UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Compiler-Driven Dynamic Reconfiguration of Architectural Variants

Dissertation

A thesis submitted to the
Faculty of Electrical Engineering, Computer Science, and Mathematics

of the

University of Paderborn

in partial fulfillment of the requirements
for the degree of Dr. rer. nat.

by

Michael Hußmann

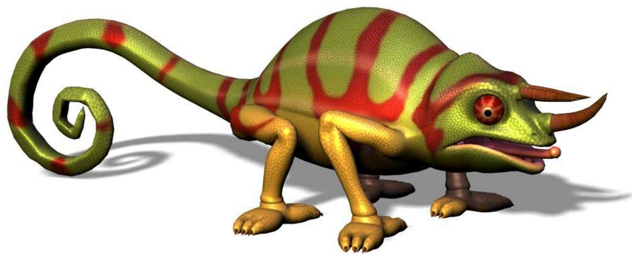
Paderborn, April 2008

Date of oral examination:
28.04.2008

Members of committee

Prof. Dr. Uwe Kastens (Chair, Reviewer)	University of Paderborn
Prof. Dr.-Ing. Ulrich Rückert (Reviewer)	University of Paderborn
Prof. Dr. Marco Platzner	University of Paderborn
Prof. Dr. Franz-Josef Rammig	University of Paderborn
Dr. Matthias Fischer	University of Paderborn

Compiler-Driven Dynamic Reconfiguration of Architectural Variants



CHARISMA

Compiler Handles
Architectural
Reconfiguration
Integrating SIMD
MIMD Automatically

CAPRICoRn

Compiler Anticipated
Processor Register
Inter-Connected
Reconfiguration

Acknowledgements

Doctoral work is never pursued by a single person. A lot of people have supported me for the last three years in different ways.

First of all, I thank the International Graduate School of Dynamic Intelligent Systems for the doctoral scholarship and the opportunity to get my Ph.D. in a comparatively short time.

I am very grateful to my advisor, Prof. Dr. Uwe Kastens, who has always given me his encouragement, guidance, and support when I encountered apparently insuperable challenges or even felt burnt out. In particular, he has proof-read my thesis or parts of it multiple times and improved its quality significantly by his helpful comments. I apologize profusely for the immense length of the thesis and the enormous effort in reading it.

A great deal of thanks deserves to him and my second supervisor Prof. Dr. Ulrich Rückert, who motivated me to start working on a Ph.D. project in the Graduate School. Together with Christian Liß, Mario Porrmann, and Michael Thies, we have evolved the first ideas of the topic.

I would like to thank Prof. Dr. Franz Josef Rammig for being my third supervisor in the Graduate School and his fruitful comments during my intermediate examination and talks given in the Graduate School. Additional thanks goes to the members of my committee and the reviewers for their comments, time, and support.

Without Michael Thies, the implementation of the CoBRA compiler would have been impossible. I am much obliged for his extensive, competent, and omnipresent support in technical and conceptual questions and apologize for making demands on his valuable time.

I am in debt to the members of the project group "Automatic Utilization of Multimedia Instruction for Reconfigurable Processing Clusters", who realized the first prototypical implementation of the Single Instruction Multiple Data (SIMD)/Multiple Instruction Multiple Data (MIMD) reconfiguration (see Part III) together with myself. Especially, I thank Ralf Bettentrup and Manuel Wickert for their excellent engagement in implementing the vectorization as well as extending the register allocation. Martin Meeser extended our simulator by the SIMD mode and realized the code integration.

Special thanks I owe to Ralf Dreesen who developed and implemented the concepts presented for the register reconfiguration (see Part IV) together with myself for his diploma thesis [46]. The extension to multiple processors has been done after finishing his thesis. While most of the ideas have been elaborated jointly, the generalization of the analysis of future register accesses to the Data-Flow Analysis (DFA) of n-liveness (see Chapter 11) was mainly done by himself. The same holds for the inter-block placement of reconfiguration instructions (see Section 12.2.2). However, he granted the permission to present both methods in this thesis in detail for completeness.

In particular, I express my gratitude to Madhura Purnaprajna for our extensive discussions about the reconfigurable QuadroCore and offering insights into the hardware background of the Ph.D. project. Augmenting the existing hardware prototype with reconfiguration enabled to test the compiler with real hardware. Further, she proof-read an early version of my thesis and gave very useful comments on the basics of reconfigurable hardware.

During research, I have also got many useful suggestions and hints from other people. I would like to thank Robert Preis for the discussion about graph partitioning, which inspired me when developing the processor partitioning method utilized by our compiler. The concept of an annotation-based load-time scheduling presented in Section 5.3.2 is also based on his suggestions. Sam Larsen provided additional information to his Superword Level Parallelism (SLP) approach [107], which is employed by our vectorization module.

A great deals of thanks goes to all my colleagues, both former and present, of the research group Programming Languages and Compilers for the both friendly and funny atmosphere.

I am grateful to my parents, who have always supported me and never lost faith that this thesis would finally come to an end.

Paderborn, 29.04.2008 Michael Hußmann

Abstract

Reconfigurable computing systems can change the functionality and structure of their components in order to improve the resource efficiency. Many existing architectures [73, 38] have to be programmed in assembly, or a related compiler does not provide full automation. Usually, a compiler is customized to a specific reconfigurable system developed for a certain application domain.

This thesis presents a unified hardware/software approach called CoBRA[1], where compiler-driven reconfiguration selects from a fixed set of modes known to the compiler. Such modes are denoted as *reconfigurable architectural variants* or briefly *variants*. Each variant relies on matching program analysis and represents optimal machine configurations for certain application domains. Typical optimization goals are a fast execution, small code size, or low power dissipation. The machine can be reconfigured by invoking special instructions between code using different configurations at run-time. A prominent example is to reconfigure between different parallelization paradigms like SIMD or MIMD. Given a program that exhibits both regular and non-regular structures, the compiler can determine the best execution mode by analyzing the parallelism. Reconfiguring the connections between ALUs and register banks at run-time allows to exploit more physical registers than architecturally available. In a multi-core, a processor can use registers of other processors temporarily to avoid spilling or communicate efficiently employing some registers in a shared manner.

Using a manageable set of variants leads to an enormous reduction in the design space, compared to fine-grained reconfiguration. The compiler then addresses this finite design space efficiently by applying well-known program analysis techniques. Reconfiguration can be performed with very low effort at run-time by switching fixed, coarse-grained components like instruction decoders, ALUs and register banks. Much complexity like generating machine code and utilizing reconfiguration is hidden from the user in contrast to other reconfigurable approaches [73, 38]. Programming reconfigurable hardware in a sequential High-Level Language (HLL) such as C using a single tool avoids further manual effort and minimizes both time-to-market and error rates.

This thesis concentrates on switching between SIMD and MIMD execution and reconfiguring register connections. In both cases, we present original methods to select variants and generate optimized machine code efficiently, or enhance existing approaches known from literature. Furthermore, we propose additional opportunities in reconfiguration, which may be part of future work.

Our approach has been evaluated using a multi-core of four-tightly coupled processors, which can be simulated using a synthesized model or a cycle-accurate software simulator. Additionally, such model can be mapped to a Field Programmable Gate Array (FPGA) based

[1] **Co**mpiler-**D**riven **D**ynamic **R**econfiguration of **A**rchitectural Variants (merge two *D*s to a *B*)

prototyping environment [93] for rapid evaluation of real-world applications on real hardware. Our measurements indicate suitability especially in audio and video processing applications. At a modest cost in reconfiguration, CoBRA achieves significant improvements of the resource efficiency in terms of execution time, code size, and power consumption.

Contents

1 Introduction 1
1.1 Motivation 1
1.2 Reconfiguration of Variants 3
1.3 Goals and Decisions 6
1.4 Methodical Contributions 8
1.5 Structure of Thesis 9

I Reconfiguration of Processors I-1

2 Reconfigurable Architectures I-5
2.1 Reconfigurable Computing I-6
2.1.1 Reconfigurable Devices I-8
2.1.2 History of Reconfigurable Computing I-10
2.2 Programming Reconfigurable Hardware I-11
2.2.1 Classification of Systems and Tasks I-12
2.2.2 Three Classical Design Flows I-13
2.2.3 Circuit Specification and Generation I-16
2.3 Existing Reconfigurable Computing Systems I-18
2.3.1 Level 1: Functional Unit I-19
2.3.2 Level 2: Coprocessor I-21
2.3.3 Level 3: Attached Processing Unit I-24
2.3.4 Level 4: Standalone Processing Unit I-25
2.3.5 Other coarse-grained approaches I-25
2.3.6 Discussion I-27

3 Dynamic Reconfiguration of Variants I-31
3.1 Reconfiguration of Variants I-33
3.1.1 Reconfiguration of Parallelization Paradigm I-34
3.1.2 Reconfiguration of Register Access I-36
3.1.3 Reconfiguration of Machine Topology I-37
3.1.4 Dynamic Assignment of Special Functional Units I-38
3.1.5 Combination of Instructions between Processors I-38
3.2 Application Scenarios I-38
3.2.1 Compiler-Driven Reconfiguration I-39
3.2.2 Need for Compiler-Supported Reconfiguration I-40
3.2.3 Compiler-Supported Reconfiguration I-41

3.3 Compiler for Reconfiguration of Variants I-43
3.3.1 Prototypical System I-44
3.3.2 Code Integration I-45

II Compilation for Multi-Cores II-1

4 Compiler for QuadroCore II-5
4.1 Reference Architecture II-6
4.2 Parallelizing Compiler II-8
4.2.1 VLIW Machines and Superscalar Processors II-8
4.2.2 Machine Model II-10
4.2.3 Structure of Compiler Backend II-11
4.2.4 Context of Scheduling Phase II-13

5 Processor Partitioning II-15
5.1 Related Work II-17
5.2 Partitioning of Data Objects II-18
5.2.1 Introductory Example II-19
5.2.2 Affinities Between Variables II-20
5.2.3 Optimal Number of Partitions II-22
5.2.4 Variable and Parameter Partitioning II-23
5.2.5 Discussion II-26
5.3 Improvements and Extensions II-26
5.3.1 Holistic Partitioning of Variables and Instructions II-27
5.3.2 Load-Time Scheduling Using Compiler Annotations II-27

6 Communication and Synchronization II-31
6.1 Communication between Processors II-32
6.1.1 Communication Mechanism and Basic Concepts II-32
6.1.2 Placement of Communication Code II-34
6.2 Barrier Synchronization II-38
6.2.1 Related Work II-38
6.2.2 Barrier Synchronization for the QuadroCore II-39
6.2.3 Placement of Local Barriers during Re-Scheduling II-41
6.2.4 Need for Global Barriers II-42
6.2.5 Placement of Global Barriers II-44

III SIMD/MIMD Reconfiguration III-1

7 Related Work III-5
7.1 Vector Machines and Classical Vectorization III-6
7.2 Compilation for Multimedia Extensions III-7
7.2.1 Challenges of Vectorization for Multimedia Extensions III-8
7.2.2 Vectorizing Compilers for Multimedia Extensions III-9
7.2.3 Vectorization by Pattern Recognition III-9
7.3 SIMD Processors III-10

7.3.1 CELL Microprocessor III-11
7.3.2 eLite DSP III-11
7.4 SIMD/MIMD Reconfiguration III-12

8 Compilation for SIMD/MIMD Reconfiguration **III-15**
8.1 Structure of the Compiler Backend III-16
8.2 Functionality of SIMD and MIMD Modes III-17
8.2.1 Machine Model in SIMD Mode III-18
8.2.2 Memory Accesses in SIMD Mode III-18
8.2.3 Branches in the Presence of SIMD/MIMD Reconfiguration III-21
8.3 Vectorization with SLP III-22
8.3.1 Adjacent Memory Accesses III-24
8.3.2 Preparation III-28
8.3.3 SLP Utilization III-30
8.4 Register Allocation for SIMD and MIMD Code III-34
8.4.1 Allocation of Vector Registers III-35
8.4.2 Efficient Placement of Transport Instructions III-37
8.4.3 Register Allocation and Spilling III-39

IV Reconfigurable Register Banks **IV-1**

9 Related Work **IV-5**
9.1 Classical Register Allocation Techniques IV-6
9.1.1 Register Allocation for Expression Trees IV-6
9.1.2 Register Allocation for Basic Blocks by Lifetime Analysis IV-7
9.1.3 Register Allocation by Graph Coloring IV-7
9.2 Restricted Form of Register Reconfiguration IV-9
9.2.1 Register Renaming IV-9
9.2.2 Register Windowing IV-9
9.3 Reconfigurable Registers IV-10
9.3.1 Multiple Register Banks IV-10
9.3.2 Register Connections IV-11
9.3.3 Register Queues IV-12

10 Register Architecture **IV-15**
10.1 Selected Register Architecture IV-16
10.1.1 Terminology IV-16
10.1.2 Examples IV-17
10.1.3 Elements of Register Architecture IV-18
10.2 General Register Architecture IV-20
10.2.1 Permitted Mappings IV-22
10.2.2 Discrimination between Read/Write Accesses IV-23
10.2.3 Operands in Multiple Registers IV-24

11 Analysis of n-Liveness **IV-25**
11.1 Definition and Representation of n-Liveness IV-26

11.1.1 Access Trees IV-27
11.1.2 Probabilities in Access Trees IV-28
11.2 Specification of Data-Flow Problem IV-29
11.2.1 Tree Operations IV-30
11.2.2 Transfer and Union Function IV-31
11.2.3 Properties of Probabilities in Access Trees IV-32
11.3 Convergence of Data-Flow Analysis IV-35
11.3.1 Complete Access Tree IV-35
11.3.2 Partial Order of Access Trees IV-36
11.3.3 Monotonicity of Transfer Function IV-36
11.3.4 Monotonicity of Union Function IV-39
11.3.5 Convergence of Data-Flow Analysis IV-40
11.3.6 Properties of Union Function IV-41

12 Register Allocation **IV-43**
12.1 Allocation of Physical Registers IV-45
12.1.1 Replacement Strategy IV-46
12.1.2 Definition of Affinity IV-47
12.1.3 Affinity to Physical Blocks IV-48
12.1.4 Construction of Affinity Graph IV-50
12.1.5 Improvement of Allocation IV-52
12.2 Reconfiguration IV-54
12.2.1 Intra-Block Reconfiguration IV-54
12.2.2 Inter-Block Reconfiguration IV-55
12.2.3 Inter-Procedural Reconfiguration IV-58
12.3 Extensions for Multi-Cores IV-59
12.3.1 Reconfiguration IV-61
12.3.2 Re-Scheduling IV-62

V Evaluation and Conclusion **V-1**

13 Evaluation **V-3**
13.1 Simulation of QuadroCore V-5
13.1.1 Specification and Implementation V-5
13.1.2 VLIW/Barrier Reconfiguration V-7
13.1.3 Description of Benchmarks V-9
13.2 Parallelizing Compiler V-10
13.2.1 Scheduling and Optimization V-12
13.2.2 Synchronization V-20
13.2.3 Processor Partitioning V-30
13.2.4 Communication V-33
13.3 SIMD/MIMD Reconfiguration V-38
13.3.1 Execution Time V-39
13.3.2 Ratio of SIMD/MIMD Execution V-42
13.3.3 Code Size V-46
13.3.4 Power Consumption V-47

13.3.5 Costs of Reconfiguration V-50
13.3.6 Costs of Communication V-50
13.4 Reconfigurable Register Banks V-51
13.4.1 Execution Time V-53
13.4.2 Code Size V-56
13.4.3 Power Consumption V-58
13.4.4 Costs of Reconfiguration V-59
13.5 Combination of SIMD/MIMD and Register Bank Reconfiguration V-60
13.5.1 Execution Time V-61
13.5.2 Ratio of SIMD/MIMD Execution V-61
13.5.3 Code Size V-62
13.5.4 Power Consumption V-63
13.5.5 Costs of Reconfiguration V-65

14 Conclusion and Future Work **V-67**
14.1 Conclusion V-68
14.2 Future Work V-71

VI Appendix **VI-1**

List of Figures **VI-3**

List of Tables **VI-9**

List of Algorithms **VI-11**

Glossary **VI-13**

1. Introduction

1.1. Motivation

A *reconfigurable system* can alter the functionality and structure of its components [165]. Basically, the term *system* can affect both hardware and software architectures.

Reconfigurable Hardware Reconfiguring hardware architectures offers a high potential for better resource utilization by tailoring the functionality of the available hardware to a set of tasks [165]. Concretely, a reconfigurable hardware architecture can adapt its structure to exploit features of a problem or a certain instance. Due to these reasons, reconfiguration can speed up the execution of a program dramatically. If the machine is reconfigured to use only a subset of its resources, the energy consumption can be reduced, which is an important aspect for embedded and mobile devices. Simultaneously, the code size of a program might be shrunk to decrease the amount of memory needed at the target machine. Reducing the memory size also results in a significant improvement concerning area need and power dissipation.

Great improvements in hardware technology have led to a multitude of different reconfigurable systems [73, 38]. Most reconfigurable computing systems combine a general-purpose core with reconfigurable logic [176, 119, 67, 180, 28, 150, 156]. In some cases [48, 158, 9], the reconfigurable hardware operates stand-alone or information about a coupling is not available. Those architectures are mostly based on Coarse-Grained Reconfigurable Arrays (CGRA) or other coarse-grained programmable logic. Section 2.1 introduces reconfigurable computing and surveys about the history of reconfigurable computing in detail. The reconfigurable approches referred to above will be discussed in Section 2.3.

General-purpose microprocessors are denoted as *programmable* hardware, because they can be used to compute any computable function in software. This offers a high flexibility over specialized hardware solutions providing only a fixed functionality. On the other hand, implementing an algorithm in hardware yields much better results concerning performance and power dissipation. Reconfigurable hardware unifies the benefits of both paradigms hardware and software.

Importantly, reconfiguration is also employed in commonly used *programmable* hardware which is often regarded as *non-reconfigurable* at the first glance: The instruction sets of general-purpose microprocessors are usually implemented by means of microprogramming [174] which was invented by Wilkes in 1951. This technique enables updating the instruction set of a processor after fabrication to improve the implementation or to fix bugs of the original design.

Reconfigurable Software The main goal for reconfiguring software architectures is the adaption to new requirements or a changing environment. For instance, long-running distributed applications should provide high availability, adaptability, and maintenability. Consequently, modifications to such a system need to happen on-the-fly during execution. Reconfiguration should minimize execution disruption and preserve a consistent state of the participating entities.

Some early work in updating a running program was done by Bloom [19] and considered dynamic replacement of modules in a distributed programming environment. Frieder et al. [59] presented a similar approach for updating certain procedures of a program.

One important challenge of dynamically reconfiguring software is to transfer state information between the old and new configurations. Hofmeister [79] targets this vital aspect in her Ph.D. thesis in detail. The introduced machine-independent concept supports dynamic reconfiguration of distributed systems based on a set of reconfiguration points specified by the programmer.

Finally, Bradbury et al. [22] published a survey of formal specification approaches used to develop dynamic software architectures. The mentioned publications refer to a multitude of further related work which is not discussed here.

Reconfigurable Devices In this thesis, we focus on reconfiguration of processors. Many approaches [63, 90, 6, 170, 167, 144, 180], especially the older ones, are based on commercial FPGAs or specifically designed FPGA-like logic. Originally, FPGAs were developed for rapid prototyping by companies like Xilinx or others [31] in the mid-1980s. Rapid prototyping aims for fast construction of early prototypes and their incremental refinement in order to test and evaluate a system before final production. Bug-fixes or modifications may also be applied after deployment, if a system is realized partly or even completely with reconfigurable logic. As a consequence, time-to-market and in particular risk can be reduced significantly.

In general, FPGAs can be used to store any function on the level of logic gates. Consequently, they combine the high flexibility of programmable hardware like general-purpose microprocessors with the performance of specialized hardware providing only fixed functionality. Modern FPGAs enable the realization of complete systems with reconfigurable logic and use techniques like partial reconfiguration, context switching, and self-reconfiguration for fast updating [38]. On the other hand, an FPGA still suffers from a comparatively long reconfiguration time as well as a huge routing area overhead.

Most recent reconfigurable computing approaches [48, 119, 28, 150, 9, 152] use CGRAs [72]. A major benefit of the coarse-grained manner is a speedup of the overall throughput, because more complex computations can be done on a greater number of inputs. Furthermore, the reconfiguration latency and configuration memory is reduced significantly, while placement and routing problems known from FPGAs are scaled down dramatically.

Compiling for Reconfigurable Architectures The majority of the reconfigurable architectures [176, 119, 28, 150] needs to be programmed in assembly language or demands manual partitioning of program code for a microprocessor and the reconfigurable logic. But many

recent projects [112, 67, 156, 180] have shown that automatic compilation for reconfigurable machines is crucial to yield best results and reduce development time. Furthermore, a lot of errors caused by human intervention are avoided, which can only be found with a high effort in most cases. Importantly, the rare set of compilers for reconfigurable hardware is usually tailored to a certain application domain. Hence, those compilers are just used to automatize code generation, but the provided reconfigurability is not well concerted with the general capabilities of compilers.

Basically, a compiler can use program analysis techniques to identify different characteristics of program code such as inherent parallelism, frequency of certain instructions, register pressure, memory access patterns, or resource utilization. From this large base of information, *suitable* configurations for a target machine as well as an optimized machine program can be generated. Each configuration represents the best machine for a certain program section from the perspective of the compiler. Thereby, different optimization goals like fast execution time, small code size, or low power consumption may be used. Configurations can be activated by executing special reconfiguration instructions inserted into the machine program.

We believe that the results of program analysis can be exploited best, if the reconfiguration is performed *in terms of the compiler*. For instance, a target machine consisting of multiple cores may adapt to different parallelization paradigms like MIMD and SIMD or just use a single core for execution. In this case, the compiler can break down a given program into sections and determine the best execution mode for each section by analyzing the parallelism in a program. The machine is reconfigured at the boundaries between the sections to adapt to the best-suited parallelization paradigm.

1.2. Reconfiguration of Variants

Generally spoken, the approach motivated above can enable different alternatives of a certain architecture. Such an alternative is denoted a *reconfiguration variant* in the following. Multiple associated variants form a *reconfiguration dimension*. For example, the three parallelization paradigms Single Instruction Multiple Data (SIMD), Multiple Instruction Multiple Data (MIMD), and single processor are reconfiguration variants of one reconfiguration dimension called *parallelization* here. From now on, the term *reconfiguration* is often omitted for simplification. In analogy to the notation formed here, we call our approach CoBRA[1]. The rest of this section introduces CoBRA incrementally and give examples for reconfiguration dimensions which are studied in this thesis.

Beneficial Restriction of Design Space The usage of a fixed small set of variants leads to an enormous reduction of the decision space for the compiler. For the sake of contradiction, assume that the compiler would target a fully reconfigurable device like an FPGA. Obviously, the high number of feasible variants would impede the selection of the best variant for a certain program section. With CoBRA, the decision space is restricted to those problems which can be solved efficiently using well-known program analysis techniques. On the

[1] **Co**mpiler-**D**riven **D**ynamic **R**econfiguration of **A**rchitectural Variants (merge two *D*s to a *B*)

other hand, a large number of different application domains can be supported by employing a small number of variants.

Ideal Reconfigurable Target Machine The ideal reconfigurable machine for CoBRA should be built of fixed, coarse-grained components which can be switched at run-time. In terms of the example given above, such components can be ALUs, instruction decoders, register banks, memories, etc. Reconfiguration may be realized using multiplexers or switch matrices. As a result, the effort in reconfiguration is minimized in order to maximize the benefits gained by using alternative variants of an architecture. Instead, commercial programmable logic devices like FPGAs provide a very fine-grained structure and suffer from a quite large reconfiguration time.

Example In order to explain the idea of CoBRA more in detail, we consider the architecture in Figure 1.1 as an example. The machine is a multi-core of four tightly-coupled processors which are connected to an external memory. Each processor consists mainly of an instruction decoder, an ALU, a register bank as well as a local memory. The solid lines represent the default connections between the components, while the dashed lines illustrate two possibilities for reconfiguration. The next two paragraphs discuss the two dimensions supported by the architecture shown in Figure 1.1.

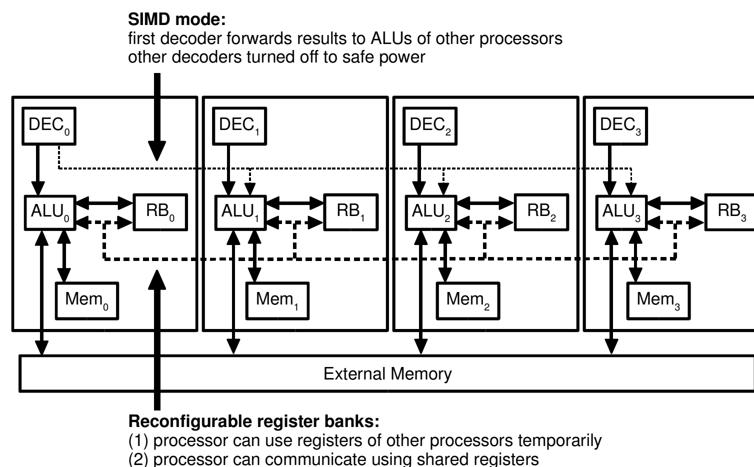


Figure 1.1.: Reconfiguration of variants

SIMD/MIMD Reconfiguration With the default configuration, all processors execute their own instruction stream on different data. Such MIMD mode is well-suited for unstructured or non-regular program code with a high degree of Instruction-Level Parallelism (ILP). But if parts of a program have a regular structure typically found in scientific or multimedia computations, a SIMD execution will be more efficient. In the SIMD mode, one machine program is executed by all processors on different data. The instructions are only decoded by the first processor which forwards the results of the decoding to the ALUs of all other processors, while the remaining instruction decoders can be turned off. By these means, the code size and power consumption may be reduced significantly.

If there is not enough parallelism available in some situations, a subset of the four processors can be used for execution. Hence, the other processors may be shut down completely to save a lot of energy. This idea works both in SIMD and in MIMD mode. A special case is the usage of a single processor when no parallelism is available or the effort in communication between multiple processors is larger than the speedup gained by parallelization.

Reconfigurable Register Banks The second reconfiguration targets the connections to the register banks: As each processor can only access its own register bank by default, the communication between different processors must be performed using the external memory. An access to this memory takes a number of clock cycles, while the registers can be read and written in one cycle. If a processor needs more registers than physically available in its register bank, expensive spilling must be done.

A reconfiguration of the connections between ALUs and register banks gives the compiler more freedom in register allocation. For instance, a processor can use registers of other processors temporarily to avoid spilling. Furthermore, the communication can be realized by utilizing a subset of the existing registers in a shared manner to speed up the execution.

Compiler-Driven Reconfiguration Compile-time analysis can be used to determine optimized machine configurations in both situations. For the SIMD/MIMD reconfiguration, the compiler may apply well-known scheduling and vectorization techniques to certain parts of the Control-Flow-Graph (CFG) like loops or basic blocks. Afterwards, the best results are chosen for execution and reconfiguration instructions are inserted at the boundaries between MIMD and SIMD sections.

When using reconfigurable register connections, physical registers of arbitrary processors can be allocated for the symbolic register values. Then, the compiler inserts reconfiguration instructions to establish the connections to the register banks as demanded by the code.

In both cases, reconfiguration costs need to be traded for execution time. For instance, the runtime of a program can only be improved, if the speedup by reconfiguration is greater than the effort in reconfiguring the machine. Additionally, the provided execution modes might have different performance in hardware. For example, a SIMD mode may require to synchronize all processors after every cycle like the functional units of a VLIW machine. In MIMD mode, a processor can execute its instructions independently of the other processors and synchronization needs only be performed explicitly where necessary.

Space of Reconfiguration Variants Up to now, a reconfiguration variant has been defined as a feasible instance of a dimension. In principle, a variant can also combine features of different dimensions. For example, a program using both SIMD and MIMD execution may also reconfigure the connections to the register banks to reduce the communication costs between the processors. Consequently, a variant can be regarded as a point in a discrete *reconfiguration space* spanned by the dimensions.

Figure 1.2 illustrates the 2-dimensional space for the paradigms introduced above. The first dimension comprises of three parallelization paradigms, the SIMD/MIMD modes as well as the trivial single processor mode. In the following, it is denoted by the term *paral-*

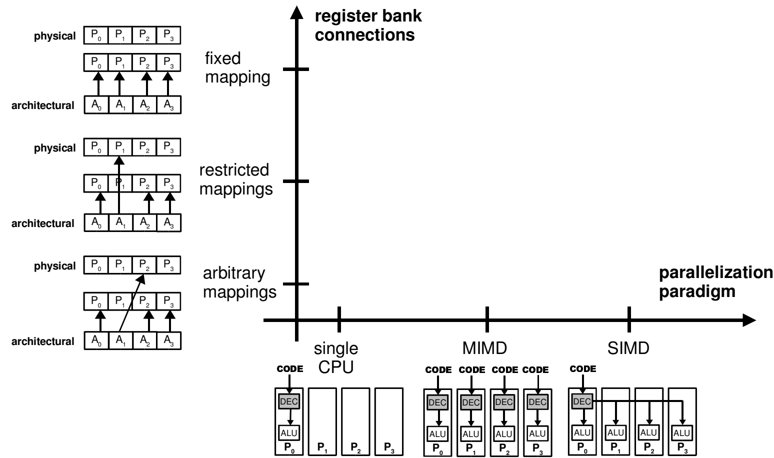


Figure 1.2.: Space of reconfiguration variants

lelization. In the second dimension, three different paradigms of connections to the register banks are located, which are explained from top to bottom. From now on, we call this dimension *register access*. The first variant is the default mapping where each processor can only use its own register bank. The second variant allows connections to other register banks, but requires that the mapping is identical. Finally, the third variant also enables non-identical mappings.

CoBRA is based on a machine consisting of fixed components like ALUs and register banks but reconfigurable connections. In general, this concept enables much more reconfiguration dimensions than *parallelization* or *register access*. Section 3.1 proposes five selected reconfiguration dimensions which can be realized with our methodology.

1.3. Goals and Decisions

This thesis proposes a holistic hardware/software approach called CoBRA for efficient utilization of reconfiguration through an optimizing compiler. Basically, the machine can be switched between different modes called *variants* by executing reconfiguration instructions at run-time. With respect to considered application domains, variants may be implemented in a target machine, if the resource efficiency can be improved through switching between the modes. Each variant needs to be supported by a corresponding program analysis and must be made known to the CoBRA compiler. The compiler determines the best variant for each piece of code and generates an optimized machine program. However, we focus on the development of reconfiguration variants and their evaluation, not on improving the resource-efficiency.

Benefits The whole process of compiling programs and employing reconfiguration is transparent for the user: Programs can be written in sequential HLLs such as C and are compiled using a single tool automatically. Implementing algorithms on most reconfigurable systems requires the application of a multitude of tools and much expert knowledge about hardware

design flow. Typically, the process demands further manual input and hencefore is quite error-prone and time-consuming.

The usage of a manageable set of variants leads to an enormous reduction in the design space, compared to fine-grained reconfiguration. Furthermore, reconfiguration can be performed with a very low effort at run-time by switching fixed, coarse-grained components like instruction decoders, ALUs and register banks.

Reconfiguration Dimensions In Section 3.1, we motivate several opportunities in reconfiguring a multi-core according to the CoBRA approach. Reconfiguring between SIMD and MIMD execution as well as reconfigurable register banks are studied in detail, which have already been motivated above. For both dimensions, we derive architectural models and present the fundamental ideas of our approaches. We introduce new methods which have been developed by us, or refer to existing work from literature which has been extended for this thesis. Finally, the evaluation of our prototypical implementation using practical benchmark programs is discussed.

Reference Architecture Throughout this thesis, a multi-core called QuadroCore consisting of four tightly-coupled RISC processors is used as a reference architecture. The QuadroCore has already been served as an example in the previous section and is presented in Section 4.1. We have a cycle-accurate simulator of the QuadroCore based on a synthesized model. For exhaustive testing this synthesized model can be mapped to our FPGA based prototyping environment [93] for rapid evaluation of large benchmarks on hardware. In addition to the hardware implementations, we have a cycle-accurate software simulator for the QuadroCore, which is partly generated from a machine specification.

Our original idea was to implement the CoBRA compiler by extending a given parallelizing compiler for the QuadroCore. As such compiler did not exist when starting this thesis, we decided to build a prototypical implementation with a reasonable effort. Consequently, the current implementation of the CoBRA compiler only exploits ILP on basic block level (see introduction of Part II).

Granularity of Parallelization Basically, a multi-core architecture can support different levels of parallelism such as Instruction-Level Parallelism (ILP), Loop-Level Parallelism (LLP), or Thread-Level Parallelism (TLP). In the future, the CoBRA approach may also support more coarse-grained parallelism. A succeeding prototype of the compiler can identify the best granularities for a given program automatically or take directives of the programmer into account. For instance, parallelism on loop-level may be detected and exploited by applying existing approaches from literature. Identifying threads for concurrent execution requires a special programming model or just additional information how to parallelize source code. Such granularities may be supported by reconfiguring between different mechanisms of synchronization and communication within the multi-core.

In the thesis, the term parallelization *paradigm* refers to SIMD and MIMD in contrast to different *levels* or *granularities* of parallelism like instruction, loop, or task.

1.4. Methodical Contributions

The main contribution of this thesis is the CoBRA approach, which provides a novel method of reconfiguring processors. In few words, a small set of variants is implemented in hardware and can be utilized automatically by the CoBRA compiler. Concretely, the compiler selects the best combination of variants for a given program and generates code, which switches between the modes at run-time using reconfiguration instructions. The intrinsic complexity of reconfiguration is hidden from the user by deciding for a basic set of modes and focusing on clear tasks well-known from compiler research. A discussion of the central benefits can be found in the previous section. This thesis concentrates on SIMD/MIMD reconfiguration and reconfigurable register banks.

SIMD/MIMD Reconfiguration The vectorization module of the CoBRA compiler is based on a customized version of the SLP approach by Larsen et al. [107]. We favoured SLP over classical techniques known from vector computers, because its simplicity and conciseness enabled an elegant implementation and easy integration in our prototype. The method (see Section 8.3) relies on adjacent memory accesses as a starting basis for SIMD code. We have evolved an original analysis of adjacent memory accesses which is based on the idea of Common Subexpression Elimination (CSE) (see Section 8.3.1). As the SLP approach has been developed for non-reconfigurable machines, we augmented its scheduling phase by placement of reconfiguration instructions between SIMD and MIMD operations. Furthermore, our scheduler tends to maximize contiguous pieces of code using a single execution mode to reduce the effort in reconfiguration.

Typically, the processors of multi-cores like the QuadroCore have separate register banks. In order to avoid major changes of the architecture for the SIMD mode, the entries of the vector register i correspond to the i-th registers of the processors (see Section 8.2.1). We have developed an appropriate register allocation, which models this property by using virtual vector registers combining the virtual registers created during code selection (see Section 8.4). The relation between both kinds of virtual registers is expressed by additional transport operations initializing vector registers used in SIMD mode based on scalar registers defined in MIMD mode, and vice versa. Such code serves as an input for register allocation by graph coloring [32].

Reconfigurable Register Banks We have derived a model of the register architecture (see Chapter 10) where physical registers are accessed indirectly using architectural registers as proposed by Kiyohara et al. [97]. The mappings can be modified by executing special instructions. In order to reduce the effort in reconfiguration, this thesis extends their work by partitioning both kinds of registers into blocks of a common size. Mappings are established block-wise between architectural and physical blocks. The influence of the block size on the number of reconfiguration instructions has been a central part of our evaluation.

Suitable for this register architecture, we present a two-phase approach which allocates physical registers by graph coloring and inserts reconfiguration instructions (see Chapter 12). Both phases aim to reduce reconfiguration code through an original method for computing the n pair-wise different values accessed after a certain program position. Such property is

denoted as n-liveness (see Chapter 11) and can be computed for each position in a given function by a data-flow analysis. The first phase tries to optimally partition the virtual registers into the physical blocks. It uses information about register values which are often accessed together as a secondary criterion during coloring. The second phase relies on the allocation of physical registers and tries to re-use mappings beyond basic blocks by identifying physical blocks which are active until the beginning of the next block.

As the first phase allocates physical registers using a well-known technique, the second phase inserts reconfiguration instructions only where necessary. If the maximum number of live registers is never larger than the number of architectural registers, reconfiguration is clearly not employed. Consequently, our method can replace a conventional register allocation on function level transparently.

Compilation for Multi-Cores In order to implement our concepts for both reconfiguration dimensions, we needed a parallelizing compiler for the QuadroCore. As motivated in the previous section, we decided for scheduling on basic block level for simplification. Instead, we focused on three conceptual challenges that need to be met when generating code for such machines.

At first, we evolved a data partitioning method using affinity graphs for homogenous multi-cores, which is outlined in Section 5.2. Functional units are allocated using the BUG algorithm by Ellis [50]. We envision a holistic processor partitioning method based on affinity graphs which considers both data objects and instructions jointly. If multiple programs compete for access to a multi-core or some processors are broken due to hardware defects, the number of available processors is first known after compile-time. The proposed partitioning method can be extended to compute compiler annotation which enable an efficient adaption of program code at load-time or even run-time of a program.

As the processors have separate register banks, a communication mechanism based on a shared memory has been chosen to transport register values between processors. Memory data are either stored in the external memory, or access to local data is ensured by a proper assignment of memory operations and their data. Section 6.1 presents the basic idea of the inter-processor communication developed for the QuadroCore and outlines the efficient placement of communication code.

The processors of the QuadroCore operate independently of each other and must be synchronized explicitly using barriers where necessary. We have developed a synchronization mechanism, which has been implemented in hardware and is supported by an automatic insertion of barriers during the re-scheduling phase (see Section 6.2).

1.5. Structure of Thesis

As this thesis alludes many different topics, the chapters have been grouped into parts in order to get a further level of structuring.

Part I deals with reconfiguration of processors in two ways: Chapter 2 provides basic knowledge about reconfigurable computing including hardware devices and architectures as well as their programming. We discuss a multitude of reconfigurable systems with respect

to a classification scheme. Thereby, we concentrate on the fundamental ideas and concepts important in the context of our work. The discussion of existing reconfigurable systems leads over to Chapter 3, which presents the CoBRA approach. At first, we characterize the ideal reconfigurable machine for CoBRA and suggest several opportunities in reconfiguration. The design of our prototypical compiler implementation is derived incrementally from a general system structure.

Compiling for multi-cores is covered by Part II. Chapter 4 introduces the QuadroCore and explains the fundamental concepts of its parallelizing compiler backend. The partitioning of data objects and operations is covered by Chapter 5, while Chapter 6 concentrates on inter-processor communication and barrier synchronization.

Part III deals with reconfiguration between SIMD and MIMD execution. After a short motivation and central decisions, Chapter 7 discusses related work about vectorization techniques and target architectures. Chapter 8 presents the basic concepts of our approach like the machine model in SIMD mode, the vectorization phase, and a register allocation for functions comprising SIMD and MIMD code based on graph coloring.

The second dimension of reconfiguring connections to register banks is the subject of Part IV. We start with a discussion of related work in Chapter 9, comprising conventional register allocation techniques, two restricted kinds of register reconfiguration, as well as architectures with reconfigurable registers. Chapter 10 introduces the register architecture used by our approach and a more general model which is part of future work. Both the allocation of physical registers and the placement of reconfiguration instructions attempt to minimize the overhead in reconfiguration by utilizing information about n-liveness of register values and physical blocks. Chapter 11 defines the property precisely and outlines a DFA to compute such information efficiently for all program positions. Finally, Chapter 12 explains the two phases of our approach with emphasis on avoiding superfluous reconfiguration code by using the n-liveness property.

Part V first discusses the evaluation of our prototypical implementation in Chapter 13. Thereby, we focus on the two considered reconfiguration dimensions as well as the parallelizing compiler backend. Our experiments based on excerpts from practical audio and video applications have proven the feasibility of the SIMD/MIMD and the register reconfiguration for QuadroCore. The resource efficiency can be improved significantly in terms of execution time, code size, and energy consumption, while the effort in reconfiguration is negligible. Chapter 14 summarizes the fundamental results of our work and gives an overview of future work.

Part I.

Reconfiguration of Processors

Reconfigurable hardware architectures can adapt their functionality and structure to a certain program in order to improve the resource efficiency in terms of performance, runtime, code size, energy consumption, or memory requirements. This part has two fundamental goals: (1) imparting knowledge about reconfigurable computing and existing approaches, as well as (2) presenting the key ideas and concepts of our methodology called CoBRA[2].

Structure The beginning of Chapter 2 introduces reconfigurable computing and outlines its role in contrast to classical solutions. Then, we deal with reconfigurable devices and systems as well as their programming. The remainder discusses a multitude of existing reconfigurable systems. Section 2.3.6 recapitulates the most relevant cognitions by considering topics like application areas, reconfigurable devices, and compiler assistance. Furthermore, it compares the related approaches with CoBRA in terms of the mentioned criteria. As this thesis concentrates on code generation for reconfigurable processors, great importance is attached to the basic concepts. In particular, hardware aspects are covered roughly to get a general idea as well as to classify both CoBRA and the work from literature.

