface module. As shown in Figure 8.4, the remaining module structure and the underlying multiplexer logic that selects single registers are very similar to the SpartanMC design. The actual structural difference between the two designs is introduced by the communication scheme that is used to communicate with the AXI4 IPIC. This interface is, by default, capable of asynchronous communication. This is achieved by the use of request and acknowledge signals that synchronize the data transfer for unexpected latencies or different clock domains. This feature is necessary as the arbitration cycle of the underlying AXI could introduce indeterministic delays. Figure 8.5 shows a sample timing diagram for a write and read transaction. The diagram starts with a write request signal (`w_req`) and a valid address (`addr`) and data value (`data_o`) from the sender. All three channels must remain stable until the next rising edge (three clock cycles in Figure 8.5) of the write acknowledge signal (`w_ack`) that finalizes the transaction. Vice versa, the read access is initialized by a `r_req` and a valid address. Both signals have to remain stable until the data is received and the read acknowledge (`r_ack`) has been sent.

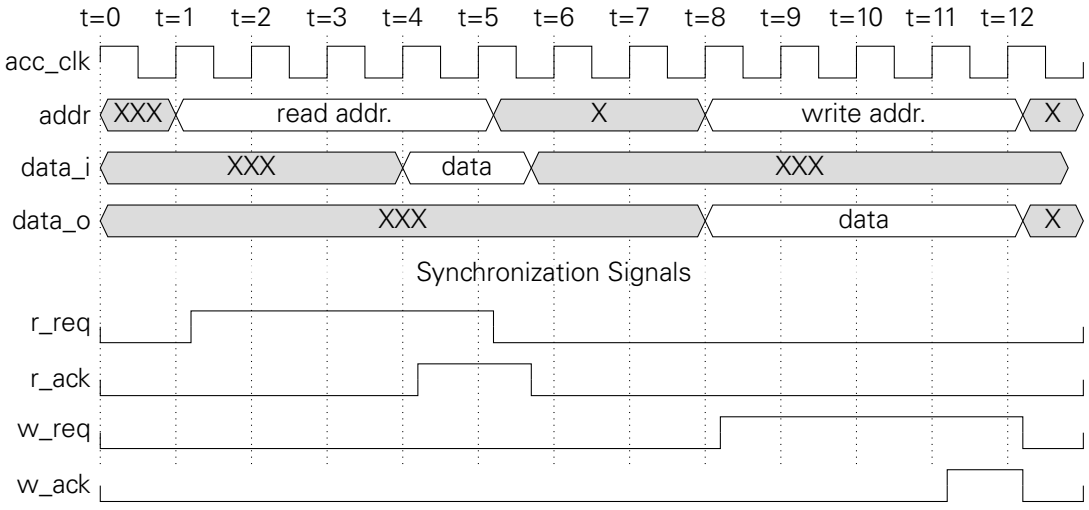

Figure 8.5: Write and read transaction accelerator – AXI4 IPIC IP

8.2 Memory Interface

Besides the register interface, almost all accelerators require an additional memory port to access the main memory of the host system during their execution. This additional port tends to adversely affect the overall system performance, as it does not provide the required data as quickly as the hardware consumes it.

This particularly holds true for custom hardware accelerators, which often execute many arithmetic operations in parallel. In spite of executing operations, the accelerator is forced to stall until sufficient data has been transferred. Hence, the implementation of the memory interface must be carried out with the required diligence in order to limit the impact of this potential bottleneck.

8.2.1 SpartanMC Memory Interface

The implemented solution for SpartanMC is based on a characteristic design paradigm for SpartanMC SoCs. Designed as an embedded SoC for small control or transformation tasks, the SpartanMC soft-core only relies on FPGA internal BRAMs for data and program memory. As BRAMs are dual ported by default, the whole memory subsystem of a SpartanMC SoC provides two ports as well. Typically, one of the ports is reserved for data access and the other is connected with the instruction-fetch stage of the pipeline. Starting from the existing memory layout, an exclusive memory port to the accelerator stub is only a small step away.

Besides the regular layout consisting of a combined data and program memory with two exclusive access ports, the SpartanMC provides an elegant solution for DMA-capable peripherals (cf. Section 3.2.1). As long as there is no need to execute code from the DMA memory, a DMA interface can be implemented by connecting one port of a BRAM to the data bus while the other is exposed to the peripheral. The peripheral now has full master access to the provided memory and can even use the interface in its own clock domain without any restrictions.

This mechanism is adapted to the need of the generated accelerators. Similar to the DMA approach, the accelerators share a larger memory section with the data access port of the SpartanMC core. But instead of an uninitialized DMA memory, the provided BRAMs contain the data sections of the corresponding program,

e.g. `.bss`, `.data`, `.rodata`, and `.stack`. This is accomplished by modifying the linker script that is generated by jConfig when the accelerator stub is enabled. Besides the adaptation of jConfig, the hardware layout of the generated DMA memory requires further adjustments to be usable as regular data memory. The data memory of the SpartanMC soft-core is organized in 9-bit words by using an additional alignment bit that distinguishes between an access to the upper 9 bit or the lower 9 bit. Among others, this allows effective string handling. However, the DMA memory does not provide this feature by default. For this reason, the existing DMA interface must be extended by the additional alignment signal. As a result, the generated hardware can access the memory in one accelerator clock cycle without interfering with the host processor. This setup would also support a real parallel execution as the running polling loop could easily start another software process in the meantime.

As shown in Figure 8.6, the resulting memory architecture now uses one distinct memory block for the program section (`.text`), which is accessed by the instruction fetch only, and another block for the data sections that provide one port for the data port of the processor and the second port for the accelerator. On these grounds, the resulting architecture can be regarded as a Harvard architecture with physically separated instruction and data memory.

In contrast to the *pseudo* Harvard architecture without separated memories[2], the *complete* Harvard architecture has fallen into disrepute due to its inflexibility. Indeed, on a static general-purpose architecture, the size for each memory part is hard to determine. Depending on the current application, there will be enough memory for the program part but insufficient memory for data or the other way around. In contrast, for a dynamic soft-core design that is, in any case, tailored to the application, this drawback must be placed in a relative perspective. The memory of a SpartanMC SoC is quantized by the size of a single BRAM (2048 x 9 bit). If the memory layout does not fit, it can be easily adjusted to the requirements of the application. Thus, a Harvard architecture was considered acceptable for the presented tool-flow.

[2]The classic design of the SpartanMC is a pseudo Harvard architecture (cf. Section 3.2.1).

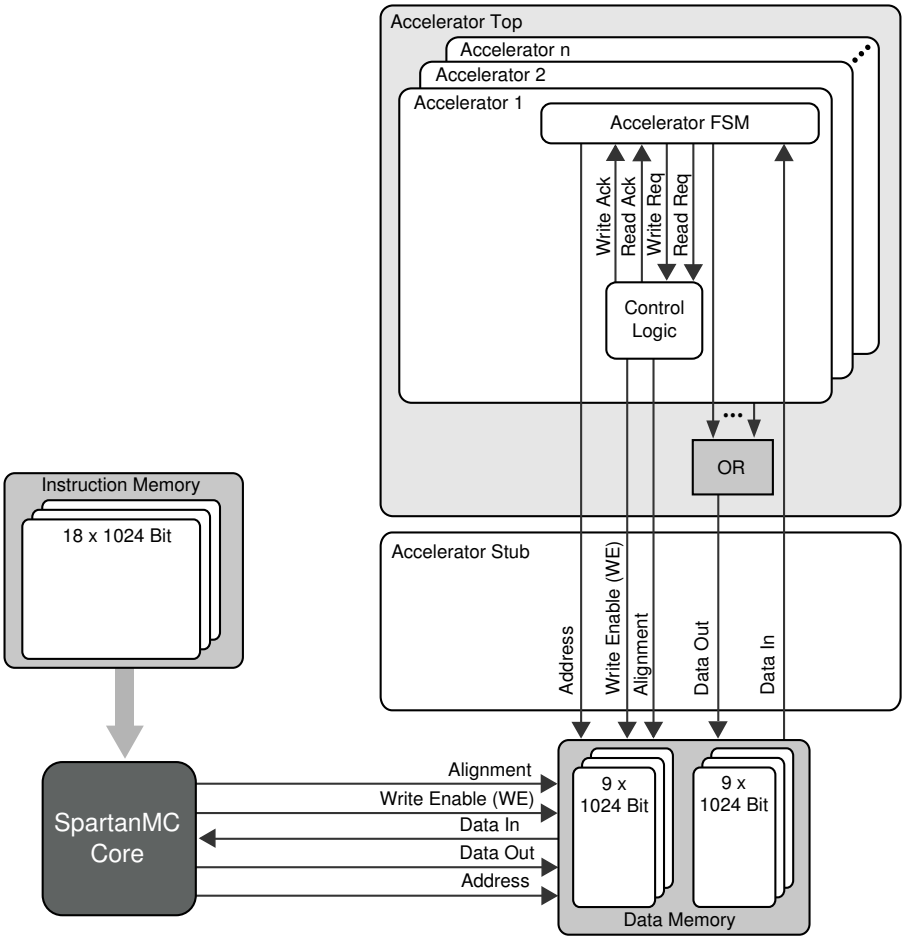

Figure 8.6: SpartanMC memory interface

8.2.2 Alternative Approaches

The current implementation of the memory interface is based on a pure Harvard architecture for the whole SoC. This can be regarded as a serious intervention in the overall architecture, which is probably not portable to other platforms. In the following, two alternative approaches to implement the same functionality on architectures similar to SpartanMC are given:

1. It is not mandatory to implement a *master mode* memory access in hardware. It is also possible for the accelerator to delegates its request to the host processor. The processor handles it in software and provide the requested data afterwards. In particular, for host processors that use an

exclusive interface to the memory, this may be a suitable yet flexible way to handle memory accesses. On the downside, this approach entails some obvious disadvantages. It introduces an additional (probably indeterministic) latency to each memory access. Furthermore, it effectively prevents the parallel execution of other software parts during accelerator execution if the memory access is not triggered via an interrupt.

2. For configurable architectures, it may also be conceivable to provide an extra multiplexer for the existing memory ports. If the accelerator conducts its memory access, the pipeline of the host processor has to stall its execution. On the SpartanMC this mechanism would have the advantage to enable the usage of two memory ports at once – the data port and the instruction port. Nevertheless, this approach requires major modifications on the processor pipeline and could affect the overall clock frequency of the host processor.

8.2.3 ARM Memory Interface

The random memory access from accelerators on the Zynq platform was implemented by following the first approach mentioned above. Although this technique introduces a high latency, it provides a simple and flexible way to perform the required address translation (cf. Section 6.2.1). Consequently, the memory access is delegated to the software that carries out the address resolution and provides the resulting data and signals in parameter registers. As shown in Figure 8.7, the implemented design reuses the hardware layout of the parameter interface described in Section 8.1.2. Nevertheless, the generated accelerator FSM requires a comprehensive interface for the AXI interface IP core (e.g. `r_req`, `r_ack`, `w_req`, `w_ack` etc.) in order to provide an address and data register to the host processor. Furthermore, the generated HDL code for memory accesses must implement an appropriate signal timing in order to use parameter registers from a running FSM. At this point, the design benefits from the synchronous implementation of the AXI interface (cf. Figure 8.5), which still allows the accelerator FSM to run in its own clock domain.

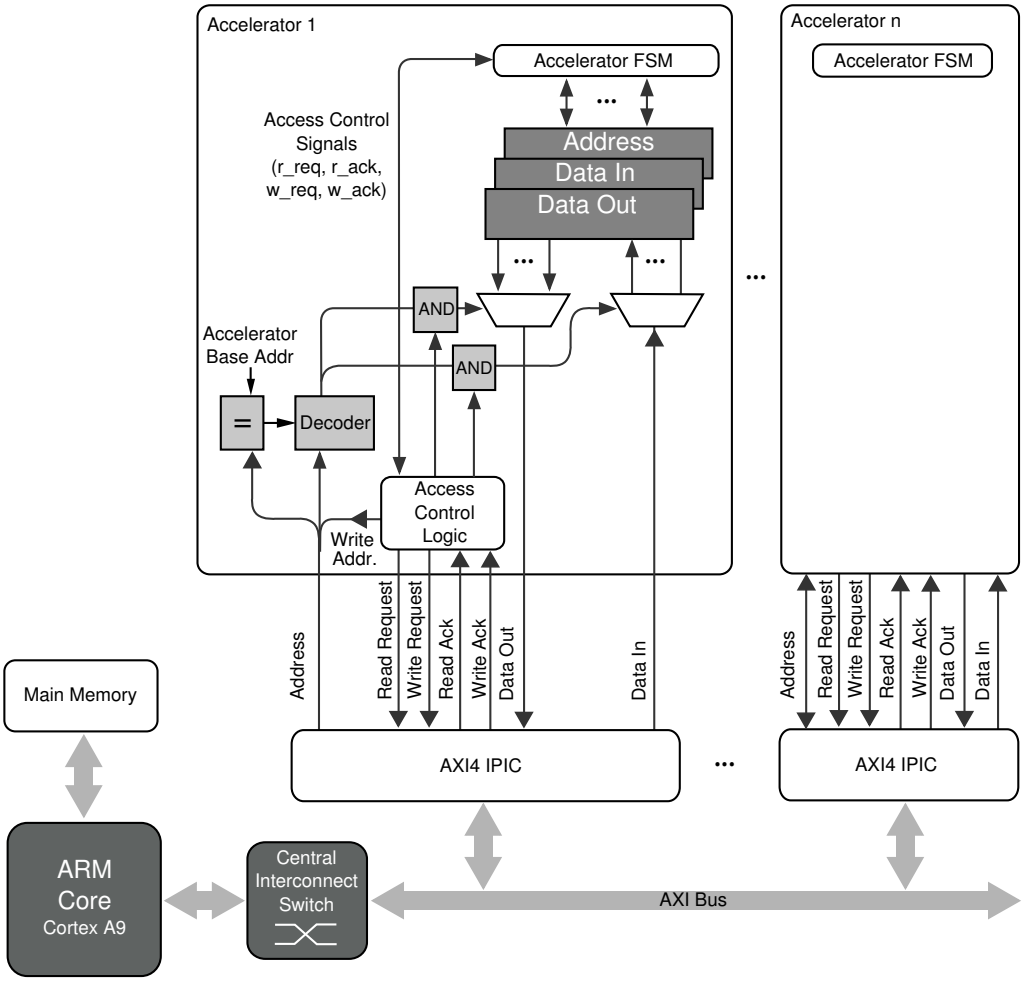

Figure 8.7: ARM memory interface

8.3 FIFO Interface

Though the ARM interface for random memory accesses is functional, it introduces large latencies to the accelerator execution. These latencies are caused, on the one hand, by the overhead that is introduced by the software code running to handle the access and, on the other hand, by the arbitration cycle of the AXI interface. As the ARM processor runs approximately seven times faster than the average accelerator, the latency caused by the software execution is nearly negligible. On these grounds, the major share of the latency goes to the AXI data transfer.

These transactions are carried out with the AXI4 IPIC interface, which is provided

as an AXI interface by XPS. This AXI interface IP requires an arbitration cycle of more than 10 clock cycles for each random memory access. Even worse, the arbitration is carried out with the clock frequency of the AXI interface IP (maximum 200 MHz [120, p. 46 ff.]). At least for the AXI4 IPIC as well as for alternative interfaces, e.g. the high-performance AXI, it must be confirmed that random memory accesses are generally unsuitable for fast data transfers. For this reason, the AXI interface provides the possibility for sequential accesses that introduce the arbitration overhead only once to setup the burst transfer.

The idea of using burst transfers was adopted in hardware by implementing the FIFO interface. Therefore, the AXI interface IP is connected with several FIFO buffers of configurable size. The current implementation uses one FIFO buffer for each identified memory reference[3]. This approach allows the usage of parallel memory access from the accelerator point of view. Nevertheless, the implementation of read memory accesses in the FSM must be aware of empty FIFOs. Thus, the access would block until data in the FIFO is available. Vice versa, a write access must block if the FIFO is full. Therefore, the *request/acknowledge* protocol of the random memory access is reused. But instead of using the synchronization from the AXI interface directly, the FIFO is now in charge of generating these signals. For that reason, the FIFO could suspend the AXI interface IP or the FSM from writing the FIFO when full or from reading the FIFO when empty. The modified interface design is shown in Figure 8.8.

The utilization of the FIFO interface by the software was described in Section 6.2.2. In order to enable burst transfers from the software, the presented accelerator function uses the driver interface for DMA transfers to the accelerator memory. The hardware module used for that purpose is the build-in DMA Controller (PL330) [90] from *PrimeCell*. The PL330 is connected with the AXI interface IP in order to access the FIFO registers. Although, the used AXI4 IPIC is only a slave interface, it is capable of handling burst transfers from another bus master.

During the development of the interface it was also considered to replace the AXI4 IPIC module by a DMA-capable AXI interface like AXI CDMA [127] or AXI DataMover [128].

[3] Assuming the memory references are not in a conditional path and do not alias with other references.

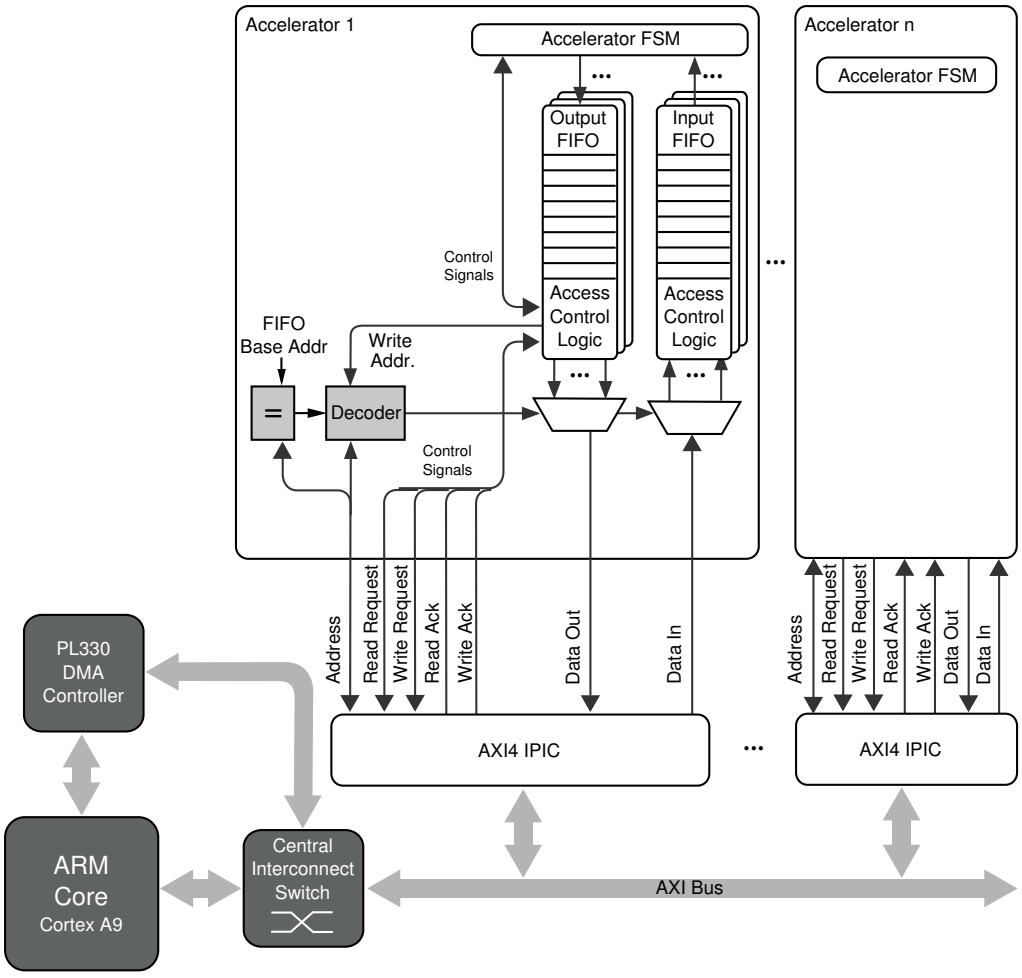

Figure 8.8: ARM FIFO interface

However, these modules are designed for direct memory transfer between AXI devices by using physical addresses. As we use a software-backed address translation the PL330 was finally considered to be the best choice.

8.4 Accelerator Address Map

Accelerators require a generic address map in order to provide a foreseeable interface for the generated software function. Nevertheless, the provided address mapping differs slightly between the described interface configurations. A comprehensive summary of the provided address layout for the possible interfaces is shown in Figure 8.9. The shown addresses start from `0x0000`. Thus, they describe the relative offset for both target platforms.

The simplest layout is used for SpartanMC (Figure 8.9 *(A)*). It basically provides only control registers and parameter registers. The first register (`control`) contains two flags to start (`st`) or to reset (`rst`) the accelerator. The `rst` bit can be used to set the FSM registers as well as the interface registers (e.g. parameters and FIFOs) to zero. Furthermore, it resets the state of the accelerator FSM to `IDLE`. The `st` bit is used to start a new accelerator run while the `rdy` bit in the second register indicates the end of the accelerator execution. For compatibility reasons between the code generation mechanisms for both address layouts, the generated parameter register starts at the offset `0x0010`. The order of the following parameters is derived from the list of input and output variables, which is also used for the C code generation.

The second layout (Figure 8.9 *(B)*) is currently used for the ARM architecture. It should be noted that the FIFO registers and the random access interface are optional. Nonetheless, one of both options must be implemented in order to enable memory accesses. The FIFOs are provided as contiguous memory blocks, with the first one starting at `0x0400`. The following FIFOs are integrated into the address space with a distance of 1024 addresses. The actual size of a FIFO is determined during compile time by a parameter. At the end of each FIFO address space, a control register is generated that provides the current FIFO filling level.

The random access interface is integrated with static registers at the beginning of the address space. It contains registers for the data input/output `data_in` and `data_out`, a write-enable register (`we`), and the address register (`addr`). To avoid time-consuming register transfers, the address register has a dual function. If set to a distinct value, it is interpreted as access request *and* as an address. Hence, the accelerator has to leave it as zero in the remaining execution time.

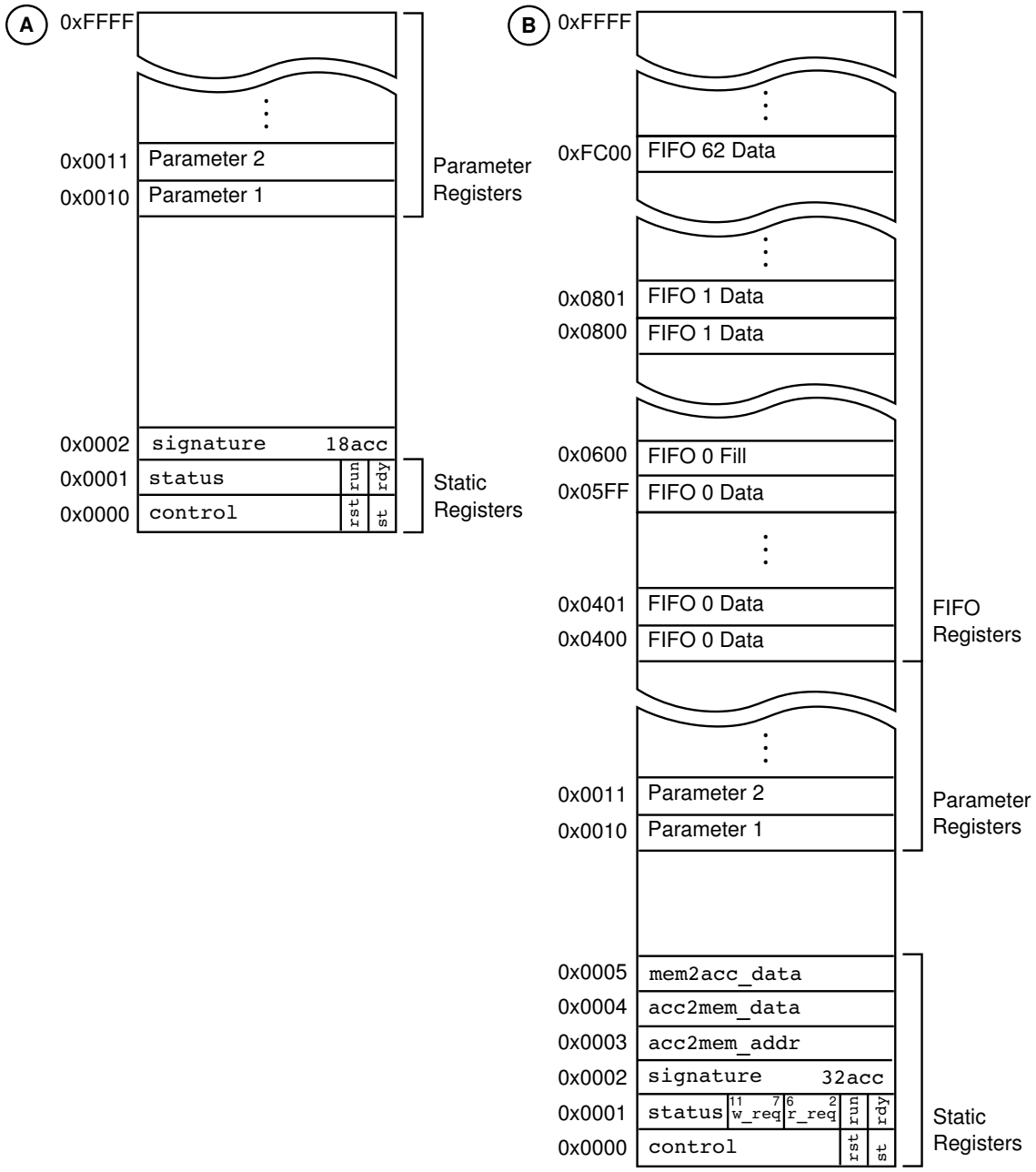

Figure 8.9: Address mapping for SpartanMC *(A)* and ARM *(B)*

Due to the large latencies of the AXI transfers, the value of the address register has enough time to stabilize. The only remaining control flag (`we`) is used to indicate a read or write access.