Chapter 3 introduces CoBRA by revising the central motivating ideas, and outlines the fundamental concepts and decisions. After defining some basic terms, we present the ideal reconfigurable target machine providing architectural variants and use this model for proposing a number of reconfiguration dimensions. The SIMD/MIMD reconfiguration will be covered by Part III, while the concepts evolved for the reconfigurable register banks are disclosed in Part IV. A discussion of the evaluation can be found in Section 13.3 and Section 13.4, respectively. Then, we motivate two application scenarios for CoBRA and suggest the system structure as well as the central tasks to be done. In order to get a prototypical implementation, such rough schemes are both substantiated and simplified. Finally, the challenges of a code integration are discussed, which selects a variant for a certain piece of code based on different results and places reconfiguration instructions.

[2] **Co**mpiler-**D**riven **D**ynamic **R**econfiguration of **A**rchitectural Variants (merge two *D*s to a *B*)

2. Reconfigurable Architectures

Contents

2.1 Reconfigurable Computing **I-6**
2.1.1 Reconfigurable Devices I-8
2.1.2 History of Reconfigurable Computing I-10
2.2 Programming Reconfigurable Hardware **I-11**
2.2.1 Classification of Systems and Tasks I-12
2.2.2 Three Classical Design Flows I-13
2.2.3 Circuit Specification and Generation I-16
2.3 Existing Reconfigurable Computing Systems **I-18**
2.3.1 Level 1: Functional Unit I-19
2.3.2 Level 2: Coprocessor I-21
2.3.3 Level 3: Attached Processing Unit I-24
2.3.4 Level 4: Standalone Processing Unit I-25
2.3.5 Other coarse-grained approaches I-25
2.3.6 Discussion I-27

The goal of this chapter is to provide basic knowledge about reconfigurable computing architectures and their programming. Furthermore, we discuss the most important existing approaches. The thesis at hand mainly concentrates on compiler techniques for reconfiguring processors, rather than circuit technology. Consequently, the following sections give only an overview of the hardware aspects and refer to the literature where necessary. Instead, the fundamental concepts of reconfigurable architectures as well as the existing alternatives of implementing algorithms are covered more in detail.

Structure At first, Section 2.1 compares reconfigurable architectures with general-purpose microprocessors and specialized hardware providing fixed functionality like ASICs. Furthermore, we outline several types of reconfiguration. Section 2.1.1 introduces common reconfigurable devices like FPGAs or coarse-grained reconfigurable arrays. Then, Section 2.1.2 surveys the most important milestones in the history of reconfigurable computing.

The beginning of Section 2.2 motivates the urgent need of automated tools like compilers to map high-level specifications to an arbitrary reconfigurable system. Section 2.2.1 classifies reconfigurable systems and major tasks of implementing algorithms using reconfigurable logic. Such tasks are concretized in Section 2.2.2, which outlines several design flows and discusses the trade-off between automated and manual processes. Specification and generation of circuits with emphasis on HLLs and Hardware Description Languages (HDL) is covered by Section 2.2.3.

Finally, Section 2.3 presents a more fine-grained classification of reconfigurable computing systems and gives an overview of a number of selected approaches. The most important conclusions are summarized by Section 2.3.6 and discussed from the perspective of this thesis.

2.1. Reconfigurable Computing

Basically, an algorithm can be implemented in software or in hardware (see Figure 2.1).

Software Microprocessors can execute software algorithms provided as a machine program. Such a program consists of elementary instructions which need to be supported by the processor. Hence, microprocessors are *programmable* hardware which can compute any computable function. As the computed function can be defined arbitrarily often after fabrication time, they offer a high flexibility, but cannot achieve maximum performance in general.

Hardware Implementing an algorithm as a specialized hardware like an ASIC yields best results concerning speed and energy consumption. But the functionality needs to be defined at fabrication time and therefore cannot be altered later. Additionally, ASICs are very costly and production first pays for at least 1,000,000 pieces. We do not provide detailed information about ASICs and related technology at this point. For further questions, the reader may refer to a special book about ASICs [153] or other standard literature. The mentioned book covers a wide range of aspects like CMOS, HDLs, logic synthesis, simulation, test, floorplanning, placement, and routing.

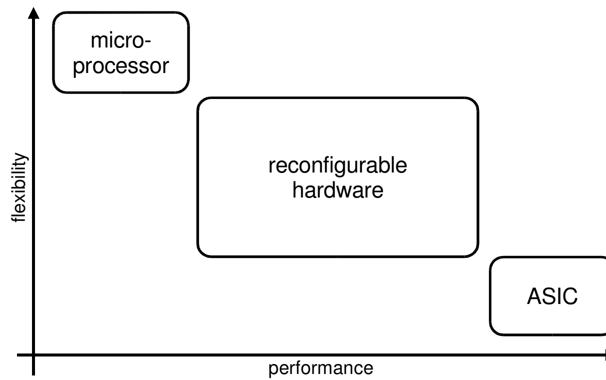


Figure 2.1.: Trade-off between programmable and fixed function hardware

Reconfigurable Hardware Reconfigurable computing is a compromise between hardware and software (see Figure 2.1): A reconfigurable architecture can adapt its functionality and structure to the properties of a certain problem in order to increase the resource efficiency. It enables a higher level of flexibility than fixed hardware, while proving a much better performance than software. Concretely, this can become obvious in significant improvements concerning performance and power dissipation. If the reconfigurable machine operates as a processor executing a machine program, code size can be shrunk, because the instruction set may be adapted to a certain application domain. This also leads to a reduction of needed memory, which consumes typically most chip area and energy.

Reconfigurable vs. Programmable The above definition implies that a *programmable* hardware like a microprocessor can also be regarded as *reconfigurable*, because all modern processors support e.g. microprogramming to update the implementation of their instructions. Furthermore, general-purpose microprocessors usually re-use single circuits for independent computations, and employ multiplexers to control the routing between these components [38].

The quite rough definitions from literature prohibit the identification of an exact border between reconfigurable and non-reconfigurable architectures. Importantly, a machine enabling the utilization of different reconfiguration variants is more likely to resemble a typical processor than a reconfigurable device like an FPGA. As a consequence, we decided to define the *conceptual reconfigurability* as considered in this thesis from the perspective of the compiler: If the modification of the functionality and structure of a microprocessor can be controlled by the compiler, such machine is denoted as *reconfigurable*. The remaining microprocessors, where changes are only visible on hardware level and transparent to the compiler, are called *programmable*. Typical reconfigurable devices like FPGAs are still regarded as *reconfigurable*.

Types of Reconfiguration If reconfiguration is performed during execution, it is said to be *dynamic*. A single reconfiguration before execution is called *static*. In the latter case, reconfigurability can be replaced by configurability, obviously. In the following, the terms *reconfigurable* and *dynamically reconfigurable* are used interchangeably. Some papers or books [165] even use the term *dynamic* for reconfigurable architectures that can change their structures

and functionality at every step of a computation. In that context, dynamic reconfiguration is only regarded as a means for enabling fast reconfiguration at run-time.

Reconfiguration can be roughly initiated in two ways: Firstly, *programmed* reconfiguration (or *ad hoc* change) is part of the system design and is scheduled to happen when certain conditions are satisfied during application execution like embedded failure recovery. Secondly, *evolutionary* reconfiguration (or *post hoc* change) refers to structure modifications caused by events independent from the application specification like maintenance actions, components upgrade, etc. Although the classification was published in a paper about reconfiguring software [14], it is also suitable for reconfigurable hardware architectures. The CoBRA approach follows the idea of programmed reconfiguration, because the compiler inserts special instructions to select variants at run-time.

2.1.1. Reconfigurable Devices

Here, Field Programmable Gate Arrays (FPGA) and coarse-grained reconfigurable arrays are introduced as the most important reconfigurable devices. Other older examples are covered by the historical overview in Section 2.1.2.

2.1.1.1. FPGA

Many reconfigurable architectures [63, 90, 6, 170, 167, 144, 180], in particular the older ones, are based on commercial FPGAs or specifically designed FPGA-like logic. Their introduction in the mid-1980s by companies like Xilinx or others [31] implied a large interest in the development of reconfigurable machines. Although the devices were initially quite small and only provided the equivalent of a few hundred logic gates, the increasing density enabled the implementation of complete systems using reconfigurable logic. Recent FPGAs contain the equivalent of over a million gates and provide a high performance compared to early prototypes.

Rapid prototyping is one key application field where FPGAs are used extensively. The production of general-purpose microprocessors or ASICs is very expensive and requires a time-consuming design and verification process before final fabrication. In order to avoid bugs in the produced chips, early prototypes are tested or evaluated using FPGAs. Furthermore, if a system is realized partly or completely using programmable logic, updates or bug-fixes can also be applied after deployment. In both situations, time-to-market can be reduced at a lower risk and higher profit.

In reconfigurable computing, FPGAs are used to speed up applications by replacing some computations that were previously executed in software with a custom-designed hardware implementation. Early approaches [13, 64] were already developed in the late 1980s and have led to a multitude of reconfigurable computing systems using that methodology. A discussion of the most important systems follows in Section 2.3.

FPGAs still have a number of disadvantages [41, 72]: One of the biggest disadvantages is the comparatively long reconfiguration time. Modern FPGAs use techniques like partial reconfiguration, context switching, and self-reconfiguration [38] in order to ease the penalty costs. We do not deal with these topics here.

Secondly, over 90% of the area is occupied for reconfiguration purposes (routing and configuration memory). Thirdly, the frequencies for real applications are too low, because the routing delays make up about 80% of the propagation time. Fourthly, FPGAs show a relatively high power consumption caused by the programmable interconnects and the configuration memory. Last but not least, the devices as well as the proprietary development tools are quite expensive.

Structure and Types Each FPGA consists of an array of Configurable Logic Blocks (CLB) (see Figure 2.2). These logic blocks contain simple circuits like Look-Up Tables (LUT), multiplexers or flip-flops. In some FPGAs, the logic blocks also include additional blocks of memories. The interconnection structure between the CLBs can also be configured in order to realize arbitrary systems on an FPGA. The FPGA logic can be linked with external components via programmable I/O connects.

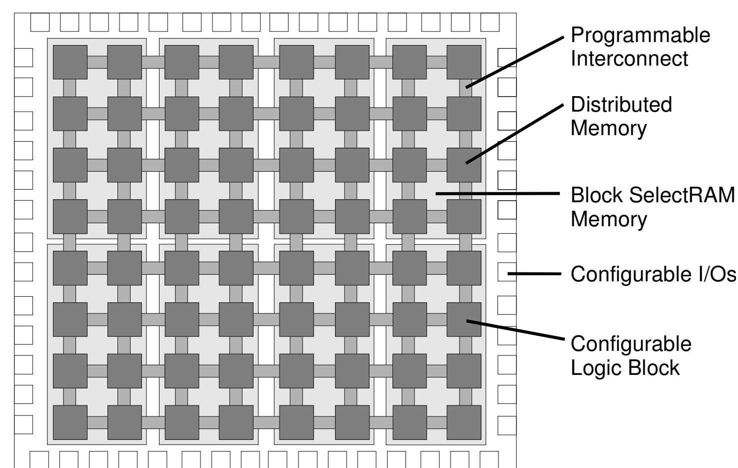


Figure 2.2.: Structure of FPGA

Modern FPGAs load configuration data into SRAM on chip. The SRAM cells are connected to the configuration points like logic blocks and interconnects. Obviously, this offers both very fast reconfiguration at run-time as well as an infinite number of updates. As a drawback, power supply needs to be maintained constantly to retain the configuration. SRAM-based FPGAs are configured at the beginning of each usage epoch, i.e. before running it with a certain configuration. During startup, the configuration data may be loaded from a PROM.

FPGAs realized with Flash memory or EEPROM can be reconfigured several thousands of times and retain the configuration even after power-loss. By these means, a system can be updated with a low effort. On the other hand, the configuration update even needs a few seconds and is therefore quite slowly. Furthermore, such devices are quite expensive.

An anti-fuse based FPGA can only be configured once before product use. It is programmed by a high voltage (10-20 V) which permanently creates an electrically conductive path. The comparatively high signal is the big problem of this technology, because additional circuits and special transistors are needed. Further information about the different types of FPGAs can be found in two books [25, 163].

2.1.1.2. Coarse-Grained Reconfigurable Arrays

Most recent reconfigurable systems [48, 119, 28, 150, 9, 152] are based on Coarse-Grained Reconfigurable Arrays (CGRA) [72]. They make use of operator-level Configurable Function Blocks (CFB), which may contain ALUs, functional units, or even complete processors. CGRAs have word-level data-paths, while the CLBs of an FPGA have only single-bit width. The hard-wired arithmetic units are much more efficient in terms of frequency, area needs, and power consumption.

Consequently, such devices can perform more complex operations on a greater number of inputs in order to speed up overall throughput. As a drawback, the increased data-path width only pays for implementing functions close to their basic word size. But if just 1-bit values are required, the use of that architecture may suffer from an unnecessary area and speed overhead.

A further benefit of the the coarse-grained manner is a significant reduction of reconfiguration latency and configuration memory. Finally, placement and routing is simplified dramatically by the area-efficient routing switches. Examples for reconfigurable computing systems based on CGRAs are given in Section 2.3.

2.1.2. History of Reconfigurable Computing

Microprogramming General-purpose microprocessors frequently make use of techniques which allow to update the functionality of a machine after fabrication. The most important among such techniques is microprogramming [174], which was already invented in 1951 and enables the specification of microprograms for implementing machine instructions. Originally, machine instructions were realized by designing hard-wired circuits which cannot be modified and debugged after fabrication. In the 1960s and 1970s, the concept of microcoding was extensively used for reconfigurable architectures to implement the large, complex instruction sets for these machines. For instance, Rauscher and Agrawala [138] developed a compiler which automatically generated microcode for machine instructions by analyzing a source code program. But in the late 1980s, microprogramming was also integrated into RISC computers, driven by significant improvements to memory technology. Consequently, using this concept for reconfigurable machines became obsolete, because it was now provided by widely available computers.

First Reconfigurable Computer Architecture Estrin et al. [52] did the earliest work on reconfigurable computer architectures with the development of the *Fixed-Plus-Variable* (F+V) machine. Its original intention was to accelerate the Eigenvalues computation of matrices [51]. The machine consisted of a standard processor, a variable hardware, and a supervisory unit. The variable hardware was based on simple, standard hardware modules which had to be exchanged manually, because an automated solution was not possible at that time. The approach relied on automatic transformation of program code into circuit specifications [53]. However, the HLL compiler was too complex in using the application-specific hardware elements.

Fairchild Micromosaic The first microchips were designed without the help of computers, textbooks or lectures about this challenging task. Instead, a team of engineers developed the design on a simple drawing board in a mainly undefined process. If possible, designs developed beforehand or experience from past projects were re-used.

In the mid-1960s, *Fairchild*, one of the first companies in semiconductor industry, began experimenting with the application of CAD for chip design. CAD enabled the faster or more reliable development of more complicated chips and still is an urgently needed technique in modern chip production. The first large-scale CAD chip was called *MicroMosaic* and contained about 150 bipolar AND, OR, and NOT gates. The transistors could be hooked up in almost any pattern by re-arranging the interconnections of the chip. A pattern was determined by a computer using a specification of a customer. By these means, the chip could be optimized rapidly and cheaply for specific applications.

First Programmable Logic Devices At the beginning of the 1970s, the first Programmable Logic Devices (PLD) appeared, which were based on a fixed matrix of macro cells with configurable interconnection structure. These devices were already cost-effective with a small number of pieces and could be developed quite fast in a mostly automated design process. Some PLDs could even be programmed multiple times or during usage. On the other hand, the complexity and possibilities in optimization were restricted by the pre-defined matrix structure.

PLDs can be classified in three groups: Simple Programmable Logic Devices (SPLD), Complex Programmable Logic Devices (CPLD), and FPGAs. An SPLD had a structure as described above, while a CPLD consisted of multiple SPLDs on a single chip with programmable interconnects. FPGAs are handled in the next paragraph.

Typically, the macro cells of an SPLD are grouped into two series connected AND and OR planes. Combinatorial logic circuits can be implemented by blowing the fuses between certain interconnects. PROMs only allow to define the connections in the OR plane, while the AND plane is already fixed. Instead, the AND planes of PLAs can also be programmed which reduces the number of minterms for the OR plane. A third variant is the PAL where only the AND plane can be configured, but the OR plane is pre-determined.

In the mid-1980s, PLDs were succeeded by the famous Field Programmable Gate Arrays (FPGA), which are used to implement arbitrary logic from simple combinatorial functions to complete systems. Their high capacity, flexibility, and performance encouraged their utilization in modern reconfigurable computing. Detailed information about FPGAs can be found in Section 2.1.1.

2.2. Programming Reconfigurable Hardware

As outlined at the beginning of Section 2.1, reconfigurable hardware has many benefits in terms of performance, flexibility, and energy consumption, to mention only a few aspects. Despite of these advantages, reconfigurable architectures still live in a niche and are mostly used in research. Importantly, there is only a small number of commercial reconfigurable systems like XPP [9], which is presented in Section 2.3.5.3. One major reason for this sit-

uation is the lack of programming environments for most reconfigurable systems to ease their usability even for non-experts. Instead, many reconfigurable architectures require a deep knowlege in circuit technology and design as well as experience with the particular reconfigurable system employed.

Reconfigurable computing can only proceed the final step from niche to mainstream, if its capabilities can be exploited automatically using tools like a compiler. Concretely, software engineers without detailed technical background should be able to map algorithms to reconfigurable hardware by writing high-level specifications.

This section deals with the implementation of algorithms on reconfigurable hardware. For simplification, we often use the term *compiler* to abstract from certain tools, set of tools, or programming environments. Especially, synthesis and configuration tools are assumed to be integrated in a compiler or invoked transparently from the user.

Structure At first, Section 2.2.1 presents a classification scheme of reconfigurable computing systems and introduces the major tasks of compiling for such systems. Section 2.2.2 motivates three design flows for the creation of circuits and explains fundamental phases like hardware/software partitioning or low-level tasks. Thereby, the advantages and disadvantages of automated and manual implementation are compared. Finally, several alternatives to specify the functionality of circuits are covered by Section 2.2.3. In all cases, the classical solutions from literature and the introduced terms or models are compared with CoBRA.

2.2.1. Classification of Systems and Tasks

2.2.1.1. Types of Systems

In principle, three major types of reconfigurable architectures should be distinguished with regards to implementing algorithms for such systems: At first, a reconfigurable system can be based on a fixed processor coupled with reconfigurable logic. This scheme has been adopted for the most existing reconfigurable systems. Section 2.3 gives an overview of some selected approaches. Secondly, a processor may be based on a CGRA or even implemented completely on a reconfigurable device like an FPGA. Such systems are surveyed in Section 2.3.5. Finally, reconfigurable logic can be used to implement a specialized, but non-programmable hardware.

CoBRA is a special case, because feasible target machines mostly resemble non-reconfigurable microprocessors. Reconfiguration is realized by switching the fixed, coarse-grained components like ALUs, decoders, and register banks at run-time.

2.2.1.2. Types of Tasks

According to the mentioned classification of reconfigurable systems, a compiler must deal with three central tasks depending on the targeted system. We characterize the basic properties of such tasks and discuss the relation to our CoBRA approach.

Hardware/Software Partitioning Firstly, the compiler should partition a program into sections to be executed on the reconfigurable hardware and into sections to be executed in software on a microprocessor. This task is only necessary for the first kind of system, which consists of both a fixed processor and reconfigurable logic. For CoBRA, hardware/software partitioning can be neglected, because no parts of a given source code program are compiled for the execution in hardware. Instead, the compiler has to determine the best variant for a certain piece of code by using program analysis, or applying several techniques and selecting the best results. Hence, we actually get a partitioning of a program into fragments that are executed on particular variants. The architectural variants themselves need to be implemented into a target machine and must be made known to the compiler before handling source code programs.

Code Generation Secondly, the compiler must produce machine code, if the reconfigurable system behaves like a processor executing instructions. This task can be skipped for the latter mentioned system, where only a specialized hardware must be created which is not programmable like a microprocessor. A typical example is the configuration of an FPGA with a specific behaviour which can only be altered by reconfiguration, but not via software programming. Another prominent instance is the No-Instruction-Set Computer (NISC) approach [142], where a C program is compiled directly for a given data-path and no instructions are provided. In CoBRA, the compiler has to generate code for the target machine which reflects the semantics of a given program and contains additional reconfiguration instructions to select appropriate variants during execution.

Hardware Synthesis Thirdly, hardware configurations need to be generated for the reconfigurable logic. This task must be done for all three mentioned types of reconfigurable architectures. With CoBRA, hardware configurations correspond to the architectural variants which need to be implemented before compiling source code programs. Usage of hardware configurations is reduced to inserting reconfiguration instructions into a machine program, which activate the desired variant implemented by the machine.

Both the first and latter topic will be covered more in detail in the following subsections. The second task is neglected, because the basic concepts of compiling HLL programs into machine code are assumed to be known [1, 95].

2.2.2. Three Classical Design Flows

Here, we substantiate the fundamental tasks of algorithm implementation introduced in Section 2.2.1 and present three possible design flows. Further information can be found in a survey [38] or in a special book on reconfigurable computing with FPGAs [62]. The cycles vary in the degree of automation and the input, i.e. the kind and level of specification. For simplification, we focus on creating hardware configurations or circuits. Code generation can be performed by compiling source code [1, 95] or writing assembly programs. However, we outline the alternatives of generating code and their relation to the design flows.

The left-hand cycle in Figure 2.3 corresponds to a fully automated process, where a program written in a HLL is partitioned for the execution in both hardware and software. The

distinct phases generate circuits automatically and are explained below. The other extreme shown by the right-hand flow is based completely on manual effort and targets a system lacking a fixed microprocessor. Its input is a network of gate-level components which are mapped to the actual target architecture. Hence, the developer has to define the inputs, outputs, and operations of each basic building block by hand. The middle design cycle represents one possibility between automatic and manual design. It demands manual partitioning of an algorithm and a circuit specification using generic complex components which is mapped to the actual hardware in the design process. The microprocessor may be programmed in a HLL or in assembly. For all three design flows, the results of a process can be improved incrementally by repeating a subset of the phases.

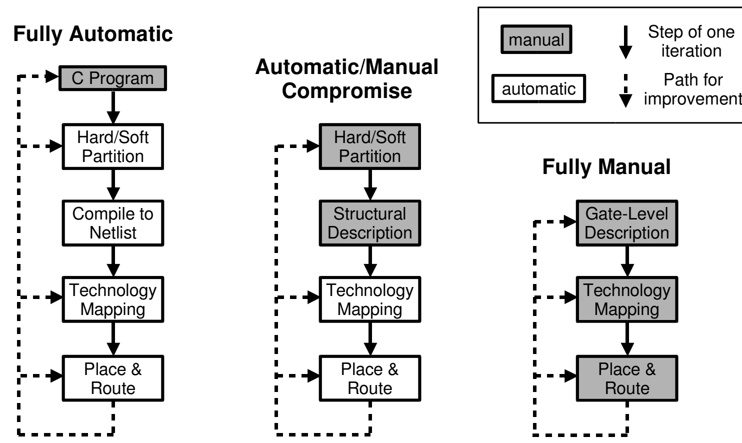


Figure 2.3.: Three design flows for algorithm implementation on reconfigurable systems [38]

Hardware/Software Partitioning and Compilation to Netlist If the targeted reconfigurable system combines a fixed processor with reconfigurable logic, a partitioning of the functionality between hardware and software may be necessary. In terms of Figure 2.3, this task must be performed for the left-hand and middle design cycle. Typically, program control code like variable-length loops or branch control cannot be implemented efficiently in reconfigurable hardware. Instead, executing arithmetic or logic operations in hardware can yield significant speedups.

A hardware/software partitioning can be either done manually by the user or automatically by a compiler. Manual partitioning requires to separate the program into special files or to use compiler directives to mark code which should be implemented in hardware [65]. Tools for automatic partitioning evaluate the benefit of implementing certain parts in hardware and consider costs in reconfiguration [34, 102, 28, 113].

After the partitioning, the parts to be realized in hardware are compiled into a netlist of gate-level components. This is either done by an additional tool or may be integrated into a single programming environment for simplification. In case of the right-hand design flow in Figure 2.3, the gate-level description is exactly the input of the process. Hardware compilation is also considered by Section 2.2.3, which mostly deals with languages for reconfigurable systems.

Technology Mapping, Placement and Routing Such detailed gate- or element-level description of the circuit needs to be transformed to the actual logic elements of the reconfigurable hardware. This phase is called *technology mapping* and is dependent on the exact target architecture. Consequently, the input for the third design flow exhibits the lowest abstraction from the actual machine, as it expects a kind of common description which is implemented in hardware later.

After mapping the circuit to the logic elements of the targeted architecture, the resulting blocks must be arranged to the reconfigurable device. Such placement has two fundamental goals: At first, the available area should be exploited best to minimize wasted space. Furthermore, blocks that communicate extensively with each other or are on a critical path, should be placed close together in order to simplify routing and minimize wire lengths.

Floorplanning can be used to speed up the placement. The idea is to partition logic blocks resulting from technology mapping into clusters according to the communication and critical path. Then a coarse-grained mapping places the clusters on the reconfigurable device. Finally, a fine-grained mapping of the blocks within the clusters is conducted.

Detailed information about these tasks can also be found in the books [25, 163, 153] already mentioned above.

Automated and Manual Design For all three mentioned flows, the degree of automation can vary significantly, while the first type exhibits the widest range among them: In the best case, a compiler performs the three mentioned phases automatically. It partitions a program for execution in hardware and software, and generates both hardware configurations as well as machine code. The opposite extreme would be to conduct all three tasks by hand without any software support.

A fully automated process may be suboptimal in some situations, if the applied methods are incomplete or poorly conceived. Instead, a human expert might yield better results when performing all tasks manually. On the other hand, this requires a lot of background knowledge, much time, and is quite error-prone. From a long-term perspective, automated solutions are most promising, because they are faster, easy to use, and avoid bugs. Furthermore, the quality of final results can be improved incrementally by developing new methods.

A prominent example is the comparison of compiler and assembler: The first computers were programmed in assembly, because no optimizing compilers were on-hand and computation time was very expensive. In 1957, the first FORTRAN compiler was introduced, which was able to optimize a HLL source code program to achieve results comparable to hand-coded assembly programs. Today, assemblers are only employed for special software with hard performance requirements like device drivers, because computation time has become much cheaper than labour costs.

CoBRA Fully automated compilation is crucial for CoBRA: The compiler must generate an optimal machine program with respect to the variants supported by the targeted machine. This requires to perform a number of complex program analyses, apply several scheduling or code generation techniques, and to combine the best results to an executable program afterwards.

2.2.3. Circuit Specification and Generation

In the following, several alternatives to specify the functionality of circuits of reconfigurable systems are presented. Finally, we consider the parallelization of sequential programs.

Hand-Mapping of Circuits Hand-mapping of circuits means to configure the system directly by using the basic building blocks of a reconfigurable device. Obviously, this strategy will only be useful, if no tools for circuit design are on-hand, or either small circuits or circuits with very high performance requirements must be implemented.

Structural Design Languages In order to abstract from certain reconfigurable architectures, structural design languages were developed. Such languages allow to describe a circuit with basic building blocks like gates, flip-flops, and latches. The descriptions are mapped to the logic elements of the actual reconfigurable system later. Both the design based on structural languages and the mapping process is supported by a multitude of commercial tools for the different FPGAs available on the market. As a benefit of the solution, the designer does not need any detailed knowledge about a certain reconfigurable architecture, but still requires experience in hardware design. The most famous examples for structural design languages are VHDL, Verilog, SystemC, and Handel-C.

Behavioural Description Languages The next step is to abstract from actual circuit realizations and to describe the behaviour of a circuit. Behavioural description languages are very similar to common HLLs such as C and provide e.g. function calls, looping constructs, and a sequential instruction stream. On the other hand, generalized pointer references or dynamic memory allocation cannot be synthesized in hardware, of course. In general, the term *behavioural* refers to the high-level modelling and simulation constructs of existing HDLs. VHDL and Verilog are mature, industry standard HDLs, which offer both behavioural and structural design, and are supported by a multitude of simulation and synthesis tools.

Algorithmic RC Languages Algorithms for reconfigurable systems can be compiled to the hybrid model, which consists of a fixed microprocessor coupled with reconfigurable logic.

Basically, two major approaches are used: Algorithms may be written using common *sequential languages* such as C, whereas computing-intensive kernels are mapped to hardware and the remaining parts are executed in software. This strategy benefits from the wide application of such languages, because they are also easy to use for people with some experience in software development, but no or just small hardware background. Furthermore, compilation can target both general-purpose processors and reconfigurable hardware. *Parallel languages* better express the capabilities of targeted hardware and can yield significant speedups compared to their sequential counterparts [62].

The PRISM compiler [6] was the first to apply hardware compilation for sequential C kernels. Transmogrifier C [61] can be used to describe hardware circuits and supports a subset of the C language for behavioural specification. But multiplication, division, pointers, and

arrays are not available. The P1 system [167] uses a C++ programming environment which provides a hybrid description method based on a combination of behavioural and structural design. The hardware/software partitioning conducted by the NAPA C compiler [65] is based on pragma directives to mark compute intensive parts of a program. Other approaches like the Nimble [113] or the Garp compiler [28] perform the partitioning automatically using profiling.

Handel-C is a commercial language used by the first parallel C compiler, which was derived from the Occam communication parallel process model [128]. The CoCentric compiler by Synopsys [89] for Xilinx Virtex FPGAs uses SystemC for behavioural compilation of C/C++ programs with the assistance of a set of additional hardware-defined classes. The Streams-C compiler [66] supports parallel processes that can either run on hardware and software. Communication between processes is realized with buffered FIFO streams.

Circuit Libraries and Generators Circuit libraries encapsulate complex designs to re-use them via a HDL like adders, multipliers, or counters. By these means, the design process can be simplified and accelerated dramatically. The user just integrates the predefined structures to his system without knowing their detailed implementation. Compilation can be done faster, because the library structures may have been pre-mapped, pre-placed, and pre-routed beforehand.

Circuit generators can create circuit specifications from a set of optimized high-level structures for certain architectures. Similar to circuit libraries, the user does not need to know the target system in detail. Furthermore, circuit generators provide more flexibility, because they can create structures according to the exact specification by the user. Circuit libraries provide only a restricted set of alternatives among the designer has to choose.

Parallelization As hardware is inherently parallel, parallelization is also an important topic for reconfigurable computing. Typically, this essential topic is targeted at three different levels: Instruction-Level Parallelism (ILP), Loop-Level Parallelism (LLP), and Thread-Level Parallelism (TLP).

Parallelism on instruction or loop level is often exploited automatically by a parallelizing compiler for reconfigurable hardware. The NAPA C compiler [65] detects fine-grained parallelism within computations to be executed in reconfigurable logic. The compilers of RaPiD [39] or XPP [172], for instance, unroll innermost loops completely and try to create a heavily pipelined structure. Some compilers even consider all loops within a program [169, 26] and do not rely on manual loop re-ordering. Manual parallelization is mostly used on thread or function level. In such cases, the programmer has to mark sections of code that should be executed in parallel by special compiler directives. One example is the RaPiD-C language [39] where TLP needs to be specified by hand, but LLP is exploited automatically.

In addition to this overview, Section 2.3 presents a multitude of reconfigurable systems and considers their compiler assistance more in detail. Gokhale and Graham [62] also consider task-level parallelism for processes and threads.

Summary As a drawback of the improved usability, all approaches relying on behavioural circuit description or HLLs produce larger and slower designs than those resulting from manual specification in general [38]. Furthermore, high-level description leaves some hardware aspects like mapping of control structures or bit width of data paths unspecified. This implies that several optimizations performed in a manual process cannot be applied. On the other hand, compilation for reconfigurable systems will improve in the future and seems to be most promising from a long-term view. In principle, automated solutions are easier to use, less error-prone, and faster than manual effort.

CoBRA The CoBRA compiler accepts source code programs written in ANSI C and produces an optimal machine program transparently for the user. Instead of generating hardware configurations, the architectural variants need to be implemented before the actual compilation. The variants are known to the compiler, which determines the best variant for each part of a given source code program. At run-time, the corresponding variants are activated by executing reconfiguration instructions. Importantly, difficult tasks like hardware/software partitioning or generation of hardware configurations can be neglected by the CoBRA compiler. Hence, most of the challenges which may lead to suboptimal results for common reconfigurable systems are avoided. Parallelization is obviously a central topic for CoBRA, because different parallelization paradigms like MIMD or SIMD can be provided easily by both the reconfigurable machine and the optimizing compiler.

2.3. Existing Reconfigurable Computing Systems

Most reconfigurable computing systems combine a general-purpose core with reconfigurable logic [176, 119, 67, 180, 28, 150, 156]. Consequently, the execution of applications can be improved by mapping expensive computations to the reconfigurable hardware. The fixed microprocessor acts as a controller for the reconfigurable logic and executes code which cannot be accelerated efficiently. For example, programmable logic is quite inefficient to implement variable-length loops or branch control.

Some reconfigurable architectures [48, 158, 9] operate stand-alone or information about a coupling is not available. Those systems are mostly based on Coarse-Grained Reconfigurable Arrays (CGRA) or other coarse-grained programmable logic.

Many researchers use reconfigurable logic to implement arbitrary functions in hardware to achieve both performance and flexibility. Such implementations often resemble specialized devices like ASICs, but not software-programmable microprocessors.

Focus of Section This section categorizes and discusses several reconfigurable approaches. As this thesis concentrates on reconfiguration of processors, we only consider systems operating as a processor which executes machine code. Concretely, our explanations concentrate on the fundamental ideas and concepts with respect to the associated application domains. Additionally, we evaluate the existance and quality of compiler support as well as proper hardware implementations.

For simplification, many technical details from system and circuit technology and related

areas are skipped. In order to get more detailed information with close relation to reconfigurable computing, the reader may refer to the excellent survey of Compton and Hauck [38]. Further knowledge is provided by two surveys of Hartenstein [74, 73].

Levels of Coupling Figure 2.4 illustrates the levels of coupling a microprocessor with reconfigurable logic which are explained using practical examples in the following four Sections 2.3.1 to 2.3.4. Section 2.3.5 deals with other coarse-grained architectures which are not coupled to a processor or where the kind of coupling is not known. Finally, Section 2.3.6 summarizes the most important aspects and discusses the approaches from the perspective of application domains, hardware platforms, and compiler assistance. As the description of related work is quite detailed, the reader may just turn over to this summary in order to get the fundamental information.

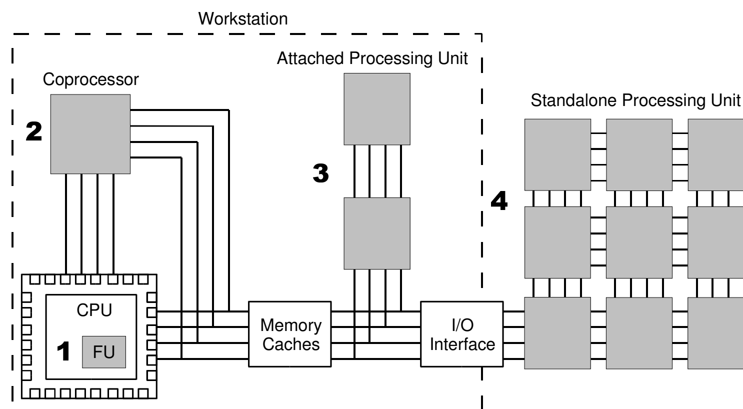


Figure 2.4.: Levels of coupling in a reconfigurable system. Reconfigurable logic is drawn shaded [38].

2.3.1. Level 1: Functional Unit

On the first level, the host processor is augmented with Reconfigurable Functional Units (RFU) which enable the implementation of custom instructions that may be changed during run-time. An outstanding characteristic is the tight coupling due to the integration into the data-path of the original processor.

2.3.1.1. PRISC

PRISC [140] extends a conventional RISC instruction set with application-specific instructions that are implemented in FPGA-based RFUs. The compiler couples program analysis routines and hardware synthesis tools to automate the process.

The hardware extraction phase first identifies sets of sequential instructions which can potentially be implemented in a RFU. These sets are mapped to boolean operations and forwarded to a hardware synthesis package, which outputs a netlist of LUTs. Then the number

of LUTs and interconnect resources is optimized using logic minimization algorithms. Finally, placement and routing is performed to determine if the LUT netlist fits in the available area. If not, the hardware extraction will be called again to reduce the input functions.

2.3.1.2. Chimaera

The RFU of Chimaera [180] acts as a cache for instructions, which have been executed recently or might be needed soon. It is realized using FPGA technology and offers partial run-time reconfiguration to reduce latency costs.

RFU operations have unique IDs and are called via a special instruction that is executed by the main processor. If the requested operation is not currently loaded into the RFU, the host processor will be stalled while the RFU fetches the operation from memory and reconfigures itself afterwards. As this reconfiguration is quite costly, several techniques such as prefetching [75], compression [76, 114], caching algorithms and hierarchies were investigated. In [181], a C compiler for Chimaera is presented, which automatically maps computations for execution in the RFU. After applying standard compiler optimizations, some branches are transformed into single macro instructions to gain larger basic blocks. Then all loops are inspected for subword operations and unrolled if necessary. At last, multiple-input-single-output patterns are extracted from the Data-Flow Graph (DFG) and combined to RFU operations if possible. The generated code is currently run on a Chimaera simulator to gather performance information. Hardware synthesis will probably be integrated into future versions of the compiler.

2.3.1.3. Other approaches

Older approaches which use the concept of RFUs are e.g. the PRISM systems [6, 170] or Spyder [90]. The prototype systems are realized as board-based solutions where the reconfigurable hardware appears as a loosely coupled coprocessor. However, both approaches were assigned to the first level, because the actual reconfiguration is performed on the Functional Unit (FU) level.

The PRISM compiler synthesizes entire C functions as new operations to adapt the main processor to the characteristics of a certain application. But the approach is not fully automated and requires some user interaction during compilation.

Spyder is a superscalar processor with three RFUs which are implemented using FPGAs. Unfortunately, there exists no appropriate compiler to partition a source program automatically. Instead, a programmer has to provide three different programs for the host workstation, the configuration of the RFUs as well as the control program. Furthermore, the lack of fixed FUs for typical integer or floating-point operations reduces the performance significantly.

Pozzi [134] developed a formal design methodology to customize the instruction set of a VLIW machine. Additionally, an algorithmic approach was evolved to partition application code for the execution on fixed and reconfigurable FUs. Consequently, this work combines the time efficiency of application-specific FUs, the flexibility of programmable devices as well as an architecture supporting ILP.

To our best knowledge, their work has not yet been used for a holistic approach consisting of a hardware implementation and a suitable compiler. However, it should be mentioned when discussing approaches using (multiple) RFUs.

2.3.2. Level 2: Coprocessor

The second level introduces reconfigurable units as a coprocessor to execute more coarse-grained computations without the constant supervision of the host processor. In contrast to a RFU, the processor first sends the necessary data or a memory reference to the reconfigurable device. Then, the coprocessor operates on the given data independently of the main processor and returns the results after completion. By these means, the communication overhead is reduced compared to a RFU which needs to communicate with the main processor for each reconfigurable instruction.

2.3.2.1. OneChip

OneChip [176] integrates a RFU into the pipeline of a superscalar RISC processor. It was designed to speed up computing-intensive applications of the multimedia or DSP domain by optimizing instructions at the loop-level.

OneChip offers dynamic scheduling and reconfiguration, pre-loading of configurations, as well as Least Recently Used (LRU) configuration management. The RFU consists of at least one FPGA as well as a controller. In favour of a fast switching among configurations, the FPGAs provide multiple contexts [40]. While only one context may be active at a given time, each context can be configured independently of the others. The FPGA controller switches between different contexts and replaces the configurations in the FPGAs.

Currently, only a working prototype exists that emulates the OneChip configuration. To evaluate the performance of the OneChip architecture, a simulator called Sim-OneChip was developed. While the default programming model for OneChip is the use of circuit libraries, Sim-OneChip can be programmed in C.

A programmer can re-use existing configurations or create custom ones which need to be implemented in C using several macros for accessing memory or instruction fields. All specified program configurations are then compiled with the simulator source code to produce a special simulator for the current configurations. A program is also written in C and contains calls to RFU instructions. It is compiled using a compiler for the RISC processor and then executed using the previously built simulator.

2.3.2.2. Garp

Garp [77, 28] was designed to fit intoan ordinary processing environment. It combines a single-issue MIPS processor core with a primarily mesh-based Reconfigurable Array (RA) to accelerate loops. The 32 by 24 RA is composed of logic blocks, which resemble the CLBs of common FPGAs, as well as a control block for each row.

The first prototype of Garp [77] did not provide compiler support for a HLL. Configurations for the RA had to be written in an assembler-like specification and were converted to a binary representation using a special configurator. The remainder of a program was written in C, compiled with an ordinary C compiler and executed on the main processor. To make the configuration accessible to the program, a configuration had to be linked into it and additional assembly code for invocation had to be provided.

With the new prototype [28], Garp can be programmed in ANSI C without any special directives for hardware/software partitioning. The compiler uses the concept of a *hyperblock* [116], which was developed for VLIW machines originally. A hyperblock consists of all basic blocks along frequently executed control paths of a loop body. Such a hyperblock is mapped to the RA, while the excluded paths are executed on the fixed processor. If the execution reaches an excluded path from a hyperblock, an exception will be raised to give control back to the MIPS core. The number of exceptional exits is minimized by using profiling to reject loops which cannot be mapped efficiently to reconfigurable hardware.

Currently, Garp programs are executed on a simulator, because there exists no complete hardware implementation. Only critical parts of the circuit layout were developed and evaluated in terms of clock speed, power consumption, and area needs.

2.3.2.3. MorphoSys

MorphoSys [150] is targeted at applications with inherent data-parallelism, high regularity and high throughput requirements. It combines a RISC processor with a mesh-connected 8 by 8 Reconfigurable Array (RA) to accelerate loops. The RA is organized as a SIMD array of coarse-grained reconfigurable cells to handle computing-intensive operations. The RISC processor is used for sequential processing and controls the RA.

The programming environment consists of a C compiler accepting hybrid code as well as a graphical user interface and an assembler to create configurations for the RA. The C code for an application must be partitioned manually by inserting a particular prefix to functions that should be mapped to the reconfigurable hardware. The compiler generates machine code for the RISC processor, which contains special instructions to invoke execution on the RA. Furthermore, MorphoSys comprises two simulators to analyze the mapping of applications and to evaluate the performance. The C++ simulator supports cycle-accurate simulation and debugging of an application, while the VHDL simulator is used for runtime measurements, especially. The evaluation has shown that MorphoSys can yield similar performance compared to ASICs and a significant speedup over optimized Pentium MMX instructions [118].

2.3.2.4. REMARC

REMARC [119] is a reconfigurable coprocessor that was designed to accelerate multimedia applications, such as video compression, decompression, and image processing. It is tightly coupled to a RISC processor and consists of a global control unit as well as 64 programmable logic blocks called nano processors. The logic blocks are organized in a 8 by 8 array, which is connected to the global control unit. REMARC has a pipelined execution which overlaps with the pipeline stages of the host processor.

Configuration data for the global control unit and the nano processors need to be generated from assembly code using the REMARC assembler and the nano assembler, respectively. The configuration data are written in text and can be imported into C source code, which is compiled by a standard C compiler using the REMARC assembler. The executable program includes the new main processor instructions as well as the global and nano configuration data.

2.3.2.5. NAPA

NAPA [144] combines a small RISC processor with reconfigurable logic. In contrast to REMARC, NAPA suspends the main processor during execution on the reconfigurable hardware. Simultaneous computation can only be realized using fork-and-join primitives known from multi-processor programming.

The authors developed a special language called NAPA C as well as a proper compiler [65]. NAPA C provides a simple set of pragma and intrinsic function directives to define the partitioning between fixed processor and reconfigurable hardware. The compiler generates both a conventional C program for the RISC processor as well as configurations in the form of Verilog netlists. Each loop that should be realized in hardware is mapped to a hardware pipeline automatically, if possible. The resulting C source code needs to be compiled using a native compiler for the RISC core. It contains calls to a runtime software library to load configurations, invoke execution on the reconfigurable hardware, and to read back state after returning.

Although the NAPA compiler does not provide a fully automated partitioning, it offers to iteratively refine a program written in NAPA C. The user may start with a program which is executed completely on the fixed processor and then move the functions one-by-one to the reconfigurable logic.

2.3.2.6. Chameleon

In the Chameleon project [152, 88], a coarse-grained reconfigurable core was developed for DSP algorithms in wireless devices such as handheld devices, mobile multimedia players, etc. The research concentrated on important aspects like small size, flexibility, performance and especially energy-efficiency next to reconfiguration. The realization of the hardware architecture could re-use well established development tools like VHDL simulators and synthesizers. Instead, the implementation of the compiler backend has been started from scratch and is still ongoing.

2.3.2.7. TU Dresden

The group of Spallek at TU Dresden conducted extensive research to accelerate DSP and multimedia applications, which consist of many computing-intensive loops. Their first target architecture was a hybrid VLIW machine [99, 23] consisting of hard-wired FUs and additional RFUs. The RFUs are implemented by the means of FPGA technique and represent additional data paths inside the processor. In contrast to common instruction-level FUs, the

RFUs exploit (more) coarse-grained parallelism to map whole loop bodies. Later, this approach was extended to RISC microprocessors with a tightly coupled reconfigurable ALU array [98].

All approaches use a retargetable compiler back-end based on the SUIF compiler kit [175] and a special simulation environment. The compiler includes an intermediate representation profiler to estimate control/data flow, especially loop execution count. Furthermore, the compiler backend performs hardware synthesis and produces FPGA bitstreams for the RFUs. The recent approaches comprise a framework for design space exploration which includes an architecture description language and a hardware model.

2.3.3. Level 3: Attached Processing Unit

The third possibility is to attach a Reconfigurable Processing Unit (RPU) as an additional processor to a multi-processor system. Alternatively, a RPU can be used as a further compute engine which is accessed through external I/O.

2.3.3.1. PipeRench

PipeRench [67, 110] has three fundamental motivations: Firstly, future applications will predominantly execute relatively simple computations on a huge data basis. Secondly, most reconfigurable approaches are restricted to the area provided by certain programmable logic devices. Finally, many existing systems cannot be adapted to exploit additional resources available in future process generations automatically.

PipeRench enables the implementation of large logical designs on small pieces of hardware by fast run-time reconfiguration of that hardware. It uses a technique called pipeline reconfiguration [145] that virtualizes hardware applications by partitioning them into virtual pipeline stages called stripes. Those virtual stripes can be loaded separately, one per cycle, into the physical stripes of the device. Consequently, one physical stripe can perform its computation on the incoming data, while the next stripe configures itself for the next pipeline stage. By these means, a v-stage application can be mapped to a p-stage device ($p < v$). Currently, PipeRench is used as an attached processor and might be integrated into a CPU in the future.

The compilation process maps source code written in a Dataflow Intermediate Language (DIL) to a particular instance of PipeRench. DIL is a single-assignment language with C operators and a type system that allows the bit-width of variable to be specified. The compiler converts the source into a dataflow graph first. After applying module inlining and loop unrolling, a Single Assigment Program (SAP) is generated. Finally, the SAP is broken into nodes fitting on one stripe and a placement/routing algorithm tries to add the nodes to stripes.

2.3.3.2. PAM, Splash

PAM [13, 167] was one of the first reconfigurable computing systems to solve domain-specific problems. The approach partitions computations between a host processor and pro-

grammable boards from Xilinx connected via the I/O bus. Synchronous circuits need to be specified as C++ programs which use a special library to describe combinatorial logic. The execution of such programs builds a netlist in memory that can be analyzed, transformed in an appropriate format, or just used for simulation. PAM showed good results for long integer multiplication [148] and RSA encryption [149].

Splash [63] mimicks the PAM model and was used successfully for text searching or DNA comparison. For instance, it yields a speedup of factor 300 compared to a Cray-II concerning the searching of DNA sequences. Like PAM, Splash is programmed by specifying combinatorial logic. Instead of using a special C++ library, existing templates describing logic functions need to be manipulated via a Common Lisp language.

Unfortunately, the board-based manner of both approaches implies a high communication overhead. This limits their applicability to a class of algorithms with both a high computational complexity and a low communication overhead.

2.3.4. Level 4: Standalone Processing Unit

On the fourth level, an external stand-alone processing unit is connected to a workstation and communicates infrequently with a host processor (if present). Example systems are quite rare, because the approach suffers from a high communication overhead. On the other hand, it allows much more parallelism than the three levels discussed beforehand.

Some practical examples [136, 135] were developed by Quickturn systems, but are more focused on hardware emulation than reconfigurable computing. For instance, the Mercury system was used successfully to verify the OneDSP architecture [178].

2.3.5. Other coarse-grained approaches

In this subsection, those reconfigurable computing systems which could not be assigned to a level according to Figure 2.4 are considered.

2.3.5.1. RaPiD

RaPiD [48] is a coarse-grained architecture that allows the configuration of very deep application-specific computation pipelines. Consequently, it is optimized for highly repetitive, computing-intensive tasks. Each pipeline consists of ALUs, multipliers, registers, and local memory. Input and output data enter and exit the datapath via streams at each end of the datapath. These streams act as an interface to the external memory.

RaPiD is programmed in RaPiD-C [39], a C like language with extensions to explicitly specify TLP, data movement, and partitioning. Parallelism on loop-level is exploited automatically by the RaPiD compiler. Innermost loops are unrolled completely and each iteration is mapped to a pipeline stage of the reconfigurable hardware.

According to [48], RaPiD should be closely coupled to a generic RISC processor, but there are actually no concrete ideas how this could be realized. To our best knowledge, there exists no recent publication which describes the implementation of a complete system. If RaPiD is

used as a coprocessor, it can obviously be assigned to level 2. Otherwise, it would probably be attached as an additional processor which corresponds to level 3.

2.3.5.2. RAW

The philosophy of RAW [168, 158] is to fully expose all details of the hardware architecture by replicating simple tiles and arrange them in a network. This allows the compiler to determine the best schedule for each application by using static program analysis. RAW machines can be much simpler designed than modern general purpose processors. They need only short internal chip wires which offers high clock speeds that scale with the feature size. Furthermore, an architecture consisting of simple, replicated tiles can be verified quickly. Last but not least, a RAW machine is well-suited for pipelined parallelism found in typical multimedia applications.

Each tile contains instruction and data memories, an ALU, registers, and reconfigurable logic. The tiles are arranged in a network and linked using programmable, tightly integrated interconnects. The network interface operates in a synchronous manner which allows a fast communication with latencies similar to those of register accesses. Additionally, static scheduling avoids explicit synchronization by arranging instructions such that operands are really available when needed. If the compiler fails to find a static schedule, a RAW machine can be reconfigured for dynamic routing between the tiles.

As the RAW architecture distributes all its resources to the different tiles, a compiler exploiting ILP needs to perform both spatial as well as conventional temporal instruction scheduling. In [112], a technique called *space-time scheduling* used by the RAW compiler is presented.

The prototype chip [158] contains 16 tiles arranged in a 4 by 4 array. Alternatively, there exists a cycle-accurate simulator. According to the publications, RAW is intended to be used as a stand-alone processor. Consequently, it cannot be assigned to one of the four levels illustrated in Figure 2.4.

2.3.5.3. PACT XPP

XPP [9] is a commercial runtime-reconfigurable data processing architecture which was designed to support different types of parallelism: pipelining, instruction-level, data-flow, and task-level parallelism. It is based on a large number of parallel scalable processing and routing elements organized in a hierarchical array. Reconfiguration is either controlled from outside or can be initiated by special events within the array to enable self-reconfiguration. Next to reconfiguration at run-time, even parts of the array can be modified while the remaining elements are operating.

A configuration consists of parallel computations derived from the DFG of an algorithm, which are mapped to machine operations such as multiplication, addition, etc. The operations are implemented by configurable ALUs in the processing elements. The ALUs communicate via an automatically synchronizing, packet-oriented network based on Terabit communication channels. During a computation, no operations or connections are modified.

The fundamental idea of XPP is the combination of data-stream processing in an array configuration with dynamic reconfiguration. For each configuration, as long data streams as possible are processed in parallel to compensate the effort in reconfiguration. After finishing the computations for a configuration, a new configuration is applied which updates the operations of the ALUs and their interconnections. Then, new computations are started which can re-use results of previous configurations stored in distributed memories or FIFOs.

Programming and Compilation In order to achieve best results, the Native Mapping Language (NML), a proprietary structural language with reconfiguration primitives, was developed. It exposes all hardware features to the programmer and allows allocation and placement of objects as well as specification of connections. NML source files are compiled using a mapping utility which generates code for either the hardware implementation or a simulation environment.

Additionally, a vectorizing C compiler [30] was realized which transforms C functions to NML modules supporting a restricted subset of the C language. Concretely, struct and floating-point data types, pointers, irregular control-flow, and recursive or system calls are excluded. The compiler vectorizes innermost for-loops by a special technique called *pipeline vectorization* [171], which unrolls loops and overlaps iterations for pipelined execution.

Due to the authors, XPP is well suited for a number of applications in multimedia, telecommunications, simulation, signal processing (DSP), graphics, and similar stream-based domains. This wide applicability is enabled by the many different supported types of parallelism like pipelining, instruction-level, data-flow and task-level parallelism.

A prototype of the XPP architecture was implemented as a chip by the PACT company. The evaluation has shown that the approach yields outstanding improvements compared to standard processor and DSP implementations, while maintaining much more flexibility than ASIC implementations.

2.3.6. Discussion

In this section, the lessons learned from the presentation of the related work in Section 2.3.1 to Section 2.3.5 are summarized. At first, we review the application areas of the studied approaches as well as the classification according to the four levels of coupling. Then we discuss briefly the hardware platforms used for implementation. The main part concentrates on the fundamental properties of provided compilers if present. Finally, we draw a conclusion for the compiler assistance, because this is the core topic of this thesis. In all cases, the approaches are compared with CoBRA.

Application Areas The majority of the outlined approaches is focused to special application domains. Only a small number of systems like Garp (2.3.2.2) is tailored to general-purpose software applications. But even in this case, reconfiguration just aims for accelerating loops.

Our reference architecture, the QuadroCore consisting of four tightly coupled S-Core processors [105, 21], has been developed originally for networking applications. By using Co-

BRA, it can support a number of different application areas. For instance, programs with both regular and non-regular structures can be executed faster and more energy-efficient when using a SIMD/MIMD reconfiguration. Additionally, reconfiguring the register banks can speed up applications with a high register pressure or significant communication overhead.

Classification In most related architectures, reconfigurable logic works as a coprocessor, operates stand-alone, or the kind of coupling to a fixed core is not known (2.3.5). The first type is very successful, because it enables the exploitation of coarse-grained parallelism on loop-level and is not restricted to ILP or Instruction Set Extension (ISE). Furthermore, the effort in communication is kept low in contrast to more loosely coupled styles. Systems belonging to the latter type often consist of a highly-parallel network architecture, which offers a high performance for suited applications.

CoBRA cannot really be classified according to the four levels, because supported architectures do not consist of two coupled components, fixed processor and reconfigurable logic. Instead, reconfigurability can be provided in many different ways: For instance, the connections between ALUs and register banks could be reconfigurable to allow more freedom in register allocation. On the other hand, multiple processors can be combined to a SIMD machine. Consequently, CoBRA belongs to the special group discussed in Section 2.3.5.

An alternative perspective is to abstract from the multitude of reconfiguration variants supported by CoBRA and to treat the reconfigurable components as a whole. Then, CoBRA could be assigned to level 1 or 2, because it mostly targets reconfiguration in a processor or between processors. Obviously, coupling a fixed machine with external reconfigurable logic via an I/O bus or a network connection is not part of our research.

Hardware Support Most approaches have been implemented using commercially available FPGAs or comparable hardware-programmable logic. Garp (2.3.2.2) and MorphoSys (2.3.2.3) are based on more coarse-grained RAs. Other systems like e.g. REMARC (2.3.2.4) or RAW (2.3.5.2) even consist of simple processor tiles organized in a network. The network of RAW can even be reconfigured to enable dynamic routing at run-time. Some approaches like Garp or OneChip (2.3.2.1) provide no complete hardware implementation or only an early prototype. In these cases, the code is executed on a software-based simulator only. On the other hand, RAW, XPP (2.3.5.3), and Chameleon (2.3.2.6) were even implemented as ASICs. As this thesis is focused on compiler techniques rather than hardware aspects, detailed information can be found in the papers referred to above.

Our long-term goal is an ASIC implementation of the reconfigurable QuadroCore to evaluate its performance. Obviously, we expect best results, because reconfiguration is restricted to modifying connections between fixed components in a mainly hard-wired processor.

Compiler Support The compiler assistance of the related approaches can be classified in four groups: The first category (full) comprises all approaches providing *full* compiler support for a HLL such as C. The compilation process is transparent to the user, i.e. no further input is necessary to partition the source code between a general-purpose processor and the

reconfigurable logic. With respect to the related work presented in this thesis, about 37.5% of the approaches like Chimaera (2.3.1.2), PRISC (2.3.1.1), and RAW (2.3.5.2) belong to this category. A special case in this group is the C compiler for XPP which supports only a restricted subset of C, but still hides decisions and technical details from the programmer.

Approaches of the second group (hybrid) also provide a compiler for the C language or a variant, but the program code has to be partitioned manually. The group itself consists of three subgroups: Firstly, the compilers of Garp (2.3.2.2), MorphoSys (2.3.2.3) and RaPiD (2.3.5.1) accept modified C languages where the partitioning must be specified explicitly by the programmer. Secondly, the PRISM (2.3.1.3) compiler even requires some user input during compilation to decide for a certain HW partitioning. Thirdly, Spyder (2.3.1.3) is programmed by writing three different programs which are compiled separately. Hence, the only benefit of a compiler in this case is the usage of a HLL instead of assembly language. OneChip (2.3.2.1) also belongs to the third subgroup, when its simulator is targeted, because the user has to implement both the entire program as well as the configuration in C. All in all, about 37.5% of the approaches presented beforehand make up this group.

The third group (assembly) contains about 6% of the outlined approaches like REMARC (2.3.2.4) where the behaviour of the reconfigurable logic needs to be specified in assembly language. Only the program code running on the hard-wired microprocessor can be implemented in C and must be linked against the configuration files.

Approaches of the fourth category (circuit) have to be programmed by specifying combinatorial logic, possibly in a HLL such as C or C++. Next to PAM and Splash (2.3.3.2), OneChip belongs to this group, when the protypical hardware is targeted. The group consists of about 19% of the approaches presented beforehand.

Conclusion The majority of about 60% of the considered related work has no full compiler assistance. Instead, the programmer has to partition the program code manually and must use complex programming environments with a lot of tools. Some approaches even require implementing configurations in assembly language or specifying circuit behaviour.

Reconfiguration of variants is based inherently on a comprehensive compiler support. The CoBRA compiler transforms C source code automatically into a machine program which contains special instructions to reconfigure the target machine. Consequently, no further input or additional tools are needed to yield best results in a short time.

In the next section, we present the fundamental ideas of CoBRA and discuss some key challenges.

3. Dynamic Reconfiguration of Variants

Contents

3.1 Reconfiguration of Variants **I-33**
3.1.1 Reconfiguration of Parallelization Paradigm I-34
3.1.2 Reconfiguration of Register Access I-36
3.1.3 Reconfiguration of Machine Topology I-37
3.1.4 Dynamic Assignment of Special Functional Units I-38
3.1.5 Combination of Instructions between Processors I-38
3.2 Application Scenarios **I-38**
3.2.1 Compiler-Driven Reconfiguration I-39
3.2.2 Need for Compiler-Supported Reconfiguration I-40
3.2.3 Compiler-Supported Reconfiguration I-41
3.3 Compiler for Reconfiguration of Variants **I-43**
3.3.1 Prototypical System I-44
3.3.2 Code Integration I-45

Motivation and Introduction Automatic compilation is only supported by about 40% of the existing reconfigurable systems (see Section 2.3.6). Typical approaches are restricted to certain application domains like multimedia or DSP. The primary goal is to speed up execution by mapping certain parts of a program like loops to reconfigurable logic. Such a compiler identifies loops and selects the best candidates for execution in hardware. Hence, the compiler is customized to a special hardware developed for a certain application domain. Generally spoken, those compilers are merely a means to an end in order to automatize code generation for typical reconfigurable systems.

From our perspective, it will be much more beneficial, if the type of reconfigurability is adapted to the abilities of a compiler. For example, a multi-core may operate in a MIMD or a SIMD mode dependent on the regularity of program code. The compiler can identify those parts of a given program suited for MIMD or SIMD execution.

With our methodology called CoBRA[1], a reconfigurable machine offers different alternatives of a certain architecture, which are supported directly by a compiler. Such an alternative is called *reconfiguration variant* or briefly *variant* in the following. The compiler applies program analysis techniques to partition given program code into sections and to determine the best variant for each section. Then it generates a machine program which contains special reconfiguration instructions to activate a certain variant where needed.

Importantly, the usage of several architectural variants provides an adaption to a wide range of application areas. For instance, a SIMD execution is well-suited for regular program structures which can be found in scientific or multimedia computations. The remaining non-regular parts of a program can be executed in a MIMD manner to exploit the inherent ILP. As a consequence, a number of different application domains can be supported by offering just two architectural variants.

Structure This chapter covers the fundamental aspects of CoBRA and presents key decisions as well as basic concepts. Most explanations refer implicitly to the QuadroCore used as a reference architecture throughout this thesis (see Section 4.1).

Section 3.1 outlines the central ideas of an ideal reconfigurable machine offering architectural variants and proposes five selected types of variants. Two of them, the SIMD/MIMD reconfiguration (see Part III) as well as the reconfiguration of register banks (see Part IV) are studied in this thesis and will be handled in detail.

Section 3.2 motivates two application scenarios mostly differing in the knowledge the compiler has about the machine. In both cases, we explain the structure of the systems based on compiler and reconfigurable target machine. This includes a rough overview of the basic tasks and their temporal order.

Section 3.3 substantiates the first scheme and suggests the design of a compiler from a practical perspective. Concretely, we first outline how the structure could look like in principle and then motivate some simplifications and restrictions applied for the prototypical implementation. Furthermore, challenges like selecting the best results for certain variants as well as placement of reconfiguration instructions are discussed.

[1] **Co**mpiler-**D**riven **D**ynamic **R**econfiguration of **A**rchitectural Variants (merge two *D*s to a *B*)

3.1. Reconfiguration of Variants

From the perspective of an optimizing compiler, the ideal reconfigurable machine consists of fixed components like ALUs or register banks. Reconfiguration means to switch the building blocks alternatively during program execution. Figure 3.1 shows our reconfigurable QuadroCore (see Section 4.1), which offers two possibilities for reconfiguration. The solid lines represent the default connections between the components: Each processor consists of an ALU which is connected to an instruction decoder, a register bank, and a local memory. Consequently, the default mode corresponds to a MIMD execution where each processor can only access its own register bank.

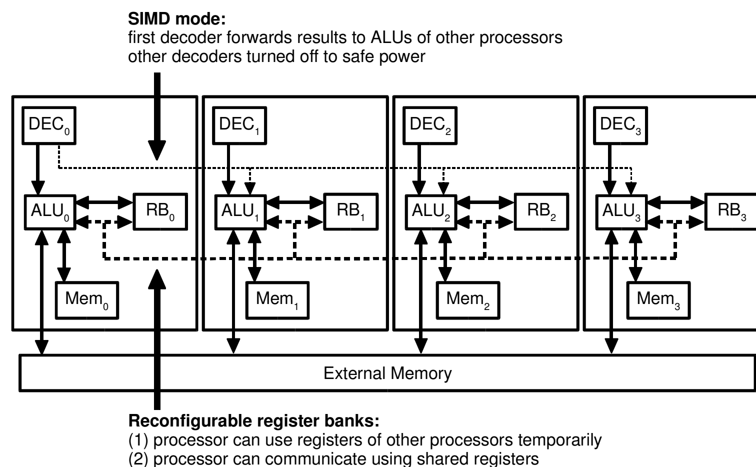


Figure 3.1.: Reconfiguration of variants

In the SIMD mode, there is only one instruction stream decoded by the first processor, but executed on all processors with different data. This is indicated by the thin dashed lines between the instruction decoder of the first processor as well as the ALUs of all processors.

The second reconfiguration targets the connection between the ALUs and the register banks and is signified by the thick dashed lines. A processor can use registers of another processor temporarily to avoid expensive spilling. Furthermore, registers could be used in a shared manner to establish a fast communication.

As mentioned above, the feasible alternatives of an architecture enabled by such reconfiguration are denoted as *variants*. Multiple associated variants like MIMD and SIMD form a *reconfiguration dimension*. In general, a variant might also combine multiple variants of different dimensions. Figure 3.2 illustrates the 2-dimensional space for the two dimensions offered by the architecture shown in Figure 3.1. The first dimension called *parallelization* consists of the SIMD/MIMD modes as well as the trivial single processor mode. The second dimension *register access* comprises multiple variants of the connections between ALUs and register banks (see Section 10.2.1), whereas our approach is limited to arbitrary mappings and focuses on the actual reconfiguration of register connections.

Structure In the following, we consider five selected reconfiguration dimensions more in detail with respect to the QuadroCore. Figure 3.3 illustrates the proposals which are num-

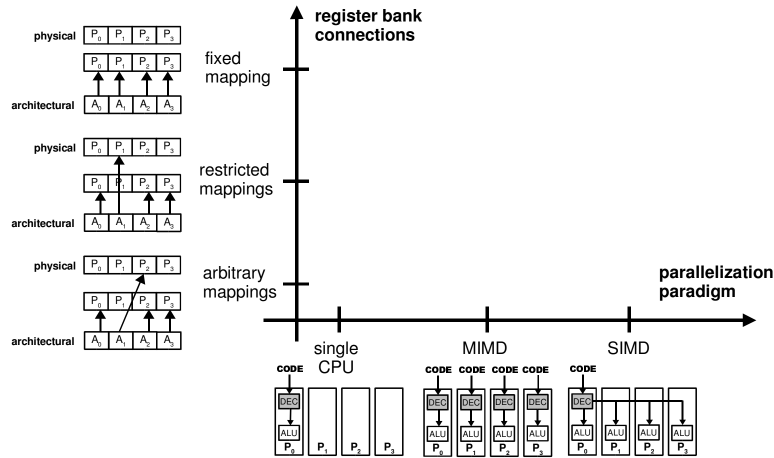


Figure 3.2.: Space of reconfiguration variants

bered in clock-wise order. This thesis focuses on the first two dimensions, which are characterized in Section 3.1.1 and Section 3.1.2. The remaining ones are mentioned briefly as further ideas which could be studied in future (see Section 3.1.3 to Section 3.1.5). The concepts evolved for the first two dimensions are presented in Part III and Part IV, while the evaluation can be found in Section 13.3 and Section 13.4, respectively.

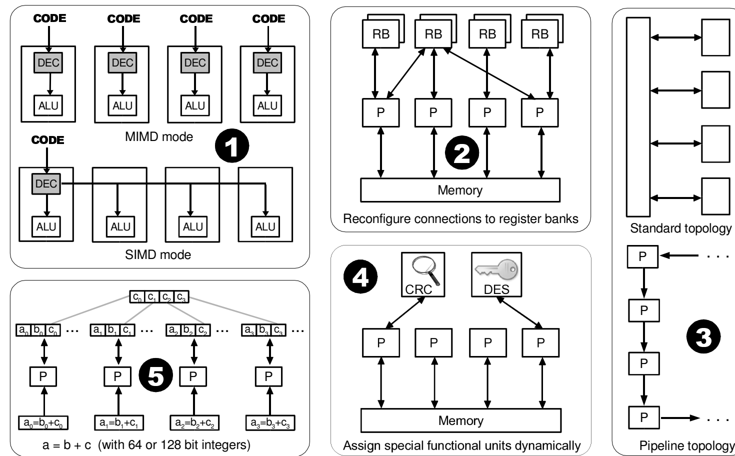


Figure 3.3.: Reconfiguration variants of CoBRA

3.1.1. Reconfiguration of Parallelization Paradigm

In the MIMD mode, all processors execute disjoint instruction streams on different data. This behaviour is beneficial for unstructured or non-regular program code with a high degree of ILP. Importantly, this does not imply that the data accessed by the processors is always disjoint in general.

Many programs of the multimedia domain or scientific computations have a regular structure which is best suited for SIMD execution. Instead of providing four machine programs,

only one instruction stream is executed by all processors on different data simultaneously. The instructions are decoded by the first processor which forwards the results of the decoding to the ALUs of all processors. As a result, the remaining instruction decoders can be turned off in order to lower the energy consumption. The usage of one instruction stream instead of n ones, where n corresponds to the number of processors, may reduce the code size by factor n. With respect to the QuadroCore comprising four processors, the code size would be shrunk by 75%.

SIMD/MIMD Reconfiguration Switching between MIMD and SIMD execution becomes useful, if programs executed on the multi-core contain both regular and non-regular structures. Instead of deciding statically for one execution mode, the compiler can identify the parts of a program suited for MIMD or SIMD execution and insert appropriate reconfiguration instructions between them. Surely, turning off the instruction decoders in SIMD mode only makes sense if its duration is greater than the startup time of the decoders.

For instance, aggregation network access node, like DSL Access Multiplexers (DSLAM), may transcode multimedia data to suit the customer's equipment. Consequently, the software executed by such devices probably contains regular structures for multimedia computations as well as non-regular structures for tasks related to the actual transmission. The QuadroCore represents the ideal target machine for such reconfiguration, because it has originally been developed for networking applications and consists of four homogenous cores.

Low Parallelism If there is not enough parallelism available in some situations, a subset of the four processors can be used for execution. Hence, the other processors can be shut down completely to save a lot of energy. This idea works both in SIMD and in MIMD mode. A special case is the usage of a single processor when no parallelism is available or the effort in communication between multiple processors is larger than the speedup gained by parallelization.

For simplification, this thesis concentrates on the reconfiguration between SIMD and MIMD execution. We have developed an original approach called CHARISMA[2], which is presented in Part III. Switching between different numbers of processors is not handled. In a prior research project, we already studied a similar scenario where machine code needs to be adapted to a superscalar processor at load-time by regarding the types and numbers of its functional units. Thereby, we developed an original approach for efficient load-time scheduling based on code annotations generated by the compiler [85]. The annotations enable a fast adaption of a program to a certain target machine in linear-time. Obviously, the evolved analysis techniques could be re-used to realize switching to modes using only a subset of the processors.

Reconfiguration of Granularity In principle, a multi-core architecture may also adapt to different levels of parallelism like instruction, loop, or task. Currently, the CoBRA compiler only exploits fine-grained parallelism within basic blocks (see introduction of Part II). The basic idea idea of reconfiguring the granularity has been discussed briefly in Section 1.3.

[2] **C**ompiler **H**andles **A**rchitectural **R**econfiguration **I**ntegrating **S**IMD **M**IMD **A**utomatically

3.1.2. Reconfiguration of Register Access

With the default configuration, each processor can only access its own register bank. If a processor needs temporarily more physical registers than available, it must spill register values to memory, while a different processor may not use all of its registers. A reconfiguration of the connections between ALUs and register banks offers more freedom in exploiting *all* registers of the multi-core. A processor can temporarily use registers from other processors instead of executing costly spill code.

Furthermore, reconfiguration may establish shared registers which can be used to speed up communication. By default, register values are exchanged via the external memory or a dedicated register bank (see Section 6.1.1), whose access times are much larger than reading or writing the registers of a processor.

Consequently, reconfiguring the connections to the register banks has two benefits: (1) exploiting additional registers and (2) fast communication using shared registers. The second advantage will only be visible, if multiple processors are targeted, of course. The first benefit can also be observed for a single processor which owns more physical registers than architecturally available. One example is the S-Core processor employed in the QuadroCore, which has two register banks that can only be used alternatively by default. Reconfiguration helps to exploit additional registers normally not available to a program in order to avoid expensive spill code.

Reconfiguring the connections to the register banks is addressed in detail by Part IV, which presents our approach denoted as CAPRICoRn[3]. The first prototypical implementation requires that reconfigurable register access is used throughout a complete program. Furthermore, we allow arbitrary mappings to the physical registers and do not distinguish between local and remote accesses (see Section 10.1.3). A more general register architecture is proposed in Section 10.2.

Emulation of Register Queues Now, we focus on a special alternative of reconfigurable register access, which may be used to compensate the high register pressure of software-pipelined loops [104, 2]. The fundamental idea of software pipelining is to interleave multiple iterations of a loop in order to speed up its execution. But such restructuring increases the life-time of variables as well as the number of simultaneously live instances of a variable as illustrated in Figure 3.4. Typical VLIW machines provide register queues [164] to reduce the life-times of values. Such behaviour can be emulated by using the registers of all processors block-wise for different instances of variables from interleaved loop bodies. However, this proposal is not studied further in this thesis.

Combination of SIMD/MIMD and Register Banks The key idea of CoBRA is to combine multiple variants in order to get the optimal machine configuration for a certain piece of code. As this thesis concentrates on SIMD/MIMD reconfiguration and reconfigurable register banks, we study also the joint application of such methods. For instance, a SIMD mode may use reconfigurable registers for fast communication or to permute variables, which is a quite common operation for SIMD.

[3]**C**ompiler **A**nticipated **P**rocessor **R**egister **I**nter-**Co**nnected **R**econfiguratio**n**

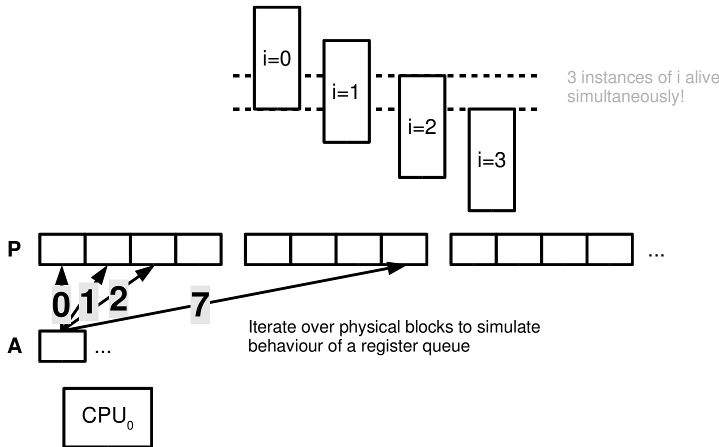


Figure 3.4.: Alternative usage of reconfigurable register banks for software pipelining

Figure 3.5 gives an example of a loop which benefits from both variants: Vector parallelism can be exploited by unrolling the loop according the number of processors and vectorizing the code. Additionally, the value f used as a factor in each computation may be stored in a shared register accessed by all processors to speed up communication.

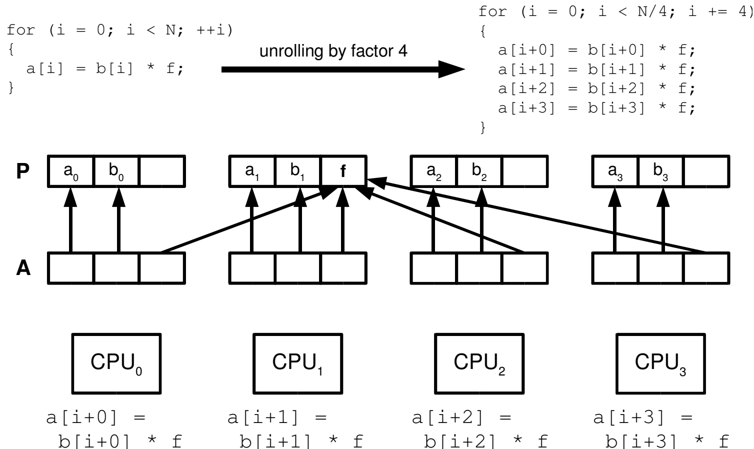


Figure 3.5.: Example of jointly using SIMD mode and reconfigurable registers

3.1.3. Reconfiguration of Machine Topology

The parallel execution of code is only useful if a program offers enough parallelism to use all the processors. For some applications which consist of dependent tasks like e.g. encryption, compression, or error correction, a pipelined execution could yield much better results. An obvious benefit is the interleaving of multiple computations to achieve a higher throughput compared to a MIMD execution. As each processor operates only on the local data of its subproblem, the data caches should be exploited better. The communication between the pipeline stages could be realized using shared registers established by a proper reconfiguration.

3.1.4. Dynamic Assignment of Special Functional Units

Instruction Set Extension (ISE) is a common technique to better support certain application domains. For instance, networking applications perform CRC checks to detect errors after transmission or use cipher like DES to encrypt strictly confidential information. The execution of such applications can be accelerated, if special instructions are introduced that combine frequently used sequences of operations. The new instructions can either be added to an existing processor or implemented in new functional units. In order to save chip area, the number of those special units would probably be smaller than the number of processors in our example.

Reconfiguration could be used to dynamically connect the special units to the processors where needed. Given a source code program, the compiler could first determine a set of hardware accelerators to speed up the execution of that application. In a second phase, it could compute a mapping of the processors to this set for each time instant.

3.1.5. Combination of Instructions between Processors

ISE means to add new operations to an existing instruction set. Such operations often result from a combination of already existing instructions. Operations could also be joined on a higher level: Multiple equivalent instructions executed on different processors could be combined to achieve 64-bit or 128-bit arithmetics. This would demand a preceding synchronization of the participating processors before the wide computation. As such synchronization comes at a certain costs, the feasibility of combining instructions has to be estimated carefully by the compiler.

Additionally, special instructions may be useful, which are executed on multiple processors only if some cores are not used temporarily. For instance, a 64-bit addition can be executed on one processor, if no further unit for a parallel computation is available. Alternatively, two processors would be combined to a 64-bit core to perform the addition as a single operation.

3.2. Application Scenarios

Compiler-Driven Up to now, we have assumed implicitly that the compiler knows the targeted reconfigurable machine and especially the provided variants precisely. With these prerequisites, a compiler can use program analysis to determine the best variants for a given application. The resulting machine code can be executed directly on the hardware which reconfigures itself according to the reconfiguration instructions in the program. This scenario is called *compiler-driven reconfiguration* in the following, because the compiler is the central tool in this case, which makes all decisions before execution.

Remarkably, machine code generated in this scenario does not need exclusive access to a target machine. With respect to the QuadroCore, a program may only use two of the four processors for execution. This is no restriction, as long as the compiler is aware of this information or a multi-tasking operating system is used.

Compiler-Supported In some situations, the compiler might lack information to decide e.g. which variants are best-suited for a certain program or when the machine should be reconfigured. For instance, the compiler might not know how many processors will be available for execution. Further motivating examples are given in Section 3.2.2.

Hence, some decisions and computations have to be deferred to the load-time or even run-time of a program. Basically, scheduling and preparing reconfiguration would need to be delayed partly or even completely. For efficiency reasons, some analysis results needed at load-time may already be computed at compile-time and stored as code annotations. In the following, this scenario is called *compiler-supported reconfiguration*, because the compiler can prepare some decisions made later.

Structure In this section, we suggest the structure of systems for the two mentioned scenarios. This includes a rough overview of the tasks which need to be performed by the compiler or other tools of the systems. Furthermore, we deal with the temporal order of such tasks according to the considered scenarios. Section 3.2.1 presents the structure of the system for the compiler-driven reconfiguration, while Section 3.2.3 concentrates on the compiler-supported reconfiguration. Section 3.2.2 discusses a couple of situations where compiler-supported reconfiguration is needed.

The prototypical implementation of CoBRA considers only compiler-driven reconfiguration. The concepts for the compiler-supported counterpart have to be refined and will be evaluated in the future.

3.2.1. Compiler-Driven Reconfiguration

First Proposal In the standard scenario, the compiler knows all details of the reconfigurable machine. Consequently, it can exploit the results of program analysis best to generate a machine program which uses the variants supported by the target machine to improve resource efficiency. Importantly, all decisions about reconfiguration and scheduling can be made at compile-time.

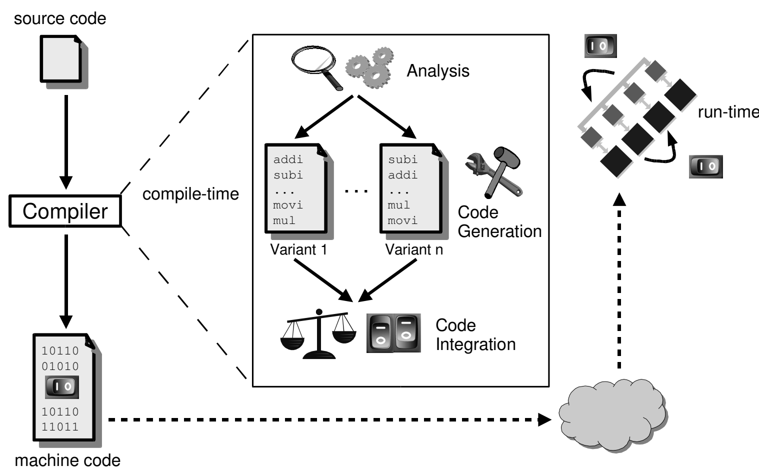


Figure 3.6.: Structure of system (compiler-driven reconfiguration)

Figure 3.6 illustrates the rough structure of the system consisting of compiler and reconfigurable machine. At first, the compiler analyzes the given program in order to get information required by succeeding phases. For example, a reconfiguration between SIMD and MIMD execution requires an analysis of inherent parallelism in a given piece of code. Additionally, the program code might be split into several parts which are handled separately later. Such a partitioning can also rely on the structuring of a function into basic blocks.

The second phase generates machine code for all parts of a program by using the n available variants. For instance, it may employ well-known scheduling and vectorization techniques for a SIMD/MIMD reconfiguration. As a result, we get n schedules for each part according to the number of supported variants.

Hence, a *code integration* is performed which comprises two basic tasks: (1) selection of the best variant for a certain program section, and (2) placement of reconfiguration instructions.

Finally, the resulting machine program can be improved by applying local optimizations like peepholing or performing techniques with a larger context like code re-ordering or even re-scheduling. This phase is not shown for simplification.

Second Proposal Alternatively, the compiler could first perform some analyses in order to decide which variants could be useful for the execution of a given program. For example, a high register pressure could indicate a need to reconfigure the connections to the register banks. After restricting the sets of feasible variants, machine code would be generated based on the analysis results gained so far or by conducting more specific analysis. Then the best results would be chosen as also done by the above strategy.

At the first glance, this idea is better, because it would reduce the effort for the compiler by restricting the set of considered variants early. On the other hand, the quality of generated code would depend heavily on quite rough decisions based on imprecise information. In terms of the example given above, the analysis could only make a rough estimate of the need for a reconfiguration by considering the register pressure. But it could not anticipate precisely which decision is the best one. Furthermore, it would not be clear how to resolve the interaction between several variants. The compiler would have to foresee if the application of a certain variant could impede the simultaneous usage of another one or even excludes it. Such a problem cannot occur with the proposed concept, because it considers all combinations and selects the best one.