Part IV

Evaluation

9 EVALUATION

This chapter gives an impression of how the current tool-flow works and what results can be expected. Two sample applications with different data access models will be evaluated here. The examples were adapted to both platforms in order to point out the influence of the chosen target platform on the overall performance of the hardware extension.

In addition, a set of benchmarks is used to examine different optimization strategies. In this context, the evaluation focuses on different characteristics of the generated system. The most obvious one is the impact of the hardware accelerators on the runtime of the entire benchmark. Furthermore, the capabilities of the application analysis are evaluated, and examinations on the size and structure of the generated state machines are presented.

9.1 Implementation Example

To show the capabilities of the implemented plugin, two sample loops incorporating different memory access models were presented: First, a numerical algorithm that provides a fully predictable memory access is shown. Second, a loop that traverses linked data structures, which typically requires random memory accesses is presented.

Both examples should be simple enough to illustrate the hardware generation process. They are not intended to show quantitative results in terms of possible speedups or complex hardware structures.

9.1.1 Fletcher Checksum

The first example is based on Fletcher's numerical checksum algorithm [100]. This algorithm is one of the alternative checksums for the Transmission Control Protocol/Internet Protocol (TCP/IP) and is further used for link-state-routing protocols such as *Open Shortest Path First* (OSPF). The Fletcher algorithm provides a position-dependent checksum for error correction. It is intended for embedded systems, as it provides a comparable quality to the well-known cyclic

redundancy check (CRC) but with lower computational effort. In this work, the original *Fletcher-32* algorithm was modified slightly to process fixed data chunks of 32 bit. The main loop of the algorithm is shown in Listing 9.1.

```
uint32_t fletcher32( uint8_t *data) {
    uint32_t sum1 = 0xffff, sum2 = 0xffff, idx=0;
    size_t tlen = CHUNK;
    size_t len = LEN;

    while (len) {
        tlen = CHUNK;
        len -= CHUNK;
        do {
                sum2 += sum1 += data[idx];
                idx++;
        } while (--tlen);
        sum1 = (sum1 & 0xffff) + (sum1 >> 16);
        sum2 = (sum2 & 0xffff) + (sum2 >> 16);
    }
    sum1 = (sum1 & 0xffff) + (sum1 >> 16);
    sum2 = (sum2 & 0xffff) + (sum2 >> 16);
    return sum2 << 16 | sum1;
}
```

Figure 9.1: Fletcher-32 algorithm

The algorithm consists of two nested loops that were separately evaluated as candidates for hardware acceleration. Note that the evaluation of the outermost loop also involves all GIMPLE nodes of the inner loop. This allows a speedup estimation for all nesting levels of the given loop nest in order to choose the best candidate (nesting level) for hardware generation. Though the outermost loop is typically the best choice for hardware generation, the process also allows excluding outer loops if they contain unsuitable GIMPLE statements.

The resulting DFG and the state machine of the generated hardware accelerator are shown in Figure 9.2. The DFG contains several unnecessary assign statements derived from the original GIMPLE code and some consecutive arithmetic operations. Such statements are perfectly suited for chaining and can be composed to *supernodes*. The found *supernodes* are illustrated with dashed lines in Figure 9.2.

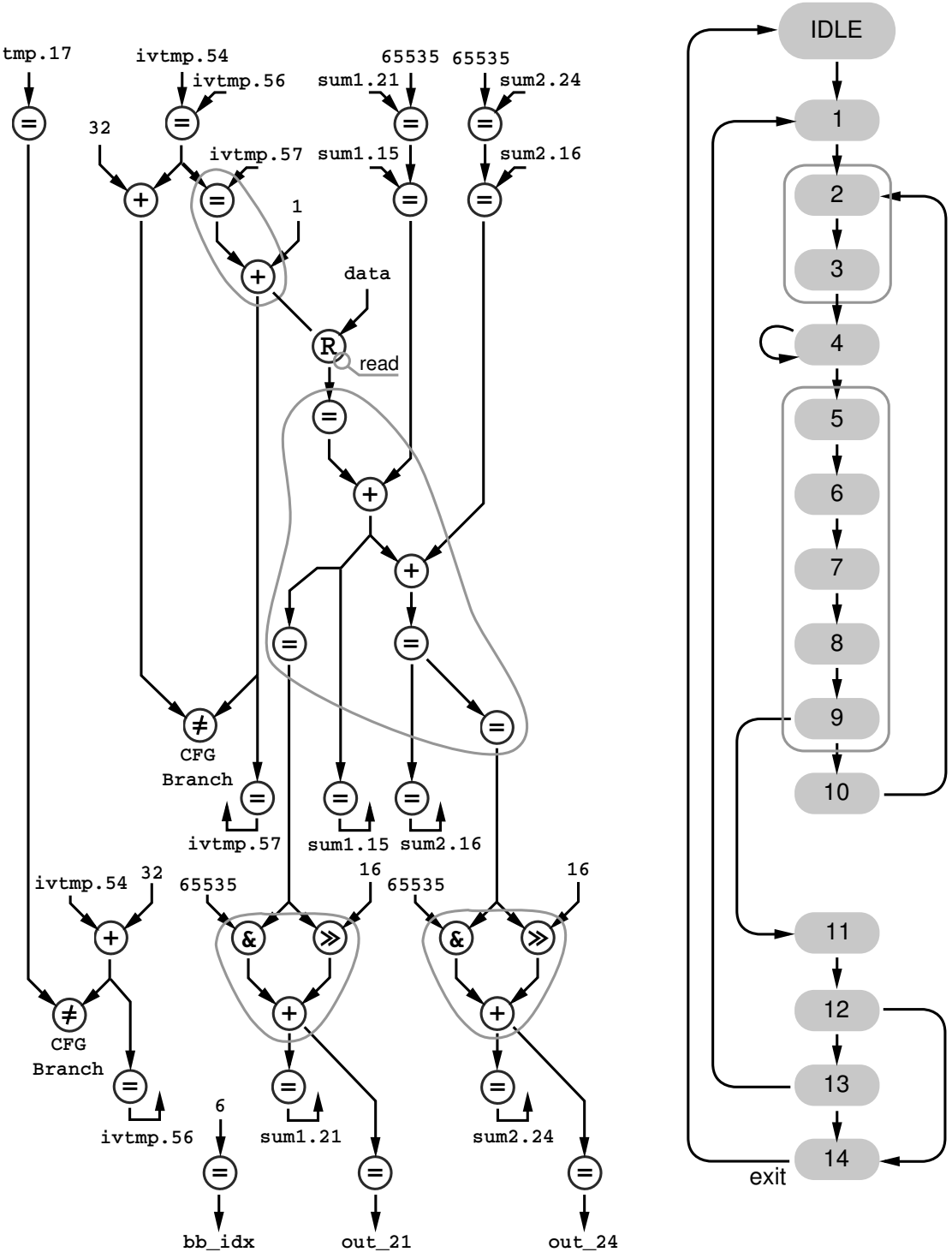

Figure 9.2: Datapath and FSM of the Fletcher-32 algorithm

Note that the applied optimizations do not provide the optimal scheduling for the memory access node in State 4. This is due to an implicit dependency between the address calculation in State 3 and the respective memory access in State 4. Though this dependency is obsolete when using FIFOs, it still exists in the domination tree of the given DFG and will be considered during the scheduling. Later improvements of the optimization algorithms should address this issue.

Due to the fixed chunk size of the given Fletcher-32 implementation, the loop boundaries for both loops can be determined at compile time. This allows successful analysis of the memory-read operation within the inner loop. When using ARM as the target architecture, the resulting hardware accelerator contains one input FIFO providing an accurate data sequence for the given loop nest. Listing 9.3 shows the code of the generated fill-loop[1] of the accelerator function in case of a chunk size (CHUNK) of 32 and a length (LEN) of 128.

```
...
cpy_cnt = 0;
for(i_0=0; i_0<=4; i_0++) {
    for(i_1=0; i_1<=31; i_1++) {
       ivtmp_buf[cpy_cnt++] = ((char*)ivtmp)[i_0*32 + i_1];
       ...
    }
    ...
}
```

Figure 9.3: The generated fill-loop within the accelerator function

Running this example on the Zynq platform with a FIFO size of 128x32 produces only one hardware stall in order to receive the first data chunk at the accelerator startup. The following data accesses are provided without further latencies.

9.1.2 Binary Tree Search

The loop of this example was taken from an own implementation of a binary tree search algorithm. The algorithmic task is to search for a specific key in an ordered binary tree. The code of the loop is given in Listing 9.4. The tree elements are structures containing the key and two pointers referencing the left

[1] The generated names and loop boundaries are simplified for the sake of clarity.

or the right sub-tree, respectively. The given function traverses the tree starting at a given root node. As the function uses a tail-recursive search method, the source code does not show any potential loop candidate. Nevertheless, the actual loop candidate of this function is generated by the GCC loop optimizers during the compilation process (requires at least `-O2` as optimization level). Consequently, the recursive function call, which would hinder accelerator generation, is transformed into a non-recursive loop that is selected for hardware generation afterwards. The resulting GIMPLE structure of the loop candidate is shown in Figure 9.5. The call of the accelerator function is inserted within the dashed basic block.

```
struct tree_node {
    unsigned int key;
    struct tree_node *left;
    struct tree_node *right;
};
typedef struct tree_node node;

node* searchtree(int key, node *root) {
    node *current_node = root;
    if (current_node == NULL) return NULL;
    if (current_node->key == key) {
        return current_node;
    }
    else if (current_node->key > key) {
        searchtree(key, current_node->left);
    }
    else {
        searchtree(key, current_node->right);
    }
}
```

Figure 9.4: Binary tree search algorithm

From the compiler's point of view, the memory accesses of this example are completely unpredictable, as they depend on the given input tree. Consequently, the scalar evolution framework of the GCC results in an unknown *chrec* for all three memory references. This implies that all memory accesses must be carried out *just-in-time* during accelerator execution, which introduces severe slow-downs on the ARM architecture. Even though the given code could benefit

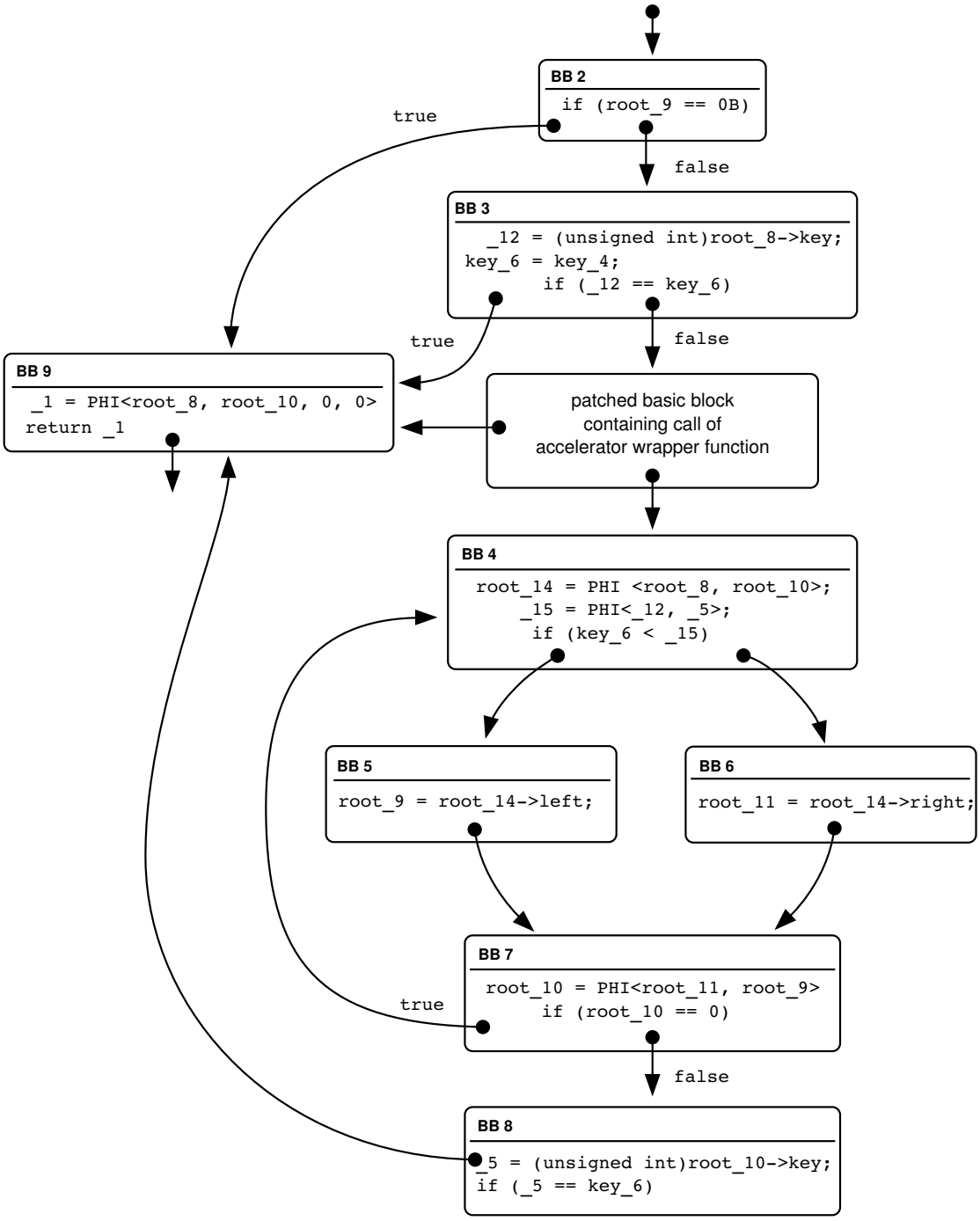

Figure 9.5: GIMPLE graph for the binary tree search

from an arbitrary hardware cache between accelerator and main memory, this feature is not implemented yet. The resulting DFG and the generated FSM of the accelerator are presented in Figure 9.6.

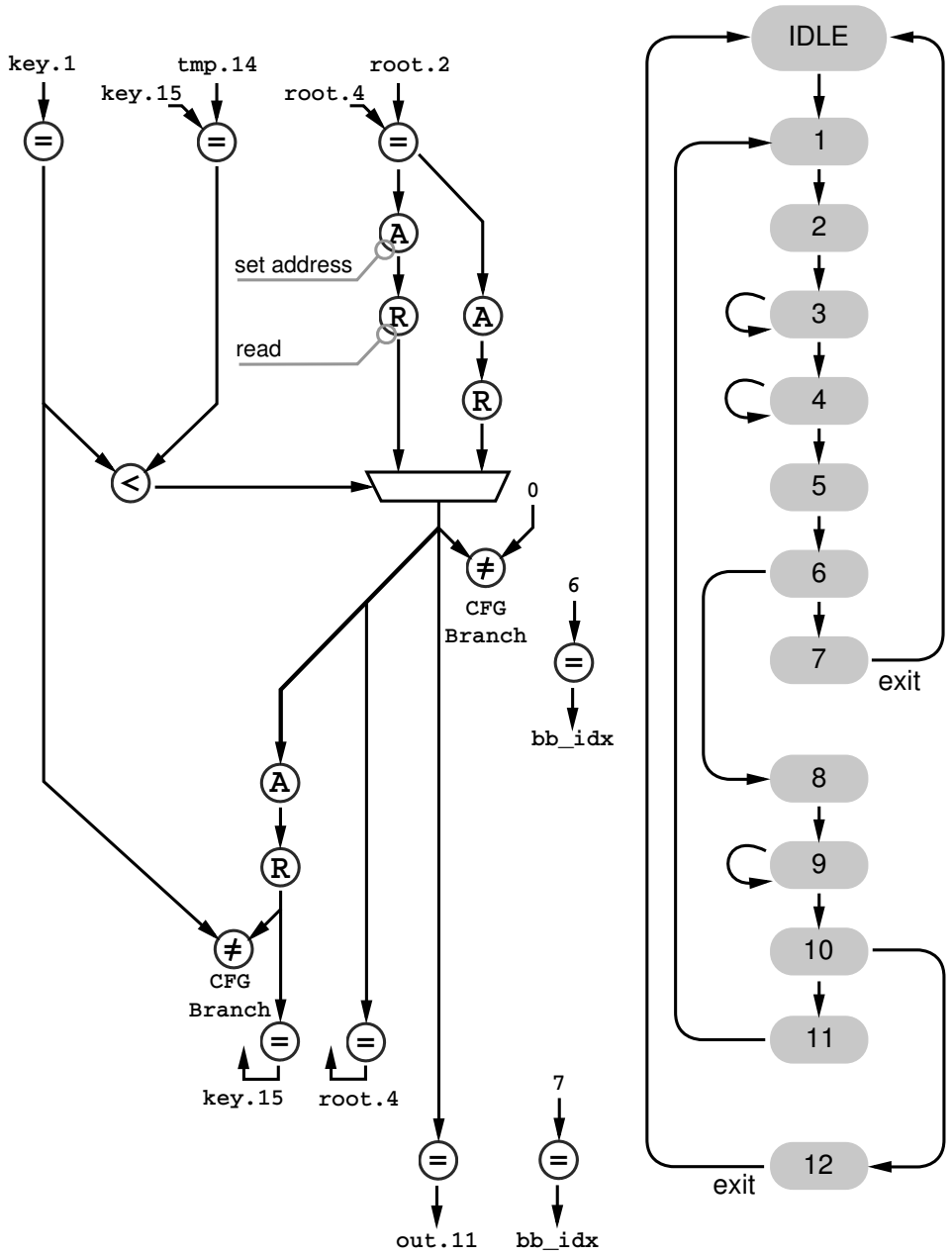

Figure 9.6: Datapath and FSM of the binary tree search

The code provides a speculated execution path for evaluation of the left and right sub-tree. Furthermore, the generated DFG also illustrates some limitations of the current HLS optimizations. For instance, the chaining of the condition check in State 5 with the memory read operation in State 4 is not carried out. This is caused by the conservative implementation of the chaining heuristic, which

reserves a whole clock period for each memory access.

This example was used to evaluate the impact of random memory access on the execution time of the accelerator. Therefore, a single memory access was evaluated with Chipscope[2] for the ARM architecture. The results show that each random access in the current implementation consumes approximately 15 accelerator cycles, which is more than the total number of state machine cycles. Hence, the given example encompassing 10 FSM states spends more than 80% of its execution time on the three memory accesses. On these grounds, hardware accelerators with random memory access for the ARM platform, which lack a prefetching or caching mechanism are useless in most cases.

In contrast to these results, the evaluation on the SpartanMC soft-core provides a different picture. Even such a small accelerator without parallelizable arithmetic operations produces a slight speedup on this architecture due to fast and direct memory accesses and a balanced clock frequency for both system parts.

Nevertheless, this example shows that PIRANHA can still handle such arbitrary memory accesses on C code level successfully.

9.2 Evaluation of Memory Access Analysis

The analysis of memory accesses is essential for the automatic generation of prefetching mechanisms. In this context, the presented implementation of FIFO buffers may be regarded as a first step toward the generation of a full-featured streaming interface.

Nonetheless, the alias analysis requires several preconditions to produce valid results, which raises doubts on its usability for arbitrary C code. For this reason, the loops within the benchmarks from Section 9.3 were evaluated. The set of benchmarks contain 57 out of 162 loops that provide suitable code for hardware generation. Loops without memory accesses or loops that are not fit for hardware generation were excluded from the evaluation. Not all remaining loops are predicted to produce speedups. However, they were still evaluated in context of the memory analysis.

Figure 9.7 shows the ratio of *fully* analyzable and *partially* analyzable loops for

[2] Chipscope is a configurable logic analyzer IP for FPGAs.

each benchmark. Loops that contain at least one undefined access are not suitable for the prefetching mechanism. Only fully analyzable loops with a completely known access pattern for each memory reference are usable for the presented FIFO interface.

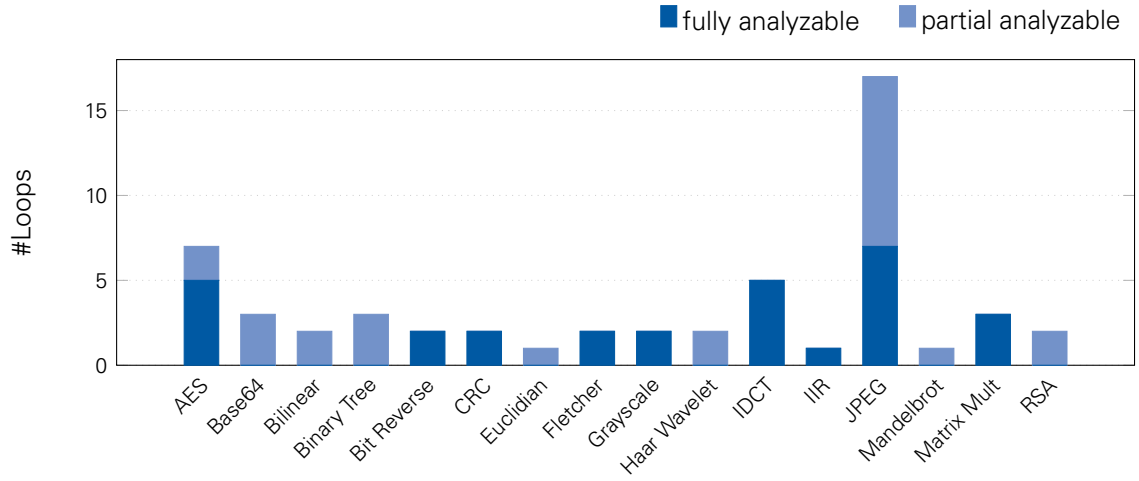

Figure 9.7: Analyzable loops and partial analyzable loops

The yield of analyzable loops suggests that approximately 53% of the given loop candidates were fully analyzable. The reasons for rejecting specific loops can be summarized as follows[3]:

Undefined Number of Iterations Several algorithms work on data fields of variable length, e.g. on a file input. Naturally, the processing loops of such algorithms do not provide fixed boundaries. This prevents a proper alias analysis as the overlapping of different accesses cannot be ruled out.

Unknown Access Subscript Unknown subscripts occur if scalar evolution cannot determine the access pattern from a given subscript. Typically, this occurs for arbitrary pointers, for complex (non-constant) strides or for subscripts that do not directly depend on the loop index variable (cf. examples in Listing 9.8 `unknown_{1,2,3}`).

Undefined Base Addresses In some cases, it is not possible to determine the origin of the base address. This occurs if the actual base address is calculated within the loop using multiple input parameters, as well as if the access does not match any input parameter of the loop.

```
void foo(const unsigned char* idx, char* buf1, char* buf2) {
    char j, k, unknown_1, unknown_2, unknown_3;
    for(i = 0; i < MAX; i++) {
        j = *idx++
        k = buf2[i];
        unknown_1 = buf1[j];
        unknown_3 = buf1[k];
        unknown_2 = buf1[(i & 0x3F)<<3];
    }
}
```

Figure 9.8: Not analyzable array subscripts (`unknown_[1,2,3]`)

An analysis of the given benchmark set implies that 77% of all unresolved accesses are caused (among others) by an unknown iteration count. For this reason, the plugin was extended by an additional runtime parameter (`assume-upper-bound`). This parameter allows the specification of a notional upper bound for uncountable loops and can be used to generate alias sets for such loops. For instance, setting this parameter to "100" means that an interference between two symbols is detected if the distance of both base addresses is less than 100 times of the stride length.

9.2.1 Runtime Costs of ISL Queries

As mentioned in Section 6.3.1, it is necessary to evaluate alias sets at program runtime right before the accelerator code starts. For simple alias sets, this is carried out by conditions in the generated file (`*_memref_info.c`). The more complex sets, which comprise several variables and conditions, could be evaluated with a runtime version of LibISL. Hence, LibISL must be regarded as a performance critical piece of code. In order to evaluate the impact of an ISL call, five problem sets have been defined. These represent the typical results of an alias analysis. According to the nomenclature given in Section 5.6 the variables α_i and β_i describe the access subscript while d specifies the distance between the base addresses of two accesses. The first group of access patterns (P_{hole}) is generated from interleaved accesses, which is typical for arrays of structures. The second group ($P_{contiguous}$) is derived from the first group of access patterns

[3] Note that a loop could be rejected due to several reasons simultaneously.

but was modified to form contiguous accesses.

$$
\begin{aligned}
P_{\text{hole1}} &= \{\ (\alpha_1, \beta_1, d) \in \mathbb{Z}^3 \mid \\
&\qquad 0 \leq \alpha_1 \leq 99 \wedge 0 \leq \beta_1 \leq 99 \wedge 4\alpha_1 - 4\beta_1 + d = 0\ \} \\
P_{\text{hole2}} &= \{\ (\alpha_1, \alpha_2, \beta_1, d) \in \mathbb{Z}^4 \mid \\
&\qquad 0 \leq \alpha_i \leq 3 \wedge 0 \leq \beta_i \leq 3 \wedge 16\alpha_1 + 4\alpha_2 - 4\beta_1 + d = 0\ \} \\
P_{\text{hole3}} &= \{\ (\alpha_1, \alpha_2, \beta_1, \beta_2, d) \in \mathbb{Z}^5 \mid \\
&\qquad 0 \leq \alpha_i \leq 3 \wedge 0 \leq \beta_i \leq 3 \wedge 16\alpha_1 + 4\alpha_2 - 16\beta_1 - 4\beta_2 + d = 0\ \} \\
P_{\text{contiguous1}} &= \{\ (\alpha_1, \beta_1, d) \in \mathbb{Z}^3 \mid \\
&\qquad 0 \leq \alpha_1 \leq 99 \wedge 0 \leq \beta_1 \leq 99 \wedge \alpha_1 - \beta_1 + d = 0\ \} \\
P_{\text{contiguous3}} &= \{\ (\alpha_1, \alpha_2, \alpha_3, \beta_1, \beta_2, \beta_3, d) \in \mathbb{Z}^7 \mid \\
&\qquad 0 \leq \alpha_i \leq 3 \wedge 0 \leq \beta_i \leq 3 \wedge \\
&\qquad 16\alpha_1 + 4\alpha_2 + \alpha_3 - 16\beta_1 - 4\beta_2 - \beta_3 + d = 0\ \}.
\end{aligned}
$$

As shown above, each problem set consists of an equality and several static conditions specifying an interval for α_i or β_i respectively. The performance of LibISL for testing a given d is evaluated in two different ways:

- The first approach (*preconstrained*) initializes the ISL-solver with all preconditions denoting a static interval. Afterwards, the ISL-solver evaluates the equality of the given problem set for a constant d. The test is executed several times using a measurement loop.
- The second approach (*intersect*), uses the LibISL to eliminate all variables but d (cf. *projection*, as described in Section 5.6.1). Later, the measurement loop uses the projected set. The ISL-solver tests the *intersection* of this set with a constant set containing only d. If the resulting set is not empty, aliasing occurs.