3.2.2. Need for Compiler-Supported Reconfiguration

Load-Time If the compiler lacks crucial information about the targeted machine, some decisions concerning reconfiguration or scheduling cannot be made at compile-time. In terms of the QuadroCore, the number of available processors may not be known to the compiler: Firstly, multiple programs could compete for access to the machine and no multi-tasking operating system is available. Instead of granting exclusive access to only one program, the four processors could be split among multiple programs to ensure optimal resource usage.

Secondly, one or more processors might be defect due to a hardware failure. This case comprises a number of very challenging dependent problems like recognizing hardware

bugs or determining the effects to other components. For simplification, we abstract from the hardware details and conclude that the number of available processors is first known at load-time in the two mentioned situations.

Run-Time The number of available processors might also change at run-time. For instance, a program could finish execution or request more processors from the operating system. We omit any extensive discussions about handling requests from programs as well as associated aspects like process management and priorities.

Furthermore, hardware defects might crash some processors used by a currently executing program. We believe that this situation is the most complicated one, because there is a multitude of challenges for both hardware and software. The hardware aspects have already be mentioned above. On the software level, a transaction concept would be necessary to restore a previous program state for the affected processors. Obviously, a potential solution would probably be too complicated and inefficient.

As a consequence, we consider only the two situations where a program finishes execution or successfully requests more processors. In both cases, the other programs could be notified of the event by a special interrupt. Basically, a concept for the load-time scenario could be reused here in order to adapt program code to the new requirements. Additionally, the current state of a running program needs to be saved temporarily and restored after scheduling has been finished. Such a state has to be suited according to the number of processors which introduces further challenges.

Conclusion This thesis contributes a unique methodology for reconfigurable computing based on variants of an architecture. Our effort has been focused to the compiler-driven reconfiguration so far. Hence, we neglect situations where the machine is first known at load-time or might even change unexpectedly during run-time. For completeness, the next subsection presents some rough, but not yet concise ideas.

3.2.3. Compiler-Supported Reconfiguration

If the properties of the target machine are not known precisely at compile-time, some effort must be delayed to the load-time or even run-time of a program. Several example situations are discussed in Section 3.2.2. For simplification, we focus on the load-time scenario and assume that the number of processors is not known at compile-time, but first at load-time.

Concretely, two fundamental tasks can first be done at load-time: *Scheduling* needs to know the exact number of targeted processors in order to generate a proper executable machine program. Many decisions concerning *reconfiguration* are also dependent on the number of processors. For instance, it does not make any sense to generate code for MIMD or SIMD execution, if actually just one processor is available.

Scheduling At first, we concentrate on the scheduling aspect: A naive solution would be to perform the entire scheduling including dependence analysis at load-time. Instead, we propose to use code annotations which are computed by the compiler and contain all

necessary information to enable an efficient load-time scheduling. We already developed a similar approach [85, 86] for superscalar processors which could be adapted for multi-cores. Hence, an obvious idea is to treat the processors of the QuadroCore as functional units of a superscalar processor or a VLIW machine (see Section 4.2.1). More details about our concept are presented in Section 5.3.2.

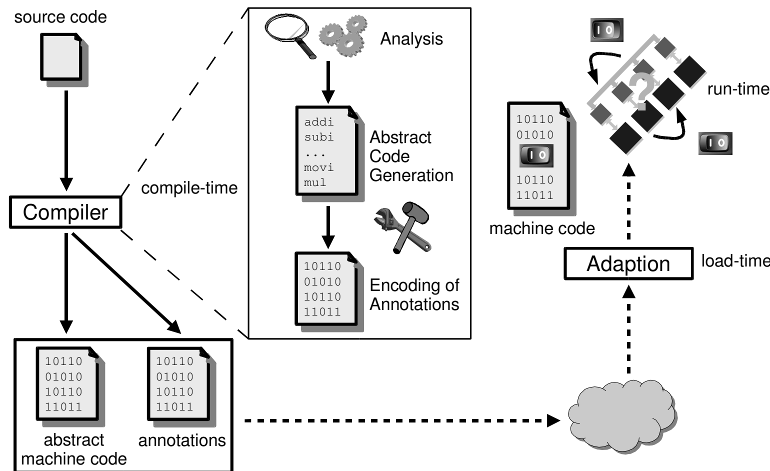


Figure 3.7.: Structure of system (compiler-supported reconfiguration)

Reconfiguration To our best knowledge, no approaches exist yet where reconfiguration is prepared at compile-time by computing code annotations. At first, we propose a rough structure of the system for this scenario. Then we discuss briefly the challenge of partitioning the effort among compile-time and load-time.

A feasible structure could be organized according to Figure 3.7. In contrast to Figure 3.6, the compiler only analyzes a program to determine some information like dependence relations, parallelism, etc. Then it generates abstract code which has not been scheduled yet for a concrete machine. Finally, the information are encoded in annotations and attached to the code.

At load-time, the abstract machine code is adapted to the concrete target machine by scheduling and selection of appropriate reconfiguration variants. For further information about load-time scheduling based on annotations, the reader may refer to Section 5.3.2. The annotations might also be used to generate code for several variants. For instance, dependence relations and parallelism are a necessary input for any scheduling or vectorization technique. A succeeding code integration phase chooses the best results and inserts reconfiguration instructions between program sections using different variants. As a last step, an executable machine program is generated which can be invoked immediately by the multi-core.

Balancing of Tasks Obviously, the annotations can reduce the effort in scheduling at load-time dramatically. But the code generation phase of CoBRA can only benefit partly from it: If we consider different parallelization paradigms like MIMD or SIMD, information about

dependences and parallelism will probably be quite beneficial. But preparing a reconfiguration of the connections to the register banks at compile-time seems to be impossible, if register allocation is performed after the scheduling. The precise schedule is first known at load-time which implies that life spans can first be computed then. Hence, the main effort would be shifted from compile-time to load-time. We believe that similar observations could be made for other reconfiguration dimensions like those covered in Section 3.1. Up to now, we have not yet developed any concepts for either scenario.

Summary As a summary, we presented the rough structure of a system for the compiler-supported scenario. For a practical implementation, it seems to be useful to realize the desired reconfiguration dimensions for the compiler-driven scenario first. This should yield important experience concerning the partitioning into subtasks as well as the appropriate times when to perform such tasks. Afterwards, the evolved concepts can be extended to the scenario handled here by deciding which tasks can or have to be done at compile-time or load-time.

3.3. Compiler for Reconfiguration of Variants

Section 3.2.1 proposed a structure of CoBRA for the compiler-driven scenario. From now on, the compiler-supported case (see Section 3.2.3) is ignored. This section concretizes the structure of the compiler step-wise with respect to the reconfiguration dimensions considered in this thesis (see Section 3.1).

In principle, the relevant excerpt from the structure looks as shown in Figure 3.8. After an analysis, code generation is performed for several variants. Finally, the code integration phase selects the best resulting schedules for each program section and places reconfiguration instructions between them. This scheme will be denoted as *CoBRA component* in the following.

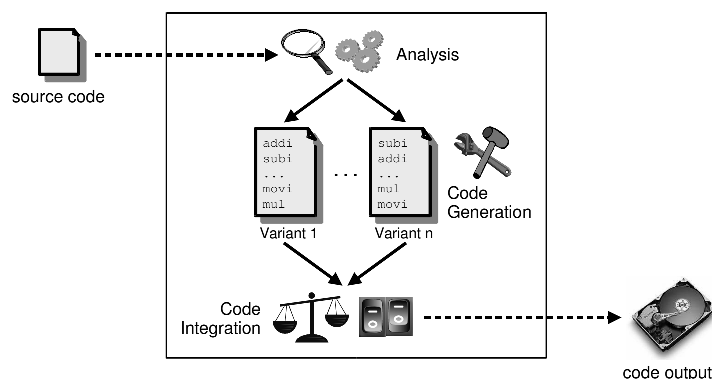


Figure 3.8.: Structure of compiler (in principle, single decision point)

In general, a compiler supporting reconfiguration of variants may have an arbitrary number of dependent CoBRA components (see Figure 3.9). Such structure can be useful when employing multiple variants that belong to different dimensions not related to each other.

Suitable examples are the two dimensions *parallelization* and *register access*. At first, a sequential program may be parallelized using different paradigms like SIMD, MIMD, or single processor. Then, a register allocation can be performed based on reconfiguring the connections to the register banks. Thereby, different variants of connections like arbitrary or restricted mappings may be considered (see Section 10.2.1).

Alternatively, the compiler may just produce several results by parallelization and register reconfiguration followed by a single code integration phase.

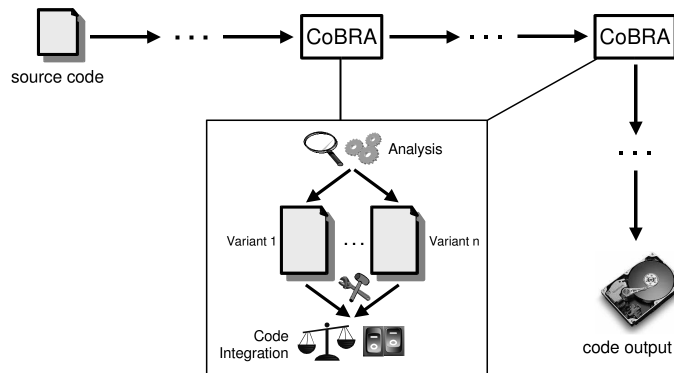


Figure 3.9.: Structure of compiler (in principle, multiple decision points)

The next subsection discusses the exemplary integration of this scheme into a parallelizing compiler backend. Section 3.3.2 deals with the code integration phase.

3.3.1. Prototypical System

Figure 3.10 illustrates the structure of a compiler backend for the dimension *parallelization* in detail. At first, the compiler performs some classical tasks to optimize the intermediate representation and to generate an abstract machine program. Then, it computes the dependences between the machine operations and determines the fine-grained parallelism. The core part of the backend parallelizes the sequential code using a couple of scheduling [82, 56, 104, 2, 116] and vectorization techniques [127, 5, 107, 91]. The code integration selects the best results and places reconfiguration instructions between them, if necessary. Registers can be allocated using conventional techniques like graph coloring [32] or by employing reconfiguration of register banks (see Part IV). Finally, a peephole optimization is performed.

An important challenge affects the granularity of the code integration: The referred scheduling and vectorization techniques operate on quite different contexts such as basic blocks, loops, or traces. For the first prototypical implementation, it seems to be easier to focus on one context like a loop or a basic block as a reference unit. The benefit would be a much easier implementation, because the results for each unit would have an equivalent functionality. Hence, the code integration could compare these results easily and select the best ones. Else, a schedule for a loop might correspond to multiple schedules for the basic blocks of the loop. Furthermore, additional glue code would be needed for software-pipelined loops in order to integrate them into a machine program.

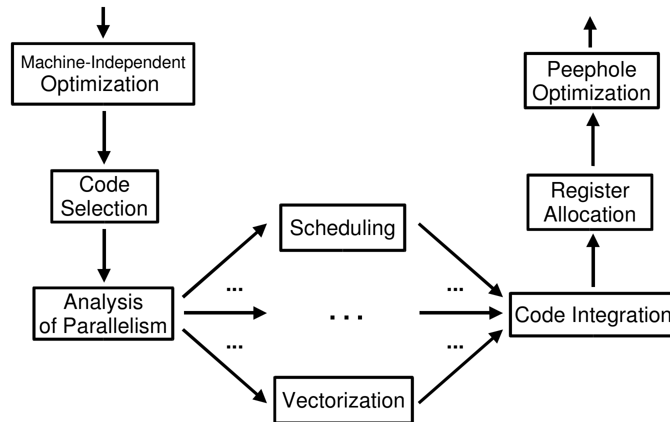


Figure 3.10.: Structure of compiler (for dimension *parallelization*)

Consequently, we decided to focus on the basic block level for simplification. The introduction of Part II refers to additional discussions about the level of the parallelization.

3.3.2. Code Integration

The code integration has to perform two tasks: (1) selection of the best variant for a certain program section, and (2) placement of reconfiguration instructions *between* the resulting schedules. In this section, we focus on the selection task which is often denoted as *code integration* for simplification. The placement task is neglected, because it is irrelevant for the current prototype of the CoBRA compiler as motivated below. The development of a concept for the code integration which handles both issues will be part of future work.

According to the previous subsection, we require that the variants have been applied on basic block level and consider a reconfiguration between SIMD and MIMD execution. The sequential code of each basic block is parallelized in two ways using both a scheduling and a vectorization. Hence, the code integration gets two schedules for each basic block which can be evaluated using different characteristics like runtime, code size or energy consumption (see Figure 3.11). A combination of such properties can be used to select the best schedule for each basic block. The current implementation of the SIMD/MIMD reconfiguration considers the estimated execution time only.

In general, a schedule can consist of multiple sections employing different execution modes with reconfiguration code between them. For instance, the vectorization technique used by the CoBRA compiler may also produce some MIMD code, if there is not enough SIMD parallelism available (see Section 8.3). If adjacent basic blocks end or start in different modes, reconfiguration code must be inserted between them. The overhead in reconfiguration can be used as an additional criterion for the selection of the best results.

As a consequence, the selection can be performed using two kinds of strategies: A *local* strategy just considers single basic blocks. Hence, it can estimate the locally best results, but neglects the effort in reconfiguration between blocks. This may lead to suboptimal results, if high penalty costs are caused by many reconfiguration points or data re-organization. For instance, a reconfiguration from MIMD to SIMD mode requires to arrange the register values

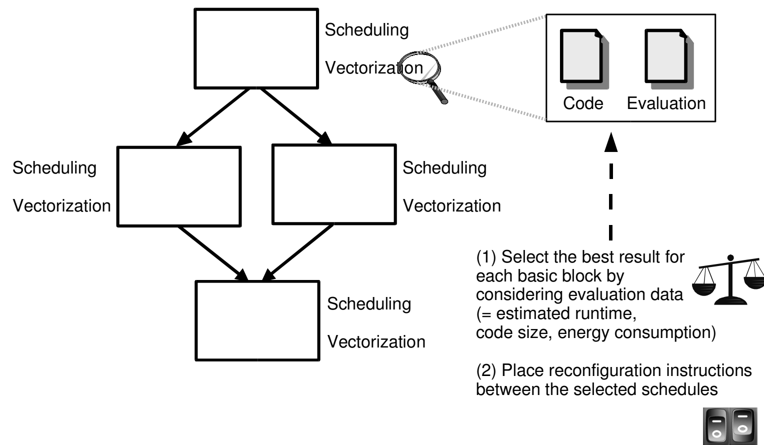


Figure 3.11.: Model of code integration

in a certain manner not guaranteed in MIMD mode (see Section 8.4).

A *global* method can yield better results by taking a larger context of the Control-Flow-Graph (CFG) into account. For instance, it may consider the predecessors and successors of a basic block, frequently executed paths of the CFG (traces) [56], or a complete function. Please note that the code generation is still assumed to be done on basic block level. Schedules for the considered basic blocks are selected by taking the mentioned properties as well as the overhead of reconfiguration between basic blocks into account.

The current implementation requires that branches are executed in MIMD mode (see Section 8.2.3). If a basic block ends with SIMD code, the execution mode will be switched to MIMD afterwards explicitly. As a consequence, there is no need for a global selection heuristic or placement of reconfiguration code between basic blocks. Hence, the prototypical implementation of the CoBRA compiler provides a local code integration only.

For simplification, the code integration is performed immediately after applying scheduling and vectorization. In order to get best results, the effect of succeeding phases like insertion of communication instructions (see Section 6.1.2), data re-organization (see Section 8.4), register allocation, and re-scheduling must be taken into account.

Part II.

Compilation for Multi-Cores

CoBRA[4] has been evaluated using the QuadroCore, a multi-core of four tightly-coupled processors called S-Core [21, 105]. The processors of this homogenous machine own separate register banks and can communicate via a shared memory. Programs executed on the QuadroCore can benefit from reconfiguring the parallelization paradigm or the connections to register banks. Hence, the QuadroCore is used as a reference architecture throughout this thesis.

Challenges A fundamental aspect of our research was the development of a parallelizing backend for the QuadroCore as a starting basis for the CoBRA compiler. The present part concentrates on three major challenges which had to be solved during the development:

The data objects and operations of a sequential program must be partitioned across the processors such that the inherent parallelism is exploited best while the number of inter-processor dependences is kept low. Communication code has to be inserted into the schedule to transport register values between the processors. Memory data objects are not affected by the communication as motivated later. The overhead of communication can be minimized by proper placement strategies. As the processors execute their instruction streams asynchronously, inter-processor dependences need to be respected by inserting synchronization operations.

Our current prototype of the CoBRA compiler only exploits Instruction-Level Parallelism (ILP) in basic blocks. Section 4.2.4 motivates our decision from the perspective of parallelizing sequential programs for the QuadroCore. Section 3.3.1 argues that the code integration is simplified a lot due to this decision. Reconfiguring the granularity of parallelization has been discussed in Section 1.3.

As a consequence of our decision, this part does not cover other levels of parallelism like Loop-Level Parallelism (LLP) or Thread-Level Parallelism (TLP), which may be supported in a future version of CoBRA. Furthermore, we do not deal with concrete scheduling techniques like software pipelining [104, 2] or trace scheduling [56, 37]. For both topics, the reader may refer to the literature.

Structure According to the three mentioned challenges, this part is structured as follows: Chapter 4 outlines the basic structure and concepts of the parallelizing compiler backend, which is generated partly from a specification of a VLIW model for the multi-core. Before, we give a short overview of the QuadroCore.

Chapter 5 explains the two-phase processor partitioning method of the CoBRA compiler, which is based partly on affinity graphs. Furthermore, a holistic concept is proposed, which assigns both data objects and instructions to the processors of a homogenous multi-core jointly. Last but not least, we introduce a concept enabling load-time scheduling for such machines, that also relies on the mentioned affinity graphs. This idea can be applied for the second application scenario discussed in Section 3.2 which requires an adaption of the machine code at load-time.

The first part of Chapter 6 explains the communication mechanism of the QuadroCore, which is used to transport register values between the processors. Then we present sev-

[4] **Co**mpiler-**D**riven **D**ynamic **R**econfiguration of **A**rchitectural Variants (merge two *D*s to a *B*)

eral strategies to place special copy instructions supported by the implementation of the mentioned mechanism. The second part handles inter-processor dependences, which are caused by the communication of register values as well as accesses to memory data objects. We introduce the mechanism for barrier synchronization provided by the QuadroCore and present the barrier placement strategy of the CoBRA compiler.

4. Compiler for QuadroCore

Contents

4.1 Reference Architecture **II-6**
4.2 Parallelizing Compiler **II-8**
4.2.1 VLIW Machines and Superscalar Processors II-8
4.2.2 Machine Model II-10
4.2.3 Structure of Compiler Backend II-11
4.2.4 Context of Scheduling Phase II-13

This chapter introduces the QuadroCore, which is used as a reference architecture in this thesis, and explains the fundamental aspects of its parallelizing compiler backend. At first, Section 4.1 provides some basic knowledge about the QuadroCore. The description comprises a short characterization of a massively parallel architecture based on the QuadroCore, as well as the employed S-Core processors.

Section 4.2 deals with the basic structure and concepts of our parallelizing compiler backend for this multi-core. As our compiler exploits ILP, the QuadroCore was modelled as a VLIW machine, which was specified using our high-level processor specification language UPSLA. The corresponding UPSLA compiler can generate machine-specific parts of a compiler backend as well as a cycle-accurate software simulator. Scheduling is performed on basic block level using list scheduling for simplification.

By default, all processors operate in a MIMD manner, i.e. they execute disjoint instruction streams on different data. For program parts with a regular structure, the QuadroCore can be switched to a SIMD mode where a single instruction stream is executed by all processors on different data simultaneously. The SIMD/MIMD reconfiguration is covered by Part III.

4.1. Reference Architecture

Most of the explanations in this thesis are based on the QuadroCore as a reference architecture, which was developed in the GigaNetIC project [21]. The project aimed at developing high-speed components for networking applications based on massively parallel architectures. A central part of the project was the design, evaluation, and realization of a parameterizable network processing unit. The developed architecture is shown in Figure 4.1 and consists of several embedded QuadroCore multi-cores, which are arranged in a hierarchical system topology with a powerful communication infrastructure. Each multi-core is connected via an on-chip bus to a so-called switch box, which allows to emulate arbitrary on-chip topologies. Additional hardware accelerators can be used to improve the flexibility and throughput of the system as well as to reduce the energy consumption.

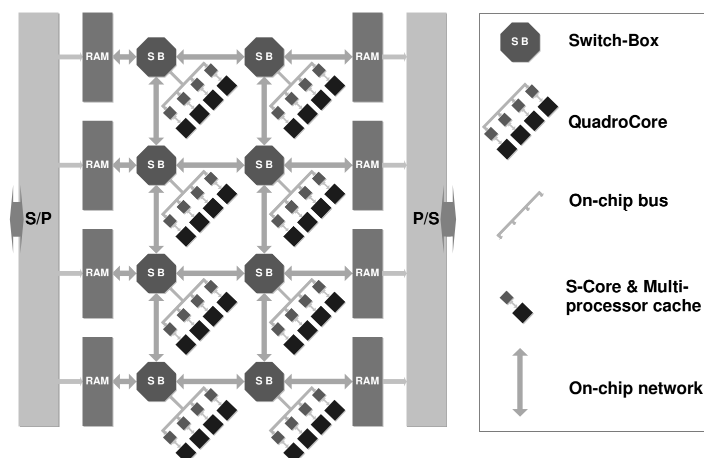


Figure 4.1.: System architecture based on embedded multi-cores

QuadroCore The structure of a QuadroCore is shown in Figure 4.2. Each multi-core consists of four 32-bit RISC processors called S-Core [105], which is explained below. The memories provide 8, 16, or 32-bit transfers. Accesses to the external memory are managed by the bus arbitration, which works in a round-robin fashion.

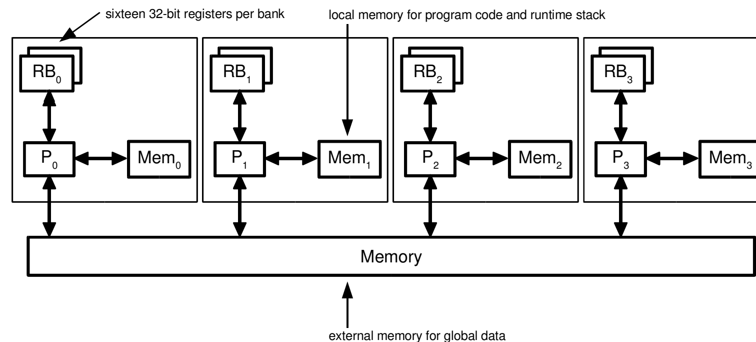


Figure 4.2.: Structure of QuadroCore

S-Core The S-Core is a straightforward 32-bit load/store architecture with big-endian byte order, which is binary compatible with Motorola's M-Core M200 architecture [122]. The processor has been developed as a soft core using the hardware description language VHDL. Additionally, we have an ASIC implementation using the ULSI technology provided by Infineon, which currently allows feature sizes of 130 nm. Currently, the chip area needed for one S-Core is less than 0.2 mm^2.

Each S-Core exhibits two register banks of 16 registers each, which can be used alternatively in user mode. This offers an efficient realization of context switches or interrupt handling. The parallelizing compiler presented in this section supports only the first register bank. Obviously, the usage of both register banks is a special case of reconfiguring the register banks covered in Part IV. Furthermore, each S-Core has a local memory, which contains its executable machine code and its runtime stack.

All machine instructions have a fixed length of 16 bits with 2-address format. Consequently, this results in high code density and reduces memory demands for the application in embedded systems like network devices.

Local and External Memory Accesses An access of the external memory has a greater latency than a local memory access due to the bus arbitration. In order to compute a parallelized schedule which resembles the actual execution as good as possible, the CoBRA compiler tries to anticipate the execution time of a memory instruction by a static analysis. It decides if such an operation will access the local or external memory during run-time.

Clearly, the analysis cannot determine the accessed memory in all cases, because of incomplete information at compile-time. Furthermore, a pointer passed to a function may point to data in local or external memory for two distinct calls. If the accessed memory cannot be determined with safety, the scheduling phase makes an optimistic assumption, i.e. treats an operation as an access to the faster local memory.

4.2. Parallelizing Compiler

The QuadroCore was modelled as a VLIW machine, because the CoBRA compiler exploits ILP. Section 4.2.1 introduces the VLIW paradigm and compares it with superscalar processors. Then, Section 4.2.2 outlines a high-level machine model of the multi-core based on a VLIW architecture. The specification of this model is used to generate machine-specific parts of the compiler backend and our cycle-accurate software simulator. The structure of the compiler backend as well as important conceptual details of certain phases are explained in Section 4.2.3. Finally, Section 4.2.4 motivates our decision to parallelize on basic block level using list scheduling.

4.2.1. VLIW Machines and Superscalar Processors

VLIW Machines A VLIW machine contains multiple functional units to exploit inherent fine-grained parallelism on instruction-level. The program code consists of very large instructions words, which specify independent scalar instructions for each functional unit. All functional units operate in lock-step according to a global clock. Figure 4.3 illustrates the fundamental properties.

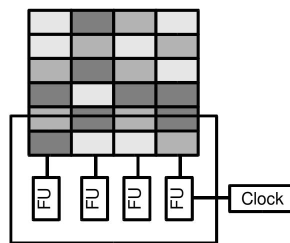


Figure 4.3.: Processing of very large instructions words in a VLIW machine

The concept of the VLIW architecture was developed by Fisher at Yale University in the early 1980's. Before he had worked on scheduling code for functional units and Instruction-Level Parallelism (ILP) at New York University [55]. Fisher invented trace scheduling [56] as a parallelization technique for VLIW to exploit fine-grained parallelism beyond basic blocks. He recognized that the compiler and the architecture for the VLIW machine must be co-designed in order fully expose the capabilities of the targeted hardware to the compiler. His Ph.D. student Ellis developed the first VLIW compiler called Bulldog [50].

In 1984, Fisher left Yale University and co-founded the company Multiflow which produced the famous TRACE series of VLIW computers [37]. The TRACE machine could execute up to 28 instructions in parallel and was shipped around 1988. Unfortunately, Multiflow ended operation in 1990, because newer techniques enabled the production of multiple issue processors on a single chip. A basic problem of the VLIW paradigm itself was the object code incompatibility across VLIW architectures with different numbers or types of functional units.

Superscalar Processors At the end of the 1980's, superscalar processors became popular with the introduction of the first commercial single-chip superscalar microprocessors by

Intel and AMD. The mainframe computer CDC 6600 developed 1965 by Seymour Cray is often mentioned as the first superscalar design. Today, most RISC processors are based on a superscalar design and even the instruction sets of CISC machines are implemented on superscalar RISC micro-architectures.

Superscalar processors have much less functional units than a VLIW machine. A dispatcher issues machine instructions from a sequential code stream to the functional units, where they are executed in parallel (see Figure 4.4). As a benefit, superscalar processors do not require a code parallelization, while VLIW machines demand special instructions with one operation per unit. On the other hand, a dispatcher can re-arrange instructions only in a very constrained context of e.g. 4 consecutive operations in case of PowerPC hardware [173]. A VLIW machine can exploit ILP best, because a corresponding compiler can consider a larger analysis context than dispatcher hardware at run-time, e.g. a basic block or a loop body. Additionally, more precise information about data dependences between operations, especially memory accesses, are available due to prior program analysis.

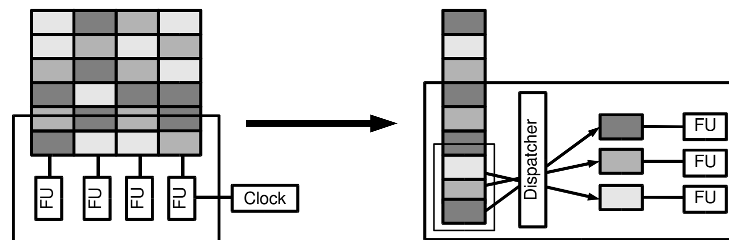


Figure 4.4.: Differences between VLIW machine and superscalar processor

Stümpel et al. [155] demonstrated that an additional scheduling phase in the compiler can increase efficiency of the dispatcher in a PowerPC 604. Scheduling in a compiler also accounts for the types and numbers of functional units in the processor and hence optimizes a program for a concrete target machine. But for mobile code in a heterogenous network with superscalar processors, the types and number of functional units is not known at compile-time. Hence, scheduling must be deferred to a later phase immediately before program execution. We have developed an approach where the compiler determines code annotations to enable efficient scheduling and register allocation with linear effort at load-time. The method has been published in [85] and is also described more detailed in Section 5.3.2.

For simplification, we use the term VLIW interchangeably for architectures with multiple functional units, although superscalar processors have much less functional units than the famous TRACE computer. Hence, the QuadroCore is also regarded as a VLIW machine with four general-purpose functional units. In contrast to a real VLIW machine, the processors of the QuadroCore operate on separate instruction streams in an asynchronous manner. If one processor needs more time for an operation than assumed by the compiler, the remaining processors will *not* be stalled. Explicit synchronization is achieved via barriers (see Section 6.2).

4.2.2. Machine Model

In the GigaNetIC scenario [21], the compiler was primarily used to evaluate proposed variants of the base architecture. One important goal was cycle-accurate performance prediction based on realistic application software, before corresponding hardware designs are available. Clearly, the compiler had to be easily retargetable when adding specialized super-instructions or changing instruction timings. Hence, we described the QuadroCore by an abstract machine model which was specified using our high-level machine specification language UPSLA. The UPSLA compiler is employed to generate machine-specific parts of the compiler backend as well as a cycle-accurate software simulator.

General Techniques The machine model provides a high-level view of the target processor, i.e. it is reduced to just those aspects which are relevant for scheduling decisions at compile-time. Consequently, the model describes the available functional units and register banks of the processor as well as their capabilities and their connectivity [21]. A further building block of the machine model is a description of the instruction set. The model defines execution time, latency, semantics, and resource requirements in each clock cycle for all machine instructions. The set of processor resources used by instructions abstracts far from the actual hardware design. Hardware resources that contribute no discriminating scheduling constraints can be safely omitted. Complex interactions between multiple interrelated hardware resources are represented by a single *virtual resource* that only captures the resulting behaviour on the instruction-level [155].

VLIW Model of QuadroCore As mentioned in the previous subsection, the CoBRA compiler regards the QuadroCore as a VLIW machine with four general-purpose functional units. Behaviour that affects more than one S-Core processor is modelled by two additional pseudo units of the VLIW. The pseudo branch unit manages the common control-flow for all four processors in concert, while the virtual synchronization unit is used to store barrier instructions.

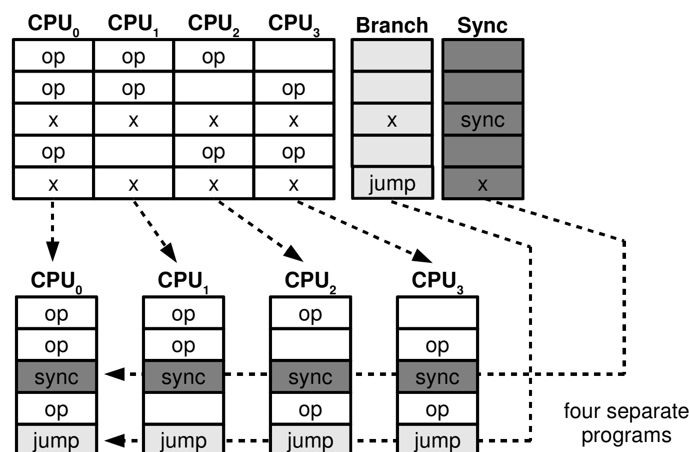


Figure 4.5.: Scheduling model and integration of pseudo unit instructions

The instructions of the pseudo units are integrated into all four code streams (see Fig-

ure 4.5) immediately before emitting the code streams. However, the concept of centralized pseudo units guarantees structurally correct code after scheduling, without any artificial scheduling constraints. Additionally, the usage simplifies code transformations in later phases like e.g. peephole optimization.

4.2.3. Structure of Compiler Backend

Figure 4.6 illustrates the structure of the compiler backend, which has been derived from an existing backend for VLIW machines. In contrast to the original backend, it features three additional phases which are highlighted in the picture and explained in the following.

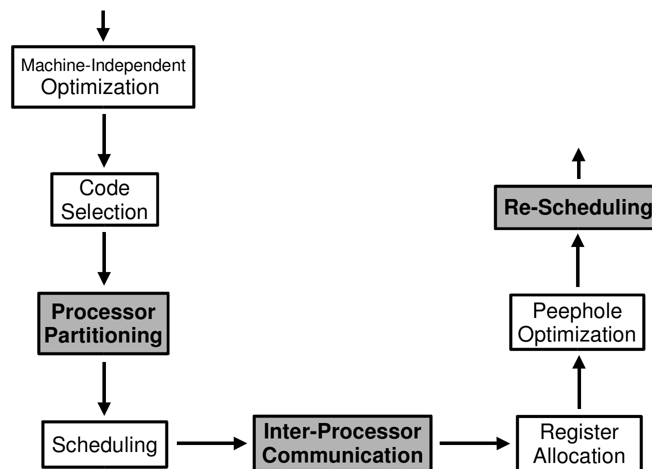


Figure 4.6.: Structure of parallelizing compiler backend

At first, the intermediate representation is optimized using a set of well-known *machine-independent optimizations*. Worth mentioning are copy propagation, common subexpression elimination, strength reduction, and loop unrolling [1, 95]. The *code selection* phase transforms an intermediate language tree representing one statement of the original source code program into a sequence of machine instructions. Our code selection is based on a variant of Bottom-up Rewrite Systems (BURS) technology [58] and generated from a tree grammar to specify the capabilities of the target processor.

Processor Partitioning, Scheduling, Communication *Processor partitioning* decomposes into partitioning of data objects and allocation of functional units, which is needed as input for the scheduling. The data partitioning method is based on affinity graphs, while the functional units are allocated using the BUG algorithm by Ellis [50]. Our concept is motivated and explained in Chapter 5.

Scheduling performs automatic fine-grained parallelization for the S-Core based multi-core. Fischer et al. [54] demonstrated that parallelizing compilers are very efficient for a small number of processors. The parallelization is based on both machine-specific code generated from a description of the target machine as well as a library of hand-written scheduling algorithms. The library includes several algorithms like list scheduling [82] and software pipelining [104]. Currently, we use only list scheduling (see Section 4.2.4).

According to Section 4.2.2, the scheduling phase treats the multi-core similar to a VLIW machine with four general-purpose functional units and two pseudo units for branch and synchronization instructions, respectively. *Barrier synchronization* is handled in detail by Section 6.2.

Immediately after the scheduling, the *remote data dependences* between different processors are determined. This information is used for the placement of *communication code* to exchange register values, which is described in Section 6.1.

Register Allocation The *register allocation* is performed simultaneously for all processors. In the non-reconfigurable case, it considers only the first register bank of each processor. The allocation heuristic is based on global register allocation by graph coloring [32]. Additionally, it incorporates several improvements to reduce spill code and to avoid conflict edges in the interference graph presented by Briggs et al. [24].

As each processor of the QuadroCore can only access its own registers by default (see Section 4.1), registers of processor p must be allocated for an instruction executed by p. Simultaneously, a virtual register v may be accessed by different processors, although the data partitioning (see Section 5.2) has assigned it to a single processor. Hence, v must be duplicated before the register allocation in order to allocate physical registers of the corresponding processors. Concretely, v is replaced by a newly created virtual register v_p for each processor p accessing v. Furthermore, communication code is added to copy its value from the defining core to the using processors (see Section 6.1). As a consequence, a machine program resulting from a parallelization and the described transformation looks like written explicitly in parallel (see Figure 4.7).

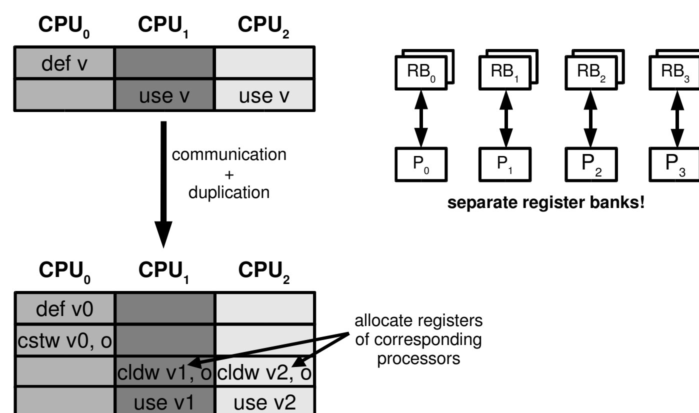


Figure 4.7.: Duplication of virtual registers

Re-Scheduling and Insertion of Barriers After register allocation and *peephole optimization*, a *re-scheduling* is performed to produce a more compact schedule. Instead of just applying local optimizations to the existing schedule, the Data Dependence Graph (DDG) is re-constructed and scheduled again. This phase also inserts barrier instructions on-the-fly to ensure that remote dependences between processors are respected. Our compiler represents parallelized code as a VLIW schedule until synchronization code is inserted in order to

yield separate code streams for an asynchronous multi-core. Section 6.2 introduces barrier synchronization and compares asynchronous with synchronous execution. Furthermore, we present the concept of the re-scheduling phase and especially the integration of barrier instructions.

4.2.4. Context of Scheduling Phase

Our first prototypical implementation of the CoBRA compiler parallelizes sequential code by exploiting fine-grained parallelism on basic block level using list scheduling [82]. In principle, our library of scheduling algorithms also includes different variants of software pipelining [104] described in [133]. But the prototypical implementation was not technically mature in order to apply it for real existing machines like the QuadroCore. We decided to abstrain from an improvement of our software pipelining implementation due to several reasons: Firstly, this thesis concentrates on reconfiguration of processors and *not* on implementing a parallelizing compiler for the QuadroCore. Secondly, as such a compiler was not available, unfortunately, we set focus on the most important challenges like processor partitioning (see Chapter 5) and barrier synchronization (see Section 6.2). Thirdly, our simulator does not provide a debugger or at least an interface to common debugging tools. Hence, the debugging of parallelized code was already a very time-consuming task when using list scheduling. The introduction of this part refers to further arguments with respect to the CoBRA approach.

Management of Control-Flow The decision to schedule on basic block level implies that the processors of the QuadroCore have a common control-flow during the execution of machine programs generated by the CoBRA compiler. Surely, more sophisticated scheduling techniques like software pipelining or even the exploitation of coarse-grained parallelism could enable a more independent instruction processing of the processors. This would also reduce the number of barriers needed for explicit synchronization significantly.

Despite of this common control-flow, each processor of the QuadroCore has its own compare register, which is also used by some arithmetic instructions. By default, a single compare instruction is executed by one of the processors per conditional branch. A special `crsync` instruction copies the result of a comparison to the other processors (see Figure 4.8) to ensure that the branch condition is equal among all processors. This technique is also known as *collective branching* [69]. In our case, the instruction writes the compare bit of the denoted processor in the first half of the cycle and reads it in the second half. The processors are synchronized by a barrier operation beforehand.

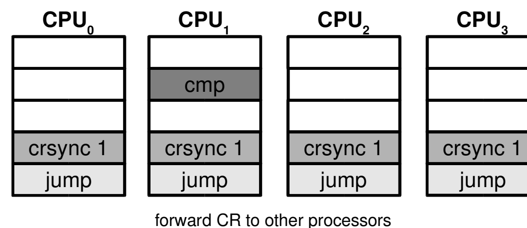


Figure 4.8.: Exchanging of compare register

Consequently, conditional branches have a significant overhead due to the additional barrier and copy operations. In order to minimize such penalty costs, a compare instruction can be executed in parallel by all processors. But this requires to copy the compared register values to each processor beforehand. Hence, the CoBRA compiler offers an alternative strategy which creates multiple independent copies of induction variables and the accessing code to reduce the costs of communication between the processors.

5. Processor Partitioning

Contents

5.1 Related Work **II-17**
5.2 Partitioning of Data Objects **II-18**
5.2.1 Introductory Example II-19
5.2.2 Affinities Between Variables II-20
5.2.3 Optimal Number of Partitions II-22
5.2.4 Variable and Parameter Partitioning II-23
5.2.5 Discussion II-26
5.3 Improvements and Extensions **II-26**
5.3.1 Holistic Partitioning of Variables and Instructions II-27
5.3.2 Load-Time Scheduling Using Compiler Annotations II-27

Motivation and Introduction Recent processor architectures often consist of multiple clusters or cores to avoid the scaling problem caused by bottlenecks of centralized register files. Instead, functional units, register files, and memory subsystem are partitioned to decentralize the hardware design and meet technology constraints in terms of cycle time, area, and power consumption. Some well-known examples are clustered VLIW machines, the IBM Cell processor [132], and the RAW machine [158] mentioned in Section 2.3.5.2.

A major challenge for a compiler targeting such architectures is to partition data objects *and* operations effectively to achieve a high throughput. Conventional approaches like the famous BUG algorithm by Ellis [50] (see Section 5.1) focus only on distributing the operations of a program but ignore the influences of partitioning data objects. This restriction might be neglected for some VLIW machines, where the data partitioning is trivial (see Section 4.2.1). One example is the architecture on the left side of Figure 5.1 which has only a single memory and two register files for integer and floating-point values, respectively. Consequently, the partitioning of register values and structured data objects is quite straightforward.

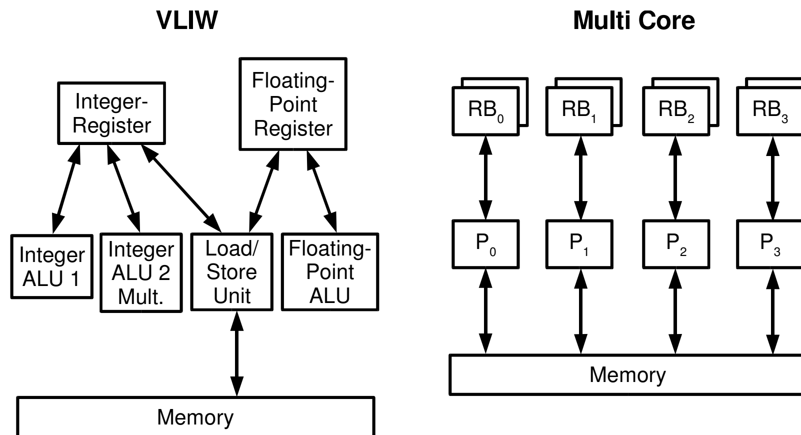


Figure 5.1.: Comparison of VLIW machine and S-Core based multi-core

In contrast to the asymmetric structure of a VLIW machine, the clear symmetry of the QuadroCore (see Section 4.1) allows completely arbitrary assignments of register values and machine instructions to the four processors (see right side of Figure 5.1). Obviously, the quality of scheduled code depends mainly on overhead of communication and synchronization due to such assignments.

As a consequence, the current prototype of the CoBRA compiler partitions data and operations in two phases. Firstly, a data partitioning is performed with an original method using an affinity graph. Then, the functional units are allocated by the mentioned BUG algorithm (see Section 5.1). We envision a holistic processor partitioning method based on affinity graphs which considers both data objects and instructions jointly.

Structure At first, the work related to our approach is discussed in Section 5.1. Section 5.2 presents the Variable Affinity Graph (VAG) which is used to partition the variables of a function. Section 5.3 outlines two interesting directions of future research: In Section 5.3.1, we discuss a rough concept to merge both partitioning phases into a single approach. Early ideas to an extension of our method for load-time scheduling are presented by Section 5.3.2.

5.1. Related Work

The first and most famous approach for partitioning operations across functional units was developed by Ellis [50] for the Bulldog compiler targeting early VLIW machines. The algorithm called BUG performs a two-phase traversal on the DDG of a function. The bottom-up phase determines those functional units which can be used for the execution of an operation. On the way back, that functional unit is chosen for each operation which can make the result available as early as possible for succeeding operations. The decision is based on both the execution time of an instruction as well as the communication costs between a functional unit and the register banks containing the operands. Consequently, the heuristic relies on a partitioning of register variables to achieve satisfactory results. Otherwise, it would lack a necessary basis for decision and could distribute the instructions based on their execution times only. The outstanding advantage of BUG is its intuitiveness and simplicity compared to more complex approaches outlined below. But as it does not produce the final schedule, it cannot estimate the resource usage precisely. Consequently, the later scheduling could cause disarrangements compared to the preliminary resource allocation determined by BUG.

Approaches like [92, 126] try to overcome this problem by integrating cluster assignment, instruction scheduling, and register allocation in one procedure. The benefit is a holistic method which is aware of all resource constraints in contrast to other compiler backends with multiple phases for these tasks. But like BUG, these approaches focus only on partitioning the operations and ignore the data objects.

Capitanio et al. [29] developed a code partitioning method for Limited Connectivity VLIW architectures which consists of three phases: Firstly, a DDG is built from code for an ideal VLIW machine. Secondly, a graph partitioning algorithm is applied in order to assign the operations to the clusters. The partitioning uses a cost function which is based on the cutset. Finally, data movement instructions are inserted to transport values between the clusters. As in our approach, the problem of processor partitioning is reduced to a graph partitioning problem. But to our best knowledge, this idea is not used for partitioning the data objects and does not rely on edge weights to model affinities between operations.

Terechko et al. [159] studied the effects of partitioning values that are alive in a whole function or across multiple scheduling units for a clustered VLIW processor. Clearly, such global values have a big impact on the schedule because of their long live ranges. Their results have shown that trivial assignments, like mapping all these values to one cluster, can result in a significant loss in performance. Furthermore, they present three advanced algorithms for partitioning global values based on multi-pass scheduling and affinity of variables. We consider only the first and most relevant solution at this point.

The affinities are computed for all pairs of variables and model the benefit of assigning them to the same cluster. The computation is based on the number of operations on a path in the DFG, the priority of an instruction [81, 80] as well as the execution frequencies of the scheduling units. According to the paper, the latter data is gained by a prior profiling. But to our best knowledge, there is no information how the parallelization for the profiling is performed to exploit all available scheduling units. Due to the usage of affinities between variables, the mentioned approach is partly similar to our concept. But the affinities are not regarded as edge weights of a graph that is partitioned afterwards. Additionally, no affinities between instructions are determined to achieve an allocation of functional units

with a similar methodology.

Chu et al. [36] proposed an integrated technique for partitioning both data objects and operations across multiple clusters. As both problems are quite complex and also interact with each other, the approach is divided into two simpler subproblems that are solved in a phase-ordered manner. The first phase performs a global partitioning of the data objects which uses a rather simple view of the operations and data communication. The second phase partitions the operations based on the results of the first phase.

At first, a program-level DFG is constructed where the nodes correspond to the operations in the program code. Memory operations and calls to memory management functions (like `malloc()` in C) are annotated with information about the associated objects. By these means, the effects of the operations can be estimated best. Then, all operations accessing the same memory objects are combined followed by merging these objects. Finally, the DFG is partitioned using the graph partitioning tool set METIS [94] which is also applied by our approach.

The second phase uses an enhanced Region-based Hierarchical Operation Partitioning (RHOP) which is explained in [35]. An outstanding property of the algorithm is the modeling of the resources and estimates of the schedule length. Edge weights are used to model the affinities between operations.

As outlined later, the method by Chu et al. is more powerful than our approach and also applicable for heterogenous clustered architectures in general. Our technique was developed primarily for our parallelizing compiler which has been used as a starting point for the CoBRA compiler. We decided to keep the partitioning method as simple as possible while trying to achieve a good performance for the QuadroCore.

5.2. Partitioning of Data Objects

In terms of the QuadroCore, three types of data objects need to be distinguished for the partitioning: The register variables of a program have to be mapped to register banks of the multi-core such that the costs in inter-processor communication are minimized. Local stack variables need to be distributed among the processors whereas the access is restricted to just one processor. Global variables do *not* need to be considered, because they are stored in external memory and hencefore are accessible by all processors.

This section addresses partitioning of local variables using the Variable Affinity Graph (VAG) with respect to the QuadroCore. In Section 5.2.5, we discuss some restrictions of our approach motivated mainly by the symmetric structure of this machine. If the approach should be used for asymmetric clustered architectures, the partitioning method must be enhanced by taking number and types of resources into account.

The nodes of a VAG correspond to the variables of a function. The affinities between variables are modelled as edge weights and express communication costs that will occur, if such variables are stored on different processors. The size of variables can be represented by node weights in order to balance the register and memory requirements. The resulting graph is partitioned using common graph partitioning techniques and additional heuristics mentioned later to optimize the results.

From the compiler's view there are three important requirements: Firstly, the number of VAG partitions should be at most as large as the number of target processors. Secondly, the number should be optimal such that the total affinities of cut edges are minimized. As a benefit, the communication costs between processors would be minimized, too. Thirdly, the algorithm should partition a VAG into approximately equally sized partitions. The CoBRA compiler uses the graph partitioning tool set METIS [94], which meets the first and last requirement. The second goal is achieved by a special heuristic.

Structure This section is structured as follows: At first, Section 5.2.1 introduces an example to understand the basic idea of computing the affinities between variables and partitioning them afterwards. Furthermore, variables and life spans are discussed as two feasible levels of granularity. Our approach is based on the variables in the intermediate representation of a program.

In Section 5.2.2, we present the computation of affinities in detail, which mainly depends on the number and weight of statements where two variables are used together. The latter input may either be determined statically or by considering profiling results.

The remaining parts deal with the actual partitioning of the VAG. Section 5.2.3 outlines a simple heuristic to determine the optimal number of partitions. The decision is based on the ratio between the sum of affinities of cut edges and the total sum of affinities.

Finally, Section 5.2.4 explains how to consolidate the partitioning of function parameters predetermined by the calling conventions of the CoBRA compiler with the present method of partitioning variables.

5.2.1. Introductory Example

We start with an introductory example (see Figure 5.2) to illustrate the basic idea of partitioning variables using a VAG. The nodes of the VAG correspond to the four register variables a, b, x, y, the parameters c and d, as well as the loop variable i. For simplification, node weights are neglected here. Edge weights accord to the affinities between two variables, which arise from the number of statements where both variables occur together. To consider (nested) loops, affinities could be multiplied by the loop counts of outer loops. In our example, the affinities between register variables used together in the for-loop are always 100. This leads directly to a partitioning into two parts.

Loop variables should be duplicated, because a static assignment to one register bank could yield bad performance results when a loop variable is accessed by multiple processors. The same applies for induction variables in general. Additionally, all instructions accessing those variables need to be duplicated and distributed to the relevant processors.

In general, the concept outlined above can be imprecise, because it is based on variables instead of life spans. Assume that a variable is re-defined multiple times and used at least once after each definition. Obviously, each instance of that variable occurs in another context of a program and probably might have different affinities to other variables. Instead, each instance of a variable could be replaced by a auxiliary variable before applying our method.

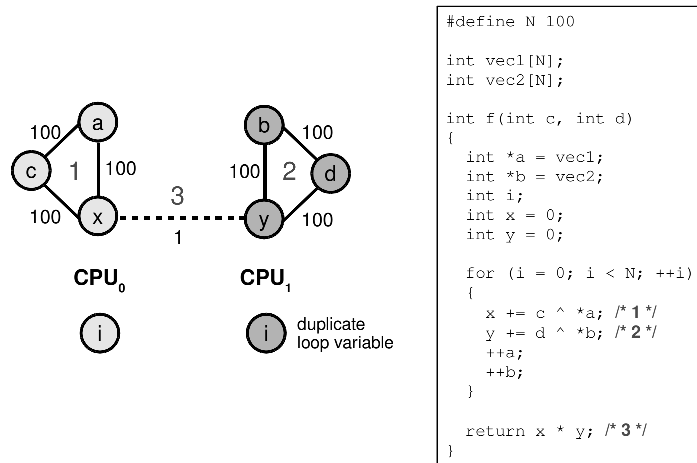


Figure 5.2.: Example of variable affinity graph

5.2.2. Affinities Between Variables

In the CoBRA compiler, the affinities are computed by considering the intermediate representation of a function: The affinity of two variables equals the sum of affinities between these variables in all statements. The affinity with respect to a certain statement corresponds to the weight of the basic block which contains the statement, if both variables occur in the statement, otherwise 0. The weight of a basic block depends on its loop depth. Alternatively, profiling results could be exploited in order to weight basic blocks according the loop iteration counts and branch probabilities.

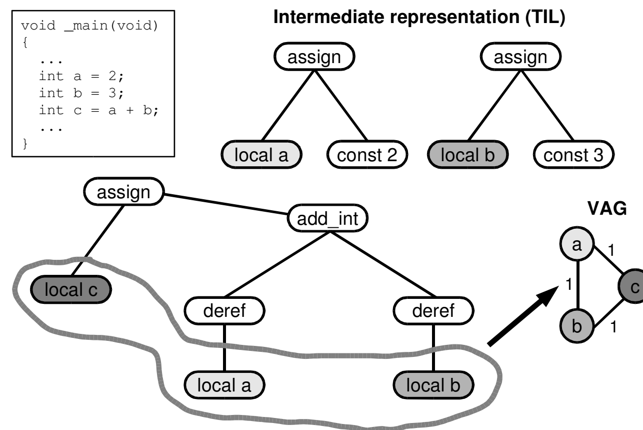


Figure 5.3.: Computation of affinities between variables

Intermediate results which are only used in the machine code generated for one intermediate tree are not considered. Obviously, the affinities would be too small and a graph partitioning would probably cut the corresponding edges. Instead, those intermediate results are implicitly partitioned during the allocation of functional units.

Concretely, the affinity between two variables is defined as follows:

Definition 5.1 (Affinity of variables)
Let x and y be two variables. Then their affinity $\alpha(x, y)$ is defined as:

$$\alpha(x,y) = \sum_{\forall \text{stmts s}} \alpha_s(x,y)$$

$\alpha_s(x, y)$ is the affinity between x and y with respect to the statement s:

$$\alpha_s(x,y) = \begin{cases} \omega(\beta(s)) & : \quad x, y \in s \\ 0 & : \quad x \notin s \vee y \notin s \end{cases}$$

The basic block containing s is denoted by $\beta(s)$ and its weight by $\omega(b)$:

$$\beta(s) = \text{basic block which contains } s$$

$$\omega(b) = \begin{cases} 1 & : \quad \lambda(b) = 0 \\ 10 & : \quad \lambda(b) = 1 \\ 100 & : \quad \lambda(b) = 2 \\ 1000 & : \quad \lambda(b) \geq 3 \end{cases}$$

$$\lambda(b) = \text{loop depth of } b$$

Loops are modelled by static values instead of considering the actual number of loop iterations. Theoretically, the loop counts could be determined by the compiler where possible. But for some loops the number of iterations is influenced by dynamic effects and is therefore only known at run-time. Furthermore, an intermixing of static ($\omega(b)$) and dynamic weights (loop counts) for basic blocks is not useful. We aim to improve the incorporation of loops for computing affinities in the future.

Finally, a VAG and its partitioning is defined as follows:

Definition 5.2 (VAG and partitioning)
Let

- $G = (V, E)$ be a VAG where V corresponds to the set of variables and E to the pairs of variables with non-zero affinities
- $w : E \rightarrow \mathbb{N}$ a weight function representing the affinities
- p the number of partitions
- $\pi : V \rightarrow P := \{0, \ldots, p-1\}$ an assignment of nodes to partitions
- $C \subseteq E$ the set of cut edges, i.e. $\forall \{c_1, c_2\} \in C : \pi(c_1) \neq \pi(c_2)$

5.2.3. Optimal Number of Partitions

In this section, we explain a simple heuristic to determine the optimal number of partitions, which is based on the fraction of the sum of affinities of cut edges and the total sum of affinities. To better understand the basic idea, we first consider an example (see Figure 5.4) where the optimal number is clearly 2, because the graph consists of two cliques of the same size and all affinities are 1.

Initially, the heuristic partitions the VAG into four parts corresponding to the maximum number of target processors. Then the result is evaluated by comparing the affinities of the cut edges A_{cut} with the sum of all affinities A_{sum}. Concretely, the heuristic checks if A_{cut}/A_{total} is less than a pre-defined constant $\sigma \in [0, 1]$. In our example, we assume $\sigma = 1/6$. Consequently, the result of the partitioning is rejected by the evaluation, because $2/3$ of the affinities belongs to cut edges. Then the algorithm proceeds with 3 target processors which leads to the same result. Finally, a satisfactory partitioning of the VAG into 2 parts is found which is also an optimal one, because it yields no cut edges.

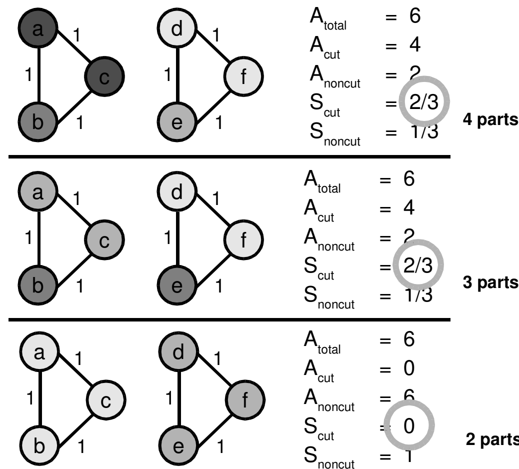


Figure 5.4.: Example of computing the optimal number of partitions

Before we take a look at the concrete algorithm, we need to define some metrics for the evaluation of a graph partitioning:

Definition 5.3 (Evaluation of VAG partitioning)
A_* denotes the sum of affinities for a certain set of edges:

$$\begin{aligned} A_{total} &= \sum_{\forall e \in E} \alpha(e) \\ A_{cut} &= \sum_{\forall e \in C} \alpha(e) \\ A_{noncut} &= \sum_{\forall e \in E \setminus C} \alpha(e) \end{aligned}$$

S_* corresponds to the relative affinity with respect to a set of edges:

$$\begin{aligned} S_{cut} &= A_{cut}/A_{total} \\ S_{noncut} &= A_{noncut}/A_{total} \end{aligned}$$

The result π of a partitioning is called *good* if $S_{cut} < \sigma$ where $\sigma \in [0, 1]$ is a quality constant defined in the compiler. Otherwise we call it a *bad* result. In our case, $\sigma = 0.2$ has led to satisfactory results during the evaluation (see Section 13.2).

Initially, p is set to the maximum of the number of processors and the number of nodes in the VAG. If the computed result is good, it will be marked as a candidate. Then the algorithm iteratively decrements p by 1 and partitions the VAG until $p < 2$.

The desired optimum corresponds to the candidate with minimal p among all candidates with minimal S_{cut}. If no candidate can be determined or the VAG contains only one node, all variables will be assigned to one processor.

5.2.4. Variable and Parameter Partitioning

In this section, we outline how to consolidate the partitioning of variables presented above with the partitioning of function parameters according to the calling conventions of the CoBRA compiler.

5.2.4.1. Calling Conventions

To start with, we first explain the calling convention: The first 12 words of parameters are passed in registers and distributed to the first three parameter registers on each processor in a cyclic fashion. All remaining words are passed on the stack of the first processor. Structures are passed as pointers to a duplicate (call-by-value) or just as a pointer without copying (call-by-reference). A result is returned in the first parameter register of the first processor. Figure 5.5 illustrates the explanations.

If a function parameter is passed on processor x, but used on processor y, it must be copied from x to y before its use on y. Obviously, this is a special case of the placement of communication code (see Section 6.1). Figure 5.6 shows a small example where the second parameter b is passed by processor 1 but used by processor 0.

5.2.4.2. Adaption of Variable Partitioning to Parameter Partitioning

Up to now we have presented a new method for the partitioning of register and local variables and a fixed partitioning scheme for function parameters defined by the calling conventions of the CoBRA compiler. Now both partitionings have to be matched as Figure 5.7 illustrates for our initial example (without loop variable i): Parameter c is passed by the first processor while the variables a and x are assigned to the second processor although connected by edges with weight 100. The same applies to b, d, and y. If these partitions are used without further optimization, high communication costs will arise.

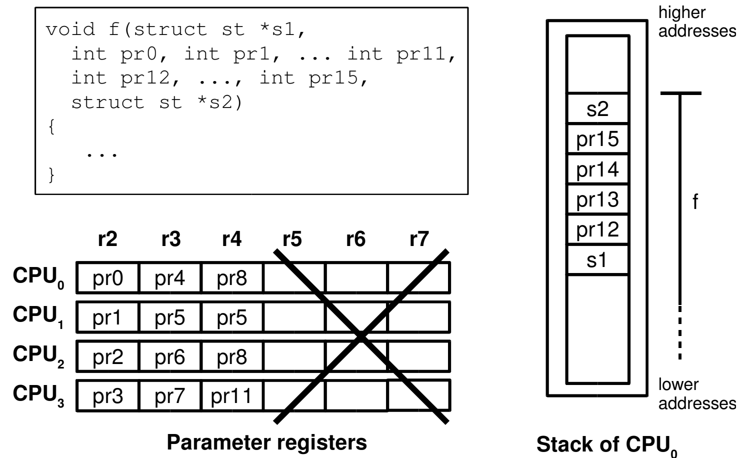


Figure 5.5.: Example of partitioning of function parameters

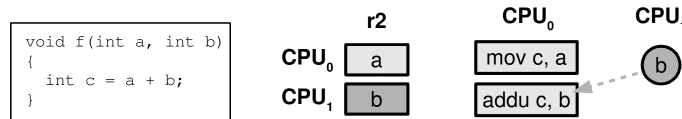


Figure 5.6.: Example of distribution of function parameters

The basic idea of our solution is to permute the partitions of variables such that the mentioned penalty is minimized or even removed completely. As a consequence, a mapping from partitions of variables to processors must be determined by considering their affinities to the already assigned parameters (see bold numbers in Figure 5.7):

Definition 5.4 (Mapping between VAG partitions and processors)
Let

- n be the number of processors
- $m \leq n$ the optimal number of partitions (for VAG partitioning)
- $P_0, \ldots, P_{m-1}$ the VAG partitions
- $C_0, \ldots, C_{n-1}$ the processors

Then the affinity $\alpha(P_i, C_j)$ between a partition P_i and a processor C_j is defined as the sum of affinities between all pairs of variables in P_i and parameters of C_j:

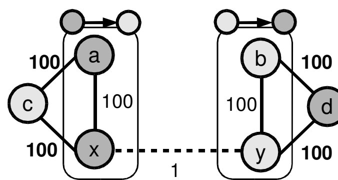


Figure 5.7.: Matching of variable and parameter partitioning

$$\alpha(P_i, C_j) = \sum_{\text{var } v \in P_i} \sum_{\text{param } p \text{ on } C_j} \alpha(v, p)$$

In principle, a partition P_i is mapped to that processor C_k where $\alpha(P_i, C_k)$ is maximal with respect to all processors $C_k, k \in \{0, \ldots, n-1\}\}$:

$$\Pi(P_i) = \{j \mid \alpha(P_i, C_j) = max\{\alpha(P_i, C_k) \mid \forall k \in \{0, \ldots, n-1\}\}\}$$

To avoid surjective mappings, processors are allocated on a FIFO basis.

5.2.4.3. Extension of Original Concept

The mentioned solution demands some modifications to the partitioning of variables (see Figure 5.8): Firstly, the VAG is constructed by considering register and local variables *as well as* function parameters. Secondly, a subgraph of the VAG containing only the variables is partitioned as described before. Finally, the results from variable and parameter partitioning are matched to get a partitioning of the original VAG.

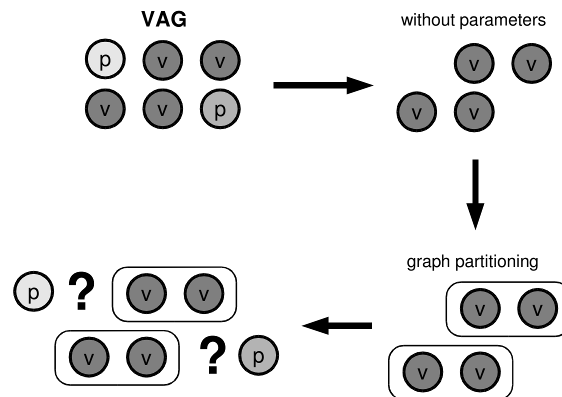


Figure 5.8.: Overview of variable/parameter partitioning

One remarkable property of the matching is that all processors are considered, independent of the optimal number computed for the partitioning of variables. This follows directly from the calling convention of the CoBRA compiler, because parameters are passed by all processors while variables might be divided into fewer parts.

Figure 5.9 clarifies the difference for a small example, where the optimal number of VAG partitions is obviously 2. If only two target processors are considered by the matching, the final partitioning will be poor, because the variable b has been assigned to the second processor while the parameters $q0$ and $q1$ are passed by the third and fourth processor, respectively. Instead, using four target processors leads to an optimal matching, because the variables a and b are assigned to one of the two processors where the associated parameters are passed. Hence, this example has four optimal partitionings of variables and parameters.

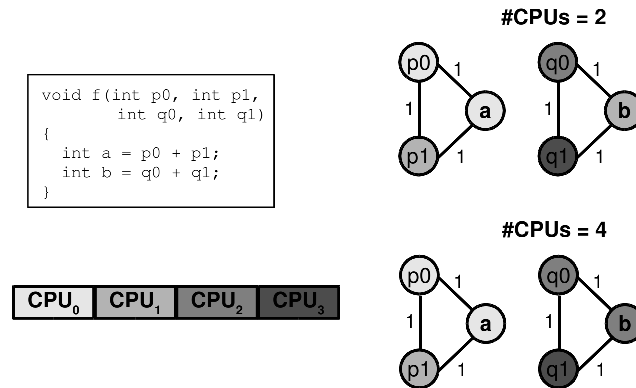


Figure 5.9.: Number of target processors for matching

5.2.5. Discussion

We have presented a method for partitioning the local variables of a program with regards to the QuadroCore. The fundamental ideas are summarized at the beginning of this section. Obviously, the approach is restricted to symmetric architectures with homogenous processors like the QuadroCore, because the method does not distinguish certain features of the cores. This property is most apparent for the actual partitioning of the VAG as well as the heuristic to determine the optimal number of partitions (see Section 5.2.3).

In order to apply our method for asymmetric architectures with heterogenous processors, the partitioning method must be aware of the number and types of their resources. This would probably require to develop a new graph partitioning algorithm which tries to achieve optimal ratio of partition sizes and distribution of nodes with respect to the targeted machine. Even the very powerful tool set METIS [94] used by the CoBRA compiler is not capable of partitioning a graph into partitions with different sizes. The properties of nodes can only be modelled by node weights, which is probably not sufficient in this case. A completely new approach might be the best solution when partitioning variables for heterogenous multi-cluster architectures. Chu et al. [36] (see Section 5.1) developed an appropriate technique which can partition both data objects and operations.

Nevertheless, our method has been applied successfully by the CoBRA compiler for the QuadroCore where such restrictions can be neglected. Implementing a more complex partitioning approach would have caused a much higher effort without any visible benefits for this thesis.

5.3. Improvements and Extensions

In Section 5.2, we have presented a novel approach to partition data objects based on an affinity graph. Currently, the CoBRA compiler allocates functional units by employing the BUG algorithm [50] (see Section 5.1). In the future, both data and operations should be partitioned among the processors by a holistic approach using affinity graphs. Section 5.3.1 discusses an early concept.

The second direction of our research concentrates on extending the approach to enable an efficient load-time scheduling using code annotations computed by the compiler [85]. First ideas are discussed in Section 5.3.2 with regards to the QuadroCore.

5.3.1. Holistic Partitioning of Variables and Instructions

Currently, the partitioning of register variables is mainly used as an input for the allocation of functional units. More importantly, the assignment of a register variable to a processor is implicitly given by the allocation of a functional unit for an instruction using that variable. As a consequence, the data objects should be partitioned during the allocation of functional units by a holistic approach.

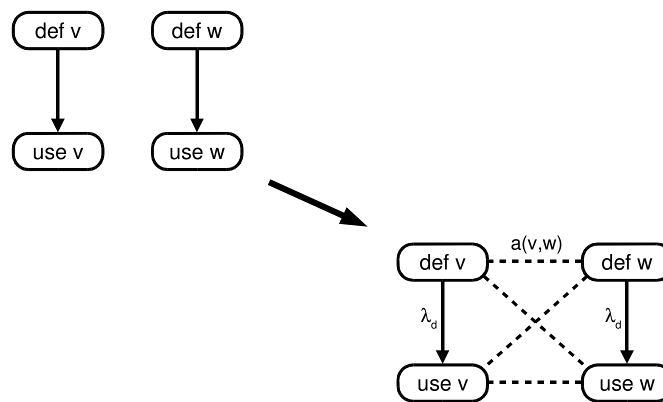


Figure 5.10.: Partitioning of variables and instructions

We suggest a concept where the DDG is augmented by further edges to model the affinities between variables (see Figure 5.10). The original DDG edges get weights according to the represented dependences. Data dependences should have the highest weight among all dependences, because expensive communication using shared memory must be avoided (see Section 6.1). Anti and write dependences between different processors disappear after the register allocation, because each processor reads and writes its own registers by default. Hence, non-data dependences should have a low affinity or even negative affinity to enforce using different cores. A substantiation of the relation between VAG affinities and dependences weights can be part of future research.

5.3.2. Load-Time Scheduling Using Compiler Annotations

Until now, we assumed implicitly that a program can use all four processors of the QuadroCore for execution. In practice, only a subset of the four processors might be available due to several reasons. Section 3.2.2 discusses some motivating examples. In all scenarios the number of available processors is only known at load-time or run-time of a program. Consequently, a program needs to be adapted to the actual number of processors which can be determined by an additional mechanism or the operating system if present. Here, we focus only on code adaption at load-time.

Load-Time Scheduling for Superscalar Processors In [85] we presented a similar approach called CALS for superscalar processors. We assumed that the target machine belongs to a so-called family of processors. A family contains several machines which all use the same instruction set and encoding, but differ in types, numbers, and effectiveness of their parallel execution units. In principle, all processors of a family can execute parallelized machine code. However, maximum performance is only achieved for the processor model that was used as the target machine during scheduling. If the types and numbers of the functional units are not known at compile-time, scheduling must be deferred to the load-time of a program.

Concretely, the CALS compiler prepares parallelization by analyzing the fine-grained parallelism in a program and computing code annotations. Then code and annotations are sent to the target machine in one object file. At load-time, very efficient scheduling can be performed in linear time using an one-pass algorithm and the supplied annotations. Our recent prototype SALT yields comparable or even better results than standard scheduling techniques for fixed target machines [86].

Load-Time Scheduling for Multi-Cores In the following, we give a rough overview of an adaption of the introduced concept for the multi-core. The fundamental challenge is to prepare an efficient load-time partitioning of data objects and instructions by producing a suitable representation at compile-time. For the scheduling we can re-use the linear-time algorithm presented in [85].

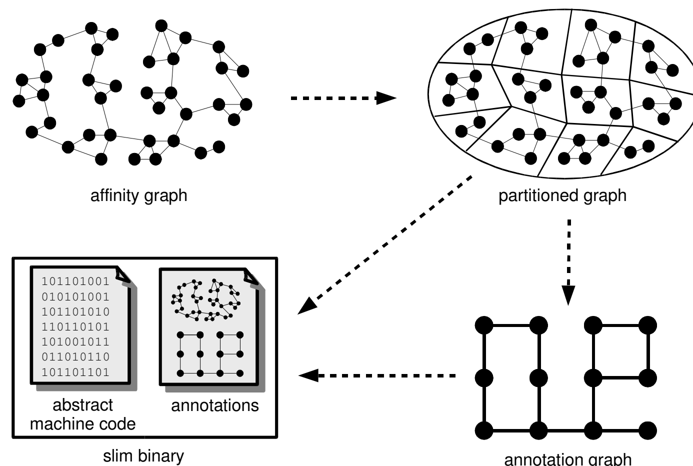


Figure 5.11.: Preparation of load-time scheduling at compile-time

Obviously, a proper representation has to support an efficient partitioning in at least two and at most n parts, where n is the maximum number of processors ($n = 4$ for the multi-core). We envision the following concept: At compile-time (see Figure 5.11), an affinity graph is partitioned into $lcm(2, \ldots, n)$ parts where *lcm* is the *least common multiplier*. For the multi-core with $n = 4$, a graph would be partitioned into 12 parts. Then a so-called *annotation graph* is constructed whose nodes correspond to the partitions of the affinity graph. An edge between two nodes exists if there is at least one edge between the nodes of the corresponding partitions in the affinity graph. The weight of an edge equals the sum of weights for all edges

between the corresponding partitions. Finally, the two graphs and a mapping between their nodes are attached to the unscheduled machine code as code annotations.

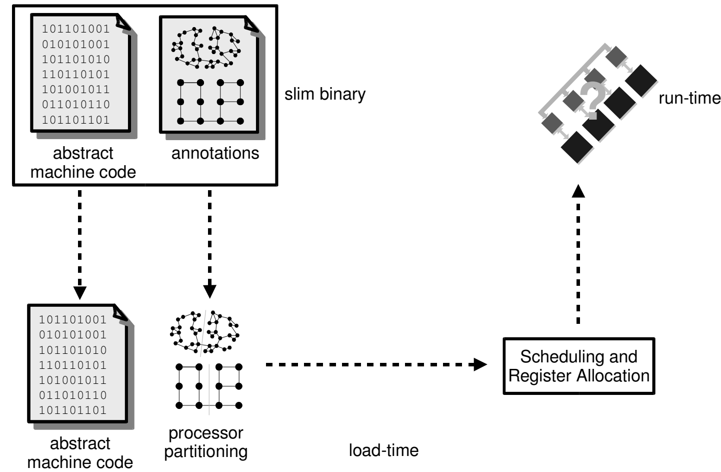


Figure 5.12.: Load-time scheduling by compiler annotations

At load-time (see Figure 5.12), the data objects and machine instructions are first partitioned using the two provided graphs. Then scheduling and register allocation are performed to generate an executable machine program. For simplification, the further annotations needed for scheduling and register allocation have been omitted in the picture.

The affinity graph can be partitioned easily by partitioning the annotation graph according to the actual number of processors. Consequently, the usage of a simplified annotation graph reduces the effort in partitiong at load-time significantly. Furthermore, the size of the annotations is minimized in contrast to encoding all different possibilities of partitioning the original graph.

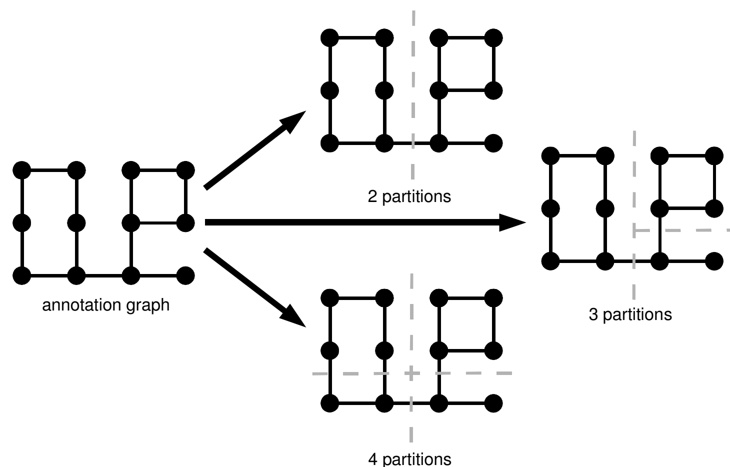


Figure 5.13.: Partitioning of annotation graph

Alternatively, the results of partitioning the annotation graph into $2, \dots, n$ parts could also be provided in the annotations. Hence, the scheduler would just read the partitioning for a certain number of processors from the annotations and then perform the actual scheduling

process. Figure 5.13 shows the results of partitioning the annotation graph from Figure 5.11 using 2, 3, and 4 partitions, respectively.

6. Communication and Synchronization

Contents

6.1 Communication between Processors **II-32**
    6.1.1 Communication Mechanism and Basic Concepts II-32
    6.1.2 Placement of Communication Code II-34
6.2 Barrier Synchronization **II-38**
    6.2.1 Related Work II-38
    6.2.2 Barrier Synchronization for the QuadroCore II-39
    6.2.3 Placement of Local Barriers during Re-Scheduling II-41
    6.2.4 Need for Global Barriers II-42
    6.2.5 Placement of Global Barriers II-44

Without register reconfiguration (see Part IV), a processor of the QuadroCore cannot access the registers of other processors directly. If a register value v is defined by a processor x and used by processor y, v needs to be transported from x to y via a shared memory. As the four processors of the QuadroCore do not operate in lock-step, they need to be synchronized explicitly where necessary.

Structure As communication and synchronization are closely related with respect to the QuadroCore, this chapter handles both topics. At first, Section 6.1 presents our concepts for both compiler and hardware to copy register values between the processors. Importantly, we outline several optimizations and placement strategies in order to minimize the overhead for communication.

Section 6.2 introduces the mechanism for barrier synchronization provided by the QuadroCore and compares it with related work. Then, our strategy for placing barriers is presented: Barriers for dependences within a basic block are inserted on-the-fly during the rescheduling phase. Synchronization code for global dependences is placed by an additional heuristic.

6.1. Communication between Processors

This section deals with the communication of register values between the processors of the QuadroCore. Section 6.1.1 first presents the communication mechanism implemented in the QuadroCore. We introduce the basic concept of identifying those data dependences where communication is necessary. Furthermore, two optimizations are outlined to reduce the effort in communication before computing the actual placement. Section 6.1.2 proposes several strategies for placing communication code based on the determined data dependences.

6.1.1. Communication Mechanism and Basic Concepts

Basically, there are two fundamental concepts to realize a communication between processors [78]: At first, communication can be based on common variables stored in a shared memory. In addition to the shared memory, a synchronization mechanism is needed, which can either use critical sections protected by semaphores or barrier synchronization.

Secondly, message-oriented architectures have no common address space, but the data is partitioned to the different processors. Communication is realized by messages which copy data objects between the local memories. Message passing is more suited for large amount of data exchange but infrequent communication. As the processors of our architecture demand fast and frequent exchange of register values, we decided to use a shared memory for communication. Such shared memory can be either based on the external memory of the multi-core or a dedicated register bank. The current implementation of the QuadroCore features such a dedicated register bank that is used exclusively for communication. The register bank contains 32 entries of size 32 bits each, that can be written or read by two special instructions `cstw` and `cldw`, respectively.

In order to ensure downward compatibility, our system also offers a legacy communication using the external memory of the QuadroCore. For simplification, we often abstract from the actual implementation in the following and just use the terms *shared memory* or *communication buffer*. The designations *register* or *register bank* always refer to the registers of the processors, which can be accessed much faster than the dedicated register bank or the external memory.

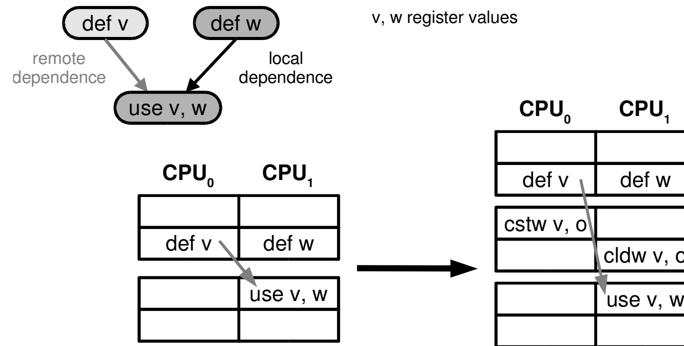


Figure 6.1.: Communication of register values

Remote Data Dependences Figure 6.1 illustrates the organization of the communication buffer and the basic principle of integrating copy instructions into the schedule. In the upper left corner of the picture, an excerpt of a DDG with three nodes is shown. Clearly, the use node depends on the two definition nodes. The right hand data dependence is called a *local* dependence, because the participating nodes are scheduled on the same processor. Instead, the left hand data dependence is denoted a *remote* dependence, because the nodes are executed by different processors.

Communication code is needed to transport the value v defined by `def v` from processor 0 to 1 where it is used by `use v, w`. The communication consists of two parts: Firstly, v is written into the communication buffer by executing a `cstw` instruction on processor 0. Secondly, v is read from the communication buffer via a `cldw` instruction executed by processor 1.

Importantly, communication is only necessary for remote *data* dependences concerning register values. Anti- and write-dependences do not need to be considered, because the unintentional overwriting of values is already avoided by using register banks of different processors. Consequently, this offers much freedom for the re-scheduling phase to produce a very compact schedule.

Optimization In order to reduce the total communication costs, the compiler can perform two optimizations: Let d be a definition of a value v on processor p_d and $u_1, \ldots, u_n$ multiple uses of v on processors $p_1, \ldots, p_n$ different from p_d. If $p_1 = \cdots = p_n = p_u$ for a processor $p_u \neq p_d$, it is sufficient to copy the value v only once (*Each CPU*). If the $p_1, \ldots, p_n$ are pairwise different, the value v can be copied by a fast 1:n communication using one entry of the communication buffer (*Broadcast*). A naive solution would be to perform the communication n times for each use which would slow down the execution significantly (*Each Use*).

For many practical applications, $p_1, \ldots, p_n$ are neither equal to a certain using processor p_u nor pair-wise different. In such a case, a 1:m communication is used, whereas $m < n$ is the number of pair-wise different using processors.

6.1.2. Placement of Communication Code

The placement of communication code is a very important challenge for realistic applications as demonstrated in Figure 6.2. The value v needs to be copied to processor 1 only if the right branch is taken. Hence, copying this value in the upper basic block would cause additional penalty costs if the left branch is executed. The value w is incremented in the loop by processor 0 and used by processor 1 after the loop. Clearly, performance can degrade, if the value is already copied after each increment instead of communicating it once after the loop.

In the following, we present three placement strategies used by the CoBRA compiler. The first two strategies consider remote dependences at single definitions. The simple method places communication code directly after a definition and may produce suboptimal results in case of structured control-flow and loops. The second strategy attempts to determine the most suited position for copy code in terms of execution time. Thereby, communication instructions are moved out of loops. Such heuristic is extended by the third method which tries to merge multiple definitions for common uses. The evaluation of these strategies can be found in Section 13.2.4.