The tests were carried out on the Zynq-7000 SoC using *libisl-0.15*, which was compiled with the default configuration. The benchmark code was compiled with `-O2` and the measurement loop was iterated 10^5 times. The results of the performance measurements are shown in Table 9.1. As the values suggest, in most cases, the *intersect* method seems slightly faster for alias set evaluation than

Table 9.1: Benchmark results for using LibISL for alias tests. [135]

Scenario	**Test Method**	**Cold Time** [µs]	**Warm Time** [µs]
hole1	preconstrained	608	135.638
hole2	preconstrained	1105	357.441
hole3	preconstrained	1253	455.936
contiguous1	preconstrained	526	51.641
contiguous3	preconstrained	110	84.193
hole1	intersect	589	195.065
hole2	intersect	460	201.132
hole3	intersect	755	326.896
contiguous1	intersect	221	54.832
contiguous3	intersect	202	55.754

the *preconstrained* method. For that reason, *intersect* was used for runtime evaluation. Nevertheless, the results of the performance measurement were not promising. The evaluation of a single alias set can cost more than 1 ms on a cold processor cache[4]. Even the results on a warm processor cache cannot compete with a typical accelerator runtime. The required computation time starts at ≈ 50 µs for rather simple problem sets describing contiguous accesses and ends at up at over ≈ 450 µs for complex problem sets. Note that this corresponds to 45000 cycles at an accelerator speed of 100 MHz, which could easily exceed the total runtime of an accelerator.

In conclusion, the raw performance of LibISL must be considered to be unsuitable for running right before accelerator execution. Nevertheless, the option to use LibISL at runtime is still available in the code of the plugin.

It should be noted that none of the evaluated examples contain such complex access patterns that a call of the runtime libISL was required. Thus, the much faster evaluation of runtime C conditions (mostly significantly less than 1000 clock cycles) turned out to be sufficient for all available benchmarks.

For target platforms that do not provide a runtime version of LibISL, the usage of this library can be turned off by using a compiler parameter (cf. Appendix C.2). In this case, the evaluation always fails if the given alias sets require complex

[4]The result of the first run have been taken as *cold time*.

conditions.

Furthermore, the strategy discussed in Section 6.3, using a hash table for solved alias sets in order to reduce the cases where LibISL needs to be run, is integrated in the current implementaion.

9.3 Benchmarks

To evaluate the performance of the presented HLS approach, a set of compute-intensive examples from typical application domains for embedded systems was chosen. The benchmarks were adapted to both target platforms without major changes. Nevertheless, the uncommon data width of 18 bit for the SpartanMC sometimes required a slightly different implementation that introduced additional arithmetic operations. Consequently, the HLS also provides different data-paths for both platforms. Moreover, the data width hinders a trivial adaptation of commonly used benchmark suites. For that reason, most of the benchmarks are derived from algorithmic descriptions or reference implementations.

Several benchmarks contain only one compute-intensive loop nest that is identified for acceleration. These benchmarks are marked with in the following list. The benchmarks marked with can be regarded as complete applications containing multiple loops for acceleration.

The following list gives a brief overview of the implemented applications:

- **AES:** An 8-bit AES implementation containing code to encrypt and decrypt a sample message.
- **Base64:** A base 64 encoder and decoder.
- **Bit Reverse:** Swaps the bits of an integer to reversed order.
- **Binary Tree:** Generates a random binary tree and searches for an item in this tree afterwards.
- **Biliniar:** Interpolates a 64x48 sized RGB-picture with a bilinear filter algorithm.
- **CRC:** Performs a cyclic redundancy check using a polynomial of the 16th degree.

- **Euclid:** Implementation of the Euclidean algorithm computing the greatest common divisor of two numbers. Besides the input data, this algorithm contains no further memory accesses.
- **FFT:** Performs a complex radix-2 fast Fourier transformation. The accelerator also handles the reordering of the output array.
- **Fletcher:** Performs the calculation of a Fletcher-32 checksum. The SpartanMC implementation performs, adapted to the 18-bit architecture, a Fletcher-16 calculation.
- **Grayscale:** Performs a grayscale-filter algorithm on a 8x8 sized RGB-picture. Each color is encoded with 4 bits. The size of the input data is adapted to the host architecture.
- **Haar Wavelet:** Performs a 1-D Haar wavelet transformation.
- **IDCT:** Performs an inverse discrete cosine transformation on an 8x8 input buffer.
- **IIR:** Performs a high-pass Butterworth infinite impulse response (IIR) filter of the fifth order.
- **JPEG:** A complete JPEG decoder optimized for architecture with limited bit width.
- **Mandelbrot:** Calculates whether a given complex number is a member of the Mandelbrot set or not. Besides the input data, this algorithm contains no further memory accesses.
- **Matrix Mult:** Performs a matrix multiplication of two 8x8 integer matrices.
- **RSA:** An implementation of the RSA cipher containing code to encrypt and decryt a sample message.

9.4 Influence of Register-Allocation Strategies

The impact of register-allocation strategies was evaluated and presented in [143]. The experiments of this publication were carried out in a very early stage of the plugin. For that reason, the evaluation was limited to the eight applications of the benchmark set of Section 9.3 which were available at that time. Due to the comparable hardware structure of all accelerators, the results should also be applicable for other benchmarks.

The resource utilization of the accelerators was measured by synthesizing the hardware description for different FPGAs. Later, the influence of the register layout on the resource footprint was evaluated based on this data. In addition, the maximum clock frequency for each accelerator was determined in order to identify a potential frequency reduction. A drop in the maximum clock frequency was anticipated for strategies that generate better register utilization. This was expected, as a better utilization of registers also implies a larger number of shared registers, which may result in an extended critical path due to additional multiplexers on the register inputs.

In order to evaluate different register-allocation strategies, the PIRANHA HLS code was equipped with different variations of the left-edge algorithm [102]. After that, the HLS tool could be parametrized to generate hardware designs with different register utilizations. The implemented strategies could be summarized as follows:

- A classic left-edge algorithm (*le_full*) that tries to map as many variables as possible to one register. This strategy probably results in an increasing number of multiplexers and leads to a more complex control logic on the FPGA. Note that the resulting allocation is not a global optimum in terms of utilized registers. However, in the context of this experiment, it provides the minimum number of registers for the evaluated accelerators.
- Four variants of a modified left-edge algorithm that use a slightly different allocation strategy by altering the maximum number of variables per register from two to five (*le_2* – *le_5*).
- An allocation strategy (*le_uid*) that tries to avoid unnecessary variable swaps. This is accomplished by mapping related GIMPLE variables (referencing the

same C variable) together into the same register. Appropriate variables are identified by their reference to an *id* in the GIMPLE structures. This mapping should result in good register utilization, in combination with a low number of multiplexers and control logic.

- A naïve approach *le_simple* that maps each variable to a fresh register. Obviously, this approach neither requires multiplexers nor any additional control logic. Additionally, it provides the maximum scope for optimizations for the following vendor tools.

The evaluation was carried out by applying the designs to a Xilinx Artix-7 FPGA (xc7a200tfbg), which provides the same internal structure as the FPGA fabric (xc7Z020) on the Zynq-7000 device. Furthermore, the optimization strategy for the vendor synthesis tool (Xilinx XST P.49d) was altered between area and frequency optimization. The measured speed is determined by the critical path delay that is given in the post place and route report from Xilinx.

At this point, it must be made clear that this approach may lead to inaccurate results. The Xilinx synthesis tools typically stuck to the given constraints for the current synthesis run. In case of the experiment the synthesis was constrained by an unattainable frequency value. As a result, the post place and route report show a negative slack for the critical path. Unfortunately, the slack and, therefore, also the derived frequency could spread considerably. Consequently, this method is not a good choice to determine an accurate value of the best achievable frequency [5]. However, the evaluation of the results will show, that the impact of the allocation strategy on the frequency is evident even for slightly inaccurate results.

The resource footprint is quantified by the utilization of LUTs, which is also provided by the post place and route report. The resulting measurements of the resource consumption and the clock frequency are illustrated in Figure 9.10. The raw data can be found in Appendix E.1.

First, the results show that the difference between area and speed optimization of the vendor toolchain is negligible for the given accelerator set. But, even more surprising, the register allocation strategies only have a small effect on the overall resource consumption. Nonetheless, there is a tendency for the *le_simple* and

[5] The best practice to perform an accurate measurement of the frequency would be an approach that determines iteratively the highest *working* frequency of the given design.

le_uid algorithms to achieve a slightly lower resource consumption for almost all benchmarks. Keeping in mind that both algorithms are designed to require a minimum number of multiplexers, this goal of optimization should be considered as a guideline for the generation of resource-efficient hardware accelerators.

Besides the resource consumption, the achievable clock frequency was the second subject of the evaluation. As expected, the approaches that use extensive register sharing (*le_full*, *le_1* to *le_5*) suffer from long critical paths that negatively affect the maximum clock frequency. Taking a closer look at the clock frequency measurements, reveals that the results can be divided into two groups of benchmarks: first, a group that only shows a minor impact on the register-allocation strategy and the achievable frequency, and second, a group (FFT, IIR, Grayscale and Matrix Mult.) that seems to be significantly influenced by the chosen allocation strategy. Further evaluations of the generated accelerators suggest that this phenomenon is introduced by the usage of DSP cores in the accelerator datapath (cf. Appendix E.1.1). As DSP cores typically provide internal registers [124], they also add additional constraints to the datapath. In combination with a multiplexer tree, this results in the demonstrated performance degradations.

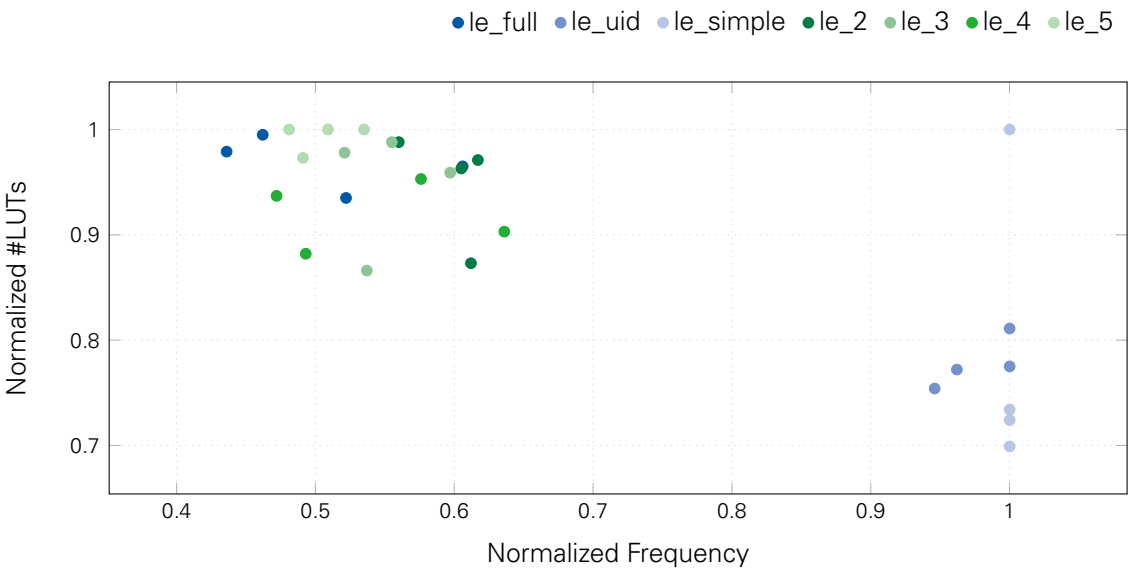

Figure 9.9: Normalized frequency plotted against the respective normalized resource consumption for different register-allocation strategies applied on *FFT, IIR, Grayscale and Matrix Mult.*

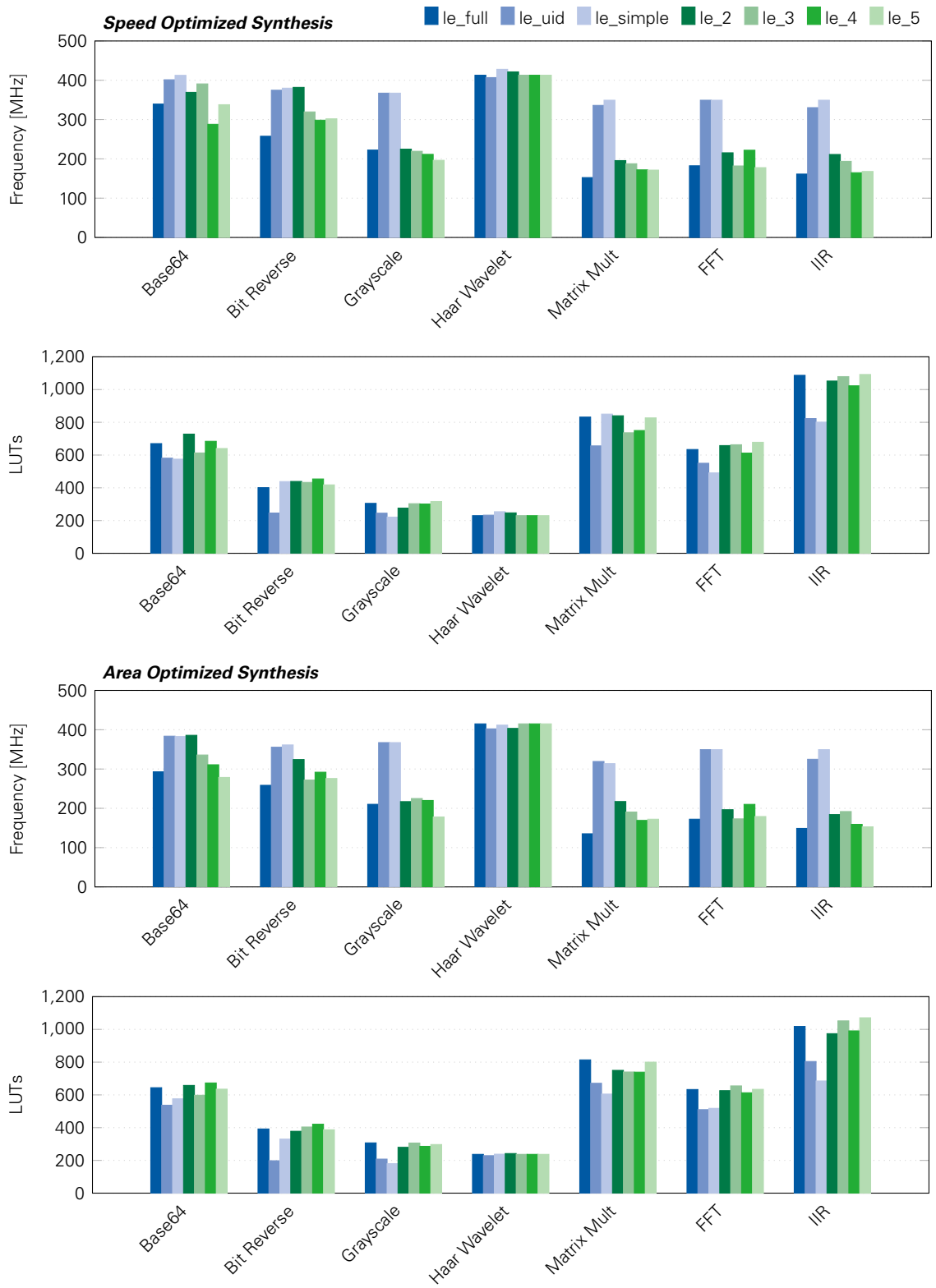

Figure 9.10: Resource and frequency comparison of different register-allocation strategies (LUTs – small values are better; frequency – large values are better)

Figure 9.9 shows the results of all allocation strategies for the second group of benchmarks. The frequency and the utilized LUTs are normalized to the best result of the respective benchmark. This plot shows the predominance of the simple register-allocation strategies *le_simple* and *le_uid*, in terms of resource consumption and achievable clock frequency. Remember that these values suffer from a certain degree of inaccuracy. Nevertheless, a difference of ≈40% (cf. Figure 9.9) between the two groups of allocation strategies imply a clear correlation between the allocation strategy and the achievable clock frequency.

To summarize, the experiments show that the effort of an elaborated HLS register-allocation strategy does not provide any significant benefits to the overall resource footprint or frequency of the accelerators. In some cases, the performance can even degrade significantly. For that reason, it was considered to use the naïve approach (*le_simple*) for register-allocation strategies in this thesis.

9.5 Influence of HLS Optimizations

The evaluation of different optimization strategies was carried out on the set of benchmarks described in the previous section. Therefore, the optimization strategies namely, *speculation* and *chaining* were applied consecutively to each CDFG. After each modification, list scheduling was applied to the resulting graph.

Note that the implemented algorithms for *speculation* and *chaining* could influence the runtime of the resulting state machine negatively even when the number of states decreases. However, the presented evaluation gives a good overall impression of the effectiveness of particular HLS optimizations.

The following Figure 9.11 shows the *improvement* of state machines compared to the *plain* number of states after applying list scheduling only. The resulting state machines for ARM and SpartanMC are identical. Consequently, the FSMs can be used to evaluate the HLS optimizations for both target architectures.

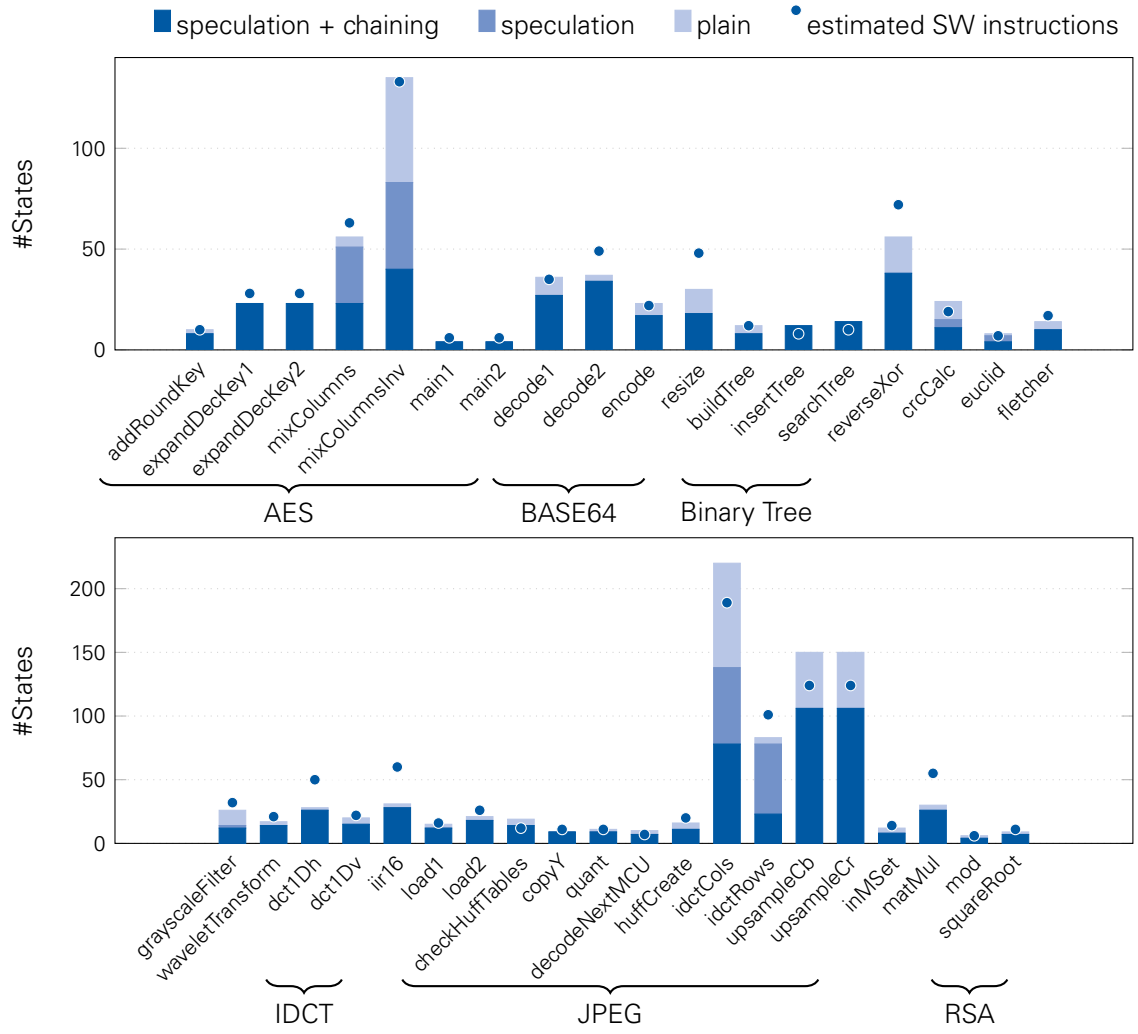

Figure 9.11: Comparison of FSM states after applying different HLS optimzations

First, speculation is applied to the basic CDFG. The speculation algorithm merges conditional basic blocks, if possible. The resulting merged block contains all operations of the conditional execution paths and calculates multiple results in parallel. The final output is selected at the end of the new block in a generated multiplexer node. After using this optimization, the execution time of the new block is dominated by the longest execution path. Assuming an unfortunate control-flow, this could increase the overall execution time, even if the number of states was reduced. The actual impact of speculation on the overall application speedup is presented in the following Section 5.4. Notably, speculation is available only for CDFGs that contain conditional branches; hence, only six benchmarks pro-

duce relevant results for this optimization. This is caused by some restrictive limitations of the speculation algorithm: the rejection of blocks containing memory accesses, loop exit edges, and conditions with embedded condition nodes. Furthermore, only approximately 32% of the identified loop candidates contain conditional branches at all.

Second, chaining was evaluated. This optimization eliminates unnecessary intermediate state changes between operations. Besides reducing the number of states, it lengthens the critical path, which can reduce the overall clock frequency of the accelerator. However, as long as the clock frequency is dominated by another part of the system, e.g. the interface for memory accesses, chaining does not introduce any unfavorable side effects. Figure 9.11 shows the influence of chaining on the number of states for the loop candidates of all benchmarks.

Even if the current chaining algorithm does not use the full potential of the existing CDFG, the resulting state machines can be improved by an average of 26%. Moreover, the results show that chaining is more effective for large state machines with a relatively small number of memory accesses.

Figure 9.11 shows the GCC-internal estimation of required instructions for the respective loop candidates (blue dots). On a RISC architecture, this value should be equivalent to the number of clock cycles required for the loop body. Besides, this value should be also higher than the required states for the *plain* state machine, unfortunately this is not the case. Due to this inaccuracy, the automated estimation of speedups (cf. Section 5.4) can give only a very coarse impression of the expected performance.

Nearly 15% of the found loops shown in Figure 9.11 are smaller than 10 instructions. Typically, these small loops are hard to accelerate as they contain few operations and short datapaths only. Hence, they do not provide enough code for an effective usage of optimization strategies. Such benchmarks could benefit from larger datapaths. This can be achieved by optimizations that generate larger accelerator-kernels such as loop merging or partial loop unrolling.

The raw data of the presented measurements can be found in Appendix E.2.

9.5.1 Performance Measurement

One of the most interesting parts of the evaluation is the measurement of the overall performance of a system. In this thesis, performance is evaluated by comparing the execution time of an application running with a hardware accelerator against the pure software execution time of the same application. Hence, the baseline for the performance measurement is based on the execution time of the benchmark application on the respective host processor.

SpartanMC SoC

The results of the benchmark applications for the SpartanMC SoC are shown in Figure 9.12.

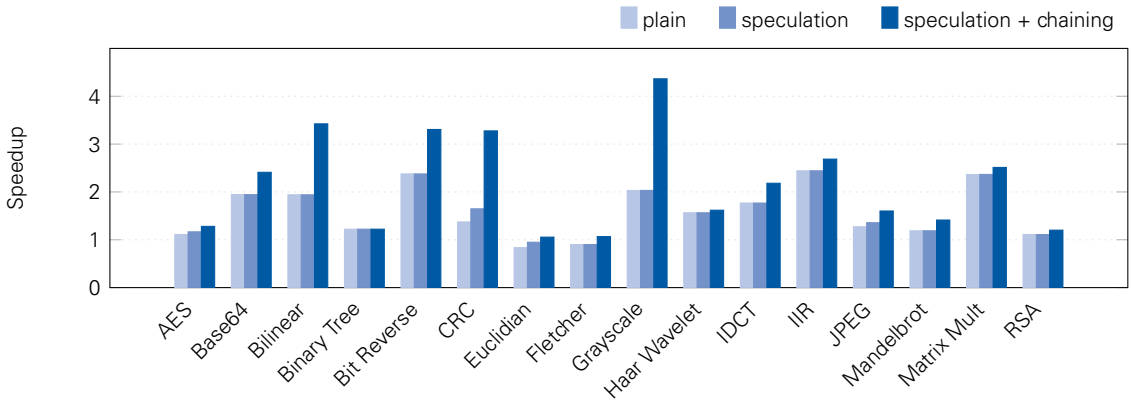

Figure 9.12: Whole application improvements for SpartanMC

According to the results, all SpartanMC benchmarks achieved performance improvements. Thus, the presented plugin has shown a general viability for this target platform. As mentioned in the previous section, programs with rather small accelerator-kernels, e.g. *Fletcher*, *Euclidian* or *RSA* only show marginal speedups. Especially, without optimizations they could also suffer from slight slowdowns. Such slowdowns occur if the actual hardware mapping incidentally results in a one-to-one implementation of the assembler code without any parallel operations. In this case the additional overhead caused be the accelerator call and the static data transfer causes a slowdown.