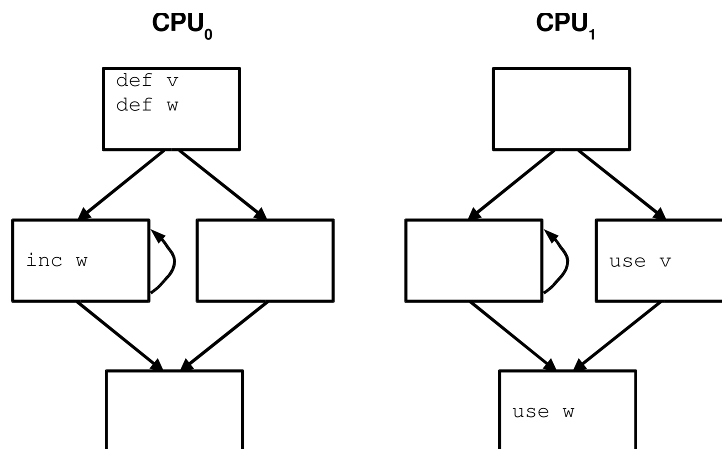


Figure 6.2.: Motivating example of placement of copy code

For simplification, the CoBRA compiler neglects replacing the transportation of a register value v from processors x to y by the re-computation of v by processor y. In order to reduce the complexity of our heuristic approach, we require that associated `cstw` and `cldw` instructions are always placed in a common basic block. Additionally, a `cstw` always corresponds to at least one `cldw`, but not vice versa. Merging multiple `cstw` operations is handled by our third strategy.

Directly After Definition An obvious simple strategy is to insert copy code directly after a definition. The correctness can be shown easily using Figure 6.3. Let us assume that the processors would have only one common register bank. Then, defining a register value in such fictive architecture would correspond to defining a register value and copying it to all other processors in terms of the QuadroCore. A similar observation can be made for using register values. Consequently, accesses to the shared register bank of the architecture in Figure 6.3 correspond to register accesses in the actual QuadroCore combined with a communication of latency 0.

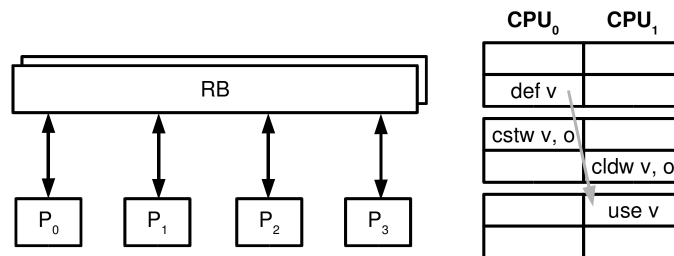


Figure 6.3.: Correctness of communication after definition

But unfortunately, the strategy is not optimal: If it is applied for the example of Figure 6.2, the result will look like illustrated in Figure 6.4: The value v is copied even if the left branch is executed. w will be communicated in each iteration of the loop, although its result is first used after the loop.

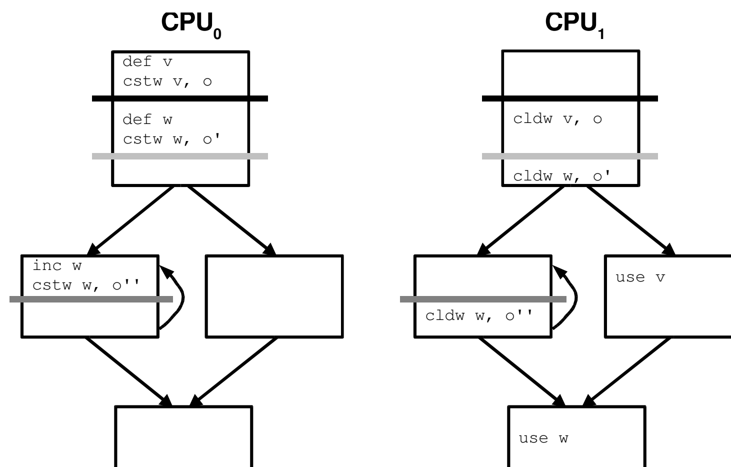


Figure 6.4.: Result of placement strategy *Directly After Definition* for Figure 6.2

Alternatively, communication code may be placed immediately before a use. But such strategy cannot be applied, if multiple definitions are executed by different processors in the presence of structured control-flow. In Figure 6.5, it cannot be decided with safety at compile-time which value of v is relevant at the use. Clearly, v should be copied in the same basic blocks where it is defined to avoid this problem. If v was defined in different branches but on a single processor, copying before a use would be correct, because the value of v would be stored into a unique register. Concretely, each defining processor would need to store v into a common entry of the communication buffer, which would be read before

executing the use. This contradicts with our assumption that a `cldw` instruction is never associated with multiple `cstw` instructions (see beginning of Section 6.1.2).

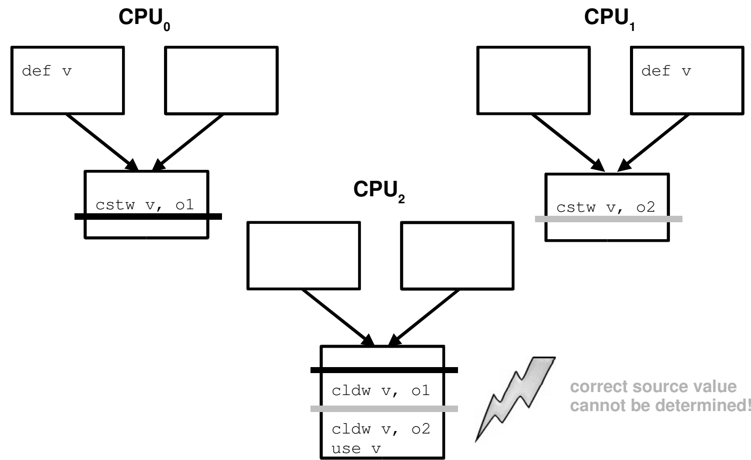


Figure 6.5.: Infeasible communication directly before use

Common Cheap Basic Blocks Assume that a value v is defined in a basic block x and used in a basic block y. Importantly, we restrict our explanations to single definitions here. Let $\mathcal{B}_{xy}$ be the set of basic blocks, which are located on *all* paths from x to y. Obviously, it holds $x, y \in \mathcal{B}_{xy}$. Hence, copy code can be placed at a single position by selecting a block in $\mathcal{B}_{xy}$. In order to reduce the effort in communication, copy code is inserted into a basic block $b \in \mathcal{B}_{xy}$ with the lowest execution frequency, that can be determined by a profiling. The CoBRA compiler uses a static estimation based on the nesting depth. If there are $n > 1$ uses in blocks $y_1, \ldots, y_n$, respectively, then $\mathcal{B}_{xy} = \bigcup_{i \in \{1,\ldots,n\}} \mathcal{B}_{xy_i}$.

Figure 6.6 shows the result of applying this strategy for the example from Figure 6.2: The value v is transported in the right branch now, i.e. only when really needed by processor 1, because such block is executed less often than its predecessor. Value w is still copied within the loop body and not after the loop (before the use), because the use is reached by two different definitions of w.

Currently, the set $\mathcal{B}_{xy}$ containing those blocks located on *all* paths from x to y is computed in a naive way, which may lead to significantly increased compilation times for functions with many control-flow paths. At the first glance, the desired information could be determined using the dominator relation. But the dominator relation is only defined with respect to the entry and the exit node of the CFG for a single function, while $\mathcal{B}_{xy}$ needs to be computed through the dominator for all blocks on paths between x and y. Hence, determining such dominator seems to have the same complexity as our naive solution. Our long-term goal is to improve the computation of $\mathcal{B}_{xy}$, although the actual placement strategy is independent of its procedure.

Let there be multiple remote dependences $d_1 \rightarrow u, \ldots, d_n \rightarrow u$ for a single use u and a basic block $b \in \mathcal{B}_{d_1,u}$. As the current strategy is restricted to single definitions, it will not insert copy code to b, if a program position in a block b is reached by multiple definitions in $\{d_1, \ldots, d_n\}$. Such basic blocks can be identified using def-use information and are removed

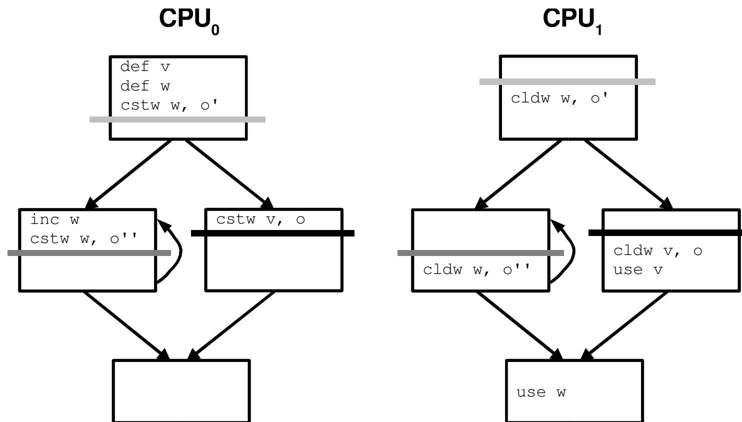


Figure 6.6.: Result of placement strategy *Common Cheap Basic Blocks* for Figure 6.2

from $\mathcal{B}_{xy}$ before selecting the cheapest block. If the definitions are executed by different processors, placing communication code in such blocks is even not allowed. This problem corresponds to the infeasibility of copying directly before a use, that has already been discussed above.

Merge Definitions The third strategy is an extension of the previous one and merges definitions of remote dependences with common uses, if possible. Let b be a basic block located on all paths between the definitions and uses and $d_1, \ldots, d_n$ be the definitions reaching b. Basically, copy code can be inserted into b, if $d_1, \ldots, d_n$ are executed by the same processor. Hence, the code size will be reduced compared to the previous strategy, which would insert communication code into a block reached by a single definition only.

As demonstrated in Figure 6.7 for the example from Figure 6.2, this strategy also moves the instructions transporting the value w outside the loop. Consequently, we have found the optimal solution by (i) selecting a basic block with minimum execution frequency which occurs on all paths from definitions to uses and (ii) merging definitions to fulfill the given constraints to our approach (see beginning of Section 6.1.2).

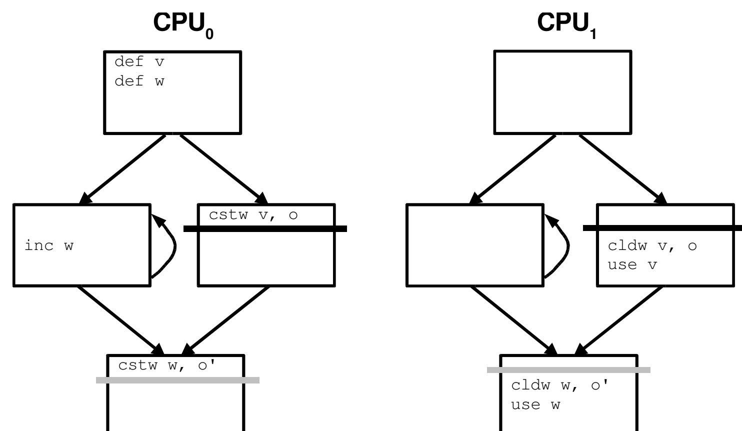


Figure 6.7.: Result of placement strategy *Merge Definitions* for Figure 6.2

6.2. Barrier Synchronization

The processors of the multi-core operate in an asynchronous manner and need to be synchronized explicitly for inter-processor dependences. Our goal was to utilize a synchronization mechanism which can be supported transparently by the compiler, while the actual synchronization is implemented in hardware completely. Consequently, we have chosen barrier synchronization [78] instead of semaphores, because the latter one requires a special programming model.

In the presence of coarse-grained parallelism, processors can operate more independently and only synchronize rarely when using asynchronous execution. This implies an improvement of the performance compared to lock-step execution. Furthermore, the code size can be reduced, because VLIW code contains additional `nop` instructions to fill empty slots if there is not enough parallelism. The clock frequency of the multi-core may also be higher than that of a comparable VLIW machine where the processors have to be synchronized after each operation. If an application mainly contains fine-grained parallelism, the number of inter-processor dependences is probably very high. Hence, barrier synchronization can introduce a large overhead and therefore is too expensive in terms of both execution time and code size. As a result, a lock-step execution may be more promising.

Barrier synchronization is also applied between the different multi-cores of the GigaNetIC architecture (see Section 4.1). In contrast the barriers inserted by the CoBRA compiler, those barriers must be specified explicitly using the Bulk-Synchronous Parallel (BSP) model [166], which is introduced roughly in the next subsection.

Structure This section begins with a study of the related work about barrier synchronization in Section 6.2.1. Section 6.2.2 outlines where barrier synchronization is needed when compiling programs for the QuadroCore. Furthermore, we give a precise definition of the employed barrier mechanism. For more information about the hardware implementation, the reader may refer to [87].

Throughout the rest of this section, the placement of barriers is explained in detail: At first, Section 6.2.3 presents the re-scheduling phase which inserts barriers for dependences within a basic block on-the-fly. The need for global barriers is motivated by discussing two problem situations in Section 6.2.4. Finally, an additional heuristic to determine synchronization code for global dependences is sketched in Section 6.2.5.

6.2.1. Related Work

Typically, code generation for asynchronous multi-cores is based on a special programming model provided by the source language or a certain library. The compiler inserts barriers to synchronize certain processors where necessary. Consequently, an important goal is to speed up program execution by reducing the number of barriers using optimization techniques.

In the literature, a multitude of approaches can be found which often focus on special structures like loops. Boyle and Stöhr [125] presented an algorithm that aims for minimizing the number of barriers placed in perfect loop nests and in certain imperfect loop nest

structures. Their approach has been implemented in a prototypical FORTRAN compiler and can deal with entire, well-structured control-flow programs containing arbitrary nesting of conditional control-flow, loops, and subroutines. The referred paper also gives an excellent overview of further related work.

Han et al. [71] developed an algorithm for elimination of barriers applied at compile-time that targets distributed-memory parallel architectures like the Cray T3D. Instead of just reducing the number of barriers, their approach tries to use data and event synchronization via post/wait statements.

Gupta [68] developed another interesting approach called *fuzzy barrier* where the waiting time at a barrier is exploited by executing instructions not related to the barrier. Concretely, the compiler computes a set of instructions that can be executed by a processor p after being ready to synchronize. If all remaining processors participating in the barrier are ready for synchronization, processor p must synchronize and then continues execution.

Valiant [166] proposed the Bulk-Synchronous Parallel (BSP) model as a common scheme for parallel computation. It is both used as a machine model for hardware architectures as well as a programming model for algorithm designers. In principle, a BSP machine is based on processors which have separate local memories and are connected by a networking supporting point-to-point messages and barrier synchronization. A BSP program consists of a sequence of *supersteps* where the processors first perform local computations and can send messages to other processors. Then each processor invokes a synchronization function and waits until all processors have reached the barrier. Finally, all messages that have been sent in the previous superstep are received and made available for the next superstep. In the GigaNetIC architecture (see Section 4.1), coarse-grained parallelism among multiple embedded QuadroCores is exploited by using the BSP model.

Finally, the hardware approaches for barrier synchronization should be addressed. Dietz et al. [45] developed a static barrier mode for a MIMD machine. Barriers are realized by a memory access which implicitly synchronizes all enabled processors. Their approach also exploits *collective branching* [69] where one processor determines the branch condition and forwards it to the other processors.

Beckmann et al. [11] published a hardware scheme for barrier synchronization in a single cycle. Their paper also includes an extensive survey about the variety of hardware approaches. The reader may refer to this paper for further details.

6.2.2. Barrier Synchronization for the QuadroCore

Two processors have to be synchronized explicitly to respect remote data dependences (see Figure 6.8). Let us assume that a processor 0 defines a register value which is needed by processor 1 later. If processor 0 suffers from an unexpected delay, the other processors will not be stalled like in a VLIW machine. Consequently, a barrier must be inserted to ensure that processor 1 does not read the register value from the communication buffer before processor 0 has written it.

In addition to remote dependences concerning register values, barriers are also needed to respect dependences between accesses to the external memory. For instance, it must be

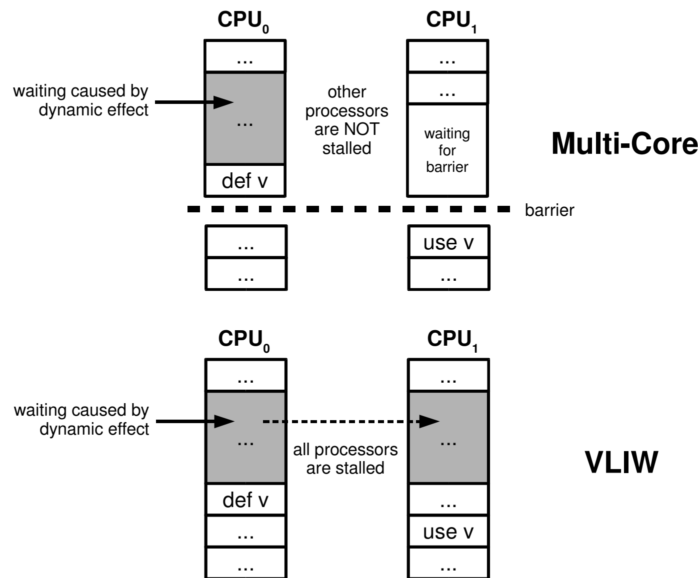


Figure 6.8.: Ensuring remote dependences by VLIW machine and multi-core

guaranteed that a read operation accessing a memory structure is always executed after the defining write operation. Last but not least, the processors will be synchronized before broadcasting the result of a comparison (see Section 4.2.4).

Definition of Barrier Mechanism Synchronization between a certain set of processors P is realized by executing a special `barrier` instruction on each processor in P. As soon as all processors in P have executed their barrier instruction, they can continue execution. If only a real subset $P' \subset P$ has reached a barrier at a particular time, the processors in P' must wait for the remaining processors.

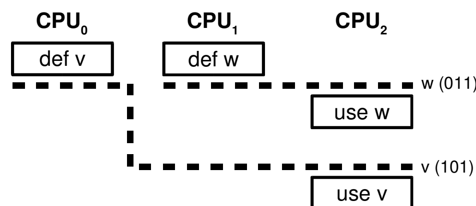


Figure 6.9.: Example of barrier synchronization for the QuadroCore

In order to synchronize disjoint sets of processors at the same time independently, a `barrier` instruction has an immediate field which represents the set P as a bitmask, called the *barrier mask*. In the following, this mask is often denoted as a *barrier mask* and equated with the corresponding set of processors.

Figure 6.9 illustrates an example. For simplification, the instructions accessing the communication buffer have been omitted in the picture. Processor 0 and 1 define values v and w, respectively, which are used by processor 2. Hence, processor 2 has to be synchronized with both processors 0 and 1 separately. The alternative solution using only one barrier is not considered here.

Now assume that processor 2 reaches its barrier with processor 1. Then it has to wait until processor 1 defines w and executes its barrier instruction. Importantly, the execution of the barrier instruction by processor 0 does not induce processor 2 to continue execution, because the barrier masks are not equal. It is not sufficient that processor 2 is also denoted by the bitmask of the barrier instruction executed by processor 0. Instead, processor 2 is first synchronized with processor 0 after the barrier with processor 1 and using the supplied value w. Consequently, the barrier mask can also be regarded as the identifier of a barrier.

In the following, the algorithm for placing barrier instructions during the re-scheduling phase of the CoBRA compiler is presented. Then, we introduce several optimizations to reduce the number of barriers and hencefore the total execution time of generated code.

6.2.3. Placement of Local Barriers during Re-Scheduling

The re-scheduling phase inserts barriers on-the-fly to respect remote dependences within a basic block. Further barriers for global dependences are placed by another heuristic presented in the following subsection. Here, we first motivate the need for a re-scheduling phase and outline the functioning of our implementation. Then an extension is outlined which inserts barriers automatically where necessary during scheduling.

Motivation and Functionality of Re-Scheduling The re-scheduling computes a more compact schedule by re-arranging instructions. With respect to the S-Core used in the QuadroCore, it has three different motivations: Firstly, the code selection of the CoBRA compiler generates many transport instructions (`mov`) between registers, because the S-Core is only a 2-address machine. The majority of these instructions becomes superfluous because of the register coalescing performed by the CoBRA compiler and is eliminated by the peephole optimization. Consequently, new opportunities for arranging instructions are given.

Secondly, the register allocation can add spill code which implies empty slots on the other processors. Thirdly, remote anti and write dependences disappear after the register allocation, because each processor reads and writes its own register by default. As a consequence, more Instruction-Level Parallelism (ILP) is uncovered. In both cases, compacting the schedule can yield significant performance improvements.

The re-scheduling phase of the CoBRA compiler re-constructs the DDG and performs list scheduling [82] which operates on basic block level. List scheduling uses a so-called ready list which contains all operations that are not yet scheduled, but whose predecessors are scheduled. Selection of nodes is mostly done by using heuristics such as earliest scheduling time or height in DDG which corresponds to the sorting of the ready list.

An alternative idea for re-scheduling is to apply local optimizations like re-ordering to the existing schedule. Typically, the results are worse unless complex transformations are used. Hence, we decided for using list scheduling, because it computes a complete re-arrangement of all instructions in a basic block and exhibits a reasonable complexity.

Placement of Barriers We extended the concept of list scheduling as illustrated in Figure 6.10: When the scheduler selects a node u from the ready list, all successors v which are

executed on a different processor than u will be marked with the barrier mask $\{u, v\}$. In the example, the two right-most uses (gray and dark-gray) are marked when the definition is extracted from the list.

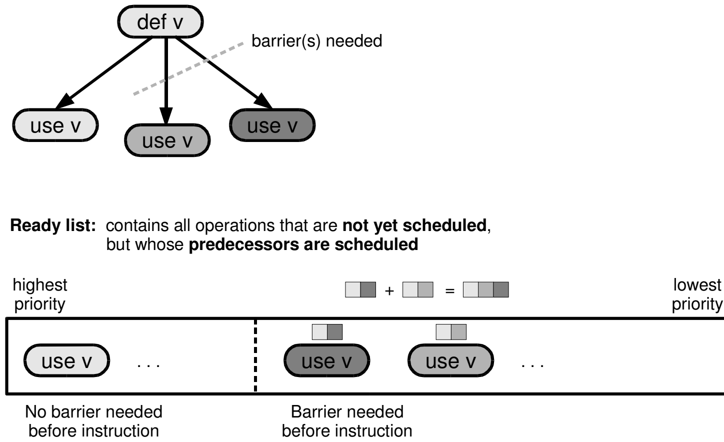


Figure 6.10.: Barrier insertion by list scheduling

Each time an instruction with a marker is selected, a barrier will be inserted before this instruction. Such barrier synchronizes the processors denoted by the markers of the instructions in the ready list. Obviously, the combination of the barrier masks of all marked instructions reduced to a single barrier synchronizing all relevant processors. With respect to our example, the three processors used will be synchronized. Finally, the markers are removed from all instructions in the ready list. Consequently, a barrier between two dependent instructions $u \to v$ is always inserted *after* placing u and *before* placing v.

In order to minimize the number of inserted barriers, marked instructions are selected with a lower priority. Concretely, the existance of a marker is used as a primary criterion, while the original criterion becomes the secondary criterion. Hence, synchronization instructions are inserted as late as possible in order to support coalescing of multiple barriers into fewer barriers.

6.2.4. Need for Global Barriers

The re-scheduling phase presented above inserts only local barriers to respect remote dependences within a basic block. In this subsection, we motivate the need for global barriers between basic blocks or when calling functions. For simplification, we use abstract examples which consider the unintended overwriting of values instead of differing between register and memory dependences. Thereby, we neglect the original definitions of values and focus on a single use as well as a succeeding re-definition. Section 6.2.5 deals with a heuristic to place synchronization code for global dependences.

Structured Control-Flow At first, we focus on single functions with structured control-flow. Figure 6.11 shows an excerpt from a CFG where control flow (i) branches or (ii) joins. For case (i), we assume that all processors take the left branch. As processor 1 is delayed unexpectedly, processor 0 might overwrite the value v before processor 1 has read it. This case

can occur, although the processors are synchronized before the `crsync` instruction (see Section 4.2.4). Re-scheduling might re-order the instructions of a basic block such that `crsync` is executed before instructions whose execution might be delayed or implies remote dependences.

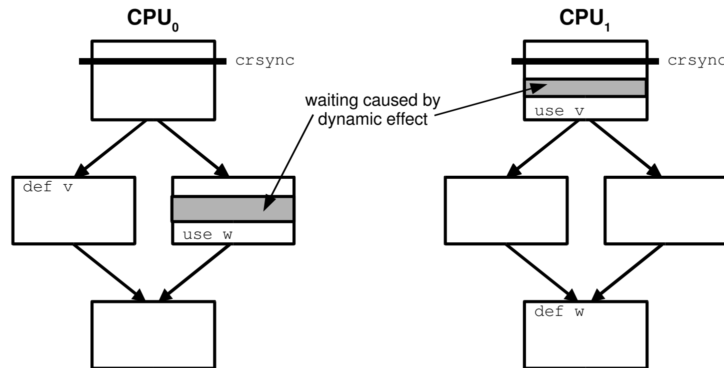


Figure 6.11.: Need for barriers between basic blocks

In case (ii), it is assumed that each processor branches to the basic block on the right. Similarly, processor 0 could read the value w after being overwritten by processor 1.

Obviously, both situations could be avoided by a naive strategy which places a barrier synchronizing all processors at the beginning of each basic block. In the next subsection, a much better heuristic is presented where barriers are only inserted if necessary.

Function Calls Explicit synchronization might also be needed when calling other functions as Figure 6.12 illustrates. Both processors call a function g from a function f. In g, processor 1 is delayed before using a value v that could be part of a global memory structure in this example. Instead, processor 0 leaves g immediately and returns to f where it might overwrite v before processor 1 has read it.

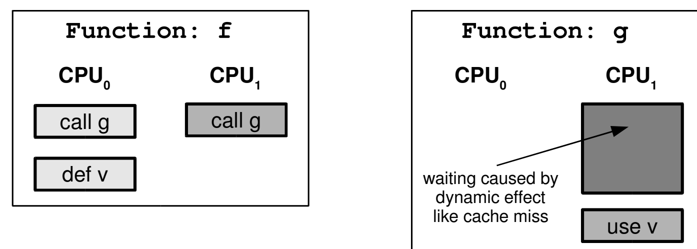


Figure 6.12.: Need for barriers between functions

Again, a naive strategy would just prepend or append barriers to the beginning or end of each function. An optimal heuristic could use inter-procedural information about barriers as well as accessed memory elements and registers to determine the needed barriers precisely.

For simplification, the CoBRA compiler inserts barriers at function calls during the re-scheduling: If a call depends on other calls, memory instructions, communication code, or vice versa, a barrier will be inserted. Surely, this solution is quite similar to the naive one,

but it allows to merge the barrier with other barriers inserted during the re-scheduling (see Section 6.2.3). Just adding barriers at the beginning and end of a function would not allow such optimizations.

6.2.5. Placement of Global Barriers

The previous subsection motivated the need for global barriers in case of structured control-flow and multiple functions. As mentioned above, barriers at function calls are already placed during the re-scheduling phase. Hence, this subsection deals with the placement of synchronization code beyond basic blocks for single functions. At first, we explain two characteristic scenarios with global dependences where barriers have to be placed between basic blocks. Then a heuristic algorithm to insert barriers in these cases is outlined briefly.

Problem Cases Firstly, overwriting of entries in the communication buffer must be avoided (see Figure 6.13). Let us assume that processor 1 reaches the lower basic block at first and executes the `cstw` instruction before processor 0 executes its `cldw` instruction. Then the value at index o of the communication buffer would be overwritten before reading it.

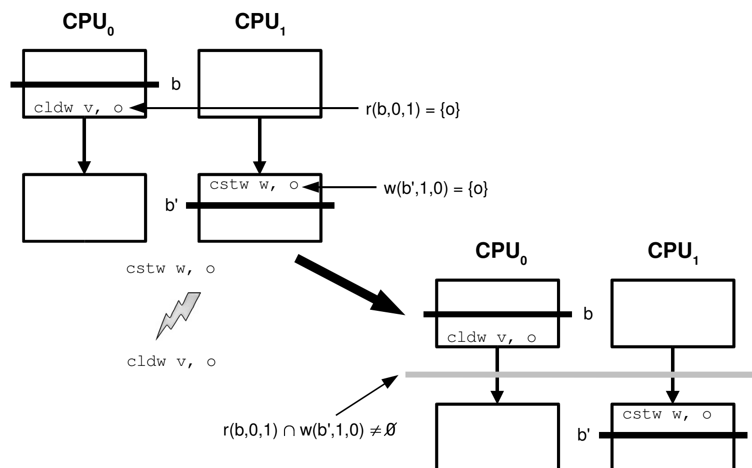


Figure 6.13.: Global barrier to avoid overwriting of entries in communication buffer

The second case concerns global memory dependences (see Figure 6.14). Imagine a similar situation as before where processor 1 executes its store instruction in the lower basic block before processor 0 can invoke its load operation.

Theoretically, a third case might be possible where associated `cstw` and `cldw` instructions are added to different basic blocks. A barrier would be needed to avoid that the transported register value is read before it has been written. But this case cannot occur in the current compiler prototype, because communication code is always placed in the same basic block (see Section 6.1.2). If the placement of communication code is improved in the future, this case can be reduced easily to the first case.

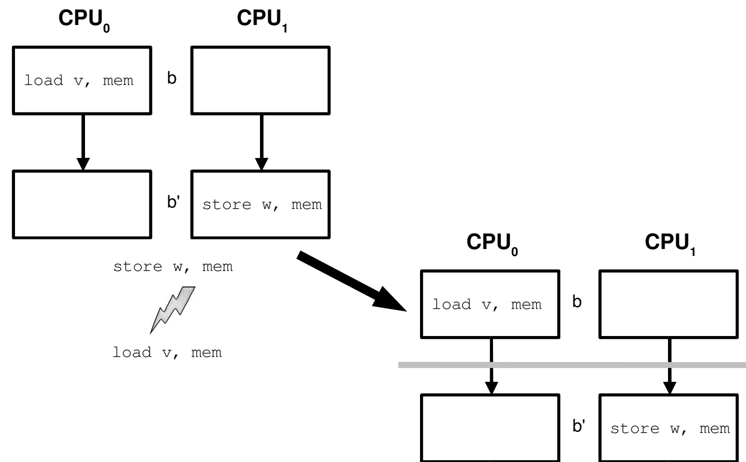


Figure 6.14.: Global barrier to respect memory dependences

Placement of Barriers Our heuristic algorithm places barrier operations for single functions. It consists of three phases: The first two phases determine processors which need to be synchronized according to the scenarios introduced above. The last phase places the barriers by considering existing synchronization code. For simplification, we explain only the first two phases in the following.

The first phase inspects the schedules to identify those entries of the communication buffer which could be overwritten unintentionally. Concretely, it computes two functions, $r : B \times P \times P \rightarrow C$ and $w : B \times P \times P \rightarrow C$. $r(b, i, j)$ models the entries of the communication buffer which are read by `cldw` instructions executed on processor i in basic block b after the last barrier with processor j. $w(b, i, j)$ represents the entries of the communication buffer that are written by processor i using `cstw` instructions in basic block b before the first barrier with processor j. If $r(b, i, j) \cap w(b', j, i) \neq \emptyset$, the processors i and j have to be synchronized explicitly between the end or beginning of basic block b or b', respectively. Let us consider the example from Figure 6.13 again. Obviously it holds $r(b, 0, 1) = \{o\}$, because of the `cldw` after the last barrier with processor 1. Similarly we can conclude $w(b', 1, 0) = \{o\}$. As $r(b, 0, 1) \cap w(b', 1, 0) \neq \emptyset$, the processors 0 and 1 must be synchronized between the end of b and the beginning of b'.

The second phase considers global memory dependences to compute further pairs of processors which need to be synchronized. Unfortunately, the concept of the first phase cannot be re-used, because the compiler cannot determine the accessed memory elements with safety. Hence, this phase analyzes the schedules of the relevant basic blocks to find a barrier synchronizing the involved processors between two globally dependent memory instructions $u \rightarrow v$. If such a barrier does not exist, synchronization of the processors executing u and v is needed.

Part III.

SIMD/MIMD Reconfiguration

Motivation and Introduction If a given program exhibits both regular and non-regular structures, a reconfiguration between SIMD and MIMD execution can be beneficial. A good example are aggregation network access nodes, like DSL Access Multiplexers (DSLAM), capable of transcoding video/audio data during transmission. Program code implementing the actual transmission may be executed in MIMD mode, while most of the regular tasks are based on the SIMD mode. As our QuadroCore has been developed for network applications originally, it is an ideal architecture to evaluate the reconfiguration. By using a single code stream for SIMD execution, the energy consumption and code size of a program can be reduced significantly. Furthermore, vectorizing a given piece of code exhibiting a regular structure surely yields comparable or even better results than a scheduling that neglects its regularity.

This part presents our approach for SIMD/MIMD reconfiguration called CHARISMA[1] with respect to both compiler and reconfigurable architecture. For simplification, a special version of the CoBRA compiler only supporting SIMD/MIMD reconfiguration is denoted as CHARISMA compiler in this part. We focus on switching between SIMD and MIMD execution and neglect the utilization of a subset of the available processors in case of low parallelism (see Section 3.1.1). Additionally, we do not deal with subword computations supported by Multimedia Extensions (MME) like MMX or AltiVec (see Section 7.2) as well as combination of instructions between processors (see Section 3.1.5).

Structure At first, Chapter 7 presents the work related to CHARISMA. This comprises a short overview of vector machines and classical vectorization as well as a survey about MMEs and associated vectorizing compilers. Furthermore, an alternative work towards SIMD/MIMD reconfiguration is discussed, where similar ideas have been established, but no compiler or hardware prototype is on-hand yet. Chapter 8 outlines the concepts of the CHARISMA compiler for our reconfigurable architecture. Thereby, we focus on the vectorization phase, the machine model in SIMD mode, as well as the conceptual extensions to the register allocation.

[1] **C**ompiler **H**andles **A**rchitectural **R**econfiguration **I**ntegrating **S**IMD **M**IMD **A**utomatically

7. Related Work

Contents

7.1 Vector Machines and Classical Vectorization **III-6**
7.2 Compilation for Multimedia Extensions **III-7**
7.2.1 Challenges of Vectorization for Multimedia Extensions III-8
7.2.2 Vectorizing Compilers for Multimedia Extensions III-9
7.2.3 Vectorization by Pattern Recognition III-9
7.3 SIMD Processors **III-10**
7.3.1 CELL Microprocessor III-11
7.3.2 eLite DSP III-11
7.4 SIMD/MIMD Reconfiguration **III-12**

This chapter presents the work related to CHARISMA[1]. We start with a short overview of classical vector supercomputers and vectorization techniques in Section 7.1.

Vector machines have inspired the development of Multimedia Extensions (MME) for general-purpose microprocessors. Section 7.2 first motivates some challenges when applying classical vectorization techniques for MMEs. Then, we characterize selected vectorizing compilers based on classical methods. Additionally, an alternative approach based on tree-pattern matching is presented.

Larsen et al. [107] developed a simple, but concise method for vectorization that operates on basic block level and exploits vector parallelism through loop unrolling. Hence, we decided to re-use their approach for the CHARISMA compiler. As we have adapted their method for our reconfigurable QuadroCore, their approach is presented in Section 8.3 of the next chapter.

Section 7.3 presents two selected SIMD processors. The famous CELL microprocessor was developed for multimedia applications and offers high performance for SIMD code, but cannot be reconfigured between SIMD and MIMD execution like our QuadroCore. The indirect access to vector element registers provided by the eLite DSP inspired our concept of memory accesses in the SIMD mode (see Section 8.2.2).

Finally, we discuss existing work from the literature, which also uses SIMD/MIMD reconfiguration. In contrast to CHARISMA, their approach lacks a proper hardware implementation as well as a vectorizing compiler.

7.1. Vector Machines and Classical Vectorization

Many early supercomputers like the famous Cray were vector processors [177]. Such machines include a large file of vector registers which may comprise up to a few hundred of elements each (see Figure 7.1). Classical vector machines process vector registers in a pipelined manner. Each operation is partitioned into suboperations, which are executed sequentially in stages of a pipeline. The speedup gained by such pipelining corresponds to the number of stages, if compared to a sequential execution.

Array computers have parallel processing units to operate on the elements of vector registers. Like the pipeline stages of vector machines, the units cannot influence control-flow. Systolic arrays [103] are a special form of array computers and are based on a 2 or 3-dimensional mesh of processing elements. The execution is performed synchronously in pipelined stages. Arrays are well-suited for processing sets of homogenous data.

In order to exploit the huge amount of parallelism offered by supercomputers efficiently, most of them are programmed using HLLs. The language of choice is Fortan, where a number of vectorizing compilers is available.

The classical vectorization algorithm by Allen and Kennedy [3, 5] is illustrated in Figure 7.2: At first, a DDG is constructed and the Strongly Connected Components (SCC) in that graph are computed. Then, each SCC is replaced by a single node to get an acyclic graph. Obviously, the instructions of a condensed node cannot be vectorized. Instead, the

[1] **C**ompiler **H**andles **A**rchitectural **R**econfiguration **I**ntegrating **S**IMD **M**IMD **A**utomatically

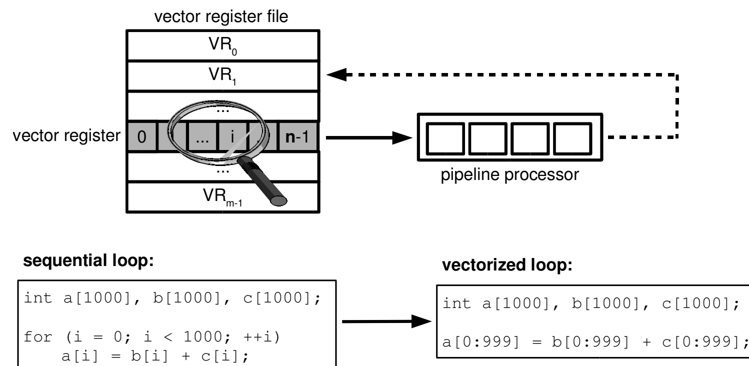


Figure 7.1.: Vector machine

cycles are broken and mapped to a sequential loop, while the remaining simple nodes are vectorized.

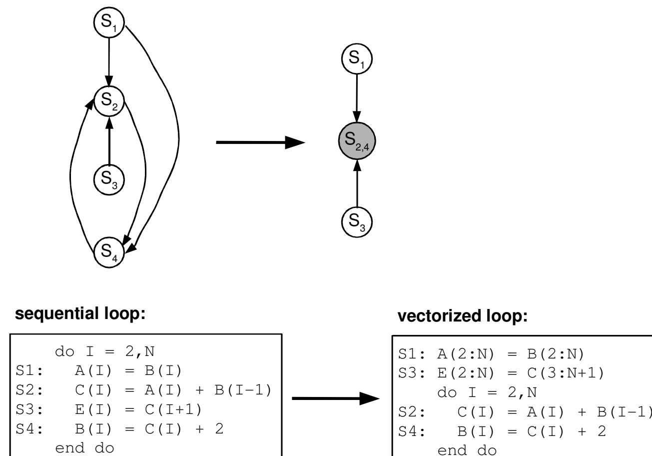


Figure 7.2.: Overview of classical vectorization

In practice, the application of this simple principle requires a couple of complex loop transformations [182, 7, 177] like scalar expansion, loop interchanging, loop fission, loop fusion, or strip mining. The basic problem is the interaction between the different techniques as well as the resulting non-determinism.

7.2. Compilation for Multimedia Extensions

In the past decade, the high computational workloads caused by multimedia applications have led to the development of MMEs for general-purpose microprocessors. The MMEs of most processors have a simple SIMD architecture based on short, fixed-length vectors, a large register file, and an instruction set targeted at the very specific multimedia application domain. Importantly, the design is inspired from array computers (see Section 7.1).

The first SIMD extension, MAX-1 [111], was introduced by HP in 1994. Today, there exist at least 10-15 commercially available MMEs like Intel Multi-Media eXtension (MMX) [118,

131] or Streaming SIMD Extensions (SSE) [43, 160], Motorola AltiVec [44, 70], Sun Visual Instruction Set (VIS) [162, 100], etc. to mention only a few of them.

MMEs are often programmed manually using assembly, system libraries, or macros in a high-level language. Such strategies are only benefical for special applications like games, where maximum performance is the most important criterion. As the execution of games mainly considers small parts of a program, it is even worthwhile to decide for their implementation by hand-optimized assembly code. On the other hand, manual effort is usually very time-consuming and error-prone. Further problems of these approaches are the low portability due to inconsistencies among different instruction sets or high development costs.

The similarity between MMEs and vector machines has resulted in the usage of classical vectorization techniques (see Section 7.1). However, there are also some fundamental differences which are discussed in the next subsection. Then, Section 7.2.2 characterizes some work towards vectorizable compilers for MMEs that use classical vectorization methods. A further technique which is based on pattern recognition is discussed in Section 7.2.3. The Superword Level Parallelism (SLP) approach used in the CHARISMA compiler targets sequential code by considering adjacent memory accesses and unrolls loops to uncover vector parallelism. It is explained in detail in Section 8.3.

7.2.1. Challenges of Vectorization for Multimedia Extensions

In contrast to classical SIMD computers, MMEs mostly operate on small data types of 8 or 16-bit width. Hence, multiple values are stored in a packed manner in one 32 or 64-bit register and processed by a single operation. Similarly, a memory access reads or writes whole packed vectors to exploit the width of the data bus as best as possible. Due to the number of supported data types, a lot of instructions are needed in principle to provide a quite small set of arithmetic operations.

In practise, multimedia instructions sets are typically very specialized and only support certain types of vectors. For instance, MMX does not provide floating-point computations for packed data. In SSE, the min/max operations are only available for vectors of signed 16-bit integers and unsigned 8-bit integers.

Another problem affects memory accesses: An MME has a much weaker memory unit and lacks native support for the gather/scatter type of memory operations to vectorize code with non-contiguous data. Architectures of MMEs typically have alignment constraints which impede the vectorization of loops. The latter problems is handled extensively by Wu et al. [179]. They have called SIMD vectorization as *simdization* to emphasize the mentioned challenges compared to classical vectorization. However, we still denote it as *vectorization*, because this thesis does not focus on generating optimal code for MMEs.

Further challenges are introduced by the style of programming MMEs like the usage of pointers instead of arrays which complicates dependence analysis or the manual unrolling of loops. Last but not least, a language like C, which is widely used for programming multimedia applications, has a lot of limitations in contrast to Fortan that is common for vector computers: For example, all subword data types are automatically transformed to the size of a machine word before performing any arithmetic operations. Additionally, special com-

putations like saturated arithmetics are not supported directly by the language, but must be expressed with native C operations. Ren et al. [141] discuss these aspects in detail.

7.2.2. Vectorizing Compilers for Multimedia Extensions

Cheong and Lam [33] have implemented a vectorizer using the SUIF compiler kit [175], which operates as a source to source optimizer and consists of two phases. At first, parallel loops are identified and conditional operations are replaced by predicated single operations. Then, scalar expansion and loop fission [182, 7, 177] are used to vectorize instructions for an abstract model supporting infinite length vectors. Finally, some optimizations like minimization of total number of vector registers are conducted and the vectorized operations are transformed into function calls of Sun VIS instructions.

DeVries [42] has developed a vectorizing SUIF compiler which targets inner for-loop bodies. Strip mining adapts them for the maximum vector length of the hardware and splits the loop body into scalar and vectorizable parts.

Krall and Lelait [101] have proposed a vectorizing compiler for Sun VIS, that can either use classical techniques or loop unrolling (see Section 8.3). Their evaluation has demonstrated that loop unrolling is as effective as conventional vectorization, but can be implemented with a much lower effort.

Sreraman et al. [154] have presented the implementation of a vectorizing C compiler for Intel MMX. Again, the vectorizer is based on the SUIF toolkit and performs a C source to source transformation, whereas the vectorized parts are encoded as inline assembly. The basic idea is to identify data parallel sections of a program and to improve the results by transformations like strip mining, scalar expansion, and condition distribution.

Bik et al. [16, 18] have developed the automatic parallelization and vectorization methods used by the Intel C++/Fortan compilers. At first, control-flow, data-flow, and data dependences are analyzed using classical techniques [177]. Then, the program is restructured to enable a parallelization. If the data dependence analysis has failed beforehand, the compiler even adds dynamic data dependence tests to improve the exploitation of parallelism. Finally, the sequential code is transformed into semantically equivalent multithreaded code or SIMD instructions. For more details about the vectorizer, the reader may refer to [15].

7.2.3. Vectorization by Pattern Recognition

Classical vectorization focuses on how to transform source code into vector form correctly. Instead, the utilization of multimedia extensions concentrates on automatic recognition and vectorization of SIMD-style parallelism. First work in this direction has been done by Bik et al. [17], which used tree-rewriting to recognize saturation and min/max operations. Boekhold et al. [20] have proposed a programmable engine for code transformations on ANSI C programs to exploit coarse-grained parallelism or multimedia instruction sets.

Fisher and Dietz [57] have developed a C-like language called SWARC to ease the writing of programs exploiting SIMD Within A Register (SWAR) parallelism. The term SWAR was formed by the authors and basically denotes SIMD parallelism for multimedia extensions

(see Section 7.2). The idea is to write portable SIMD programs in the SWARC language, which are compiled to efficient modules and interface code in order to integrate them into ordinary C programs.

Jiang et al. [91] have implemented a recognition engine based on the BURS technique [58], which is integrated into the classical vectorization algorithm (see Section 7.1). At first, the representation of a program is simplified using IF-conversion [4], which converts control dependence into data dependence. Then, vectorizable statements are identified using the BURS technique. Each rule consists of the BURS rule, the vector length, estimated costs for vectorized and sequential code, a definition as well as its uses, and a constraint to handle dependence or data type. From this information, a Directed Acyclic Graph (DAG) is constructed, where each node either represents the best rule for a statement or a match of a multi-statement rule. Edges model the inclusion relationship between multi-statement and statement nodes. Afterwards, the DAG is decomposed into a series of trees and invalid or unprofitable trees are removed. The selection of the best trees for a set-covering problem is formulated by defining the weight of each tree as the speedup of vectorization. As set-cover is an $\mathcal{NP}$-complete problem, a greedy algorithm is used, which favours trees with most number of statements and minimal costs first. Finally, the selected trees are mapped to SIMD instructions.

In principle, the approach of Jiang et al. should be quite powerful, extendable, and merely require the specification of a small amount of rules. The authors demonstrated that their method can recognize more vectorizable idioms supported by typical multimedia extensions like MMX than classical vectorization. On the other hand, the vectorization of the example discussed in the paper already requires a number of rules compared to its complexity. The effort in specification required for the bigger benchmarks considered by the evaluation is not disclosed. The presentation of their approach even suggests that the number of rules can be quite high. Last but not least, the selection of the best subtrees bears a high complexity that contrasts with the smartness of the tree pattern matching.

7.3. SIMD Processors

In this section, we discuss two selected processors which both offer SIMD execution. The CELL microprocessor (see Section 7.3.1) is aimed at multimedia and vector processing applications and is employed by the PlayStation 3 game console, for instance. In contrast to the QuadroCore, it provides much more computational power, but cannot be reconfigured between SIMD and MIMD execution.

The eLite DSP (see Section 7.3.2) offers an indirect access to the elements of vector registers in order to speed up the data-reorganization impeding conventional multimedia extensions. Such idea can also be re-used for the memory accesses in SIMD mode of the reconfigurable QuadroCore (see Section 8.2.2).

7.3.1. CELL Microprocessor

The famous CELL microprocessor [132] was developed jointly by Sony, Toshiba, and IBM for multimedia and game applications. It consists of a multithreaded Power Architecture processor as well as eight attached streaming processors on a single chip. Most instructions provided by the streaming processors operate in a SIMD fashion on 128-bit words. Each processor has seven execution units and can issue two instructions per cycle.

The CELL architecture supports a wide range of programming models [49]: At first, the streaming processors can be programmed in assembly in order to achieve best results. Alternatively, the user can write separate programs for the different processors in a HLL using distinct compilers. Intrinsic functions are always provided to optimize performance critical parts of an application.

The third level of support enables automatic vectorization for the available SIMD units, which operates on both loops and basic blocks. The alignment contraints imposed by the memory subsystem of the architecture are handled by a systematic approach [179]. At this level, the user still has to partition an application manually as well as inserting special code and data transfers.

The highest level of abstraction demands the usage of an OpenMP programming model, whereas the programmer has to specify parallel sections of a program. Due to the authors, their approach is also applicable to automatic parallelization support.

Vectorization on loop-level is based on classical techniques targeting innermost loops [5, 182]. Non-vectorizable loops are distributed to the different processors using a cost model to avoid excessive distribution. Furthermore, sequential code is inspected for SIMD parallelism, which can be found in e.g. unrolled loops. This special vectorization is performed using the SLP approach also employed by the CHARISMA compiler (see Section 8.3). The vectorizer of the CELL compiler handles basic blocks before loops, because vectorized sequential code may be re-used by the loop-level vectorization.

7.3.2. eLite DSP

The eLite DSP [120] was developed by IBM to support efficient vectorization techniques by offering two types of SIMD [124]: Single Instruction Multiple packed Data (SIMpD) refers to conventional subword SIMD operations also provided by MMEs like MMX or AltiVec. This functionality is only useful, if data are aligned properly, contiguous, and not re-used.

Single Instruction Multiple disjoint Data (SIMdD) accesses a large vector-element register file through vector pointers in order to support efficient access to non-contiguous data. Each vector pointer register consists of four elements which contain the indices of the referred vector element registers. Consequently, data re-ordering needed at each permuted access for SIMpD is avoided by modifying the vector pointers once.

A vector pointer is described by the quadruple

$$(v, (\Delta_0, \Delta_1, \Delta_2), \delta, \rho)$$

where v denotes the address of the first vector element register and $\Delta_0, \Delta_1, \Delta_2$ represent the offsets to the next registers. Hence, such vector pointer has the following value:

$$(v_0, v_1, v_2, v_3) = (v, v + \Delta_0, v + \Delta_0 + \Delta_1, v + \Delta_0 + \Delta_1 + \Delta_2).$$

The values δ and ρ specify the post-increment of each v_i:

$$v_i = v_i - (v_i \mod \rho) + ((v_i + \delta) \mod \rho).$$

Thereby, δ represents the distance between two vectors, and ρ denotes the value where a pointer is reset to v. For instance, a vector pointer toggling between two consecutive quadruples of vector element registers starting at address v is represented by $(v, (1, 1, 1), 4, 8)$.

Figure 7.3 shows the memory layout and the corresponding vector pointers for the computation of the complex inner-product. We assume that the data is given as two arrays $a1$ and $a2$ of alternating real and imaginary values. For simplification, the ρ value is ignored.

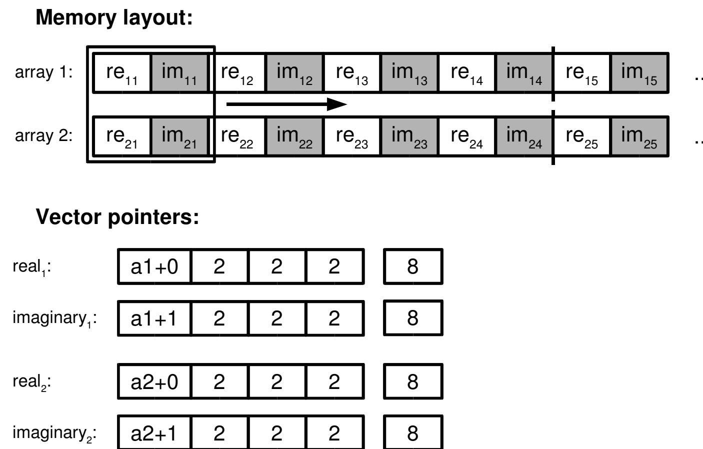


Figure 7.3.: Vector pointers for computation of inner product

The vectorizer of the eLite compiler targets loops, because the focus of the authors was set on optimizing loop kernels. At first, vectorizable loops are identified by memory and data dependence analysis [177]. Furthermore, access patterns and memory alignment of data references are determined to define a set of possible options for data layout. The information can also be re-used for the setup of vector pointers. The actual vectorization decides between SIMpD and SIMdD, assigns operations to the units, and allocates space in the vector register files.

7.4. SIMD/MIMD Reconfiguration

Barretta et al. [8] proposed a multi-clustered VLIW architecture, which can be switched between so-called ILP and SIMD modes. Similar to our CHARISMA approach, the goal is to exploit both regular and non-regular structures in a given program. Currently, there exists

only a simulator, but neither a proper hardware implementation nor a corresponding compiler. Although the approach seems to be rather incomplete, there are no follow-up papers describing any extensions. The suceeding publications of the authors have concentrated on multi-threading of multi-cluster VLIW processors. In the following, we characterize the aproach of Barretta et al. and compare it with CHARISMA.

Reconfigurable Machine The authors envision to reconfigure the machine between ILP and SIMD modes by executing special reconfiguration instructions. They argue that there can be either a single instruction which switches the current mode or two separate instructions to enable a certain mode. Barretta et al. favour the latter solution in order support further execution modes in the future. Our machine features a single reconfiguration instruction where the execution mode to be enabled is encoded as an immediate operand.

In SIMD mode, single instructions are distributed to all clusters of their VLIW architecture where they are executed on different data. It is not explained wether such instructions are still part of a very long instruction word or wether the VLIW decoder can be adapted to handle also scalar instructions. The SIMD mode of the QuadroCore executes a single instruction stream whose instructions are decoded by the first processor and forwarded to the ALUs of all processors. In the MIMD mode, the processors operate on separate instruction streams instead of combining them to a VLIW program.

Each cluster of the architecture by Barretta et al. has a private register file as well as a load/store unit for memory access. Before entering the SIMD mode, the base registers of the clusters are loaded with different addresses to operate on disjoint data. The S-Cores employed by the QuadroCore also have separate register banks and can access the external memory independently. We use common base addresses to avoid the additional effort in preparing the memory access before switching to SIMD mode. Instead, a processor c can access memory data of word size w by adding $c * w$ to a base address during load/store operations (see Section 8.2).

Level of Parallelization Barretta et al. propose a compiler that focuses on coarse-grained regular structures, because ILP techniques do not detect all data parallelism. This minimizes the communication effort between the clusters of their machine. Concretely, they consider two levels of SIMD parallelism: LLP (one iteration of a loop is executed simultaneously by each cluster) and TLP (one instance of a function call is executed simultaneously by each cluster). The relation of SIMD and TLP does not really become clear in the paper.

The SLP approach (see Section 8.3), that is employed by the CHARISMA compiler, targets sequential code in basic blocks instead of performing complex transformations on loop nests. The authors have shown that focusing on SLP leads to simple and robust compiler implementations while still achieving a good performance. In contrast to classical vectorization techniques, SLP can also be exploited when vector parallelism is scarce or loop transformations cannot be applied.

Functionality of Compiler The concepts of the envisioned compiler are presented in a very rough way: At first, it may identify portions of the source code that can be executed in

SIMD mode by determining accesses to disjoint memory blocks. Such information can either be provided in the source code or has to be computed automatically. We believe that both strategies are suboptimal, because the information is too general, even if gained automatically. The applied SLP approach starts by identifying adjacent memory accesses, which can be implemented using a couple of alignment and array analysis techniques. Our compiler uses an adapted version of CSE for this task, which is presented in Section 8.3.1.

As a next step, the desired compiler may allocate data in memory such that the locality is maximized. Finally, a scheduling is performed based on the previously identified portions of code.

CHARISMA features a fully automated compiler, which identifies parts of a given program to be executed in SIMD or MIMD mode. The reconfigurable QuadroCore is available as both real hardware as well as a cycle-accurate software simulator. Furthermore, this special reconfiguration is part of our superior methodology CoBRA.

8. Compilation for SIMD/MIMD Reconfiguration

Contents

8.1 Structure of the Compiler Backend **III-16**
8.2 Functionality of SIMD and MIMD Modes **III-17**
8.2.1 Machine Model in SIMD Mode III-18
8.2.2 Memory Accesses in SIMD Mode III-18
8.2.3 Branches in the Presence of SIMD/MIMD Reconfiguration III-21
8.3 Vectorization with SLP **III-22**
8.3.1 Adjacent Memory Accesses III-24
8.3.2 Preparation III-28
8.3.3 SLP Utilization III-30
8.4 Register Allocation for SIMD and MIMD Code **III-34**
8.4.1 Allocation of Vector Registers III-35
8.4.2 Efficient Placement of Transport Instructions III-37
8.4.3 Register Allocation and Spilling III-39

In this chapter, the concept of our CHARISMA[1] compiler for machines reconfiguring between SIMD and MIMD execution is presented. At first, we give an overview of the structure of the parallelizing compiler backend in Section 8.1. As the CHARISMA compiler utilizes the functionality of both modes, Section 8.2 derives the machine model used in the SIMD mode from the original MIMD model of the QuadroCore. The vectorization phase of the CHARISMA compiler is based on the Superword Level Parallelism (SLP) approach of Larsen et al. [107], which targets sequential code in basic blocks. Section 8.3 introduces the SLP approach and outlines some extensions developed by us. Last but not least, Section 8.4 explains some conceptual changes of the register allocation motivated by the implementation of vector registers.

8.1. Structure of the Compiler Backend

The backend of the CHARISMA compiler has been developed from the parallelizing backend for the QuadroCore (see Section 4.2.3). An overview is illustrated in Figure 8.1. For simplification, some phases like processor partitioning have been omitted. The reader may refer to Section 4.2.3 for further details about the other phases.

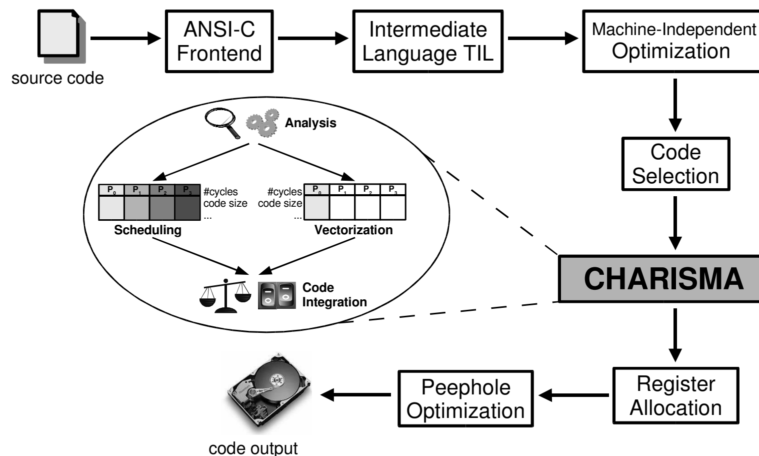


Figure 8.1.: Structure of compiler backend with CHARISMA component

Basically, the scheduling phase has been replaced by a new component called CHARISMA. Its structure is shown in Figure 8.2. At first, an analysis is performed to compute dependences between instructions as well as inherent parallelism. Furthermore, this phase may partition the code into several pieces which are considered separately during code generation. As we decided to schedule on basic block level (see introduction of Part II), our compiler just considers the control-flow graph of a function.

Then, the code of each basic block is parallelized for both MIMD using list scheduling [82] and for SIMD using the SLP approach of Larsen et al. [107], which is described in Section 8.3. Afterwards, the code integration selects the best result for each basic block and inserts reconfiguration instructions at the interfaces (see Section 3.3.2). Our current prototype of CHARISMA aims for a short execution time. In the future, other goals like small

[1] **C**ompiler **H**andles **A**rchitectural **R**econfiguration **I**ntegrating **S**IMD **M**IMD **A**utomatically

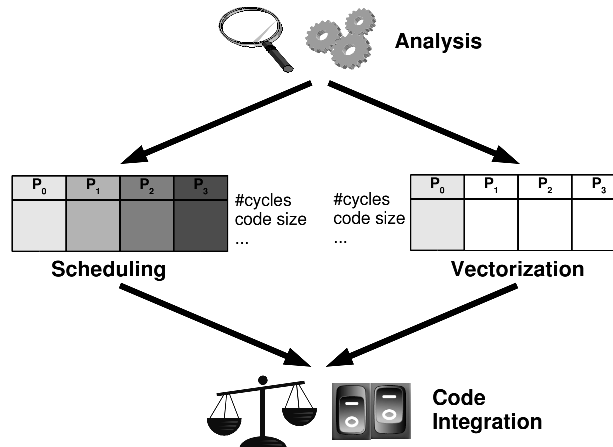


Figure 8.2.: Structure of CHARISMA

code size and low power consumption will be considered also. Importantly, high savings in code and energy are achieved also, if the vectorizer yields a faster result based on a single code stream than the scheduling phase.

8.2. Functionality of SIMD and MIMD Modes

Figure 8.3 gives a rough overview of the machine models utilized by the two modes. The SIMD mode is used for parts of a program with a regular structure, which can be found in multimedia applications, for instance. A single code stream is decoded by the first processor and executed by all processors with different data. During the execution of parts which exhibit an irregular program structure, the processors operate on different code streams in a MIMD manner.

Alternatively, the vectorization can also be utilized to produce multiple code streams for a MIMD mode. In such case, neither reconfiguration instructions nor additional code to arrange register values for the execution of vectorized code (see Section 8.4.1) are required. Hence, this so-called *pseudo* SIMD mode can be regarded as an optimistic estimation of the results gained by using the *real* SIMD mode considered in the following. The pseudo SIMD mode is employed by the evaluation of the SIMD/MIMD reconfiguration presented in Section 13.3.

In order to develop a compiler for a machine which can reconfigure between SIMD and MIMD modes, their functionality must be specified precisely. Section 8.2.1 outlines the machine model for SIMD execution, which has been derived from the default MIMD model. This comprises the implementation of vector registers as well as the semantics of non-memory instructions in SIMD mode. Section 8.2.2 explains the concept of wide memory accesses in SIMD mode and proposes an extension for non-adjacent data. Finally, a technique to handle branches in SIMD mode is proposed in Section 8.2.3.

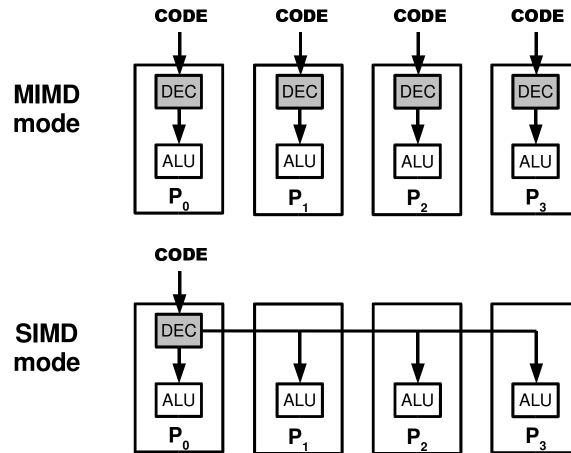


Figure 8.3.: SIMD/MIMD modes of QuadroCore

8.2.1. Machine Model in SIMD Mode

Typical SIMD architectures like famous vector supercomputers (see Section 7.1) or Multimedia Extensions (MME) of general-purpose microprocessors (see Section 7.2) have special vector registers. In the QuadroCore, each S-Core processor has a separate register bank to store scalar values. Clearly, such design is useful for the default MIMD mode (see Figure 8.4).

Let *C* be the number of processors and *R* be the number of registers per processor. Due to the presence of a single code stream, a register operand must refer to a single vector register $r_i, i \in \{0, \ldots, R-1\}$, which consists of an ordered set of C scalar registers of pair-wise different processors. Furthermore, the set of scalar registers represented by two vector registers r_i and $r_j, i \neq j$, must be disjoint. In order to minimize the changes of the existing architecture for the SIMD mode, the j-th entry of r_i is mapped to the register $r_{i,j}$ of processor j, for $i \in \{0, \ldots, R-1\}$ and $j \in \{0, \ldots, C-1\}$, as illustrated in Figure 8.5. In the following, such registers $r_{i,j}$, for a certain i and all j, are denoted as *homonymous* registers. Obviously, this decision demands a proper register allocation to ensure that *C* homonymous scalar registers of the processors are allocated for a single vector register.

Consequently, a single instruction with encoded register operand r_i is executed by all processors j with different values stored in their registers $r_{i,j}$, respectively. Hence, we decided that non-memory instructions should have the same behaviour in both modes to model the SIMD manner best. Instead, it does not make sense to execute a single memory access multiple times on different processors. The adapted semantics of memory instructions is explained in the next subsection.

8.2.2. Memory Accesses in SIMD Mode

Wide Access of Adjacent Elements In the SIMD mode, a processor c accesses memory data of word size w by using $c * w$ as an offset to a base address as illustrated in Figure 8.6. Such semantics corresponds to wide memory accesses known from vector machines (see Section 7.1) or conventional MMEs (see Section 7.2) and exploits the existing data-paths as

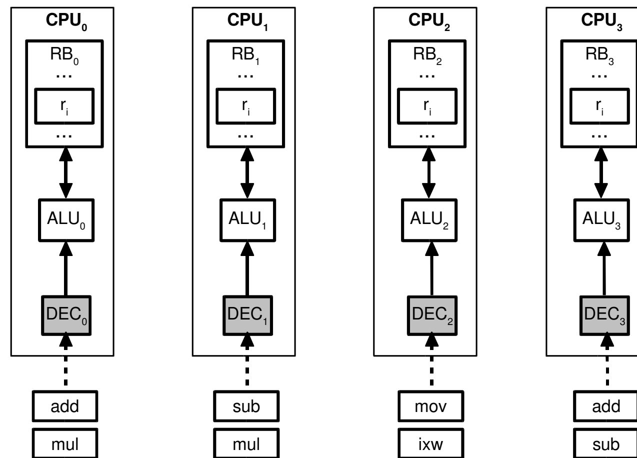


Figure 8.4.: Functionality of original MIMD mode

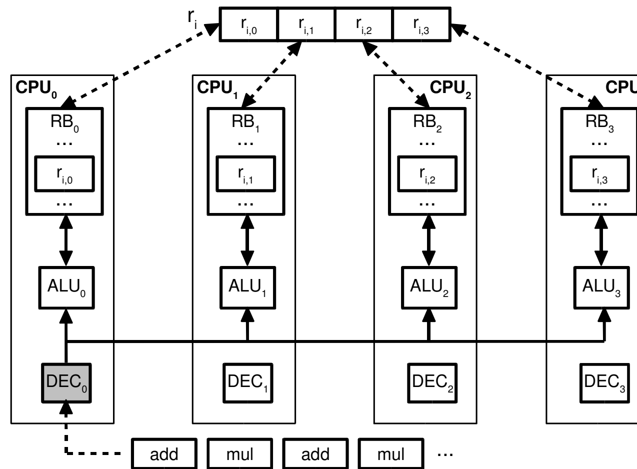


Figure 8.5.: Functionality of SIMD mode

best as possible. Furthermore, it matches the perspective of the SLP approach employed by the CHARISMA compiler (see Section 8.3), which is based on adjacent memory accesses.

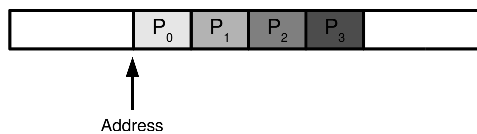


Figure 8.6.: Memory access in SIMD mode

Importantly, this decision requires that memory access and corresponding address computation are performed in the same mode. For instance, the processors may compute the addresses for each memory element from Figure 8.6 separately in MIMD mode. Hence, the memory must also be accessed in MIMD mode in order to avoid the implicit adaption of the addresses as described above. Vice versa, a common address computed by each processor in SIMD mode may only be used for a wide memory access in SIMD mode. In order

to allow that address computation and memory access were executed in different modes, two types of memory instructions would be needed: The first group might behave like the original memory instruction, while the second one corrects the address as mentioned. Our prototypical implementation does not distinguish between two types of accesses.

With respect to the structure of the QuadroCore (see Section 4.1), memory data used in the SIMD mode must be stored in the external memory to enable the access by all processors. The hardware architecture has been modified accordingly to support such wide memory access to adjacent memory elements, instead of serializing it by the bus arbitration. Our compiler transforms local arrays or structures to global memory data if necessary.

Alternatively, data-structures may be partitioned among the local memories of the processors in order to speed up their access. But this behaviour impedes a usage in MIMD mode, which assumes contiguous arrays by default. On the other hand, maintaining two different versions of those data-structures used in both modes is clearly too expensive. Last but not least, data-structures used only in SIMD mode could be stored in the local memories of the processors, while memory data used in both modes would be placed in the external memory. But this demands further interprocedural analysis at compile-time to determine all functions where memory data might be used. Hence, we decided to use the external memory for all data-structures accessed in SIMD mode.

Access of Non-Adjacent Elements In the following, we propose a concept for the implementation of wide memory accesses of non-adjacent elements. The method has been derived from a related approach and its implementation is still pending.

The eLite DSP (see Section 7.3.2) offers an indirect access to vector elements by using so-called vector pointers. One example is the computation of the complex inner product, where real and imaginary values are stored alternatingly in two arrays. Please refer to the mentioned subsection for the notation used in the following.

The concept of vector pointers can be mapped to the machine model in SIMD mode. For simplification, we neglect the ρ parameter in the following, which would allow to compute addresses modulo a certain value. This is realized by resetting a vector pointer to the base address of a vector when a specific address is reached.

Figure 8.7 illustrates the implementation of vector pointers with common offset values Δ. The basic idea is that each processor begins the iteration through the array elements at the address $r_b = \beta + cpu * \Delta$. Hence, no additional register is needed to store the offset Δ afterwards. The initialization may also be executed in SIMD mode, if the processors add their number times the offset value implicitly. In contrast to the concept presented above, no offset needs to be added by each processor before the actual memory access. Incrementing r_b by the distance δ provides access to the next elements.

If the offset values are not equal, the initialization has to be changed as illustrated by Figure 8.8. Again, no further registers are needed. But, the initialization cannot be performed efficiently in SIMD mode, because each processor has to access and sum up the offset values separately.

For a practical implementation, the first proposal seems to be more suited. Although it is simpler, the common offset values should not be too restrictive for most applications.

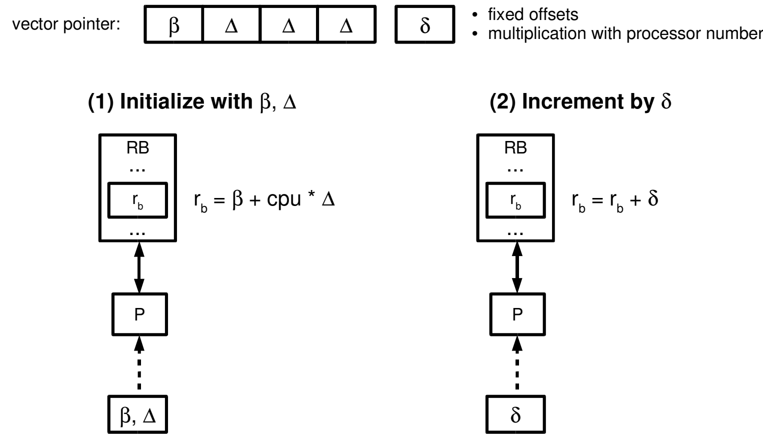


Figure 8.7.: Concept of non-adjacent memory access with fixed offsets

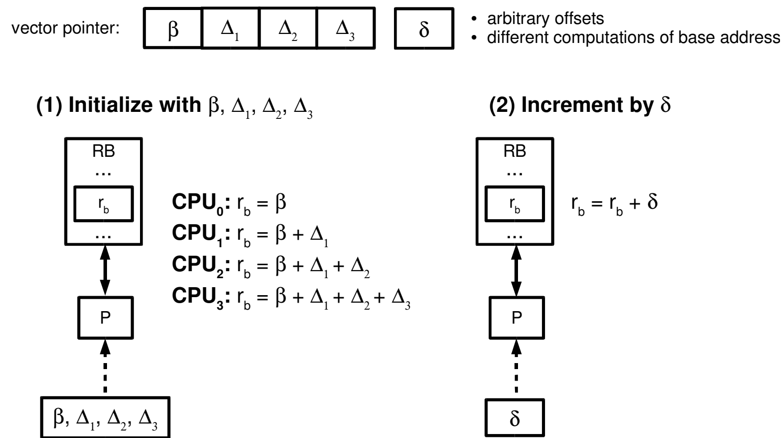


Figure 8.8.: Concept of non-adjacent memory access with arbitrary offsets

However, an implementation would require to augment the architecture by special vector registers, because parallel memory accesses to non-adjacent elements can probably not be realized efficiently in hardware. Like the eLite DSP, contiguous memory data would be read into such vector registers to perform computation and written back to memory afterwards.

Furthermore, the vectorization technique would need to be changed accordingly to recognize regular accesses to non-adjacent memory elements. Our compiler uses the SLP approach, which identifies adjacent memory accesses as a starting basis for vectorization. Further information are provided in the following section.

8.2.3. Branches in the Presence of SIMD/MIMD Reconfiguration

For simplification, branches are always executed in MIMD mode by code resulting from the current prototype of CHARISMA. Hence, all processors can directly proceed executing their own instruction stream after a reconfiguration from SIMD to MIMD mode. In the following, we illustrate the challenge of branches in SIMD mode and outline a technique which may be integrated into a future version of CHARISMA.

Branches in SIMD mode can only be executed by the first processor which fetches and decodes instructions for the other cores. Let us assume a scenario as illustrated by Figure 8.9. In the basic block b, the execution mode is switched from MIMD to SIMD at program position p. The SIMD code stream contains a branch instruction such that execution ends up in a succeeding basic block b', where it encounters a reconfiguration back to MIMD at program position p'. Please note that the program counters of the processors except the first one still point to the prior reconfiguration point p. In order to guarantee a proper execution, the program counters need to be updated with the addresses of the first instructions in MIMD mode after p'. This can be done by inserting branches after the reconfiguration code at p.

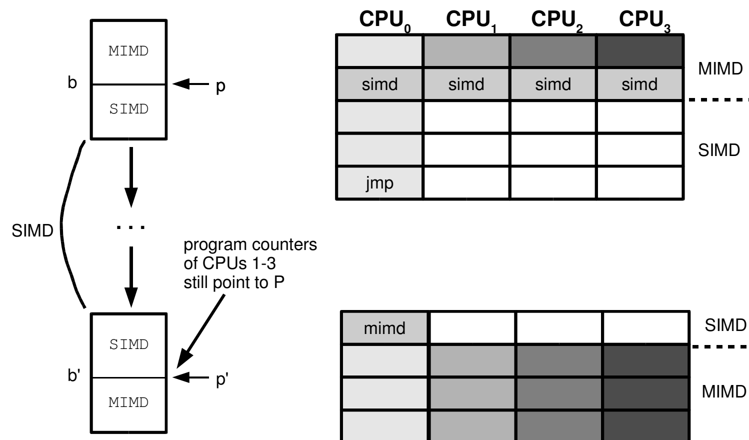


Figure 8.9.: Branch in SIMD code

8.3. Vectorization with SLP

Classical vectorization which was developed originally for vector machines (see Section 7.1) has been re-used in a number of vectorizing compilers for Multimedia Extensions (MME) (see Section 7.2.2). However, complicated loop transformation techniques are required to parallelize loops that are only partially vectorizable [3, 27, 115]. Furthermore, different loop transformations interact with each other and imply a non-deterministic, hardly manageable behaviour. Last but not least, classical vectorization is only capable of identifying SIMD-style parallelism inside loops but not within a single basic block.

Superword Level Parallelism Larsen et al. [107] introduced the concept of Superword Level Parallelism (SLP) as a fundamentally different type of parallelism than the vector parallelism associated with traditional vector supercomputers. SLP is defined as short SIMD parallelism in which the source and result operands of a SIMD operation are packed in a storage location. While vector machines require a large amount of parallelism inside loops to achieve speedups, exploiting SLP is already profitable for a low degree of parallelism. The authors have demonstrated that their method can yield significant speedups over classical vectorization. Focusing on a clear and concise method leads to a simple and robust compiler implementation. Consequently, we decided to re-use their technique for the vectorizer of the CHARISMA compiler.

The basic idea of the SLP approach is to identify so-called *isomorphic* statements within a basic block. Two statements are *isomorphic*, if they contain the same operations in the same order. Statements with adjacent memory references among corresponding operands are well-suited for vectorization, because the operands are effectively pre-packed in memory and address computation is only needed once. Hence, adjacent memory accesses provide first candidates for isomorphic statements. Further SIMD operations are determined by traversing the def-use/use-def chains of the operands.

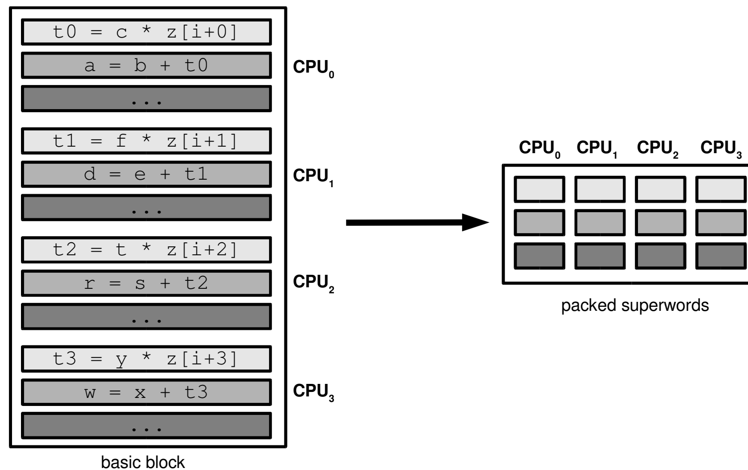


Figure 8.10.: Vectorization based on adjacent memory references

Figure 8.10 shows an abstract example based on statements in 3-address form. The three shades of gray visualize the statements which can be vectorized, i.e. packed to be executed in parallel. As the statements in the light-gray boxes access adjacent elements of the array z, they are remembered as an initial set. Traversing the def-use chains of $t0, \ldots, t3$ leads to the statements in the medium-gray boxes and may find further candidates for SIMD execution, which are not considered here for simplification.

Our compiler transforms the intermediate statements directly into machine instructions using code selection. We define that two instructions are *isomorphic*, if they have the same opcode and address adjacent memory elements (load/store instructions) or have equal immediate operands (non-memory instruction). Register operands can be neglected at this point, because registers are allocated later. The register allocation phase mainly has to ensure that registers with common indices are accessed by each processor to enable the usage of a single instruction in SIMD mode (see Section 8.4).

Vector Parallelism and SLP In order to exploit vector parallelism between loop iterations, loop unrolling is applied to create more SLP. Concretely, vector parallelism is transformed into SLP in the same way that loop unrolling translates loop level parallelism into ILP. Hence, only a single and straightforward loop transformation is needed to exploit parallelism between different loop iterations. Figure 8.11 illustrates by using different shades of gray how loop unrolling produces a basic block with SLP like in Figure 8.10.

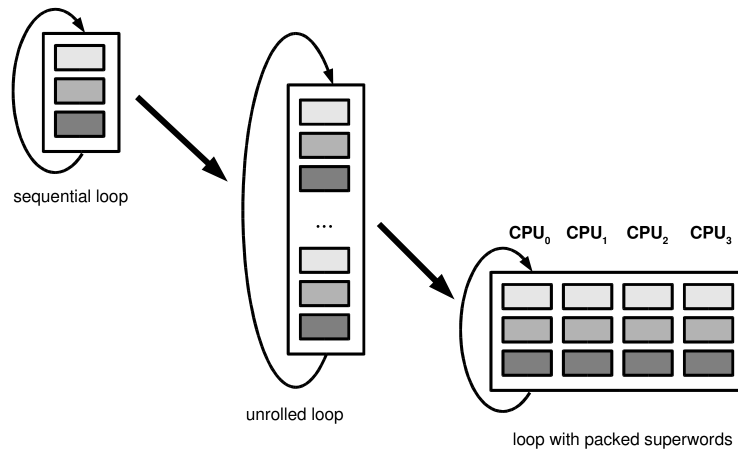


Figure 8.11.: Loop unrolling transforms vector parallelism into SLP

Structure In this section, we explain the SLP approach by using an example and outline some customizations for CHARISMA. The method consists of a preparation and a SLP utilization stage, which include four phases each (see Figure 8.12). Basically, the first stage unrolls loops and identifies adjacent memory accesses, which are used as a starting basis for the second stage which generates code by exploiting SLP. Section 8.3.2 gives an overview of the preparation, while the phases of the vectorization are handled in Section 8.3.3. Beforehand, Section 8.3.1 presents an original analysis of adjacent memory developed by us which was inspired by the data-flow problem Common Subexpression Elimination (CSE).

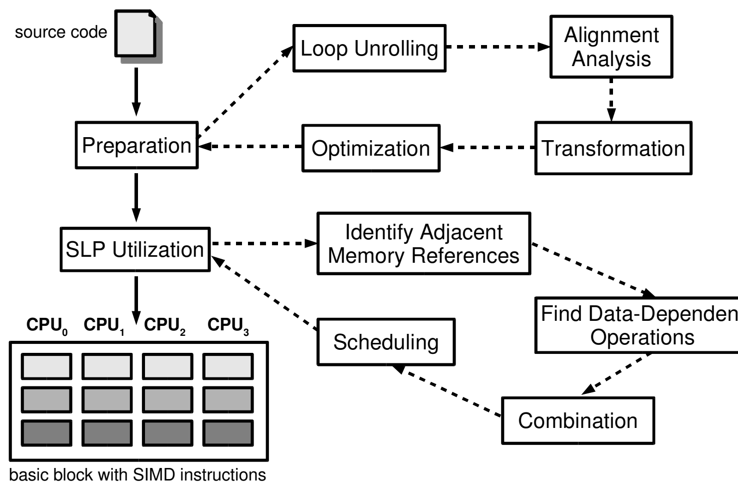


Figure 8.12.: Overview of SLP algorithm

8.3.1. Adjacent Memory Accesses

A mandatory requirement for the SLP approach of Larsen et al. [107] is an analysis of adjacent memory references. According to their paper, they utilize both alignment information [108] and array analysis for computing adjacence. Our approach is based on the idea

of the well-known compiler optimization technique Common Subexpression Elimination (CSE). At first, we introduce CSE and then present our method.

Common Subexpression Elimination Two computations are denoted as *common subexpressions*, if their evaluation gives the same result. With respect to a single tree, such expressions can be recognized by checking subtrees for structural equivalence, if the evaluation of operands does not imply side effects. The identification of common subexpressions in larger contexts such as basic blocks or functions has to ensure that two expressions are based on a common set of values also. In the following, we always refer to single functions. CSE can optimize a given function such that common subexpressions only have to be computed once. Thereby, the result of a computation is assigned to a newly introduced auxiliary variable which replaces the remaining occurrences of the expressions.

For instance, the upper left part of Figure 8.13 shows an excerpt of a program which computes two expressions and assigns them to variables x and y, respectively. As the expression $b + c$ is computed with common values for b and c each, $b + c$ is a common subexpression for the computation of x and y. Hence, the program can be simplified by computing $b + c$ first and assigning its result to a auxiliary variable h, which is used to calculate x and y afterwards.