The gained speedups can be easily improved by increasing the clock frequency

of the generated hardware accelerators. This is possible, because the resulting accelerator datapaths hardly utilize the capacity of the given duty cycle in contrast to the tightly packed pipeline of the SpartanMC core. Furthermore, the used Harvard architecture in combination with the dual-ported BRAMs (cf. Section 8.2.1) provides a separate memory interface for the hardware accelerators that runs in its own clock domain. Both features enable the usage of a higher clock frequency in favor for the accelerator. The results in Figure 9.12 were achieved by running the SpartanMC core and the accelerators on the same frequency of 50 MHz. Exemplary, the results for a higher accelerator frequency of 150 MHz are shown in Appendix E.3 (column 5). Several spot-checks for the Artix FPGA suggest that the generated accelerators could operate at a clock frequency of 150 – 250 MHz without violating any timing constraints.

ARM SoC

Unfortunately, the frequency setup of the Zynq platform was not so advantageous. The Zynq provides a powerful ARM Cortex-A9 processor running ≈7 times faster than the FPGA fabric. Together with the highly optimized ARM GCC back-end it is nearly impossible to achieve speedups with the currently generated accelerators and their software interface. The following results are intended to prove that the mechanisms used for the SpartanMC architecture could be ported to a more complex SoCs architecture without changes of the HLS approach.

Besides the difference of the clock frequencies between programmable fabric and ARM core, the implementation of the software interface introduces additional latencies to the accelerator execution. First, each call of an accelerator triggers an `open(...)` system-call of the respective hardware device in the Linux Kernel. This includes the mapping of accelerator registers into the application address space and the allocation of DMA buffers. At the end of the hardware execution these buffers are freed and the device is closed. Second, the software backed memory transfers during accelerator execution are a source of latencies. These transfers either require DMA operations to fill FIFO buffers or a software delegated transfer for each memory access. Both methods are devastating slow in the current implementation. The average values for the current latencies are shown in Appendix E.5. These latencies, together with the differential of the

clock frequency, lead to severe slowdowns of ARM programs when using accelerators. The resulting slowdowns of the benchmarks are shown in Figure 9.13. As the latencies for initializing the accelerator are in the same order of magnitude as the overall runtime of the accelerator, the penalty for running accelerators in an external loop leads to even worse results. This effect can be observed for the RSA, Fletcher, Euclidean, Mandelbrot, and AES benchmark.

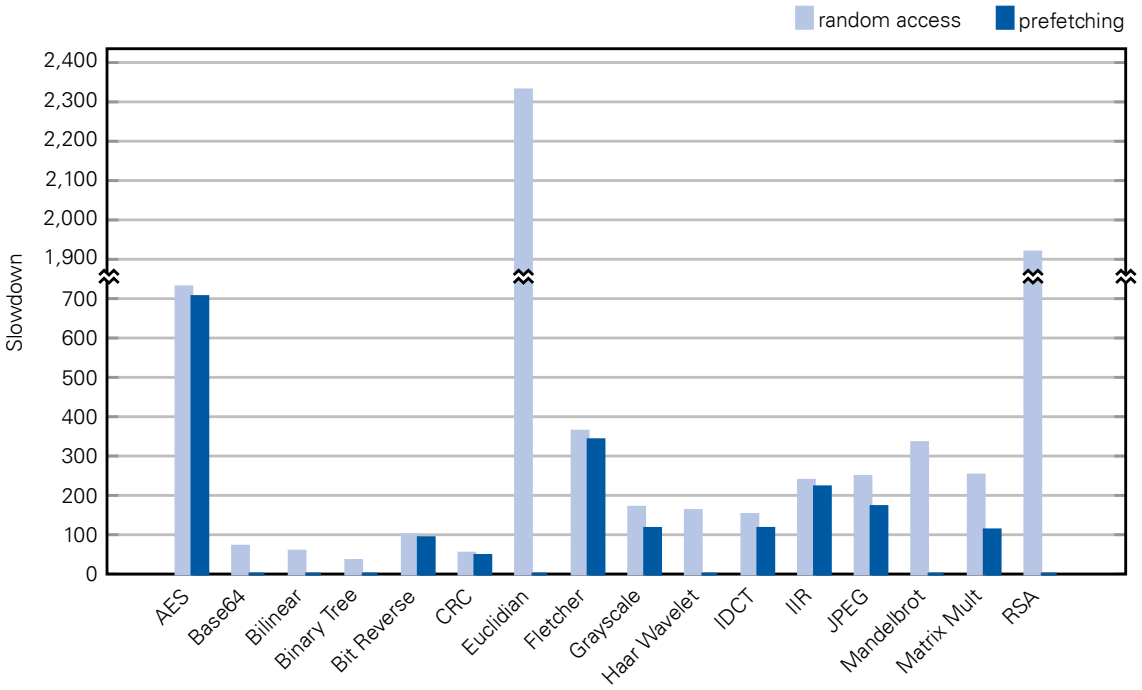

Figure 9.13: Whole application evaluation for the ARM SoC optimized with speculation and chaining (small values are better)

9.5.2 Performance Analysis

In the previous section it was claimed that the performance of accelerators on ARM is dominated by various latencies. In the following section an analytical relationship will be introduced which confirms this statement. Therefore, a runtime-analysis of suitable accelerators is carried out and compared to an analytic estimation. Firstly, the estimation was developed for the SpartanMC architecture. Since the SpartanMC accelerator interface provides hardware-triggered deterministic memory accesses, the chosen analysis approach should be cycle accurate and

easy to prove.

The reference for the evaluation was the number of clock cycles between start and return of the accelerator wrapper-function. Hence, the only software depended latency is introduced by the static offset for transferring initial values to the accelerator. This latency can be easily estimated with the known number of initial input and output variables.

According to section 5.4 the cycle-count of a loop body CP can be defined as

$$CP = T_{body} + N_{Mem} \cdot (T_{Mem} - 1).$$

Where N_{Mem} is the number of memory accesses of the corresponding loop body and T_{Mem} is the cycle-count of a memory access. The cycles of a loop T_{loop} with a fixed number of iterations n can then be defined as $T_{loop} = CP \cdot n$. As accelerator kernels usually consist of nested loops, the individual loop bodies must be estimated and summed separately to guarantee an exact calculation. This results in the following relationship for m nested loops with i and k referring to the current loop beginning with the outermost one:

$$T_{acc} = \sum_{i=1}^{m} [\prod_{k=1}^{i} \cdot n_k] \cdot CP_i \quad \text{with } m, n, i, k \in \mathbb{Z}.$$

Since loops in GIMPLE, and thus also in the accelerator FSMs, spare *one* state in the last iteration, the number of clock cycles T_{acc} must be corrected by the respective state. Therefore, the length of the execution path CP is represented as sum of the loop body CP_{body} and the state CP_{header}, which is spared in the last iteration:

$$CP = CP_{body} + CP_{header}.$$

The total number of cycles of a loop nest is then described by the following equation:

$$T_{acc} = \sum_{i=1}^{m}[\prod_{k=1}^{i} \cdot n_k] \cdot (CP_{body_i} + CP_{header_i}) - \sum_{i=1}^{m}[\prod_{k=1}^{i} \cdot n_{k-1}] \cdot CP_{header_i} \quad \text{with } n_0 = 1.$$

With the constant offset OV from Section 5.4 the number of cycles of an accelerator T_{total} can be regarded as the sum T_{acc} and OV. The constant offset is calculated from the number of input/output variables IO and a constant offset representing the assembler instruction required to transfer a single value. The used values for the SpartanMC were $T_{Mem} = 0$ and $OV = 4 \cdot IO$. Figure 9.14 shows the deviation of the estimated values to the measured runtime. Note that only accelerator kernels with bounded loops, which could be calculated by means of a static analysis, are considered for the evaluation. However, the chosen accelerator kernels may still contain several different execution paths due to conditional branches.

For that reason, Figure 9.14 distinguishes between two types of accelerators: First, accelerators with data dependencies that were shown as blue bars. The interval represents the possible runtimes between minimum and maximum execution time. Second, accelerators with a static execution time that were show as blue dots. According to the analysis results, the maximum error for accelerators with static execution time on the SpartanMC architecture is 2.9%[6]. The remaining error is induced from small deviations in the value for OV representing the static IO operations. These operations are initiated from the software interface, and therefore depend on the specific compilation results of the accelerator wrapper function.

[6]For loops without data dependencies

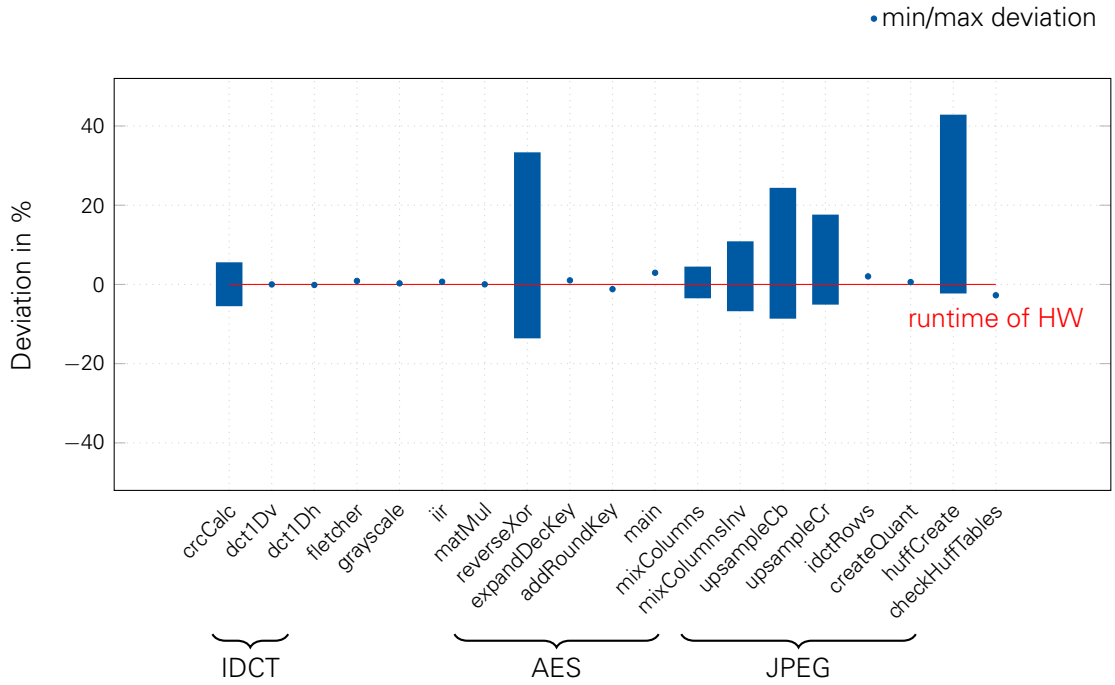

Figure 9.14: Relative error of performance analysis for SpartanMC benchmarks

After the first evaluation for the SpartanMC SoC, the same equations, with an additional scaling factor for the clock difference, were applied for the ARM-SOC. The memory access latency was adjusted according to the measurement of T_{Mem} on the Zynq platform. In addition, *OV* was supplemented by a constant summand T_{call} that represents the latency of the system calls for `open(...)` and `close(...)`. The used values were determined from the average of all kernel calls. Besides these latencies, a discrepancy for each respective first call of an accelerator was observed. These calls were characterized by a higher latency, which is probably caused by a cold cache for the first use of the accelerator device-structures. The effect was compensated by an additional latency for the first accelerator call. The measured values for specific latencies can be looked up in Appendix E.4.

The relative error of the performance analysis for the ARM SoC is shown in Figure 9.15. Since the runtime on the ARM SoC is dominated by the high latencies, the different execution paths are hardly relevant. The maximum error of 11.5% is induced by deviations of the average latencies.

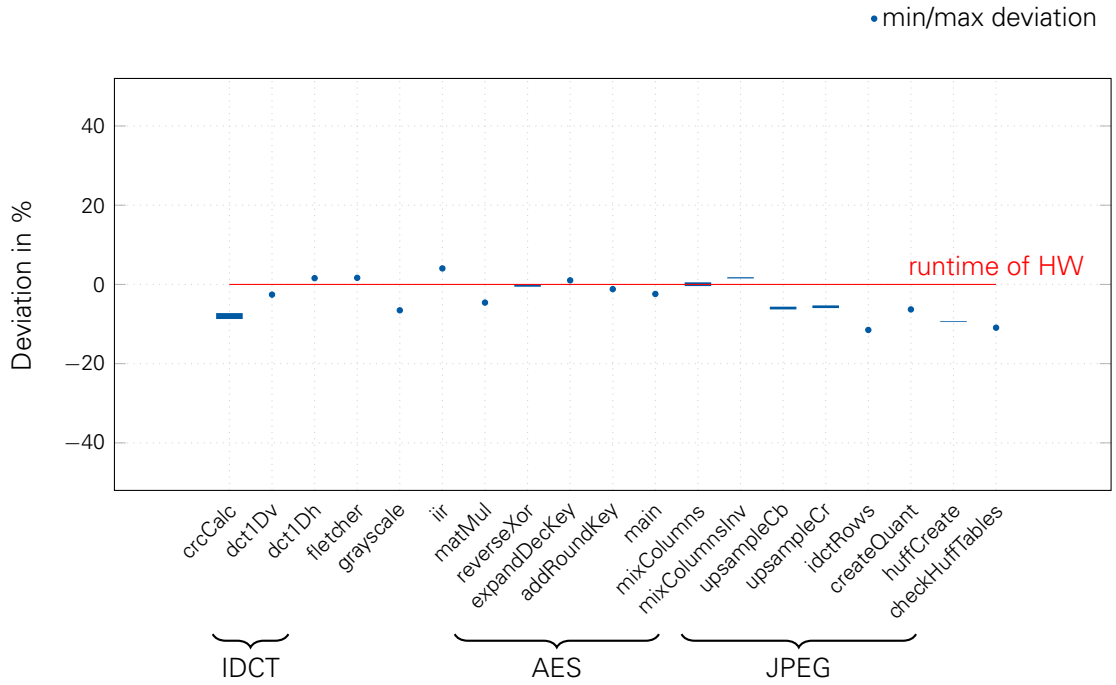

Figure 9.15: Relative error of performance analysis for ARM SoC benchmarks

Since the runtime of the accelerators can be determined with sufficient accuracy by the analysis described above, it is now possible to analyze the runtime of the individual subtasks. Figure 9.16 shows the results of this analysis for the ARM SoC benchmarks. These results imply that the actual execution time of the accelerator hardware plays a subordinate role only. In contrast, the static latency for initializing the IO and DMA driver constitutes the major part of the execution time.

Consequently, the runtime of accelerators could be significantly improved if the static latencies for the `open(...)` and `close(...)` system-call could be integrated into the application startup and termination code. In addition, the DMA controller must be modified accordingly, so that the required channels can be reserved at application startup. These modifications are not included in the current implementation and have to be performed manually.

Besides the elimination of the latencies, future improvements of the plugin should address HLS optimization in order to minimize and/or speed up the generated hardware accelerators. The given results may benefit from partial loop unrolling in combination with modulo scheduling, but the reduction of unnecessary memory accesses could also be a future goal for improvements. For instance, the

identification of constant arrays and the direct mapping of this data into hardware structures could improve several real-world applications.

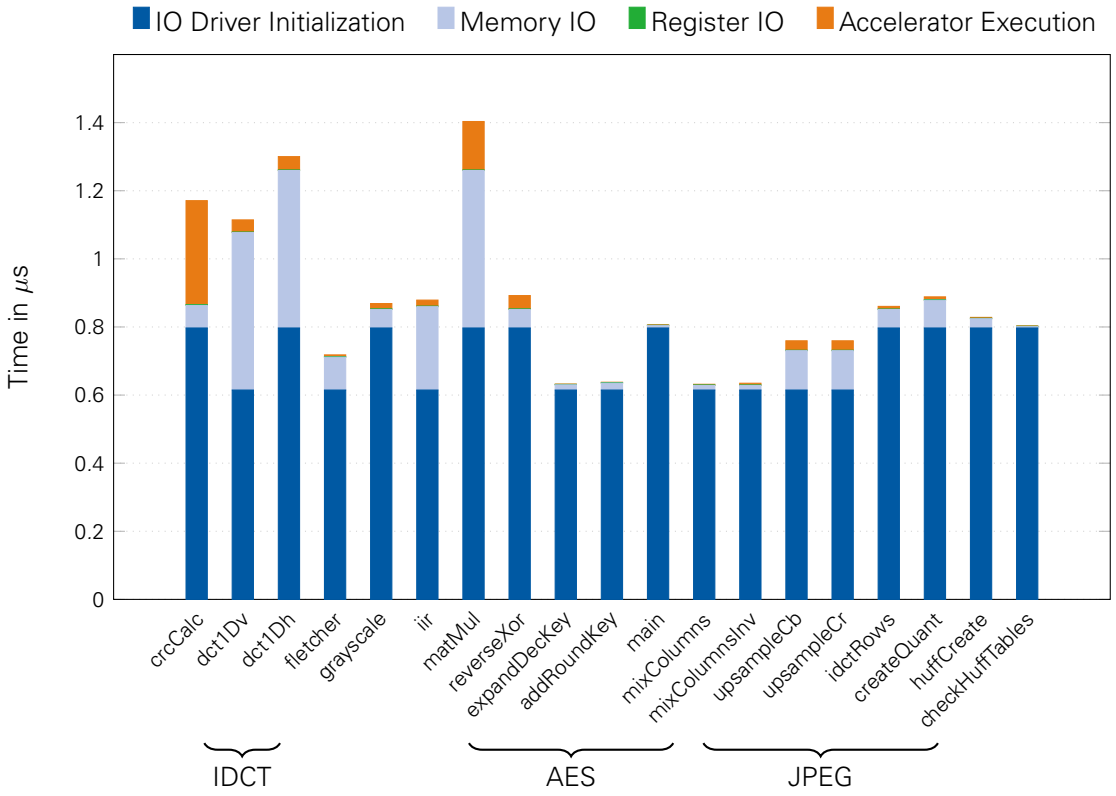

Figure 9.16: Analysis of subtask execution time for ARM SoC benchmarks

10 CONCLUSION

The final chapter of this thesis summarizes the achievements of the presented work. Therefore, a comparison of the implemented tool-flow with the objectives stated in *Introduction Part I* is given. Furthermore, it discusses the known issues of the current implementation and points out possible improvements of the presented work-flow. Finally, the thesis concludes with a short summary.

10.1 Realization of Aims and Objectives

The goal of this thesis was to implement a GCC-based work-flow for the translation of high-level software applications to a hardware/software co-design for hybrid systems. The entire work-flow was intended to be transparent for the system developer, in order to lower the entry barrier for software designers who lack specialized hardware knowledge.

These primary goals have been divided into single objectives, as described in Section 1.3. The following paragraphs summarize the realization of these objectives and point out which of them have been reached and which of them only satisfy the requirements partially .

- **Use of a high-level input language without annotations.**
  The current approach uses unrestricted, plain C as input language, as this is a common language for embedded applications. However, the GCC front-end still provides other language interfaces that could be used by the presented work-flow with little effort. Thus, support for a high-level language is given. The application does not require any annotation in order to generate an application-specific accelerator. Nonetheless, for the ARM architecture, it may be helpful to point out that a memory reference could never *alias*, even if the automatic alias analysis cannot find any evidence to prove this. This can be done with the keyword `restrict`. However, the whole process could still be used without any annotations for ARM as well as for SpartanMC.

- **Seamless integration of an HLS work-flow into a standard software compiler**
  This is accomplished by implementing all parts of the work-flow as GCC passes. Such passes can be integrated into the pass-schedule of the GCC by using the plugin interface of the compiler. Consequently, the inserted passes can take advantage of the optimized code structure that is provided by previous passes. In order to obtain a global view of the whole application, the compilation is run twice. The details of this process are integrated in a wrapper script that provides a transparent usage of the whole process from the developers viewpoint.

- **Automatic selection of promising code sequences**
  The automatic selection of code sequences for hardware acceleration is based on a simple heuristic that evaluates loops and uses the number of iterations as an indicator for a later hardware mapping. After identifying a list of accelerator candidates, the code sections in question are checked for various criteria, which could lead to exclusion of the candidate, e.g. function calls or unsupported GIMPLE statements. Although this may be regarded as a simple, yet powerful, solution, current experience shows that this approach could easily end up abandoning all candidates. For this reason, the approach offers room for improvements, which will be discussed in the following section.

- **Automatic adaptation of software code during compilation**
  After performing the HLS, this step is essential in order to call the generated accelerator in a transparent way. The current approach uses a twofold strategy: It modifies the actual GIMPLE representation of the program and generates additional compilation units to handle the data transfer. On the one hand, the direct modification of the compilers' intermediate representation guarantees a seamless integration of accelerators. On the other hand, the generation of additional compilation units provides the required flexibility to support complex interactions between software and hardware during accelerator execution. The current implementation can be easily adapted to other platforms or new functionalities.

- **Automatic generation of a memory interface**
  During the evaluation of different accelerators, the design of the memory interface turns out to be one of the most critical aspects of the system. At least, after HLS optimizations, the generated FSMs are usually memory-bound. Consequently, an accelerator design that provides only a slow memory interface runs the risk of introducing a massive performance bottleneck. The presented approach of tailored FIFOs in combination with an exact analysis of memory access patterns represents a promising concept to solve this problem. For this reason, the current implementation must be regarded as a first attempt with room for improvement.

- **Portability to other target architectures**
  In the context of this thesis, the plugin has been implemented for two different host processor architectures (SpartanMC and ARM Cortex-A9). This has been achieved by providing a unique implementation of the memory and host processor interface for each target. Nevertheless, the vast majority of the plugin sources have been used for both systems. This is supported by the decision to use the plugin interface of the GCC instead of direct modifications of the compiler. In addition, the idea to work on an architecture-agnostic intermediate code representation (GIMPLE) facilitates good portability of the plugin. On these grounds, the presented GCC plugin can be regarded as well adaptable to new GCC targets and host architectures.

Besides the solution for these technical issues, the question from Section 1.3 remains: Which improvements of application performance can be achieved for both target platforms using a static code analysis and a fully automatic accelerator generation?

It has been shown that performance improvements by a factor of approximately four are possible for the SpartanMC platform. This is supported by certain underlying conditions, e.g. the availability of a fast and direct memory interface as well as a balanced frequency setup between host processor and hardware accelerator. In fact, the results for the SpartanMC architecture can still be improved by raising the accelerator clock frequency. Considering that the overall process remains transparent to the end user and does not introduce any further efforts, the results are good.

Obviously, the ratio between host processor and accelerator clock frequency is a property of the target architecture and can hardly be influenced by the used work-flow. Unfortunately, in case of the ARM architecture, a frequency ratio of approximately seven, in favor of the host processor, is given. Hence, the current work-flow cannot provide any speedups for a single problem setup without extensive code refinements. Though a manual refinement of a C program is not comparable with the effort of custom hardware development, it would also contradict the intended approach to support legacy C code. Increasing the single-accelerator performance is, however, not the only way to gain an overall application speedup. Architectures that provide an operating system could benefit from real task-level parallelism by using accelerators. This solution could provide a speedup even with single accelerators running slower than the host processor. Thus, the presented system could still be used to extract parallel computing cores automatically.

Due to the given results, the ideal use case for the presented system would be a host processor running almost at the same clock frequency as the FPGA fabric. Prominent examples for this setup are the Microblaze [119], NIOS [89] or Leon-Family soft-cores [87]. There are also examples of state-of-the-art FPGA-SoC architectures that could be suitable for the given work-flow, e.g. Microsemi's SmartFusion Family [109], which consists of a Flash-based FPGA fabric and an ARM Cortex-M3, both running at a clock rate of up to 100 MHz.

To summarize, it can be concluded that the objectives of this thesis have been reached. Nevertheless, the effort of implementing a working tool-flow for hardware/software co-design was tremendous. In order to save time for future research projects, the presented implementation of GCC PIRANHA can be used as a starting point.

10.2 Limitations and Future Work

As mentioned already, not all GIMPLE operators and data types are supported by the HLS work-flow. For this reason, the current implementation will reject candidates containing unsupported GIMPLE expressions. The simple reason for almost all missing expressions is the lack of an adequate Verilog representation

for the respective operation or data type. Typically, such representations can be added with more or less engineering effort, which would solve this issue.

Another unsolved problem is the support for unbounded loops on ARM systems using FIFOs. This would require an additional write-back operation to be executed after reaching the exit condition of the loop for each write-FIFO. Although this is not supported at present, it could be implemented with little effort. In a next step, the FIFOs and the hardware accelerators should be adapted for continuous operation in order to provide a real streaming interface. Together with the existing memory access analysis, this would allow the generation of *streaming* accelerators without special code annotations.

Besides these minor issues, there is also one conceptional challenge that should be the subject of future research. The basic idea is simple: Available optimization passes should be actively used to modify the intermediate representation of loop candidates to the special needs of hardware generation.

At present, the candidates are identified and processed after completing all loop optimization passes. Note that the GCC is still a compiler for sequential general purpose processors; thus, possible optimizations are tailored to the typical requirements of such architectures (even on the GIMPLE-level). Therefore, the provided intermediate structures does not represent a fully optimized version of the current loop and there is no guarantee that all suitable optimizations for hardware generation have been applied. Many interesting optimizations for instance, modulo scheduling are carried out at the end of the compilation process by using target-specific RTL. This makes perfect sense, as such optimizations can only be applied for specific machines.

For such unavailable RTL optimizations, there is no other option but to reinvent these passes on the GIMPLE-level for hardware generation. Apart from that, there are many available GIMPLE passes and IPA passes that would be useful for hardware generation but have already completed their optimizations when the plugin is called. Unfortunately, due to the internal properties of GIMPLE, it is not possible to re-execute some of these passes. The quantity of suitable candidates, as well as the performance of the resulting accelerators, could be significantly improved by actively modifying the intermediate representation of a candidate to the particular needs of hardware generation by using these

optimizations[1]. Therefore, it would be necessary to restructure the internal work-flow of the plugin in such a way that the list of potential candidates is completely available in the early stages of the second GCC run. This would allow to parametrize useful GIMPLE optimizations for certain candidates before the actual HLS plugin starts. Consequently, the remaining work-flow could take advantage of selective modifications of the GIMPLE code of the accelerator candidate.

Another conceivable improvement is the replacement of the current random memory access interface for the ARM architectures with the cache architecture proposed in [144]. This would provide a prefetching mechanism even for memory accesses that are currently not analyzable at all.

Beyond that, future work should address the migration to partial reconfiguration in order to support accelerators in multiple applications on Linux systems. In addition, the impact of accelerators on the systems energy consumption and the effects of task-level parallelism should be evaluated.

10.3 Summary

In this thesis, a GCC-based work-flow for automatic accelerator generation and integration has been presented. The intended target platforms are hybrids of FPGA fabric and static host processor. The given work-flow performs an automatic hardware/software partitioning by selecting frequently executed loops for the mapping to hardware. Therefore, the underlying program analysis operates on the whole application and analyzes function calls even across file boundaries. The implemented hardware synthesis uses a target-agnostic intermediate representation of the given C sources to generate a hardware description of the accelerator. Moreover, an elaborated analysis of memory references is applied in order to identify exact memory access patterns within accelerator candidates. This enables the integration of a prefetching mechanism for the respective accelerator interface.

In contrast to recent approaches, the presented process neither requires HDL skills from the user nor any special hardware knowledge of the underlying platform. The integration of accelerators is carried out in a seamless and transparent

[1] Promising modifications would be function inlining or (partial) loop-unrolling.

way, which also includes support for modern operating systems. The approach avoids the use of special annotations or restrictions on to the input language that would substantially lower the entry barrier for most software programmers.

The presented work-flow was implemented for two platforms, an FPGA-based soft-core processor and a rather complex Zynq-7000 FPGA SoC encompassing an ARM Cortex-A9 and an Artix-7 FPGA fabric. The evaluation of both implementations shows that the soft-core processor platform achieves an speedup of up to four while accelerators on Zynq show slowdowns for unmodified C code. This loss of performance could be explained by the seven times higher clock rate of the ARM host processor and the latencies of the memory interface.

for this platform, even though, this demands additional refinements of the given C code.

At present, the development of several HLS tools is on the threshold from the prototype stage to commercially viable products. Nonetheless, most of these approaches are intended for hardware developers. The underlying design strategies are usually "obsessed" with the optimum and thus compete with hand developed HDL solutions. In contrast, the presented work-flow intends to reveal new approaches for the ongoing development of these tools, by changing the perspective to the software developer's point of view.

A EXAMPLES AND CODE LISTINGS

A.1 Alias Set Generation for Interleaved Structure Access

The following C code generates an interleaved and self-aliasing access to an array of structures. The example and the associated alias sets are taken from [135].

```
struct Foo {
  int input;
  int output;
};

int munge_data(struct Foo data[100]) {
    int  i, accum = 0;
    for (i = 0; i < 100; ++i) {
        accum += data[i].input;
        data[i].output = accum;
    }
    return accum;
}
```

Figure A.1: Access to array of structures [135]

The symbols and accesses discovered for the example above by using the `MemrefAnalyzer` are shown in Table A.1.

Table A.1: Fields of the memory access structure for Listing A.1

Symbol idx.	Name	Access idx.	Code	Type	Width	Offs.	Slices
0	data	0	data[i].input	LOAD	4	0	(4, 8)
0	data	1	data[i].output	STORE	4	4	(4, 8)

These values are integrated into the equation system to get the universe and problem sets:

$$
\begin{aligned}
U &= \left\{ (\alpha_1, \alpha_2, \beta_1, \beta_2, d) \in \mathbb{Z}^5 \right\}, \\
P &= \left\{ (\alpha_1, \alpha_2, \beta_1, \beta_2, d) \in \mathbb{Z}^5 \;\middle|\; \right. \\
&\qquad 0 \leq \alpha_1 \leq 99 \wedge 0 \leq \alpha_2 \leq 3 \wedge \\
&\qquad 0 \leq \beta_1 \leq 99 \wedge 0 \leq \beta_2 \leq 3 \wedge \\
&\qquad \left. 4 = 8\alpha_1 + \alpha_2 - 8\beta_1 - \beta_2 + d \right\}.
\end{aligned}
$$

By using the libISL to project out the variables α_1, α_2, β_1, and β_2, the following alias set A is found:

$$A = \{ d \in \mathbb{Z} \mid \exists e_0, e_1 : 0 \leq e_0 \leq 99 \wedge 0 \leq e_1 \leq 99 \wedge \\ 8e_1 \geq 1 + d + 8e_0 \wedge 8e_1 \leq 7 + d + 8e_1 \}.$$

The resulting expressions are hard to understand and expensive to evaluate. However, after applying the optimizations described in Section 5.6.4, the basic sets reduce to three disjoint sets.

$$A = A_0 \cup A_1 \cup A_2,$$
$$A_0 = \left\{ d \in \mathbb{Z} | \exists e_0 = \left\lfloor \frac{7 + 7d}{8} \right\rfloor : -1 \leq d \leq 791 \wedge 8e_0 \leq 7 + 7d \wedge 8e_0 \geq 1 + 7d \right\},$$
$$A_1 = \left\{ d \in \mathbb{Z} | \exists e_0 = \left\lfloor \frac{7 + d}{8} \right\rfloor : -799 \leq d \leq -8 \wedge 8e_0 \geq 1 + d \wedge 8e_0 \leq 1 + d \right\},$$
$$A_2 = \{ d \in \mathbb{Z} | -7 \leq d \leq -2 \}.$$

Each of the resulting sets has a constraint on the range of d but covers another part of $\mathbb{Z}$ for d. As d refers to a self-alias query (cf. Listing A.1), this implies that d must be zero. Thus, the only valid condition is A_0 which can be decomposed to the following, more trivial, conditions:

$$e_0 = \left\lfloor \frac{7 + 7d}{8} \right\rfloor = 0,$$
$$8e_0 \leq 7 + 7d \quad \Rightarrow 8 \cdot 0 \leq 7 + 7 \cdot 0 \quad \Rightarrow 0 \leq 7, \text{ which is true,}$$
$$8e_0 \geq 1 + 7d \quad \Rightarrow 8 \cdot 0 \geq 1 + 7 \cdot 0 \quad \Rightarrow 0 \geq 1, \text{ which is false.}$$

Consequently, d is not in A_0 and no aliasing occurs.

A.2 GIMPLE Example

The following listings show the *high* GIMPLE and *low* GIMPLE representation for the sample code in Listing 4.7.

High GIMPLE

```
foo_sum (int * a) {
  unsigned int i.1, D.6506;
  int D.6508, D.6511, sum;
  int * D.6507;
  {
    int i;
    i = 0; sum = 0;
    goto <D.6488>;
    <D.6487>:
    i.1 = (unsigned int) i;
    D.6506 = i.1 * 4;
    D.6507 = a + D.6506;
    D.6508 = *D.6507;
    if (D.6508 < 0) goto <D.6509>;
    else goto <D.6510>;
    <D.6509>:
    D.6511 = error ();
    return D.6511;
    <D.6510>:
    i.1 = (unsigned int) i;
    D.6506 = i.1 * 4;
    D.6507 = a + D.6506;
    D.6508 = *D.6507;
    sum = D.6508 + sum;
    i = i + 1;
    <D.6488>:
    if (i <= 99) goto <D.6487>;
    else goto <D.6489>;
    <D.6489>:
  }
  D.6511 = sum;
  return D.6511;
}
```

Listing A.2: High GIMPLE example

Low GIMPLE

```
foo_sum (int * a) {
  int i, sum, D.6511, D.6508;
  unsigned int D.6506, i.1;
  int * D.6507;

  i = 0; sum = 0;
  goto <D.6488>;
  <D.6487>:
  i.1 = (unsigned int) i;
  D.6506 = i.1 * 4;
  D.6507 = a + D.6506;
  D.6508 = *D.6507;
  if (D.6508 < 0) goto <D.6509>;
  else goto <D.6510>;
  <D.6509>:
  D.6511 = error ();
  goto <D.6512>;
  <D.6510>:
  i.1 = (unsigned int) i;
  D.6506 = i.1 * 4;
  D.6507 = a + D.6506;
  D.6508 = *D.6507;
  sum = D.6508 + sum;
  i = i + 1;
  <D.6488>:
  if (i <= 99) goto <D.6487>;
  else goto <D.6489>;
  <D.6489>:
  D.6511 = sum;
  goto <D.6512>;
  <D.6512>:
  return D.6511;
}
```

Listing A.3: Low GIMPLE example

```
unit1.c
  1451067104
  =
  function=fun2
    frequency=1
    %
    uid=0
    lineno=10
    index=3
    countable=1
    header_frequ=1
    latch_frequ=0
    count.low=29
    count.high=0
    gcountable=1
    countall.low=29
    countall.high=0
    instr_sw=9
    instr_hw=8
    deps_computed=0
    deps=0
    wrefs=0
    rrefs=0
    innermost=1
    call=1
    well_nested=0
    -fun3
    =

  function=fun1
    frequency=1
    %
    uid=1
    lineno=5
    index=3
    countable=1
    header_frequ=1
    latch_frequ=0
    count.low=9
    count.high=0
    gcountable=1
    countall.low=9
    countall.high=0
    instr_sw=8
    instr_hw=7
    deps_computed=0
    deps=0
    wrefs=0
    rrefs=0
    innermost=1
    call=1
    well_nested=0
    -fun2

unit2.c
  1451067115
  =
  function=fun3
  frequency=1
    %
    uid=0
    lineno=3
    index=3
    countable=1
    header_frequ=1
    latch_frequ=0
    count.low=99
    count.high=0
    gcountable=1
    countall.low=99
    countall.high=0
    instr_sw=6
    instr_hw=6
    deps_computed=1
    deps=0
    wrefs=0
    rrefs=0
    innermost=1
    call=0
    well_nested=1
```

Listing A.4: Complete transcript file of *unit1.c* and *unit2.c* [145]

B PLATFORM

B.1 SpartanMC Instruction Set

Table B.1: SpartanMC instruction coding matrices – Main Matrix (IR 17-13)

IR 4-0	..000	..001	..010	..011	..100	..101	..110	..111
00..	Spec. 1	Spec. 2	J	JALS	BEQZ	BNEZ	BEQZC	BNEZC
01..	ADDI	MAVI	LHI	SIGEX	ANDI	ORI	XORI	MULI
10..	L9	S9	L18	S18	SLLI	—	SRLI	SRAI
11..	SEQI	SNEI	SLTI	SGTI	SLEI	SGEI	IFADDUI	IFSIBUI

Table B.2: SpartanMC instruction coding matrices – Submatrix Special 1 (IR 4-0)

IR 4-0	..000	..001	..010	..011	..100	..101	..110	..111
00..	—	—	—	—	SLL	MOV	SRL	SRA
01..	SEQU	SNEU	SLTU	SGTU	SLEU	SGEU	—	—
10..	—	—	—	—	—	—	CBITS	SBITS
11..	—	—	—	—	—	—	—	NOT

Table B.3: SpartanMC instruction coding matrices – Submatrix Special 2 (IR 4-0)

IR 4-0	..000	..001	..010	..011	..100	..101	..110	..111
00..	RFE	TRAP	JR	JALR	JRS	JALRS	—	—
01..	—	—	—	—	—	—	—	—
10..	ADD	ADDU	SUB	SUBU	AND	OR	XOR	MUL
11..	SEQ	SNE	SLT	SGT	SLE	SGE	MOVI2S	MOVS2I

B.2 SpartanMC Pipeline

The SpartanMC pipeline consists of three stages driven by one clock signal. Instruction fetch (IF), instruction decode (ID), and the first part of operand fetch (OF1) are combined in the first stage. The second part of operand fetch (OF2), execution (EX), and the first part of the memory access (MEM1) form the second stage while the second part of the memory access (MEM2) and write back (WB) are combined in the third stage.

git-commit-hash: ebbf16479adbdf3d7b718dcb1d1c79a026013456
(internal repository of RS chair TU Darmstadt)

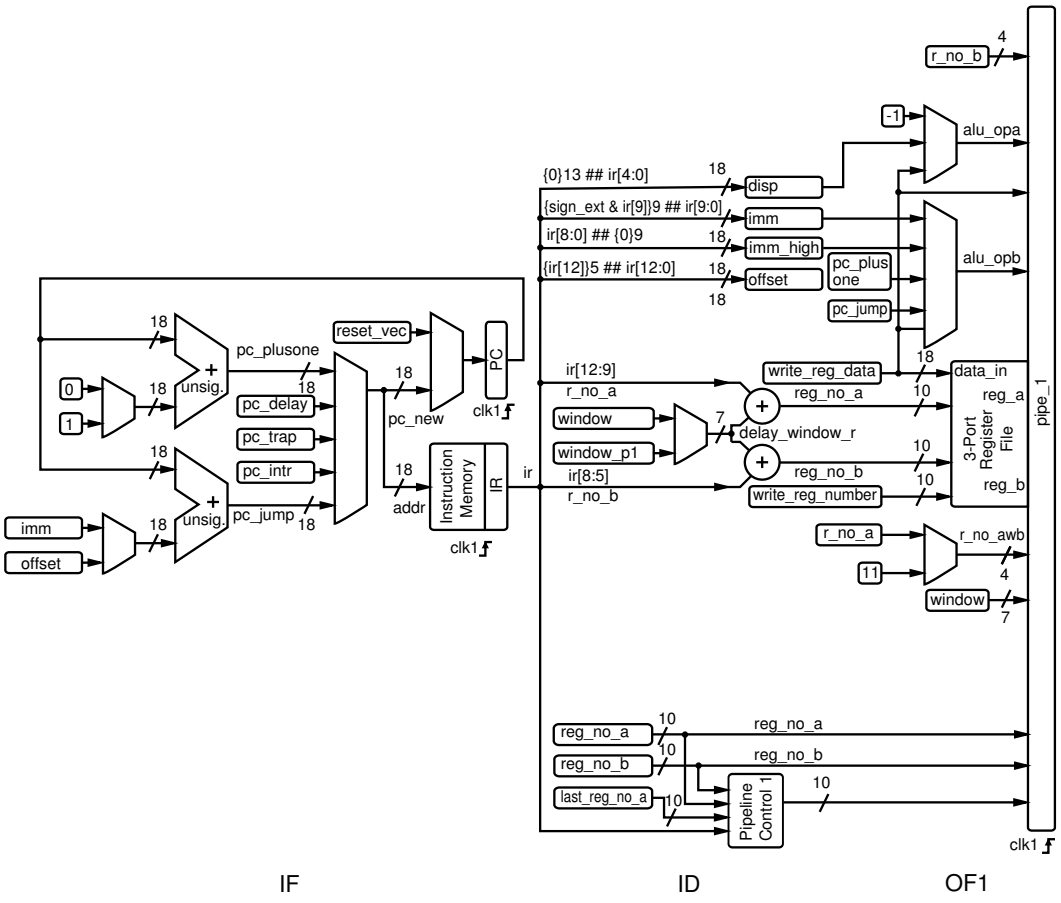

Figure B.1: SpartanMC pipeline architecture (IF, ID, OF1) [139]

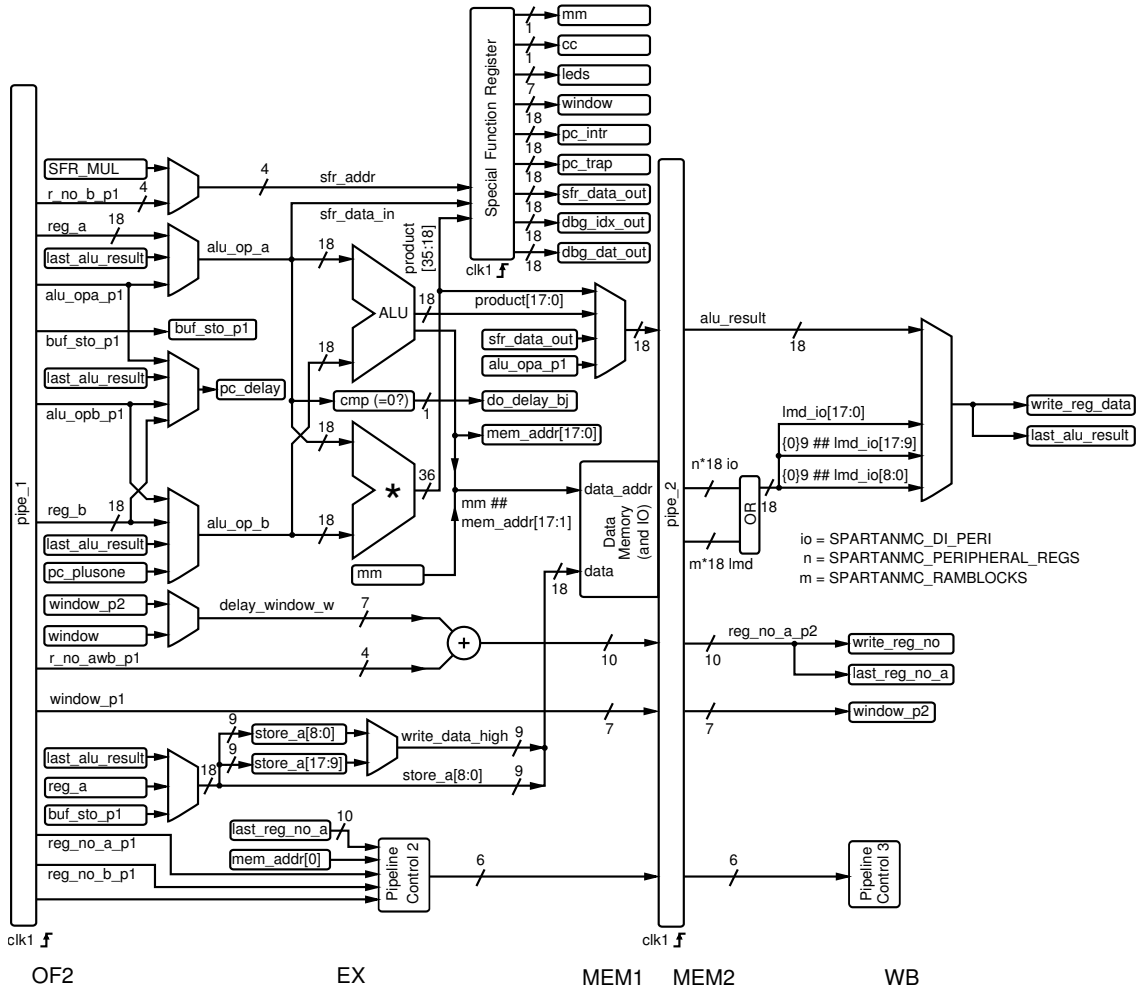

Figure B.2: SpartanMC pipeline architecture (OF2, EX, MEM1, MEM2, WB) [139]

C GCC PLUGIN

C.1 PIRANHA Parameters

Plugin parameters have to use the following synopsis `-fplugin-arg-<plugin>-<key>[=value]`. The provided shared objects `hw_analysis.so` and `hw_generation.so` determine the name of the plugin. The following listing shows only the `key` of the respective commands. If nothing different is specified, the default value of parameters is "0".

Table C.1: GCC plugin parameters

Key	Type	Description
		Analysis Plugin
`debug`	`bool`	Debug output for analysis plugin.
`extract-file`	`string`	Name of analysis transcript file.
`ignore-uncountable`	`bool`	Ignore loop in case of an unknown iteration count.
		Synthesis Plugin
`codegen-debug`	`bool`	Debug output for code generation.
`log`	`string`	Configure logging for a specific subsystem logger and logging level. The subsystem loggers (e.g. `memref` or `chaining`) specify the part of the plugin for which logging should be configured. Each subsystem logger is subordinated to the root logger. The logging level specifies the verbosity of the subsystem logger. Possible levels are `all`, `debug`, `info`, `warning`, `error`, `exception` and `nothing`. For instance, `root:all` would show the entire logging output of the plugin.
`log-file`	`string`	Configure the logging level and a file name for the file-output of the logging system, e.g. `buildlog.xml:debug`. By default, the output files are provided in XML format with integrated style information.
`log-stdio`	`string`	Configure the logging level for `stdout`. (default value is `warning`)
`stats-memory-analysis`	`bool`	Create a `*.json` file for each accelerator with memory analysis statistics

Key	Type	Description
		Synthesis Plugin
`path`	`string`	Specifies the output path for generated Verilog files.
`addr-space-begin`	`uint`	Specifies the begin of the address space for accelerators. (in hexadecimal)
`max-regs`	`uint`	The maximum number of generated registers of an accelerator. (default is 512)
`compare`	`string`	Specifies the compare function for loop ordering during analysis. Possible values are: `lcmp_iterations` Sort loops by iterations. `lcmp_iterations_strict` –"–. `lcmp_instructions` Sort loops by instructions. `lcmp_instructions_strict` –"–. The *strict* variant of the compare functions assumes an iteration count of one for uncountable functions (default is infinite).
`ninstr`	`uint`	Define maximum number of instructions for all loop candidates. This value is only used by `lcmp_instructions` (default is 100).
`nloops`	`uint`	Define maximum number of loop candidates. (default is 100)
`synth-strategy`	`uint`	Specify the candidate selection strategy. Possible values are: `0` Use the given minimum efficiency as the lower bound for candidate selection. If the efficiency is not predictable, the loop candidate is synthesized. `1` Use the given minimum efficiency as the lower bound for candidate selection. If the efficiency is not predictable, the loop candidate is rejected. `2` Take all candidates.

Key	Type	Description
		Synthesis Plugin
`min-efficiency`	`uint`	Specify the minimum efficiency. Default value is "0"; thus, all candidates are taken for synthesis if a speedup is predicted. For *Synthesis Strategy 2*, this value is irrelevant.
`freq-cpu`	`uint`	Specifies the clock rate of the host system in MHz (default is 100).
`freq-acc`	`uint`	Specifies the clock rate of the accelerators in MHz (default is 100).
`chaining`	`bool`	Activates chaining.
`spec-exec`	`bool`	Activates speculative execution.
`spec-max-wight`	`uint`	Specifies the maximum deviation of weight difference for speculation
`block-wights`	`bool`	Defines the weighting method for speculative execution. Possible values are: `instr_cnt` Take the number of instructions to weight an execution path. `block_latency` Take the latency of an execution path for weighting.
`fifos`	`bool`	Enable FIFOs for memory accesses. This parameter is only available for ARM.
`runtime-isl`	`bool`	Enable runtime ISL evaluation of alias sets.
`assume-loop-bounds`	`uint`	Assume a fixed iteration count for uncountable loops in order to enable a proper memory access analysis (default is "-1").

C.2 Heuristic Delay Times for Operations

Table C.2: Delay times for different operations

Type	Operation	Delay
Assignments	=	0
Boolean Operations	<<, >>, *XOR*, *AND*, *NOT*, *OR*	5
Comparison Operations	==, $<$, $\leq$, $\geq$, $>$, $\neq$	5
Branch Operations	`if`, `else`, `switch`, `loop_exit`, `default`	10
Simple Arithmetic	$+$,$-$	10
Complex Arithmetic	$*$,%	50
Memory Operations	`MEM_READ`, `MEM_WRITE`	101
Unknown Operations		100

C.3 Generated Structures for Memory Analysis

Table C.3: Fields of the `memref_symbol` runtime structure [135]

Type	Name	Description
`void*`	`base`	The base, if it has been determined at compile time (e.g. because it is a reference to a `static const`).
`bool`	`base_known`	Is *true* if the base has been determined at compile time.
`bool`	`random_access`	After run time alias analysis, this field indicates whether random access mode needs to be used with this symbol.
	`accesses`	A pointer to a `NULL`-terminated sequence of accesses for this symbol.

Table C.4: Fields of the `memref_access` runtime structure [135]

Type	Name	Description
`int`	`mode`	The mode of the access, either `AM_LOAD` or `AM_STORE`.
`size_t`	`word_size`	The width of a single access in bytes.
`ptrdiff_t`	`min_offset`	The constant offset from the base.
`size_t`	`bit_offset`	Only for `component_ref`s, this is the bit offset of the accessed field relative to the previous byte.
`size_t`	`slice_count`	The count of nested slices this access uses.
	`slices`	The pointer to a sequence of slices for this access.

Table C.5: Fields of the `memref_slice` runtime structure [135]

Type	Name	Description
`ptrdiff_t`	`stride`	The stride of the slice. A negative stride indicates that the access runs toward "reverse".
`size_t`	`iterations_minus_one`	The number of iterations of the slice, minus one.

C.4 GIMPLE Statements

The following Table C.6 describes the GIMPLE statements that are important for this thesis. An overview of all available GIMPLE statements and their scope of usage (*high* GIMPLE or *low* GIMPLE) is given in Table C.7.

Table C.6: Assorted GIMPLE statements

Statement	Description
`GIMPLE_ASSIGN`	This statement assigns a value to the operand on the left-hand-side (LHS). The value is either taken directly from the first right-hand side operand (RHS1) or calculated from RHS1 and RHS2 by using an expression (cf. Table C.9).
`GIMPLE_ASM`	This statement describes inline assembler instructions. Loop candidates containing this statement will be excluded from synthesis.
`GIMPLE_COND`	This statement is generated for *if-else-branches* and uses one LHS and one RHS operand. The function `gimple_cond_code` returns the actual condition status of this statement.
`GIMPLE_SWITCH`	This statement implements a *switch*-branch. The index variable included in this statement can be determined by using the GCC function `gimple_switch_index`. The labels are accessible via `gimple_switch_num_labels` and `gimple_switch_label`, respectively.
`GIMPLE_CALL`	This statement represents a function call using its LHS as a return value. A function call statement is specified by a `FUNCTION_DECL` tree and a list of trees as parameter-list. The `GIMPLE_CALL` statement is generated to patch the existing code with the call of the accelerator wrapper function.

Table C.7: All GIMPLE statements and their scope of usage

Instruction	High GIMPLE	Low GIMPLE
GIMPLE_ASM	x	x
GIMPLE_ASSIGN	x	x
GIMPLE_BIND	x	
GIMPLE_CALL	x	x
GIMPLE_CATCH	x	
GIMPLE_COND	x	x
GIMPLE_DEBUG	x	x
GIMPLE_EH_FILTER	x	
GIMPLE_GOTO	x	x
GIMPLE_LABEL	x	x
GIMPLE_NOP	x	x
GIMPLE_OMP_ATOMIC_LOAD	x	x
GIMPLE_OMP_ATOMIC_STORE	x	x
GIMPLE_OMP_CONTINUE	x	x
GIMPLE_OMP_CRITICAL	x	x
GIMPLE_OMP_FOR	x	x
GIMPLE_OMP_MASTER	x	x
GIMPLE_OMP_ORDERED	x	x
GIMPLE_OMP_PARALLEL	x	x
GIMPLE_OMP_RETURN	x	x
GIMPLE_OMP_SECTION	x	x
GIMPLE_OMP_SECTIONS	x	x
GIMPLE_OMP_SECTIONS_SWITCH	x	x
GIMPLE_OMP_SINGLE	x	x
GIMPLE_PHI		x
GIMPLE_RESX		x
GIMPLE_RETURN	x	x
GIMPLE_SWITCH	x	x
GIMPLE_TRY	x	

C.5 Tree Types

A subsequent tree for a given tree node can be determined with the GCC macro `TREE_TYPE`. This macro, again returns a tree type; thus, it can be used to examine tree nodes recursively until the `NULL_TREE` is reached. The actual enumeration value specifying the tree type can be determined with the macro `TREE_CODE`.

Sub-operands of a tree node for instance, for a TARGET_MEM_REF-type (*base*, *index* etc.), can be determined via the TREE_OPERAND-macro. The following table shows the currently supported tree types.

Table C.8: Tree types

Tree Type	Description
LABEL_DECL	This type represents a label within a function. Typically, it is used as a target for GIMPLE_GOTO statements.
INTEGER_CST	This type describes an integer constant. The actual value is accessed with the macro TREE_INT_CST_LOW.
REAL_CST	This type describes a floating-point constant. If this type is found during application analysis, the current loop candidate is rejected.
ADDR_EXPR	This type describes the value of a memory address.
MEM_REF	This type describes a memory access by two arguments (cf. 4.2.1).
ARRAY_REF	This type describes the access to the elements of an array.
TARGET_MEM_REF	This type describes memory accesses by five arguments (cf. 4.2.1).
INDIRECT_REF	A type describing a memory access by dereferencing a pointer.
COMPONENT_REF	This type describes a memory access to a member of a structure.
PARM_DECL	This type is used for function parameters.
VAR_DECL	This type is used for variables.
CONST_DECL	This type is used for enumeration constants.
REAL_DECL	This type is used for floating-point variables. If this type is found during application analysis, the current loop candidate is rejected.
POINTER_TYPE	This type represents a pointer. The actual data type of the pointer is determined via TREE_TYPE.
BOOLEAN_TYPE	A type describing a Boolean value.
INTEGER_TYPE	A type describing an integer value.
ENUMERAL_TYPE	A type describing an enumeration value.

C.6 Operations

The type of a complete GIMPLE statement is determined with the macro `TREE_CODE`. The currently supported return values are listed in the following table. Note that the occurrence of unsupported operations emits a *warning* during plugin execution.

Table C.9: GIMPLE expressions

Tree Type	Description
`BIT_IOR_EXPR`	Bitwise *OR*
`BIT_XOR_EXPR`	Bitwise *XOR*
`BIT_AND_EXPR`	Bitwise *AND*
`BIT_NOT_EXPR`	Bitwise *NOT*
`LSHIFT_EXPR`	Left shift
`RSHIFT_EXPR`	Right shift
`NEGATE_EXPR`	Sign reversal
`MIN_EXPR`	Return the minimum of two operands
`MAX_EXPR`	Return the maximum of two operands
`PLUS_EXPR`	Arithmetic addition
`POINTER_PLUS_EXPR`	Pointer increment
`MINUS_EXPR`	Arithmetic subtraction
`MULT_EXPR`	Arithmetic multiplication
`LT_EXPR`	Less than
`LE_EXPR`	Less than or equal to
`GT_EXPR`	Greater than
`GE_EXPR`	Greater than or equal to
`EQ_EXPR`	Equal to
`NE_EXPR`	Not Equal to
`NOP_EXPR`	Do nothing

D OS INTEGRATION

A common mechanism used by operating systems for managing access to critical resources or preventing processes from interfering with each other is the usage of CPU privilege levels. The kernel of the operating system typically runs on the highest privilege level, the *kernel-mode*. Meanwhile, processes that are spawned by the kernel, operate on a lower privilege level, the *user-mode*. Hence, the interface to hardware accelerators must be provided by a *kernel-mode* application that implements the bookkeeping of hardware resources and allows access to accelerators and memory from a *user-mode* application.

The integration of hardware accelerators in separate processes on a full-featured OS has hardly been explored in the research field of HLS and configurable architectures. One of the rare examples of a toolchain supporting custom accelerators within an OS was presented by A. Agne et al. in [88]. The proposed system *ReconOS*[1] defines an interface for an operating system to integrate and manage automatically generated accelerators. Therefore, the concept of hardware threads for accelerators that could be used alongside classical software threads is introduced. Both types of threads use a POSIX thread-like API. The build process of the underlying eCos OS links application threads and OS threads together to form a monolithic bare-metal binary. Consequently, the resulting system requires neither virtual memory management nor special treatment of the accelerators. The data exchange between accelerator and host system is implemented as a shared memory between hardware and software threads. The drawback of the system is the rather coarse scope on complete threads. Due to this issue, the approach can be considered as incompatible with the presented GCC-based HLS system, which works on the granularity of loops.

In the initial version, the plugin used the helper-library `libacc.a` to handle accesses to the accelerators from user space. The library maps the accelerator address space into the process address space by using `mmap`. This approach assumed the usage of only one accelerator at a time or, at least, the cooperative behavior of all involved processes and accelerators when running simultaneously. However, from the system designer's point of view, this contradicts the idea of different privilege levels. Thus, for PIRANHA, the integration of generated hardware into the application is supported by a generic OS interface that handles the accelerator invocation, the data transfer, the address translation, and the hardware resource management.

D.0.1 Customized Application Binary

The current design of the tool-flow generates an ELF file containing the patched application and a separate FPGA design file (bit-file). Virtually, the executable is logically connected with the bit-file that contains the required accelerators. The accelerators are usable only if the corresponding

[1] Contrary to what the name suggests, ReconOS is not a full OS but rather an interface for custom hardware. Nevertheless it is based on the popular embedded OS eCos [97].

bit-file could be loaded; otherwise, the running program must execute its software path.

In order to resolve this dependency, the bit-file was embedded into the executable file. This approach was implemented in [136]. Besides applying an implicit connection between bit-file and software binary, it provides the intuitive handling of applications using accelerators. It would allow the end user to load the FPGA design transparently during program execution, which would enable the usage of accelerators even for users who lack background knowledge about the hybrid design of the underlying architecture.

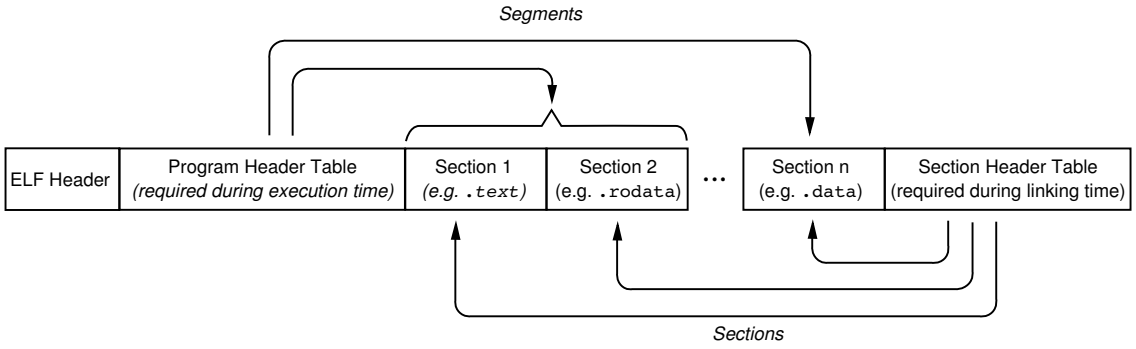

Figure D.1: Structure of an ELF file [136]

ELF is broadly used and could be regarded as the actual standard executable file format for Linux systems. One premise for the extension of this format was the avoidance of compatibility issues with traditional tools. As shown in Figure D.1, the ELF file format starts with an ELF header, followed by the program header table, and the actual data, which is organized in sections. Typically, the ELF file contains the section header table at its end. As the ELF file could be interpreted during program execution or at link time one distinguishes two different views of the file. While the program header shows the segments used at the time of execution, the section header lists the set of sections for the linking view. Note that the ordering of sections and headers shown in Figure D.1 must be regarded as best practice only. It applies for most binaries, but it is not mandatory in the file formats definition. Unfortunately, this aggravates the design of an ELF parser. Nevertheless, one great advantage of the ELF format is its flexibility. It could contain arbitrary sections that make it easily extensible and well suited for our purpose.

The connection between bit-file and ELF format could be implemented by two orthogonal methods. On the one hand, it would be possible to embed a pointer into the ELF file, referencing the external bit-file. On the other hand, the bit-file data could be integrated directly into the ELF file. The former method would be similar to the usage of shared libraries. Hence, it would be suitable for setups that share an accelerator between multiple applications. However, although possible, this is not a common use case of PIRANHA. As the extended ELF file describes a one-to-one mapping between accelerator and a specific software binary, it could also store the whole bit-file. For that reason, the latter method was preferred. This approach still allows the usage of shared objects in the ELF format and also enables the sharing of bit-files between multiple applications.

Extended ELF File Format

The ELF file format allows the definition of sections holding arbitrary data. Each section type is identified by a 32-bit value in the section header table. The mechanism is used to describe segments from the program headers point of view. The reserved range for custom types for OS-specific sections and segments is defined by `0x60000000-0x6FFFFFFF`. To identify the bit-file section, the ELF file was extended by a new section and program header of type `0x68777475`. This random type identifier lies in the middle of the reserved range. This should minimize the risk of colliding with another type that is probably defined in another system environment[2]. The newly inserted section, called *ZwoELF*, contains the bit-file and some additional fields that are shown in the C representation in Listing D.2. In order to detect corruption of the included data, ZwoELF contains a checksum field (using a SHA256 hash of the data field). Moreover, it provides a version field that can be used for future consistency checks between different tool-chain components. Finally, two length fields (`dtbfile_len` and `bitfile_len`) determine the length of the following data fields, which further consist of the DTB defining the hardware device for the OS (cf. Section D.0.4) followed by the bit-file containing the hardware design. The new section is named `.tudos.hwacc.XXXXXXXX`, with `XXXXXXXX` containing the first digits of the checksum in hexadecimal.

```
#define ZWOELF_DIGEST_SIZE   SHA256_DIGEST_SIZE
#define ZWOELF_VERSION_SIZE  32
struct zwoelf {
    __u8  checksum[ZWOELF_DIGEST_SIZE];
    __u8  version[ZWOELF_VERSION_SIZE];
    __u32 dtbfile_len;
    __u32 bitfile_len;
    __8   data[0];
}
```

Listing D.2: ZwoELF header definition [136]

Adding the ZwoELF section to an existing bit-file is carried out by a stand alone C++ program called `zwoelf`. This is necessary because the classic way – using `objcopy` from the `binutils` package – was unable to create the program headers which are required when loading executables. The implemented program takes an existing ELF binary, a bit-file, and a DTB as arguments, and outputs a new ELF file containing the additional ZwoELF section. Furthermore, it extends the program header and section header tables. The offsets within the header tables are later used by the kernel to map the sections into the process's address space. As the insertion of a new entry could invalidate the offsets of the remaining entries by changing their position in the file, it is also a task of the `zwoelf` program to realign the sections in order to correct such offsets.

[2] Most present OS environments tend to define these values at the beginning or end of the allowed range.

Bit-File Loading from ELF

After bringing bit-file and ELF binary together, the application loader process must be adapted in order to trigger the loading of the bit-file at application startup. The first and probably most obvious approach was the integration of the bit-file loader into the Linux runtime-loader `ld.so`. However, a naïve implementation of this approach is inelegant for two reasons. First, the runtime-loader `ld.so` cannot be used for statically linked binaries. Second, it would allow the user-space process to modify the FPGA content without considering other processes or the current configuration of the FPGA. For a potentially open system running third-party applications, this could induce a security problem. Hence, the kernel needs to be in charge of the bookkeeping of loaded bit-files and accelerators. As a consequence the kernel ELF loader is used directly.

The mechanism implemented in [136] registers a new binary format to the list of existing formats supported by the kernel subsystem implementing the loading of executables. By default, this list contains common formats, such as `binfmt_elf`, implementing support for loading ELF or `binfmt_aout` for `a.out` binaries. As shown in Algorithm 11, the kernel internal loader traverses each entry of this list and tries to recognize the format of the given binary in order to use the correct execution handler. Depending on the return code of the handler, the traversal stops (return "0" or *error*) or the next format is evaluated (return `-ENOEXEC`). If the traversal terminates without finding anything to execute, the code `ENOEXEC` is returned to the user-land.

Algorithm 11: Kernel internal binary loader

```
   Input: List (L) of Formats
   Output: Return Value ret
1  foreach Format ∈ L do
2      ret ⟵ execute_format()
3      switch ret do
4          case ENOEXEC do
5              continue
6          case "0" do
7              return 0
8          case error do
9              return error
10 return ENOEXEC
```

In order to utilize this process in a minimally invasive manner and without interfering with the current loading process of ELF files, an additional binary format is registered with respect to the following conditions:

- The new binary format handler supports a minimum of the ELF format just enough to identify the ZwoELF section.
- The new binary format is inserted in the list *before* the actual loading of `binfmt_elf` takes place.
- The new binary format handler returns `-ENOEXEC` instead of "0" after loading.

Hence, the loading process of the bit-file does not stop the kernel internal loader, which, in turn, allows the loading of the remaining ELF file afterwards.

The implemented format handler for the Linux kernel is called `binfmt_tudos_zwoelf`. In order to register the new format handler prior to the `binfmt_elf` handler, the kernel function `insert_binfmt(...)` is used. This function requires a pointer to the `struct linux_binfmt` as its argument, which describes the actual format. Among others, the structure contains the function pointer `.load_binary`, which refers to an actual implemented loader function for the specific handler. For `binfmt_tudos_zwoelf` this function is called `load_tudos_zwoelf_binary(...)`. It reuses some basic mechanisms from its pendant in `binfmt_elf`.

The task of this loader function is to identify the ZwoELF segment that has the program header type `PT_TUDOS_ZWOELF`. If such a section could be found within the binary, its data is extracted and handed over to the actual accelerator driver. For Linux kernel version 4.4 or higher, the loading of bit-files can be carried out with integrated kernel functions provided by the FPGA manager module (`fpga_mgr`). Older versions of the Linux kernel have to use the `xdevcfg` kernel driver provided by Xilinx. Both variants are supported by the current accelerator driver.

As already mentioned above, the return code of the loader is always `-ENOEXEC` in order to enable `binfmt_elf` to load the remaining ELF file afterwards. This even holds true in the case of an error. Instead of returning an error code, `binfmt_elf` will be allowed to try again, which would potentially lead to the pure software execution of the binary.

D.0.2 Managing Bit-Files and Utilizing the Device Tree

After extracting the bit-file on program execution, there still remains the task of loading the hardware design onto the reconfigurable fabric. This task is carried out by a device driver that should also allow the OS to handle the new hardware resources. At present, the plugin only supports generating a single bit-file occupying the whole FPGA. It may contain multiple hardware accelerators, but they always belong to the same application. Nevertheless, the device driver should be capable of keeping track of multiple bit-files, each corresponding to a free *slot* on the FPGA. However, the current implementation provides only one slot, which represents the whole FPGA. In this case, the implemented mechanism only avoids interfering with a running accelerator that may be used by another process. Nevertheless, the support for managing multiple bit-files is considered a long-term goal of PIRANHA. Although not supported by the tool-chain at present, it is conceivable to utilize partial reconfiguration techniques in conjunction with the PIRANHA

HLS tool-flow. This would allow a convenient usage of the reconfigurable fabric even for multiple applications with accelerators running in parallel.

Besides the managing of bit-files, the kernel needs to deduce which registers and base addresses are used by the current accelerator set in order to provide a convenient interface to the user-land. Unfortunately, this information cannot be extracted from the bit-file itself, which is an opaque object from the kernel's point of view. For this reason, the ZwoELF section contains the DTB field that describes the required set of devices. On embedded systems like ARM, there typically is no standardized firmware available to describe the hardware configuration of the machine[3]; instead, they usually run a rather simple boot loader from a flash memory. The actual hardware configuration is determined during system startup from a *device tree* that has to be provided to the kernel at boot-time. The *device tree* defines the configuration of all available hardware components through a file-system like tree. The DTBs contained inside the ZwoELF section represent the flattened device tree for a set of system components. They are generated from a DTS file by using the device tree compiler (`dtc`). The DTS represents a human-readable form of the DTB.

```
/dts-v1/;

/ {
    arbitrary_name {
      compatible = "tudos,hwacc";
      tudos,fpga-tile-id = <0xffffffff>
      reg = <
        0x76c00000 0x00100000
        0x78800000 0x00100000
      >;

      tudos, hwacc-name =
        "acc1_name"
        "acc2_name";
    };
  };
```

Listing D.3: Example DTS file for two accelerators

An example of a DTS file is shown in Listing D.3. It contains the register configuration of two accelerators. The accelerators are identified by their unique names defined in the `tudos, hwacc-name` field. In the context of this thesis, the `compatible` field and the `reg` field are of particular importance. The `compatible` field is used to find generated accelerators in the device tree. Formally, it is a string that specifies which device and driver is compatible with the

[3] On PC-compatible systems this task is carried out by the PC BIOS (basic input/output system) or a UEFI BIOS (united extensible firmware interface).

given tree node. It is usually specified as a <vendor>,<device> combination. The reg field is used to specify the register interface of the accelerator. It is an n-tuple of 32-bit values defining one or more register windows, each specified by a start address and a length. In order to distinguish multiple register windows within the reg field, one needs to know the number of 32-bit values that define the start and the length of the window, respectively. This is deduced from two parameters, #size-cells and #addr-cells, which are provided by the parent bus[4]. The length of the start address – defined by the number of consecutive 32-bit values – is specified by #addr-cells, while the length of the size field is defined by #size-cells. For instance, a value of #addr-cells of two and #size-cells of one would define a 64-bit start address and a 32-bit range for the register window. A value of zero for #size-cells implies that #addr-cells is directly used to address a distinct set of registers. Besides reg, compatible, and tudos,hwacc-name, the DTS contains the vendor-specific field tudos,fpga-tile-id. This field is used to identify which tiles of the FPGA are replaced by the bit-file. All bits set to one (0xffffffff) mean that the whole FPGA is used for the bit-file.

Typically, the DTB can be either provided to the kernel by the boot loader, or directly integrated into the kernel image. This means the device tree is static during system runtime. Fortunately, most recent Linux kernels provide mechanisms to dynamically add or remove devices from the tree in order to provide a hot plugging mechanism. In this context, the dynamic modification of the device tree is supported through so-called *device tree changesets*. The required branch of the device tree is generated by the plugin as DTS during compilation time. Afterwards, it is compiled into a DTB that is finally added to the ZwoELF section of the binary. The compatible field of the device is set to "tudos,hwacc". Later, this allows the accelerator driver to be registered for this type of device, which then triggers a notification by the kernel if such a device is added or removed. Finally, the generated devices are applied to the device tree during application startup.

D.0.3 The User-Land Interface

The access to the device driver is implemented by using the canonical approach provided by the Linux kernel. It uses a *device special file* that is, as the name implies, a special file-system entry typically located in /dev. The Linux device driver definition provides two types of device nodes that differ in their supported access patterns: character device and block device. In order to support a direct and fine-grained access, the character device model was chosen. As devices are represented as file-system entry, the access to the device driver is limited to regular file operations, such as open(...), close(...), read(...), and write(...). The current implementation provides one device file (/dev/hwacc/<acc_name>) per accelerator, which is identified by the unique accelerator name.

The access to accelerators is carried out by reading and writing of registers that are addressed within the corresponding IO memory. In case of generated accelerators, the address is resolved

[4] In our case, the parameters are defined by the AXI bus interface.

by the use of the base address and the offset to the specific register. As the base address is implicitly defined by the accelerator device, the actual register access only requires the relative offset within the register window. In order to access the accelerator, the standard user-land functions `pread(...)` and `pwrite(...)` are used. The parameter "`offset`" of these functions corresponds the offset in the register window. Unfortunately, standard functions such as `pread(...)` or `pwrite(...)` do not fit well into the semantic of accessing a single register. On these grounds, an additional memory mapped approach, based on `mmap(...)`, is used for non-DMA or single register accesses.

D.0.4 Prototype of a Linux Accelerator Driver

The device driver created in [136] is registered as a platform device driver. Typically, a driver needs to allocate a major device number and a minor number for registration. The major number defines the device class, while the minor number is used to manage multiple devices of this class. The driver registers a static major device number while for each accelerator a minor device number is allocated as needed.

As defined in the drivers `file_operations` structure, the device driver supports the IO operations `pread(...)`, `pwrite(...)` as well as `mmap(...)` and the file operations `open(...)` and `close(...)`.

Managing Accelerators at Runtime

Accelerators are used by calling `open(...)` and `close(...)` on the accelerator device file. Thus, the kernel can easily check the presence and status of the used accelerator. If it turns out that the accelerator does not exist or is currently in use, `open(...)` returns an error. In this case, the calling process switches to software execution for the hardware accelerated loop. The kernel checks the range of the requested memory of a read or write operation by default. If an IO operation tries to access a memory address that does not belong to the memory range of the requested accelerator's register window, the signal `SIGSEGV` is sent to the calling process. This harsh reaction is clearly intended and could help point out a potential bug in the GCC plugin that shall never generate invalid memory accesses. Hence, this behavior could indicate a severe error within the generated code.

Registration of Bit-Files

Another task is the creation of the internal accelerator representation in a consistent way during bit-file loading. Unfortunately, applying device tree changeset introduces some issues that affect proper bookkeeping of the created accelerators. The problem can be described as follows:

Before adding an accelerator to the device tree the kernel has to check the required resources for the new hardware in order to avoid multiple allocations of IO addresses. This would be achievable

by applying the complete DTB that is contained in the ZwoELF section of the executable. This is necessary, because `#addr-cells` and `#size-cells` must be derived from the parent device in the device tree in order to find the correct interpretation of the `reg` field in the DTB. The interpretation depends on the parameters of the used peripheral bus. Hence, the actual amount of required resources become available only after applying the complete device tree changeset.

At this point, the problem seems to be solvable by applying the DTB before loading the bit-file. Unfortunately, this provokes an inconsistent system state. The attaching of a new device triggers the kernel to probe for the new hardware that should provide the claimed registers at this moment. Hence, the driver must apply the bit-file beforehand, without actually knowing if it contains a valid configuration.

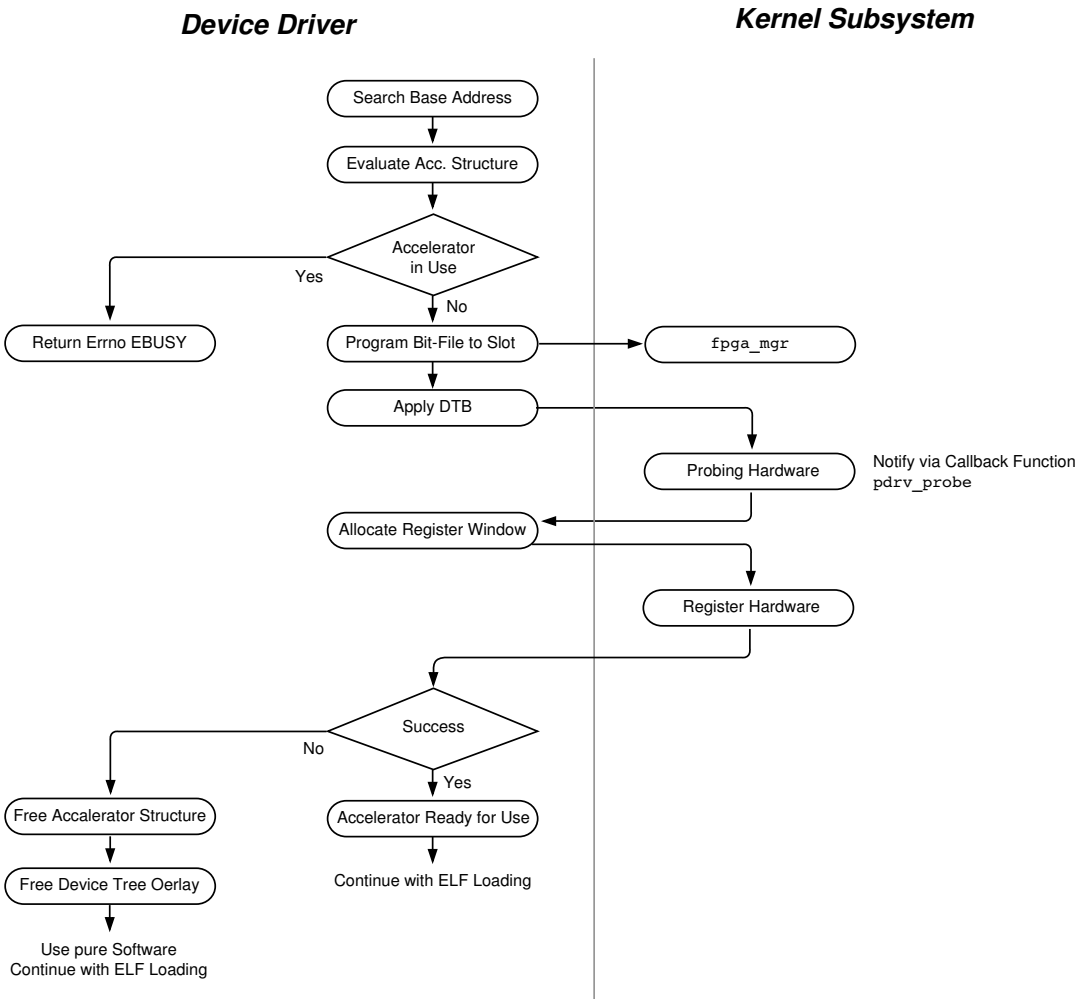

Figure D.4: Loading of bit-file and DTB in accelerator driver

This problem is circumvented by evaluating the tile ID field of the DTB at the beginning of the accelerator registration. Each tile ID corresponds to a base address. Consequently, tile IDs

must be given in a global and consistent way during compilation of the program. The current version of the plugin has no option for partial reconfiguration, thus, it always uses the whole FPGA (`fpga-tile-id: <0xffffffff>`) for all accelerators of an application. As long as the address space of those accelerators is generated in a consistent way no resource conflicts will occur. Nevertheless, the device driver already has a built-in support for reconfigurable tiles and multiple applications with accelerators.

The resulting mechanism of accelerator creation is shown in Figure D.4. It distinguishes between tasks that are carried out in the kernel subsystem (right side) and tasks that are carried out by the device driver (left side). As already mentioned, in the first step, the tile ID with the respective base address of the requesting accelerator is evaluated. In the following, all devices using the `compatible` field with the value "`tudos,hwacc`" are searched in the driver internal tree structure in order to find a violation with the base address of the requesting accelerator. If the accelerator does not exist and an FPGA slot is available, the FPGA is programmed with the bit-file; otherwise, the driver returns the busy error code (`EBUSY`). Afterwards, the DTB is used to make the accelerator available to the OS. If the final allocation of the accelerator fails for instance, due to a wrong DTB, the already allocated resources and the FPGA slot are marked as free again. As a result of this, the hardware design cannot be loaded and the program runs in pure software mode. The return status of the loading process is logged to the regular kernel message buffer.

E EVALUATION RESULTS

E.1 Comparison of Register-Allocation Strategies

The following tables contain the estimated frequency and resource consumption of eight accelerators for different register allocation strategies.

Speed Optimized Synthesis (Frequency in MHz)

Benchmark	Allocation Strategy						
	le_full	le_uid	le_simple	le_2	le_3	le_4	le_5
Base64	339.44	401.28	412.54	369.27	390.77	287.93	337.83
Bit Reverse	257.59	374.67	379.93	381.97	319.48	298.41	301.93
Grayscale	222.59	367.24	367.24	224.76	219.29	211.50	196.34
Haar Wavelet	412.88	406.83	427.89	421.40	412.88	412.88	412.88
Matrix Mult	152.43	336.13	349.28	195.65	187.58	172.08	171.46
FFT	182.41	349.28	349.28	215.51	181.98	222.02	177.71
IIR	161.34	330.36	349.28	211.23	193.72	164.77	167.89

Speed Optimized Synthesis (Resource Footprint in LUTs)

Benchmark	Allocation Strategy						
	le_full	le_uid	le_simple	le_2	le_3	le_4	le_5
Base64	670	581	575	728	613	684	640
Bit Reverse	401	246	438	439	433	454	418
Grayscale	305	245	221	276	303	301	316
Haar Wavelet	230	233	254	247	230	230	230
Matrix Mult	832	656	850	840	736	750	827
FFT	634	550	491	658	663	612	678
IIR	1087	823	801	1052	1079	1023	1092

Area Optimized Synthesis (Frequency in MHz)

Benchmark	Allocation Strategy						
	le_full	le_uid	le_simple	le_2	le_3	le_4	le_5
Base64	292.99	383.58	382.70	385.80	335.68	310.84	278.62
Bit Reverse	258.59	355.49	361.27	324.14	271.96	291.88	275.86
Grayscale	210.26	367.24	367.24	217.06	224.92	220.02	177.87
Haar Wavelet	414.93	402.25	411.69	403.38	414.93	414.93	414.93
Matrix Mult	135.28	319.18	313.67	217.29	190.51	169.26	172.29
FFT	172.35	349.28	349.28	196.46	173.16	210.08	179.14
IIR	148.94	324.78	349.28	184.12	191.79	159.23	152.85

Area Optimized Synthesis (Resource Footprint in LUTs)

Benchmark	Allocation Strategy						
	le_full	le_uid	le_simple	le_2	le_3	le_4	le_5
Base64	644	537	576	658	598	673	635
Bit Reverse	392	198	330	378	404	421	387
Grayscale	307	208	180	280	306	286	297
Haar Wavelet	237	229	238	242	237	237	237
Matrix Mult	814	671	605	750	740	739	800
FFT	633	510	518	626	655	613	634
IIR	1018	804	685	974	1052	991	1071

E.1.1 Benchmark Characteristics

This table shows the characteristics of the benchmarks for the register allocation experiment on GIMPLE-level.

Benchmark	#GIMPLE Variables	#Basic Blocks	#Branches	#ALU Operations	#DSP Operations
Base64	50	3	0	21	0
Bit Reverse	62	22	10	61	0
Grayscale	29	3	0	17	2
Haar Wavelet	14	3	0	7	0
Matrix Mult	59	3	0	25	8
FFT	51	6	1	22	4
IIR	79	6	1	29	11

E.2 State Machine Evaluation

The following tables show the number of states of the generated state machines for different HLS optimizations and give information about possible prefetching and conditional branches. Additionally, the GCC-estimated number of software instructions for each loop candidate is given.

AES

Function	State Machine States			Instr.	Cond.	Pre-
	plain	spec.	chain.	Software	Branches	fetching
addRoundKey	10	10	8	10		✓
expandDecKey1	23	23	23	28		✓
expandDecKey2	23	23	23	28		✓
mixColumns	56	28	23	63	✓	✓
mixColumnsInv	135	92	40	133	✓	✓
main1	4	4	4	6		
main2	4	4	4	6		

Base64

Function	State Machine States			Instr.	Cond.	Pre-
	plain	spec.	chain.	Software	Branches	fetching
decode1	36	36	27	35	✓	
decode2	40	40	34	39	✓	
encode	20	20	17	32		

Bilinear

Function	State Machine States			Instr.	Cond.	Pre-
	plain	spec.	chain.	Software	Branches	fetching
resize	30	30	18	48		

Binary Tree

Function	State Machine States			Instr. Software	Cond. Branches	Pre-fetching
	plain	spec.	chain.			
buildTree	12	12	8	12		
insertTree	12	12	12	8		
searchTree	14	14	14	10		

Bit Reverse

Function	State Machine States			Instr. Software	Cond. Branches	Pre-fetching
	plain	spec.	chain.			
reverseXor	56	56	38	71	✓	✓

CRC

Function	State Machine States			Instr. Software	Cond. Branches	Pre-fetching
	plain	spec.	chain.			
crcCalc	24	20	11	19	✓	✓

Euclidian

Function	State Machine States			Instr. Software	Cond. Branches	Pre-fetching
	plain	spec.	chain.			
euclid	8	5	4	7	✓	

Fletcher

Function	State Machine States			Instr. Software	Cond. Branches	Pre-fetching
	plain	spec.	chain.			
fletcher	14	14	10	17		✓

Grayscale

Function	State Machine States			Instr. Software	Cond. Branches	Pre-fetching
	plain	spec.	chain.			
grayscaleFilter	24	24	12	30		✓

Haar Wavelet

Function	State Machine States			Instr. Software	Cond. Branches	Pre-fetching
	plain	spec.	chain.			
waveletTransform	17	17	14	21		

IDCT

Function	State Machine States			Instr. Software	Cond. Branches	Pre-fetching
	plain	spec.	chain.			
dct1Dh	28	28	26	50		✓
dct1Dv	20	20	15	22		✓

IIR

Function	State Machine States			Instr. Software	Cond. Branches	Pre-fetching
	plain	spec.	chain.			
iir16	31	31	28	60		✓

JPEG

Function	State Machine States			Instr. Software	Cond. Branches	Pre-fetching
	plain	spec.	chain.			
load1	15	15	12	16		
load2	21	21	18	26		
checkHuffTables	19	19	14	12		✓
copyY	9	9	9	11		
quant	11	11	9	11		✓
decodeNextMCUsub	10	10	7	7		
huffCreate	16	16	11	20	✓	
idctCols	220	160	78	189	✓	
idctRows	83	28	23	101	✓	✓
upsampleCb	150	150	106	124	✓	✓
upsampleCr	150	150	106	124	✓	✓

Mandelbrot Set

Function	State Machine States			Instr. Software	Cond. Branches	Pre-fetching
	plain	spec.	chain.			
inMSet	12	12	8	14		

Matrix Multiplication

Function	State Machine States			Instr. Software	Cond. Branches	Pre-fetching
	plain	spec.	chain.			
matMul	30	30	26	55		✓

RSA

Function	State Machine States			Instr. Software	Cond. Branches	Pre-fetching
	plain	spec.	chain.			
mod	6	6	4	6		
squareRoot	9	9	7	11		

E.3 Performance Evaluation

This section presents the performance evaluation for each benchmark with respect to different HLS optimizations. The benchmarks for SpartanMC were evaluated after consecutively applying list scheduling (*plain*), speculative execution of branches (*spec.*), and chaining of operands (*chain.*). In addition, column five (*1:3*) shows the speedups for a host processor clock frequency of 50 MHz and an accelerator clock frequency of 150 MHz.

The evaluation of the same benchmarks for ARM[1] distinguishs between the runtime using random memory accesses and the prefetching mechanism. Note that prefetching is only applied on benchmarks that contain fully analyzable loops.

Benchmark	Speedups for SpartanMC				Slowdowns for ARM SoC		Kernel
	plain	spec.	chain.	1:3	random	prefetch.	
AES	1,11	1,16	1,28	1,57	705.45	730.17	7
Base64	1,94	1,94	2,41	8,70	70.95	–	3
Bilinear	1,94	1,94	3,42	10,96	58.52	–	1
Binary Tree	1,22	1,22	1,22	4,57	34.71	–	3
Bit Reverse	2,38	2,37	3,30	11,68	99.74	92.49	1
CRC	1,37	1,64	3,27	11,93	53.23	47.36	1
Euclidian	0,83	0,95	1,05	1,59	2316.41	–	1
Fletcher	0,90	0,90	1,06	1,38	363.33	341.46	1
Grayscale	2,03	2,03	4,37	13,16	169.82	116.15	1
Haar Wavelet	1,56	1,56	1,61	4,87	161.58	–	1
IDCT	1,79	1,76	2,18	7,61	151.50	116.21	2
IIR	2,44	2,44	2,69	7,98	238.35	221.71	1
JPEG	1,27	1,35	1,60	2,27	248.26	171.69	11
Mandelbrot Set	1,18	1,19	1,41	1,78	334.26	–	1
Matrix Mult	2,36	2,37	2,51	6,81	251.81	112.38	1
RSA	1,10	1,11	1,20	2,32	1915.25	–	2

[1] The runtime for the ARM SoC was measured after chaining.

E.4 Performance Estimation

The following tables show the hardware runtime of accelerator kernels (with wrapper function code) and an analytic estimation of the runtime for the respective accelerator. The estimation is only predictable for accelerators with bounded loops. Accelerators that contain conditional branches are shown with a minimum and maximum runtime.

The timing on the ARM SoC was measured with the C library function `clock_gettime(...)`. Each benchmark was measured at least 100 times, the average was taken for the evaluation.

SpartanMC

Benchmark	Function	HW Exec.	Estimated Runtime		Relative
		clk. cyc.	min. clk. cyc.	max. clk. cyc.	Deviation in %
AES	mixColumnsInv	340	317	377	-6.76 – 10.88
	expandDecKey	97	98	–	1.03
	mixColumns	200	193	209	-3.50 – 4.50
	addRoundKey	168	166	–	-1.19
	main	68	70	–	2.94
Bit Reverse	reverseXor	2444	2107	3259	-13.79 – 33.34
CRC	crcCalc	11264	10645	11893	-5.55 – 5.58
Fletcher	fletcher	450	454	–	-0.89
Grayscale	grayscaleFilter	1096	1099	–	0.27
IDCT	dct1Dv	4363	4363	–	0
	dct1Dh	1469	1467	–	0.14
IIR	iir16	1472	1482	–	0.68
JPEG	upsampleCr	1549	1470	1822	-8.65 – 24.38
	upsampleCb	1550	1416	1928	2.03
	idctRows	542	553	–	-8.65 – 24.39
	quant	668	672	–	0.59
	huffCreate	175	171	250	-2.29 – 42.86
	checkHuffTables	73	71	–	-2.74
Matrix Mult.	matMul	4554	4555	–	0.02

ARM SoC

Benchmark	Function	HW Exec. avg. in μs	Estimated Runtime min. in μs	Estimated Runtime max. in μs	Relative Deviation in %
AES	mixColumnsInv	0.6272	0.6373	0.6380	1.61 – 1.72
	expandDecKey	0.6345	0.6365	–	0.33
	mixColumns	0.6326	0.6360	0.6300	-0.41 – 0.53
	addRoundKey	0.6585	0.6427	–	-2.40
	main	0.8556	0.8070	–	-5.68
Bit Reverse	reverseXor	0.8919	0.8867	0.8924	-0.57 – 0.059
CRC	crcCalc	0.9260	0.8452	0.8590	-8.73 – -7.24
Fletcher	fletcher	0.8449	0.8590	–	1.67
Grayscale	grayscaleFilter	0.9306	0.8698	–	-6.53
IDCT	dct1Dv	1.1953	1.1646	–	-2.56
	dct1Dh	1.3564	1.3779	–	-1.58
IIR	iir16	0.9000	0.9364	–	4.04
JPEG	upsampleCr	0.8189	0.7700	0.7755	-5.97 – -5.30
	upsampleCb	0.9874	0.8740	0.7767	-11.49
	idctRows	0.8230	0.7710	–	-6.31 – -5.61
	quant	0.9986	0.9071	–	-9.16
	huffCreate	0.9213	0.8338	0.8347	-9.49 – -9.40
	checkHuffTables	0.9017	0.8032	–	-10.92
Matrix Mult.	matMul	1.4138	1.3489	–	-4.59

E.5 Data Transfer Evaluation

The following table give an impression of the latencies caused by the system calls of a DMA transfer using the PL330 controller. The experiment was carried out 600 times with an synthetic benchmark transferring a chunk of 1024x32 bit. The measured DMA transfers for read (rd) and write (wr) were divided into six parts.

copy Copy data to DMA buffers

prep Setup DMA registers for data transfer

sync Sync DMA controller (cache flush)

transfer Actual data transfer

wait_rdy Spinlock waits for completion of DMA transfer

back_transfer Copy data to userland

Besides the DMA transfer, the latencies for the `open(...)` and `close(...)` calls of the device were evaluated. These functions were used to setup/release the DMA buffers and channels. The first call of an accelerator causes a larger latency (*1st_call*).

Function	Time in μs			Clock Cycles (ARM at 667 MHz)		
	max.	avg.	min.	max.	avg.	min.
wr_cpy	8.769	6.38	5.23	5866	4267	3502
wr_prep	15.173	10.62	9.49	10150	7102	6346
wr_sync	21.980	16.61	14.63	14704	11112	9786
wr_transfer	202.796	55.72	31.29	135670	37275	20931
wr_waitrdy	1038.962	173.74	7.85	695065	116234	5250
wr_btransfer	11.880	10.11	8.68	7947	6766	5808
wr_total	1299.560	273.18	77.17	869405	182759	51624
rd_cpy	9.94	8.81	7.46	6649	5894	4989
rd_prep	12.33	10.71	9.66	8246	7162	6466
rd_sync	19.02	16.58	14.41	12722	11092	9641
rd_transfer	62.24	48.32	33.91	41640	32324	22687
rd_waitrdy	838.23	167.22	81.22	560777	111868	54334
rd_btransfer	16.29	13.33	11.64	10898	8917	7789
rd_total	958.05	264.96	158.31	640934	177259	105908
open	493.76	325.39	313.46	330323	217686	209702
close	302.74	289.36	210.35	202533	193587	140721
1st_call	210.43	182.23	963.24	140779	121915	64440

LIST OF FIGURES

1.1 Operating principle of the hybrid hardware/software work-flow 10

2.1 Gajski and Walker Y-chart showing high-level-synthesis 17
2.2 High-Level-Synthesis design steps 19
2.3 Liquid metal compilation and runtime system (derived from [33]) 28
2.4 Altera OpenCL hardware architecture [7] 30
2.5 Altera OpenCL tool-flow [22] 31
2.6 SystemC design of a 4-bit adder [114] 33
2.7 Vivado design-flow (adapted from [60] 35
2.8 Nymble hardware/software co-compilation [34] 37
2.9 Switch between hardware and software [34] 37
2.10 LegUp design-flow [17] 39
2.11 GCC plugin-based HLS work-flow for configurable architectures 45
2.12 GCC plugin-based HLS work-flow for FPGA-based architectures 46

3.1 Xilinx Artix FPGA layout [125, 113] 51
3.2 SpartanMC instruction types [139] 53
3.3 SpartanMC sliding register window [139] 54
3.4 SpartanMC memory Architecture [139] 56
3.5 SpartanMC tool-flow [139] 57
3.6 Zynq architectural overview [145] 59

4.1 GCC compilation flow with common optimizations, adapted from [80]. 63
4.2 Tree structure of an `PLUS_EXPR` with `TARGET_MEM_REF` 65
4.3 Example loop with memory accesses 67
4.4 GIMPLE transcript of the loop body showing memory references 67
4.5 C code with conditional 68
4.6 SSA values with *PHI function* 68
4.7 C code example 69
4.8 GENERIC code transcript 69
4.9 Example transcript of a CFG with GIMPLE basic blocks 71

4.10 Simplified call graph for `-O0` and `-O1` – `-O3` 74
4.11 Register pass-info for plugin 76

5.1 First GCC compilation run collecting loop-data 81
5.2 Example call graph *(A)* and inverted call graph for `bar` *(B)* 82
5.3 Source code of unit1.c [145] 84
5.4 Source code of unit2.c [145] 84
5.5 Analysis transcript file after compiling unit1.c and unit2.c [145] 84
5.6 Second GCC compilation run analyzing loop-data 85
5.7 Interface for compare functions [133] 86
5.8 Second compilation run with *synthesis pass* 88
5.9 Flow-chart of the pseudo-scheduling 92
5.10 Potential pointer aliasing 94
5.11 Four-dimensional matrix-vector product [135] 98
5.12 Memory access pattern produced by two nested slices S_0 and S_1 [135] 98

6.1 Overall software-flow of an accelerator call 116
6.2 Remove loop and CFG modifications 119
6.3 GIMPLE transcript of an accelerator call 121
6.4 SpartanMC assembler code with polling loop 121
6.5 Add function call to CFG 123
6.6 GIMPLE transcript of an accelerator function call 124
6.7 SpartanMC assembler code for accelerator function call 124
6.8 Control-flow of the accelerator function for Scheme I/II 131
6.9 Control-flow of the accelerator function for Scheme III 133
6.10 Nested C loop with three memory references 134
6.11 Qualitative execution trace for Listing 6.10 135
6.12 Generated and optimized alias test between output vector and matrix of the matrix-vector-product example 5.11 [135] 137
6.13 Input and output files of the GCC plugin 139
6.14 Decomposed compilation tool-flow and patching of base addresses 142

7.1 Flow-chart of the synthesis pass 145
7.2 Transcript of memory access in GIMPLE [131] 148

7.3 Transformed memory access in CDFG transcript [131] . . . 148
7.4 State machine structure (cf. [131]) . . . 150
7.5 State structure (cf. [131]) . . . 150
7.6 State branch structure (cf. [131]) . . . 150
7.7 Structure of the FSM . . . 152
7.8 Memory access state . . . 152
7.9 Generic datapath structure of the accelerator (cf. [131]) . . . 153
7.10 Merging of basic blocks for balanced *(A)* and unbalanced *(B)* branch structures [131] 156
7.11 Conditional branches – GIMPLE transcript . . . 158
7.12 Original DFG *(A)* and speculated DFG *(B)* [131] . . . 159
7.13 Original DFG *(A)* and chained DFG *(B)* . . . 163

8.1 Straight work-flow with accelerator stub module . . . 169
8.2 SpartanMC register interface . . . 170
8.3 Work-flow with manual accelerator integration . . . 171
8.4 ARM register interface . . . 171
8.5 Write and read transaction accelerator – AXI4 IPIC IP . . . 172
8.6 SpartanMC memory interface . . . 175
8.7 ARM memory interface . . . 177
8.8 ARM FIFO interface . . . 179
8.9 Address mapping for SpartanMC *(A)* and ARM *(B)* . . . 181

9.1 Fletcher-32 algorithm . . . 184
9.2 Datapath and FSM of the Fletcher-32 algorithm . . . 185
9.3 The generated fill-loop within the accelerator function . . . 186
9.4 Binary tree search algorithm . . . 187
9.5 GIMPLE graph for the binary tree search . . . 188
9.6 Datapath and FSM of the binary tree search . . . 189
9.7 Analyzable loops and partial analyzable loops . . . 191
9.8 Not analyzable array subscripts (`unknown_[1,2,3]`) . . . 192
9.9 Normalized frequency plotted against the respective normalized resource consumption for different register-allocation strategies applied on *FFT, IIR, Grayscale and Matrix Mult.* . . . 199

9.10 Resource and frequency comparison of different register-allocation strategies (LUTs – small values are better; frequency – large values are better) 200
9.11 Comparison of FSM states after applying different HLS optimzations 202
9.12 Whole application improvements for SpartanMC 204
9.13 Whole application evaluation for the ARM SoC optimized with speculation and chaining (small values are better) 206
9.14 Relative error of performance analysis for SpartanMC benchmarks 209
9.15 Relative error of performance analysis for ARM SoC benchmarks 210
9.16 Analysis of subtask execution time for ARM SoC benchmarks 211

A.1 Access to array of structures [135] 221
A.2 High GIMPLE example 223
A.3 Low GIMPLE example 223
A.4 Complete transcript file of *unit1.c* and *unit2.c* [145] 224

B.1 SpartanMC pipeline architecture (IF, ID, OF1) [139] 226
B.2 SpartanMC pipeline architecture (OF2, EX, MEM1, MEM2, WB) [139] 227

D.1 Structure of an ELF file [136] 238
D.2 ZwoELF header definition [136] 239
D.3 Example DTS file for two accelerators 242
D.4 Loading of bit-file and DTB in accelerator driver 245

LIST OF TABLES

2.-8 Overview of Related HLS Projects (1) 41
2.-9 Overview of Related HLS Projects (2) 42

5.1 Alias matrix for Listing 5.11 112

9.1 Benchmark results for using LibISL for alias tests. [135] 194

A.1 Fields of the memory access structure for Listing A.1 221

B.1 SpartanMC instruction coding matrices – Main Matrix (IR 17-13) 225
B.2 SpartanMC instruction coding matrices – Submatrix Special 1 (IR 4-0) 225
B.3 SpartanMC instruction coding matrices – Submatrix Special 2 (IR 4-0) 225

C.1 GCC plugin parameters 229
C.2 Delay times for different operations 232
C.3 Fields of the `memref_symbol` runtime structure [135] 232
C.4 Fields of the `memref_access` runtime structure [135] 232
C.5 Fields of the `memref_slice` runtime structure [135] 233
C.6 Assorted GIMPLE statements 233
C.7 All GIMPLE statements and their scope of usage 234
C.8 Tree types 235
C.9 GIMPLE expressions 236

LIST OF ALGORITHMS

1 Function `analyze_data_access_base` [135] 109

2 Function `analyze_data_access_base_chrec` [135] 110

3 Function `normalize_base` [135] 111

4 Identify input variables 117

5 Identify loop exit variables 118

6 Function `generate_state_for_block` 151

7 Merging basic blocks 155

8 Generation of a DFG [131] 160

9 List scheduling [95] 161

10 Chaining [134] 164

11 Kernel internal binary loader 240

RELATED PUBLICATIONS ON HLS APPROACHES

[1] PandA Bambu Framework. `http://panda.dei.polimi.it`. Accessed: 2015-09-30.

[2] IEEE Unapproved IEEE Draft Standard for System Verilog: Unified Hardware Design, Specification and Verification Language (Superseded by P1800/D6). *IEEE Std P1800/D6*, 2005.

[3] C-to-Silicon Compiler High-Level Synthesis – Automated High-Level Synthesis for Design and Verification. Documentation, 2011.

[4] IEEE Standard for Standard SystemC Language Reference Manual. *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pages 1–638, Jan 2012.

[5] Carte++ Application Development Process. White Paper, 2015.

[6] M. Aldham, J. Anderson, S. Brown, and A. Canis. Low-cost hardware profiling of run-time and energy in FPGA embedded processors. In *Application-Specific Systems, Architectures and Processors (ASAP), 2011 IEEE International Conference on*, pages 61–68, Sept 2011.

[7] Altera Corporation. Implementing FPGA Design with the OpenCL Standard. White Paper 01172-2.0, 2011.

[8] J. Auerbach, D.F. Bacon, P. Cheng, and R. Rabbah. Lime: A java-compatible and synthesizable language for heterogeneous architectures. In *ACM*, pages 89–108, 2010.

[9] C. Baaij, M. Kooijman, J. Kuper, A. Boeijink, and M. Gerards. CλaSH: Structural Descriptions of Synchronous Hardware using Haskell. In *Proceedings of the 13th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools*, pages 714–721. IEEE Computer Society, September 2010.

[10] N. Bansal, S. Gupta, N. Dutt, A. Nicolau, and R. Gupta. Network topology exploration of mesh-based coarse-grain reconfigurable architectures. In *Design, Automation and Test in Europe (DATE), 2004*, pages 474–479, 2003.

[11] Y. Ben-Asher and N. Rotem. Synthesis for variable pipelined function units. In *System-on-Chip, 2008. SOC 2008. International Symposium on*, pages 1–4, Nov 2008.

[12] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware Design in Haskell. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, pages 174–184. ACM, 1998.

[13] T. Bollaert. Catapult Synthesis: A Practical Introduction to Interactive C Synthesis. In P. Coussy and A. Morawiec, editors, *High-Level Synthesis*, pages 29–52. Springer Netherlands, 2008.

[14] M. Bowen. Handel-C – Language Reference Manual. Documentation, 2007.

[15] M. Budiu. *Spatial Computation*. Phd thesis, Carnegie Mellon University, 2003.

[16] T. Callahan. Kernel Formation in Garpcc. In *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, FCCM '03, pages 308–. IEEE Computer Society, 2003.

[17] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J.H. Anderson, S. Brown, and T. Czajkowski. LegUp: High-Level synthesis for FPGA-based processor/accelerator systems. In *ACM/SIGDA*, pages 33–36, 2011.

[18] A. Canis, J. Choi, B. Fort, R. Lian, O. Huang, N. Calagar, M. Gort, J.J. Qin, M. Aldham, T. Czajkowski, D. Brown, and J. Anderson. From Software to Accelerators with LegUp High-level Synthesis. In *Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '13, pages 18:1–18:9, Piscataway, NJ, USA, 2013. IEEE Press.

[19] J. Cong, F. Yiping, G. Han, Wei Jiang, and Zhiru Zhang. Platform-Based Behavior-Level and System-Level Synthesis. In *SOC Conference, 2006 IEEE International*, pages 199–202, Sept 2006.

[20] P. Coussy, G. Lhairech-Lebreton, D. Heller, and E. Martin. GAUT – A free and open source high-level synthesis tool. In *IEEE Design Automation and Test in Europe*, 2010.

[21] T.S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D.P. Singh. From opencl to high-performance hardware on FPGAS. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 531–534, Aug 2012.

[22] T.S. Czajkowski, D. Neto, M. Kinsner, U. Aydonat, J. Wong, D. Denisenko, P. Yiannacouras, J. Freeman, D.P. Singh, and S.D. Brown. OpenCL for FPGAs: Prototyping a compiler. In *Int'l Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pages 3–12, 2012.

[23] C. Dase, J.S. Falcon, and B. MacCleery. Motorcycle control prototyping using an FPGA-based embedded control system. *Control Systems, IEEE*, 26(5):17–21, Oct 2006.

[24] B. Fort, A. Canis, J. Choi, N. Calagar, R. Lian, S. Hadjis, Yu Ting Chen, M. Hall, B. Syrowik, T. Czajkowski, S. Brown, and J. Anderson. Automating the Design of Processor/Accelerator Embedded Systems with LegUp High-Level Synthesis. In *EUC*, pages 120–129, 2014.

[25] H. Gadke and A. Koch. Comrade - A Compiler for Adaptive Computing Systems using a Novel Fast Speculation Technique. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 503–504, Aug 2007.

[26] S. Gatzka and C. Hochberger. The amidar class of reconfigurable processors. *J. Supercomput.*, 32(2):163–181, May 2005.

[27] G. Genest, R. Chamberlain, and R. Bruce. Programming an FPGA-based Super Computer Using a C-to-VHDL Compiler: DIME-C. In *Adaptive Hardware and Systems, 2007. AHS 2007. Second NASA/ESA Conference on*, pages 280–286, 2007.

[28] M. Gokhale, J. Stone, J. Arnold, and M. Kalinowski. Stream-oriented FPGA computing in the Streams-C high level language. In *Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on*, pages 49–56, 2000.

[29] S.T. Gurumani, H. Cholakkal, L. Yun, K. Rupnow, and C. Deming. High-level synthesis of multiple dependent CUDA kernels on FPGA. In *Design Automation Conference (ASP-DAC), 2013 18th Asia and South Pacific*, pages 305–312, Jan 2013.

[30] M. Haldar, A. Nayak, N. Shenoy, A. Choudhary, and P. Banerjee. FPGA hardware synthesis from MATLAB. In *VLSI Design, 2001. International Conference on*, pages 299–304, 2001.

[31] R. Harr. The Nimble Compiler for Agile Hardware: A Research Platform. *System Synthesis, International Symposium on*, 0, 2000.

[32] J.R. Hauser and J. Wawrzynek. Garp: A MIPS Processor with a Reconfigurable Coprocessor. In *Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines*, FCCM '97, pages 12–. IEEE Computer Society, 1997.

[33] S. Huang, A. Hormati, D.F. Bacon, and R. Rabbah. Liquid metal: Object-oriented programming across the hardware/software boundary. In J. Vitek, editor, *ECOOP 2008 – Object-Oriented Programming*, volume 5142, pages 76–103. 2008.

[34] J. Huthmann, B. Liebig, J. Oppermann, and A. Koch. Hardware/software co-compilation with the Nymble system. In *ReCoSoC*, pages 1–8, 2013.

[35] P.O. Jääskeläinen, de C.S. La Lama, P. Huerta, and J.H. Takala. OpenCL-based design methodology for application-specific processors. In *Embedded Computer Systems (SAMOS), 2010 International Conference on*, pages 223–230. IEEE, 2010.

[36] N. Kavvadias. HercuLeS 1.0 reference manual. Menual, 2013.

[37] N. Kavvadias and K. Masselos. Hardware Design Space Exploration Using HercuLeS HLS. In *Proceedings of the 17th Panhellenic Conference on Informatics*, PCI '13, pages 195–202, New York, NY, USA, 2013. ACM.

[38] D.W. Knapp. *Behavioral Synthesis: Digital System Design Using the Synopsys Behavioral Compiler*. Electronic and digital design. Prentice Hall PTR, 1996.

[39] H. Lange and A. Koch. An Execution Model for Hardware/Software Compilation and its System-Level Realization. In *FPL*, pages 285–292, 2007.

[40] S.J. Lee, D.K. Raila, and V.V. Kindratenko. LLVM-CHiMPS: Compilation Environment for FPGAs Using LLVM Compiler Infrastructure and CHiMPS Computational Model. In *Reconfigurable Systems Summer Institute 2008 (RSSI'08)*, 2008.

[41] Y. Li and M. Leeser. Hml, a novel hardware description language and its translation to vhdl. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 8(1):1–8, 2000.

[42] D. MacMillen. Nimble Compiler Environment for Agile Hardware. report, 1998.

[43] E. Martin, O. Sentieys, H. Dubois, and J.L. Philippe. GAUT: An architectural synthesis tool for dedicated signal processors. In *Design Automation Conference, 1993, with EURO-VHDL '93. Proceedings EURO-DAC '93., European*, pages 14–19, Sep 1993.

[44] J. Matthews, B. Cook, and J. Launchbury. Microprocessor specification in Hawk. In *Computer Languages, 1998. Proceedings. 1998 International Conference on*, pages 90–101, May 1998.

[45] Mentor Graphics. Designing High Performance DSP Hardware Using Catapult C Synthesis and The Altera Accelerated Libraries. White Paper, 2007.

[46] R. Nikhil. Bluespec: A General-Purpose Approach to High-Level Synthesis Based on Parallel Atomic Transactions. In P. Coussy and A. Morawiec, editors, *High-Level Synthesis*, pages 129–146. Springer Netherlands, 2008.

[47] H. Nikolov, T. Stefanov, and E. Deprettere. Systematic and Automated Multiprocessor System Design, Programming, and Implementation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(3):542–555, March 2008.

[48] A. Papakonstantinou, K. Gururaj, J.A. Stratton, D.C. Chen, J. Cong, and W.-M.W. Hwu. High-performance cuda kernel execution on fpgas. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 515–516, New York, NY, USA, 2009. ACM.

[49] M.M. Pereira and L. Carro. Dynamic Reconfigurable Computing: The Alternative to Homogeneous Multicores under Massive Defect Rates. *International Journal of Reconfigurable Computing*, 2011:21 – 37, 2011.

[50] C. Pilato and F. Ferrandi. Bambu: A modular framework for the high level synthesis of memory-intensive applications. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pages 1–4, Sept 2013.

[51] F. Plavec. *Stream Computing on FPGAs*. Phd thesis, University of Toronto, 2010.

[52] A.R. Putnam, D. Bennett, E. Dellinger, J. Mason, and P. Sundararajan. CHiMPS: A High-level Compilation Flow for Hybrid CPU-FPGA Architectures. In *Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays*, FPGA '08, pages 261–261. ACM, 2008.

[53] S. O Settle. High-performance Dynamic Programming on FPGAs with OpenCL, 2013.

[54] S. Sharma and W. Chen. Using Model-Based Design to Accelerate FPGA Development for Automotive Applications. In *Proceedings of SAE International J. Passenger Cars – Electron. Electr. Systems*, pages 150–158. SAE, 2009.

[55] M. Sheeran. muFP, a Language for VLSI Design. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 104–112. ACM, 1984.

[56] B. Shehan, R. Jahr, S. Uhrig, and T. Ungerer. Reconfigurable Grid ALU Processor: Optimization and Design Space Exploration. In *DSD*, pages 71 – 79. IEEE, September 2010.

[57] S. Singh and D.J. Greaves. Kiwi: Synthesis of FPGA Circuits from Parallel Programs. In *Proceedings of the 2008 16th International Symposium on Field-Programmable Custom Computing Machines*, FCCM '08, pages 3–12. IEEE Computer Society, 2008.

[58] M.C. Smith, J.S. Vetter, and X. Liang. Accelerating scientific applications with the src-6 reconfigurable computer: Methodologies and analysis. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 157b–157b, April 2005.

[59] D. Soderman and Y. Panchul. Implementing C algorithms in reconfigurable hardware using C2Verilog. In *FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on*, pages 339–342, Apr 1998.

[60] T. Feist. Vivado Design Suite. White Paper, 2012.

[61] Impulse Accelerated Technology. Accelerate Software Algorithms on FPGAs. `http://www.impulseaccelerated.com`. Accessed: 2015-09-30.

[62] M. Thompson, H. Nikolov, T. Stefanov, A.D. Pimentel, C. Erbas, S. Polstra, and E.F. Deprettere. A Framework for Rapid System-Level Exploration, Synthesis, and Programming of Multimedia MP-SoCs. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2007 5th IEEE/ACM/IFIP International Conference on*, pages 9–14, Sept 2007.

[63] J.J.P. van Zuijlen. Feasibility study on Handel C for Embedded Control. Pre-msc report 014ce2007, University of Twente, May 2007.

[64] J. Villarreal, A. Park, W. Najjar, and R. Halstead. Designing Modular Hardware Accelerators in C with ROCCC 2.0. In *FCCM*, pages 127–134, 2010.

[65] F. Winterstein, S. Bayliss, and G.A. Constantinides. High-level synthesis of dynamic data structures: A case study using Vivado HLS. In *Field-Programmable Technology (FPT), 2013 International Conference on*, pages 362–365, Dec 2013.

[66] Inc. Y Explorations. eXCite – C to RTL Behavioral Synthesis. `http://www.yxi.com/products.php`, 2013. Accessed: 2015-10-05.

[67] Y. Zhu, Y. Liu, D. Zhang, S. Li, P. Zhang, and T. Hadley. Acceleration of pedestrian detection algorithm on novel C2RTL HW/SW Co-design platform. In *Green Circuits and Systems (ICGCS), 2010 International Conference on*, pages 615–620, 2010.

RELATED PUBLICATIONS ON THE GCC FRAMEWORK AND COMPILER TECHNIQUES

[68] *Compilers*. Pearson, second edition, 2007.

[69] K. Aardal, C.AJ. Hurkens, and A.K. Lenstra. Solving a system of linear diophantine equations with lower and upper bounds on the variables. *Mathematics of Operations Research*, 25(3):427–442, 2000.

[70] O. Bachmann, P.S. Wang, and E.V. Zima. Chains of recurrences–a method to expedite the evaluation of closed-form functions. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation*, ISSAC '94, pages 242–249, New York, NY, USA, 1994. ACM.

[71] D. Berlin and D. Edelsohn. High-level Loop Optimizations for GCC. In *In Proceedings of the 2004 GCC Developers Summit*, pages 37–54, 2004.

[72] D. Novillo. Design and Implementation of Tree SSA. In *Proceedings of the 2004 GCC Developers Summit*, pages 119–130, 2004.

[73] Inc. Free Software Foundation. GNU BinUtils. `https://sourceware.org/binutils/`, 2016. Accessed: 2016-01-20.

[74] L.J. Hendren, C. Donawa, M. Emami, G.R. Gao, J., and B. Sridharan. Designing the McCAT Compiler Based on a Family of Structured Intermediate Representations. In *In Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing, number 757 in LNCS*, pages 406–420. Springer-Verlag, 1992.

[75] Y. Huang, L. Peng, C. Wu, Y. Kashnikov, J. Rennecke, and G. Fursin. Transforming GCC into a Research-Friendly Environment: Plugins for Optimization Tuning and Reordering, Function Cloning and Program Instrumentation. In *2nd International Workshop on GCC Research Opportunities (GROW'10)*, Pisa, Italy, January 2010.

[76] J. Hubička. The GCC call graph module: A framework fir inter-procedural optimization. In *Proceedings of the 2004 GCC Developers Summit*, pages 65–78, 5 2004.

[77] U. Khedker. Plugin Mechanism in GCC. Technical report, GCC Resource Center, Department of Computer Science and Engineering, Indian Institute of Technology, Bombay, 2014.

[78] J. Merrill. GENERIC and GIMPLE: A New Tree Representation for Entire Functions. In *In Proceedings of the 2003 GCC Summi*, pages 171–180, 2003.

[79] D. Novillo. Tree SSA - A New Optimization Infrastructure for GCC. In *in 'Proceedings of the 2003 GCC Summit*, pages 181–194, 2003.

[80] D. Novillo. GCC—An Architectural Overview, Current Status, and Future Directions. In *Linux Symposium. Vol. 2. 2006*, pages 185–200, 2006.

[81] S. Pop, A. Cohen, C. Bastoul, S. Girbal, G.-A. Silber, and N. Vasilache. Graphite: Polyhedral analyses and optimizations for gcc. In *Proceedings of the 2006 GCC Developers Summit*, page 2006. Citeseer, 2006.

[82] R.M. Stallman and the GCC Developer Community. *GNU Compiler Collection Internals (Version 4.8.1)*. Free Software Foundation, 2013.

[83] S. Verdoolaege. isl: An Integer Set Library for the Polyhedral Model. In K. Fukuda, J. Hoeven, M. Joswig, and N. Takayama, editors, *Mathematical Software – ICMS 2010*, volume 6327 of *Lecture Notes in Computer Science*, pages 299–302. Springer Berlin Heidelberg, 2010.

UNCATEGORIZED RELATED PUBLICATIONS

[84] IEEE Standard for Verilog Hardware Description Language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, pages 1–560, 2006.

[85] IEEE Standard VHDL Language Reference Manual. *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*, pages c1–626, Jan 2009.

[86] Taming the Challenges of 20nm Custom/Analog Design. Technical report, Cadence Design Systems, Inc., 2012.

[87] Inc. Aeroflex. *Processors*, 2015. Accessed: 2015-11-20.

[88] A. Agne, M. Happe, A. Keller, E. Lubbers, B. Plattner, M. Platzner, and C. Plessl. ReconOS: An Operating System Approach for Reconfigurable Computing. *Micro, IEEE*, 34(1):60–71, Jan 2014.

[89] Altera. Nios II Processors - World's Most Versatile Embedded Processors. `http://www.altera.com/products/ip/processors/nios2/ni2-index.html`, 2015. Accessed: 2015-10-10.

[90] ARM Limited. *PrimeCell DMA Controller (PL330) – Technical Reference Guide*, 2007.

[91] Avnet, Inc. *Zedboard – Zynq Evaluation and Development Hardware User's Guide –Version 2.2*, 2014.

[92] M. Barr and A. Massa. *Programming Embedded Systems: With C and GNU Development Tools*. O'Reilly Media, 2006.

[93] Cadance Design Systems, Inc. *Xtensa LX6 Customizable DPU – Tensilica Datasheet*, 2014.

[94] C.E. Cummings. *Verilog-2001 Behavioral and Synthesis Enhancements*. Sunburst Design, Inc., 2002.

[95] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill series in electrical and computer engineering. McGraw-Hill, 1994.

[96] Inc. Free Software Foundation. GNU Automake (Version 1.15) – Manual. `https://www.gnu.org/software/automake/manual/html_node/index.html#Top`, 2014. Accessed: 2015-11-20.

[97] Inc. Free Software Foundation. eCos – Homepage. `http://ecos.sourceware.org`, 2015. Accessed: 2015-12-05.

[98] D.D. Gajski and R. Kuhn. Guest Editors' Introduction. In *New VLSI Tools*, Computer 16(12), pages 11–14, December 1983.

[99] Yan Han and E. Oruklu. Real-time traffic sign recognition based on zynq fpga and arm socs. In *EIT*, pages 373–376, 2014.

[100] J. Kodis. Fletcher's checksum error correction at a fraction of the cost. *j-DDJ*, 17(5):32, 34, 36, 38, May 1992.

[101] G. Kornaros, K. Harteros, I. Christoforakis, and M. Astrinaki. I/O Virtualization Utilizing an Efficient Hardware System-Level Memory Management Unit. In *System-on-Chip (SoC), 2014 International Symposium on*, pages 1–4, Oct 2014.

[102] F.J. Kurdahi and A.C. Parker. REAL: A Program for REgister ALlocation. In *Design Automation, 1987. 24th Conference on*, pages 210–215, June 1987.

[103] H. Lange and A. Koch. Low-Latency High-Bandwidth HW/SW Communication in a Virtual Memory Environment. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pages 281–286, Sept 2008.

[104] H. Lange and A. Koch. Architectures and Execution Models for Hardware/Software Compilation and Their System-Level Realization. *Computers, IEEE Transactions on*, 59(10):1363–1377, Oct 2010.

[105] H. Lange, T. Wink, and A. Koch. MARC II: A Parametrized Speculative Multi-Ported Memory Subsystem for Reconfigurable Computers. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pages 1–6, March 2011.

[106] Z. Li, J. Li, Y. Zhao, C. Rong, and J. Ma. A soc design and implementation of h.264 video encoding system based on fpga. In *IHMSC*, volume 2, pages 321–324, 2014.

[107] Mentor Graphics, Corp. *Functional Verification Study*, 2014.

[108] P. Metzgen. Optimizing a high performance 32-bit processor for programmable logic. In *ISSOC 2004*, page 13. IEEE Computer Society, 2004.

[109] Microsemi Corporation. *SmartFusion: FPGA Fabric Synthesis Guidelines – Application Note AC361*, 2011.

[110] University of Cambridge. The Tiger MIPS processor. `https://www.cl.cam.ac.uk/teaching/0910/ECAD+Arch/mips.html`, 2010. Accessed: 2015-09-30.

[111] Inc. Qualcomm Technologies. Qualcomm Snapdragon TM 4K Ultra HD. `https://www.qualcomm.com/documents/snapdragon-4k-datasheet`, 2014. Accessed: 2015-09-30.

[112] L. Ramachandran, F. Vahid, S. Narayan, and D.D. Gajski. Semantics and synthesis of signals in behavioral VHDL. In *Design Automation Conference, 1992., EURO-VHDL '92, EURO-DAC '92. European*, pages 616–621, Sep 1992.

[113] S. Ramalingam. Xilinx Stacked Silicon Interconnect Technology – Producing a New Class of High-capacity, Resource-rich FPGAs. *ISSUU - MEPTEC Report*, pages 22–25, 2011.

[114] D.K. Tala. *SystemC Adder*. Accessed: 2015-11-20.

[115] J. Teich. *Digitale Hardware/Software-Systeme: Synthese und Optimierung*. Springer-Lehrbuch. Springer Berlin Heidelberg, 2013.

[116] J. Trommer, M. Raitza, A. Heinzig, T. Baldauf, T. Mikoajick, M. Völp, and W.M. Weber. Reconfigurable Nanowire Transistors with Multiple Independent Gates for Efficient and Programmable Combinational Circuits. In *In publication at Conference on Design, Automation & Test in Europe*, DATE '16, 2016.

[117] P. Vogel, A. Marongiu, and L. Benini. Lightweight Virtual Memory Support for Many-Core Accelerators in Heterogeneous Embedded SoCs. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2015 International Conference on*, pages 45–54, Oct 2015.

[118] R.A. Walker and D.E. Thomas. A model of design representation and synthesis. In *Proceedings of the 22Nd ACM/IEEE Design Automation Conference*, DAC '85, pages 453–459. IEEE Press, 1985.

[119] Xilinx, Inc. *MicroBlaze Processor Reference Guide Embedded Development Kit EDK 8.2i*, June 2006.

[120] Xilinx, Inc. *LogiCORE IP AXI Slave Burst (IPIC) – Product Specification (DS769 v1.00.a)*, 2010.

[121] Xilinx, Inc. *LogiCORE IP Fast Simplex Link (FSL) V20 Bus – Product Specification (DS449 v2.11c)*, 2010.

[122] Xilinx, Inc. *AXI Reference Guide (UG761 v13.4)*, 2012.

[123] Xilinx, Inc. *Platform Studio User Guide – Embedded Development Kit EDK 6.2i (UG113 v1.0)*, 2012.

[124] Xilinx, Inc. *7 Series DSP48ES Slice – User Guide (UG479 v1.8)*, 2014.

[125] Xilinx, Inc. *7 Series FPGAs Configurable Logic Block – User Guide (UG474 v1.7)*, 2014.

[126] Xilinx, Inc. *7 Series FPGAs Overview – Product Specification (DS180 v1.17)*, 2015.

[127] Xilinx, Inc. *LogiCORE IP AXI Central Direct Memory Access v4.1 – Product Guide (PG034)*, 2015.

[128] Xilinx, Inc. *LogiCORE IP AXI DataMover v5.1 – Product Guide (PG022)*, 2015.

[129] Xilinx, Inc. *Zynq-7000 All Programmable SoC –Manual (UG585 v1.10)*, 2015.

RELATED STUDENT WORKS

[130] J. Hoyer. Automatische Erzeugung von Hardware zur Applikationsbeschleunigung. Studienarbeit, Technische Universität Dresden, 2012.

[131] J. Hoyer. Erweiterung automatisch erzeugter Prozessorerweiterungen um Speicherzugriffe. Diplomarbeit, Technische Universität Dresden, 2013.

[132] C. Lohse. Verbesserung der GCC-basierten Applikationsanalyse zur Synthese von Prozessor-Erweiterungen für den SpartanMC. Studienarbeit, Technische Universität Dresden, 2014.

[133] M. Raitza. Untersuchung von Möglichkeiten zur Beschleunigung von Anwendungen für den SpartanMC anhand des GCC. Studienarbeit, Technische Universität Dresden, 2011.

[134] J. Rohde. Erweiterung von Hardwarebeschleunigern um Chaining. Studienarbeit, Technische Universität Darmstadt, 2014.

[135] J. Wielicki. Entwicklung einer GCC basierten Analyse von Speicherzugriffen. Bachelor's Thesis, Technische Universität Dresden, 2015.

[136] A. Wiese. Integration and Management of Automatically Generated Hardware Accelerators on the Linux OS. Diplomarbeit, Technische Universität Dresden, 2015.

OWN PUBLICATIONS

[137] R. Backasch, G. Hempel, S. Werner, S. Groppe, and T. Pionteck. Identifying homogenous reconfigurable regions in heterogeneous FPGAs for module relocation. In *ReConFigurable Computing and FPGAs (ReConFig), International Conference on*, pages 1–6, December 2014.

[138] R. Backasch, G. Hempel, S. Werner, S. Groppe, and T. Pionteck. An Architectural Template for Composing Application Specific Datapaths at Runtime. In *ReConFigurable Computing and FPGAs (ReConFig), International Conference on*, Dec 2015.

[139] G. Hempel and C. Hochberger. A resource optimized Processor Core for FPGA based SoCs. In *Digital System Design Architectures, Methods and Tools (DSD), 10th Euromicro Conference on*, pages 51–58, August 2007.

[140] G. Hempel and C. Hochberger. A resource optimized SoC Kit for FPGAs. In *Field Programmable Logic and Applications (FPL), International Conference on*, pages 761–764, August 2007.

[141] G. Hempel, C. Hochberger, and A. Koch. A Comparison of Hardware Acceleration Interfaces in a Customizable Soft Core Processor. In *Field Programmable Logic and Applications (FPL), International Conference on*, pages 469–474, August 2010.

[142] G. Hempel, C. Hochberger, and M. Raitza. Towards GCC-based automatic soft-core customization. In *Field Programmable Logic and Applications (FPL), International Conference on*, pages 687–690, August 2012.

[143] G. Hempel, J. Hoyer, T. Pionteck, and C. Hochberger. Register allocation for high-level synthesis of hardware accelerators targeting FPGAs. In *Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), International Workshop on*, pages 1–6, July 2013.

[144] G. Hempel, M. Vogt, J. Castrillon, and C. Hochberger. Software-Backed Caching and Virtual Addressing for Generated Accelerators in SoC FPGAs. In *Software Engineering and Advanced Applications - Work in Progress Session (DSD), 41th Euromicro Conference on*, August 2015.

[145] M. Vogt, G. Hempel, J. Castrillón, and C. Hochberger. GCC-Plugin for Automated Accelerator Generation and Integration on Hybrid FPGA-SoCs. *CoRR*, abs/1509.00025, August 2015.

Declaration

I confirm that I independently prepared this thesis, and that I used only the indicated references and auxiliary means.

Dresden, December 20th, 2017 Gerald Hempel