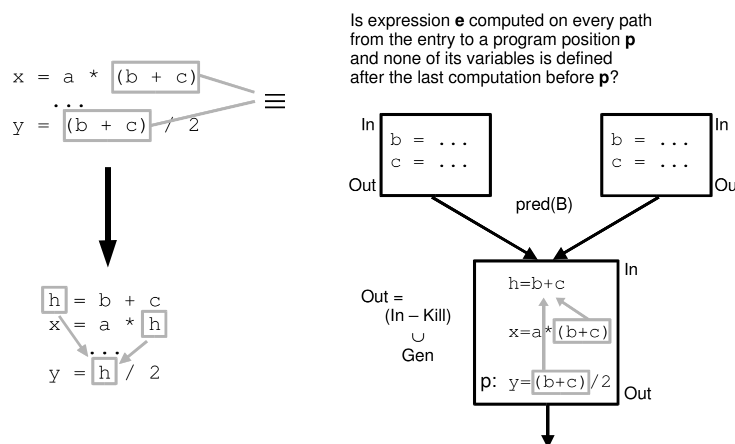


Figure 8.13.: Data-flow problem *Common subexpressions*

CSE can be formulated as a data-flow problem, which determines the *available expressions* for all program positions. An expression e is available at a certain program position s, if e is computed on every path from the entry node to s and none of its variable is defined after the last computation before s. Such information can be used to both identify common subexpressions and determine program positions for assignments to auxiliary variables.

The information about expressions is propagated according to the control-flow, because the data-flow property at a program position s depends on information about expressions computed before s. As available expressions are defined with respect to *all* paths, the join operator corresponds to the intersect function. Hence, CSE is a forward-intersect problem.

Strong and Weak Structural Equivalence Address computations for adjacent memory accesses have a similar structure and only differ in a constant. Adjacence can be shown by considering the effect of such constants on the address computation as well as the width of memory accesses. Hence, we formed the term *weak* structural equivalence, which softens the definition of the known *(strong)* equivalence by ignoring different constant values. We omit a precise definition for simplification, because this would require to specify types of operators used in expression trees and their interaction.

Figure 8.14 gives an example: The left subtrees of each intermediate tree correspond to address computations for the assignments to the elements i and $i + c$ of an array a, respectively. Both computations only differ slightly: In the tree on the left-hand side, the index i is left-shifted 2 bits and added to the address of a. The shifting corresponds to a multiplication by 4, because the array contains 32-bit words. In the right tree, the value $c*4$ is further added to $i * 4$ to get the offset of $i + c$ relatively to a.

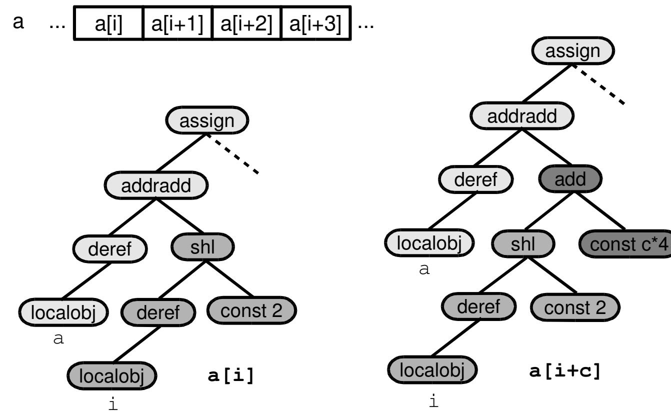


Figure 8.14.: Intermediate trees of address computations for indexed memory access

Obviously, the analysis of *weak* structural equivalence can be realized quite similar to the original definition by neglecting different constants in address computations. After the analysis, each assignment or dereference node is annotated with a triple (b, o, s) which can be used to determine the adjacence later efficiently.

The first element b of the triple denotes an abstract base address and corresponds to the index of a table entry that contains the expression for the address computation. As shown in Figure 8.15, the trees of Figure 8.14 have common base addresses, because the address computations are weakly structural equivalent. The second element o represents the constant offset in the corresponding address computation. Clearly, o is 0 for the first tree $a[i]$ and $c * 4$ for the second tree $a[i + c]$. The size of the memory access in bytes is equal to the third element s.

Computation of Adjacence Given two memory accesses, adjacence can be queried easily using the annotated triples. Obviously, non-adjacence must be assumed if the abstract base addresses are not equal. In case of *common base addresses*, the accessed memory blocks are modelled by intervals $[o, o + s]$, whereas o is the offset and s the size of the access, again. In the following, we always neglect the base address and use the terms *memory block* and *interval* interchangeably.

At first, we focus on *strong* adjacence, i.e. two intervals $[o_1, o_1 + s_1]$, $[o_2, o_2 + s_2]$ need to be tangent to each other, but must not interfere. Hence, it holds either $o_2 = o_1+s_1$ or $o_1 = o_2+s_2$. Figure 8.16 illustrates the computation and gives three examples. If two accesses are strongly adjacent, they are used as a starting basis for the SLP approach (see Section 8.3.3).

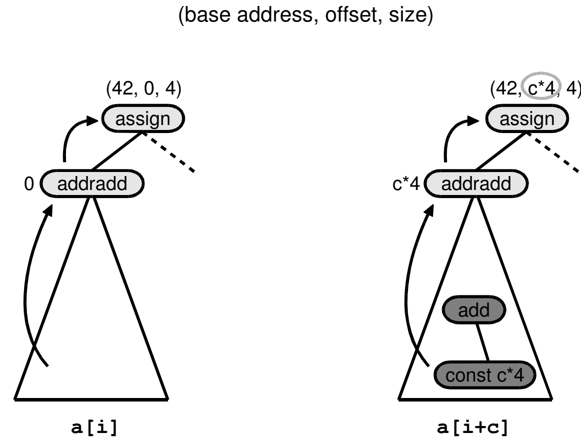


Figure 8.15.: Annotation of intermediate nodes with information about memory access

	Formal	Examples
Pure adjacency	$(b_1, o_1, s_1), (b_2, o_2, s_2)$	(42, 0, 4), (42, 4, 4) yes (42, 0, 4), (42, 8, 4) no (42, 0, 4), (23, 4, 4) no
	$b_1 = b_2$ $o_2 = o_1 + s_1 \lor o_1 = o_2 + s_2$	needed for vectorization
Weak adjacency	$(b_1, o_1, s_1), (b_2, o_2, s_2)$	(42, 0, 4), (42, 4, 4) yes (42, 0, 4), (42, 8, 4) yes (42, 0, 4), (23, 4, 4) no
	$b_1 = b_2$ $o_2 = o_1 + s_1 \lor o_1 = o_2 + s_2$	weakly adjacent memory accesses do NOT depend on each other

Figure 8.16.: Computation of adjacence information from annotations

An interesting alternative is the *weak* adjacence, which only demands that the intervals are disjoint, i.e. the memory accesses are independent. Importantly, the base addresses must still be equal, and strongly structural equivalent memory accesses are always weakly structural equivalent. Information about weakly structural equivalence is exploited by the CHARISMA compiler to optimize the dependence information for scheduling and vectorization.

As the presented method is only a heuristic, adjacence cannot be proven in all cases. Hence, if two memory accesses are *not* weakly or strongly adjacent, independence *must not* be assumed. On the other hand, adjacence is only claimed if it holds, by construction. The presented concept was implemented with a reasonable effort by re-using an existing module for CSE and has demonstrated its effectiveness.

8.3.2. Preparation

Here, we explain the preparation stage of the SLP approach, which consists of four phases as shown in Figure 8.12. The SLP utilization is outlined in the next subsection.

Loop Unrolling At first, the loops of a function are unrolled on the intermediate level. When targeting an MME, the unroll factor must be chosen according to the data sizes used within the loop. For instance, a vectorizable loop containing 32-bit values should be unrolled 4 times for a 128-bit datapath. The prototype implemented by Larsen et al. unrolls loops based on the smallest data type present.

Our compiler unrolls loops by the number of targeted cores, i.e. 4 if all processors of the QuadroCore are used. The exemplary unrolling of a loop with iteration count 11 by factor 4 is illustrated in Figure 8.17. Unrolling yields a new loop body which comprises four iterations of the original loop. The remaining 3 iterations are appended as sequential code after the new loop. This scheme can also be applied if the upper bound of loop count l is not known at compile-time, but at run-time before executing the loop. Then, the new loop is executed $\lfloor l/4 \rfloor * 4$ times and is followed by $l \mod 4$ iterations of the original loop.

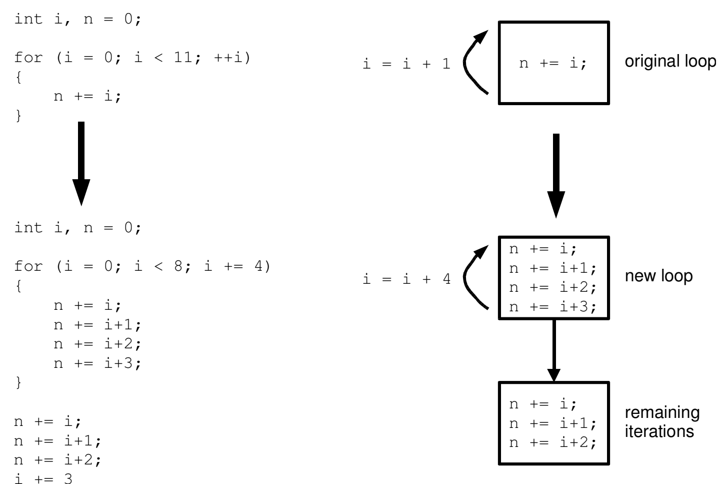


Figure 8.17.: Basic scheme of loop unrolling

Figure 8.18 demonstrates the specific benefits of loop unrolling for the SLP approach: The original loop consists of accesses to three arrays, but does *not* contain SLP. If the loop is unrolled by factor 4, adjacent memory accesses appear, which can be used as a starting basis for the SLP algorithm.

Alignment Analysis After loop unrolling, an alignment analysis attempts to determine the address alignment of load/store instructions. This information is crucial to compile for target architectures that do not support unaligned memory access or where unaligned accesses imply high penalty costs. On the Pentium 4 processor, for instance, unaligned loads even suffer from two penalties, namely the cost of handling the unaligned access and the impact of the cache line splits. If memory accesses must assumed to be unaligned, additional merging code needs to be emitted for each wide load/store operation. Such merging overhead

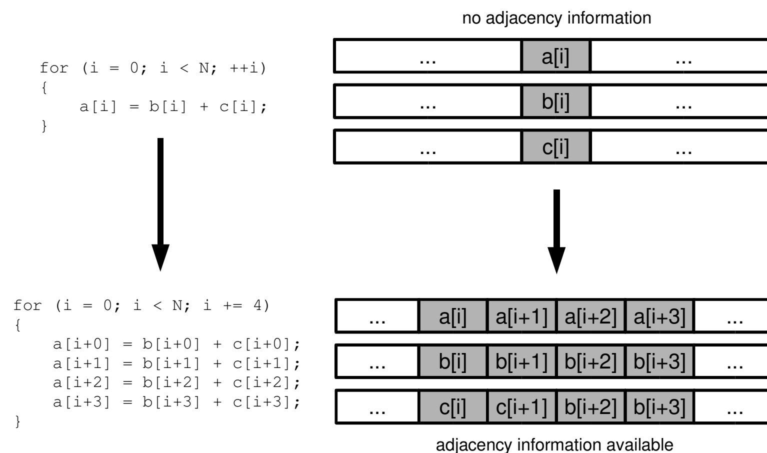


Figure 8.18.: Benefits of loop unrolling for vectorization technique

can be reduced to one additional merge operation per load and store by using data from previous iterations, if a contiguous memory block is accessed in a loop (see Figure 8.19).

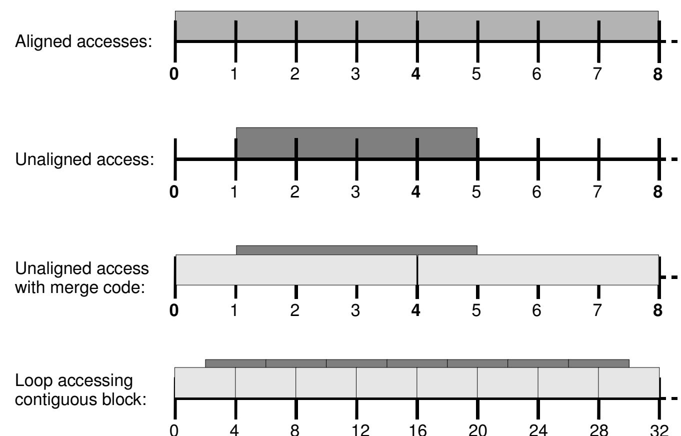


Figure 8.19.: Aligned and unaligned accesses

Alignment analysis can completely remove the overhead introduced by the merging code. Larsen et al. use an enhanced pointer analysis package developed by Rugina and Rinard [143] for C sources. A full description is off-topic and is given in [109]. Their prototype offers compilation both with and without alignment constraints.

Our prototypical implementation handles alignment only in a very simple way, because the focus of our research was the reconfiguration between SIMD and MIMD. Concretely, alignment is ignored during the vectorization. At the end of the compiler run, adjacent memory accesses are replaced by wide accesses which load or store 4 contiguous words. This transformation is only performed if the base address of a wide access is divisible by 4 times the size of one word. In principle, we can re-use the alignment analysis employed by Larsen et al. to influence the vectorization such that the base addresses of memory blocks addressed by adjacent accesses are aligned properly. For simplification, alignment analysis is neglected throughout the rest of this section.

Transformation and Optimization The two latter phases of the preparation stage transform the intermediate representation of a program into a low-level 3-address representation and apply standard compiler optimizations [1, 95]. According to the authors [107], the SLP algorithm has best flexibility when operating on a 3-address form, instead of finding isomorphic intermediate statements. As adjacent memory references can be identified much easier using an intermediate representation of address computations, adjacence information is determined beforehand and annotated to the low-level form. Our compiler transforms the intermediate statements directly into machine instructions using code selection. Optimizations are already applied on intermediate level to improve the representation early.

8.3.3. SLP Utilization

The utilization stage targets single basic blocks and relies on proper loop unrolling as well as identification of adjacent memory accesses by the preparation stage (see previous subsection). For simplification, we assume that the SLP algorithm operates on instructions like our implementation of the CHARISMA compiler. The original method by Larsen et al. is based on a low-level 3-address representation.

SLP utilization comprises four phases, which are illustrated in Figure 8.12. The first two phases determine *pairs* of isomorphic instructions, which are combined to *groups* by the third phase. Below, we discuss the usage of pairs and groups. In the last phase, dependences between the groups resulting from irregular program structures are resolved by splitting groups. Finally, code is generated by mapping groups to SIMD instructions.

Pairs/Groups According to the authors, pairs are used as an intermediate step in order to explore the space of feasible groupings by computing all pairs first and then building groups incrementally. For instance, suppose that there are four operations a, b, c, d and you can either vectorize a, b, c or b, c, d, but not a, b, c, d. Examples prohibiting the larger group may be the vector length of the machine or dependences between the operations.

Each pair consists of a left and a right operation, which must not be dependent on each other. An instruction may belong to two pairs as long as it occupies a left position in the first one and a right position in the other one. Obviously, this allows the efficient combination of pairs to groups.

Assume that loops are unrolled by a factor f (see Section 8.3.2). Then, vectorization may determine groups of size f directly instead of identifying pairs and building larger groups incrementally. In their paper [107], Larsen et al. also discussed an alternative algorithm that skips the intermediate step of computing pairs and just combines operations that originate from the same unrolled statement. Like the approach mentioned above, it enables efficient parallelization of partially vectorizable loops without complicated loop transformations. On the other hand, it may not be applicable to architectures with long vectors, because the unroll factor needs to be consistent with the vector size.

In the following, we present the general SLP algorithm by using the example shown in Figure 8.20. The program on the left-hand side consists of four statements using adjacent elements of the array z. The abstract machine code on the right-hand side is commented with the arithmetic operations and array accesses.

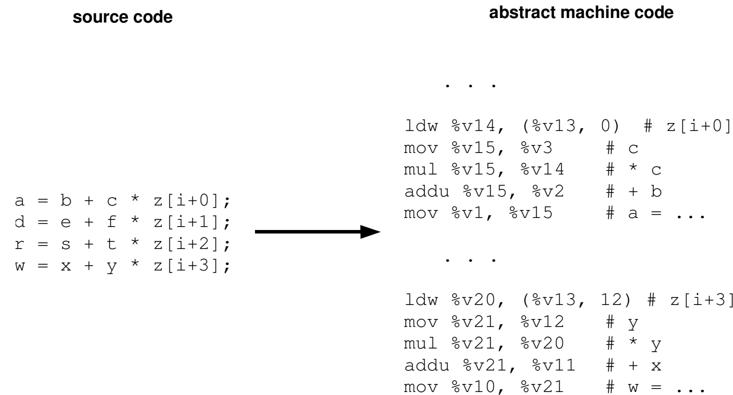


Figure 8.20.: Example of vectorization with SLP

Identifying Adjacent Memory References The initial set of pairs is made up of operations using adjacent memory references, because the operands are effectively aligned in memory and do not require further rearrangement within a register. Additionally, costly address calculations for memory access need only to be executed once and not redundantly for each element. Due to Larsen et al. [107], the latter improvement achieved the greatest speedup during their measurements. Such optimization is quite similar to the replacement of expensive multiplications in address computations by the linear increment of array addresses [95].

The compiler of Larsen et al. determines adjacent memory references by using both alignment information [108] and array analysis. Our adjacence module is based on a customized version of CSE which has been presented in Section 8.3.1. It computes all expressions which are *weakly* structural equivalent. In contrast to *strong* structural equivalence, we allow that address computations differ in constants. Such constants are annotated at the memory nodes of the intermediate representation in order to determine the adjacence afterwards.

With respect to our example, the four load instructions (`ldw`) operate on adjacent elements of the array z and are packed into pairs (see Figure 8.21).

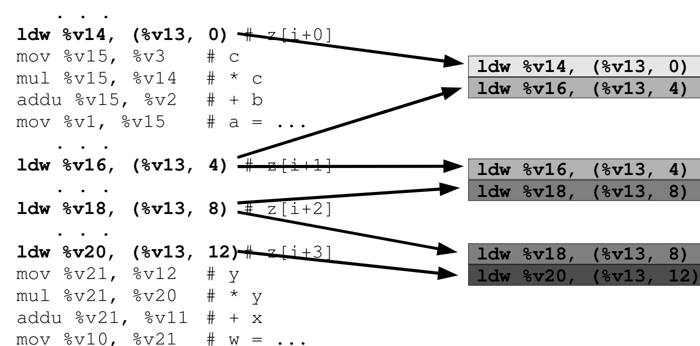


Figure 8.21.: Identification of adjacent memory references and creation of pairs

Identification of Data-Dependent Instructions In order to exploit SLP for non-memory instructions also, the second phase determines further pairs based on the current set of pairs. Given a pair of instructions (l, r), a new pair of instructions (l', r') is identified such that l is

data-dependent on l' or vice versa. A similar requirement must hold for r and r'. This can be done by traversing the def-use/use-def chains of the register operands of l and r until all pairs have been found. Clearly, l' and r' must be isomorphic and independent to each other. Furthermore, l' and r' must not already be packed in a left or right position, respectively.

Additionally, Larsen et al. demand that alignment information are consistent and the estimated execution time of the parallelized instructions is less than the sequential version. As mentioned in Section 8.3.2, we neglect alignment during vectorization. Further information about the cost model of the original SLP approach can be found in [107]. With respect to the CHARISMA compiler, estimating the penalty costs imposed by communication (see Section 6.1) or arrangement of register values (see Section 8.4) is too expensive and imprecise at this point. Even if the vectorization produces a suboptimal output, the code integration can still select a better result computed by the MIMD scheduling or a further variant using only a single processor (see Section 3.1.1).

In the example, traversing the def-use chains of the load instructions (`ldw`) leads to isomorphic multiplication and transport instructions (`mul`, `mov`) amongst others. Such instructions are packed into new pairs (see Figure 8.22).

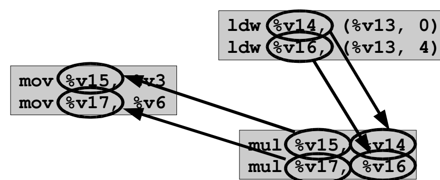


Figure 8.22.: Traversal of def-use/use-def chains to find further pairs

Combination As a third step, pairs are combined to groups which can be regarded as tuples mathematically. The size of the groups must not not exceed the superword datapath size. Our vectorizer is aimed at creating groups with a cardinality equal to 4, which corresponds to the number of targeted processors of the QuadroCore.

Two pairs are combined, if they have a common element, i.e. the left instruction of one pair is also the right instruction of the other one. An important goal of CHARISMA is to yield groups where dependences only occur between instructions with common index between the corresponding groups. Otherwise, additional communication code has to be added to transport register values between the processors. The requirement is ensured by a simple heuristic: The grouping of pairs starts with those pairs (l, r) such that there exist no further pairs (l', r') with $r' = l$. Then, each pair (x_0, x_1) with this property is combined with pairs $(x_1, x_2), (x_2, x_3), \ldots$. In terms of our example, the load instructions (`ldw`) are combined to a group, for instance (see Figure 8.23).

Scheduling The last phase schedules the instructions of a basic block according to their original order with As Soon As Possible (ASAP) strategy. A group is arranged as soon as all of its instructions are arrangeable.

The prior phases of the SLP algorithm ensure that two instructions within a group do not depend on each other. However, it may happen that the combination of pairs implies

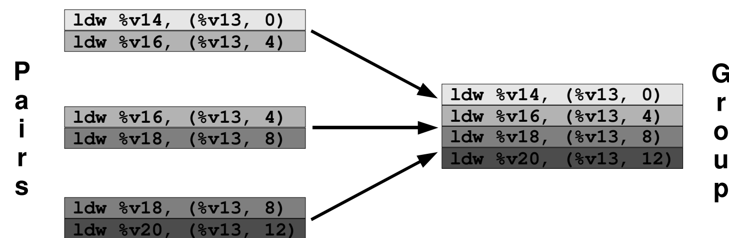


Figure 8.23.: Combination of pairs to groups

cycles between the resulting groups, which impede vectorization. This situation can only be caused by irregular structures in sequential code. Unrolling a loop always yields groups with regular forward dependences.

For instance, the upper left part of Figure 8.24 shows a DDG with two backward edges which may be caused by loop-carried dependences. The graph on the upper right side corresponds to the first graph, where the nodes of each group have been collapsed into single nodes, causing a cycle made up of three groups. Obviously, the cycle can be destroyed by splitting a single group as shown in the lower part of the figure.

In the original SLP approach, such cycles are resolved during scheduling by splitting that group containing the earliest unscheduled instruction. For further explanations, the reader may refer to the original paper [107].

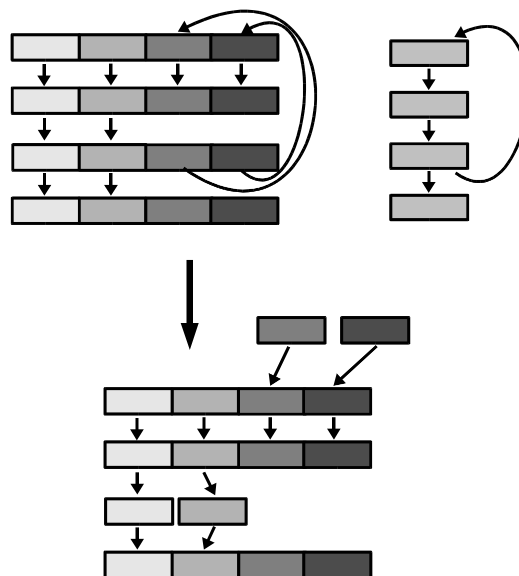


Figure 8.24.: Resolving of inter-group cycles by splitting nodes

Our vectorization destroys cycles before scheduling by using a heuristic which tries to minimize the number of split groups. We re-use a variant of the algorithm by Tarjan [157] for finding strongly connected components, which only needs linear time and is described in [123]. Our greedy strategy favours nodes with a high degree, because it is likely that splitting them destroys the SCC property. As soon as all inter-group cycles have been resolved, a scheduling is performed which is quite similar to list scheduling [82].

Generation of SIMD and MIMD Code In terms of CHARISMA, the splitting of groups can lead to schedules which consist of both SIMD and MIMD code. This is a fundamental difference to the original SLP method, where no reconfiguration is considered and parallelism may be decreased only due to the splitting of groups. In order to maximize contiguous pieces of code using a single execution mode, the scheduling phase of CHARISMA handles MIMD and SIMD operations alternately. Furthermore, it inserts reconfiguration instructions when switching between MIMD and SIMD. Figure 8.25 shows the schedule for the example from Figure 8.24.

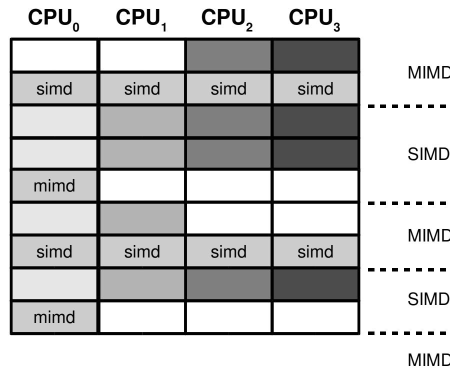


Figure 8.25.: Schedule for example from Figure 8.24

8.4. Register Allocation for SIMD and MIMD Code

Let *C* be the number of processors and *R* be the number of registers per processor. As mentioned in Section 8.2.1, vector registers exist only conceptually in our architecture and are mapped to corresponding *homonymous* registers of all processors. Hence, the vector register r_i consists of the scalar registers $r_{i,j}, i \in \{0, \ldots, R-1\}$ and $\forall j \in \{0, \ldots, C-1\}$ (see Figure 8.26).

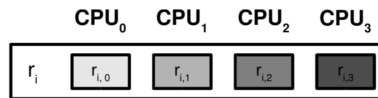


Figure 8.26.: Conceptual vector register

In this section, we outline the register allocation for functions consisting of both SIMD and MIMD code. The MIMD mode exhibits *C* code streams, whereas an instruction of the j-th stream is executed by processor j accessing its physical registers. In SIMD mode, there is a single stream of instructions which use conceptual vector registers r_i. At run-time, such instructions are executed by the *C* processors operating on different data stored in the physical registers $r_{i,j}$.

8.4.1. Allocation of Vector Registers

Let $o_0, \dots o_{C-1}$ be vectorized instructions which are processed in parallel in SIMD mode such that o_j is executed by processor j. By definition, such instructions have a common opcode. Denote the k-th virtual register operand of instruction o_j as $v_{k,j}$. Obviously, all $v_{k,j}$, for all j and a certain k, have to be considered jointly to allocate a vector register. Hence, our compiler maintains C code streams for the SIMD mode until register allocation. Then, a single instruction o is generated for the SIMD code stream, which has the same opcode as $o_0, \dots o_{C-1}$ and combines their semantics. Concretely, each register operand is set to the allocated vector register. The remaining immediate operands must be initialized according to the corresponding operands of o_0.

In detail, homonymous scalar physical registers $r_{i,j}$ need to be allocated for the virtual registers $v_{k,j}$ with respect to a common $i \in \{0, \dots, R-1\}$. This is achieved by creating virtual vector registers v_k, which combine C virtual scalar registers $v_{k,j}, j \in \{0, \dots, C-1\}$ in the context of $o_0, \dots o_{C-1}$. Hence, $v_{k,j}$ corresponds to the j-th entry of v_k. The k-th register operand of o is initialized with v_k. For each v_k, C homonymous physical registers $r_{i,j}$ are allocated for all j and a certain i. Obviously, this corresponds to allocating $r_{i,j}$ for the virtual scalar register $v_{k,j}$. Finally, v_k is replaced by r_i in the instruction o.

The entries $v_{k,j}$ of a virtual vector register v_k can be represented as a C-tuple $(v_{k,0}, \dots, v_{k,C-1})$. Let v_1 and v_2 be two virtual vector registers. v_1 is denoted to be the *permutation* of v_2, if the corresponding tuples are permutations of each other. As each virtual scalar register belongs to a certain processor (see Section 4.2.3), there cannot be two virtual vector registers v_1 and v_2, where v_1 is the permutation of v_2, or vice versa. Hence, vector registers may only vary in single scalar registers as outlined in Figure 8.27 for two vector register v_1 and v_2 differing in $v_{1,1}$ and $v_{2,1}$. There, x_i and y_j denote virtual scalar registers and the $=, \neq$ operators signal (in)equality. Such example may be caused by two groups of adjacent memory operations using the tuples of virtual scalar registers represented by the vector registers v_1 and v_2. Last but not least, two virtual vector registers v_1 and v_2 with a common scalar register in the j-entry are denoted as *overlapping* in the following. Otherwise, v_1 and v_2 are called to be *disjoint*. If there are many tuples of scalar registers, a lot of virtual vector registers will be needed. Hence, the register pressure may increase significantly, if the cut through overlapping life spans of virtual vector registers is large.

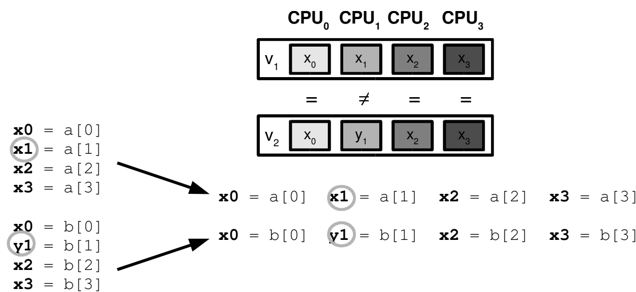


Figure 8.27.: Virtual vector registers

Transport Instructions We refer to the above definition of $o_0, \ldots o_{C-1}$ to be executed in SIMD mode and their virtual scalar register operands $v_{k,j}$. Assume that the virtual vector register v_k depends on definitions of $v_{k,j}$ in MIMD mode and is read by the newly created instruction o in SIMD mode. In order to compute life spans properly, the j-th entry of v_k must be initialized with $v_{k,j}$ for all j. Consequently, C transport instructions t_j are needed which copy the value of $v_{k,j}$ to the j-th entry of v_k on the respective processors. This is achieved by copying $v_{k,j}$ to v_k on processor j, because the access of a vector register v_k on processor j always refers to the j-th entry $v_{k,j}$ (see Section 8.2.1). Let $p_0, \ldots, p_{C-1}$ be the physical registers allocated for $v_{k,0}, \ldots, v_{k,0}$, respectively, such that p_j is a register of processor j. The $p_0, \ldots, p_{C-1}$ may be non-homonymous, because they are defined in MIMD mode. Hence, transport instructions t_j must be executed in MIMD mode also, after the last definitions of $v_{k,j}$ before reconfiguring to SIMD mode.

Vice versa, assume that the $v_{k,j}$ are used in MIMD mode and depend on a prior definition of v_k in SIMD mode. Similarly, transport instructions must be added to copy v_k to $v_{k,j}$ on the respective processors before the first use of $v_{k,j}$ after switching back to MIMD mode.

Importantly, a transport instruction may become superfluous, if a common physical register $r_{i,j}$ is allocated for a virtual register $v_{k,j}$ used in both modes. Besides, the number of transport instructions can be reduced by register coalescing [121], which is explained briefly in Section 9.1.3. Our current implementation of register coalescing can only deal with normal transport instructions between scalar registers, but not with multiple definitions for the same virtual vector register.

Example Figure 8.28 illustrates the fundamental ideas described above using an example. For simplification, we restrict ourselves to a single virtual vector register. Hence, the first index of the virtual registers can be omitted in the picture. Overlapping virtual vector registers are handled in Section 8.4.2. The virtual registers $v_0, \ldots, v_3$ are defined in both upper basic blocks in MIMD mode. The two lower basic blocks are executed in SIMD mode, where the registers are used by four instructions. According to the above description, a new virtual vector register v is created which corresponds to the tuple $(v_0, \ldots, v_3)$. The four instructions are replaced by a single instruction using v, respectively. Hence, transport instructions are inserted before switching to SIMD in the centered basic block, such that the j-th entry of v if initialized with v_j on processor j.

Register Allocation The mentioned requirements can be fulfilled by a two-phase approach: In the first phase, virtual vector registers v_k are determined by considering tuples of virtual scalar registers $v_{k,j}, \forall j \in \{0, \ldots, C-1\}$, where k is the number of a register operand. Then, C parallel instructions $o_0, \ldots o_{C-1}$ in SIMD mode are replaced by a single instruction o using virtual vector registers. Furthermore, transport instructions between scalar and vector registers are inserted at the boundaries between SIMD and MIMD code to arrange register values properly. A method for efficiently placing such transport code is outlined in Section 8.4.2.

During the second phase, the registers are allocated using conventional techniques known from literature. Section 8.4.3 describes the employed approach briefly and then discusses spilling of register values in SIMD mode.

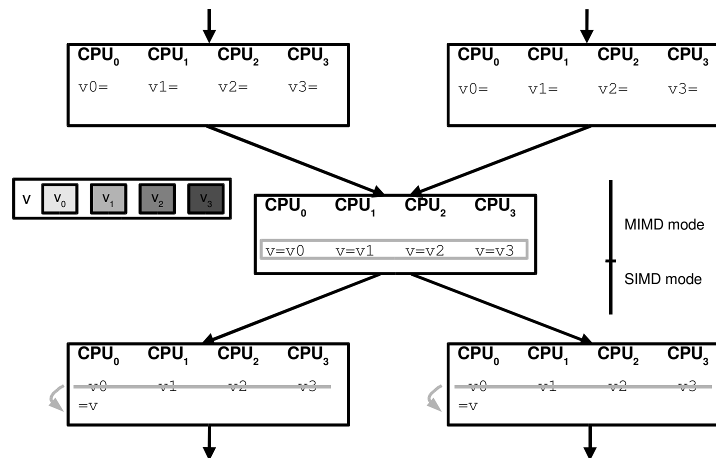


Figure 8.28.: Idea of Placement Algorithm

8.4.2. Efficient Placement of Transport Instructions

In order to minimize the additional effort in arranging register values, transport instructions should be moved out of frequently executed loops. Furthermore, transport code should be placed in basic blocks, where multiple definitions of common vector registers can be handled. In the following, we present a heuristic algorithm to place transport instructions efficiently.

For each use u of a virtual scalar register $v_{k,j}$, a set of basic blocks $\mathcal{B}$ is computed, which occur on *all* paths from the definitions of u to the use u. The computation of $\mathcal{B}$ is discussed in Section 6.1.2. If there is a block $b \in \mathcal{B}$ containing a transport instruction which copies $v_{k,j}$ to the j-th entry of v_k (u in SIMD mode) or vice versa (u in MIMD mode), we are ready. Otherwise, at most two positions in both SIMD and MIMD mode are determined, for all blocks in $\mathcal{B}$. Finally, the best positions are selected based on the estimated execution count of the basic blocks and the used execution mode, with decreasing priority. As transport code must be executed in MIMD mode, additional reconfiguration may be needed, if a selected position is located in SIMD mode. If there are no overlapping virtual vector registers and all definitions for a use are executed in the same mode, it is always sufficient to place transport code at a single position.

Equivalence Classes In the presence of structured control-flow, there may be multiple definitions of a vector register which are executed in different modes (see Figure 8.29). Furthermore, a virtual scalar register, which may be part of a vector register v, can depend on multiple definitions of different overlapping vector registers (see Figure 8.31). The examples are outlined after describing the solution. Both cases can be handled by partitioning the definitions of a virtual scalar register into equivalence classes. Such scalar register is either accessed in MIMD mode or is part of a vector register used in SIMD mode. Two definitions are denoted as *equivalent*, if they define the same virtual register.

As a consequence, the above algorithm has to be modified as follows: Given a use of a virtual register, the equivalence classes of the corresponding definitions are determined. For

each equivalence class, transport code is placed by determining all feasible positions and selecting the best position afterwards, as described above. Transport instructions can only be inserted in the basic block containing the use, if the number of equivalence classes is equal to 1. For all remaining blocks b, the number of equivalence classes which reach the end of b must not be greater than 1. Importantly, this restriction does only affect equivalence classes, but not definitions. In Figure 8.28, there is only a single vector register v and hencefore a single equivalence class for both uses of v in SIMD mode. Hence, transport instructions can be placed in the centered basic block in order to minimize the overhead, although there are two definitions of v.

Now, we consider examples for each of the two situations motivated above. Figure 8.29 deals with definitions of a vector register in different modes. In the left branch, the scalar registers $v_0, \ldots, v_3$ represented by the vector register v are defined in MIMD mode. As the right branch contains a definition of v in SIMD mode, transport instructions may only be placed in the left branch.

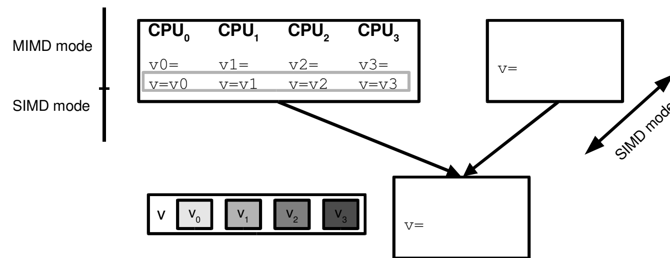


Figure 8.29.: Definition of virtual vector and scalar registers

Handling multiple definitions of overlapping vector registers is outlined in two steps: At first, the placement is explained for multiple overlapping vector registers defined in a single basic block. Such model is used to present a solution for structured control-flow.

In Figure 8.30, we get two overlapping virtual vector registers v_1 and v_2 due to the two upper groups of adjacent load operations. A third vector register v_3 represents the combination of virtual scalar registers used by the adjacent store instructions. Transport instructions must be inserted between the memory loads and stores, because each use of a scalar entry of v_3 depends on a single definition. For each processor j, the j-th entry $v_{3,j}$ of v_3 is set to the j-th entry of either v_1 or v_2 which contains the last definition of $v_{3,j}$.

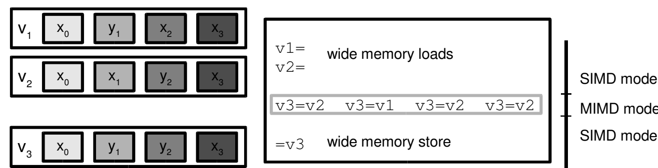


Figure 8.30.: Definition of overlapping virtual vector registers

Figure 8.31 illustrates a situation where transport code needs to be placed at two different positions, because there are two equivalence classes of definitions executed in SIMD mode. In the lower block, the vector register v_5 is used, which depends on a definition in one of the two upper basic blocks. Hence, v_5 must be defined based on either v_1, v_2 or v_3, v_4 in the

corresponding blocks. The equivalence classes for the use of y_2 in the lower block are the definitions of v_2 and v_3, for instance.

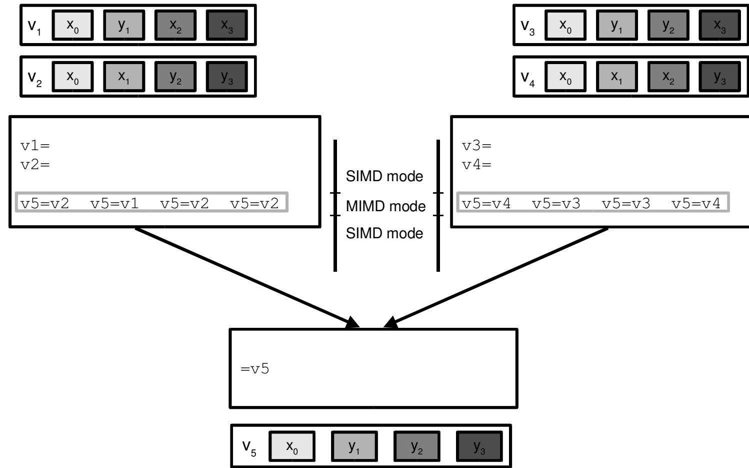


Figure 8.31.: Overlapping registers and structured control-flow

8.4.3. Register Allocation and Spilling

After replacing virtual scalar registers by virtual vector registers and inserting appropriate transport code (see previous subsection), the actual allocation can be performed using conventional techniques known from literature. Our compiler uses global register allocation by graph coloring [32], which is described in Section 9.1.3.

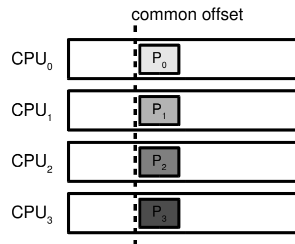


Figure 8.32.: Spilling areas for vector registers

In principle, spilling of a vector register r_i can be reduced to spilling the corresponding scalar registers $r_{i,j}, j \in \{0, \ldots, C-1\}$ on processors j, respectively. For simplification, the scalar registers are stored at a common offset in the stack frame of each processor (see Figure 8.32). Hence, the given address is not incremented by the shifted processor number as done when accessing data structures in external memory (see Section 8.2.2).

Alternatively, the content of a vector register may be stored in a contiguous memory block. As the processors of the QuadroCore have separate runtime stacks (see Section 4.1), vector registers would need to be spilled to adjacent elements of the external memory by a wide access (see Section 8.2.2). But this would require to spill already when only a single free register is available, because at least one register is needed for address computation. Alternatively, a certain register must be reserved for this purpose. When spilling to the runtime

stack of a processor, the stack pointer can be incremented temporarily if the offset within the stack frame is too large to be encoded. Reload code can always use the target register for address computations.

Independent of the solution, scalar and vector registers can only be spilled in the respective mode. If no appropriate program location is available, explicit reconfiguration is inserted to create a position.

Part IV.

Reconfigurable Register Banks

Motivation and Introduction Reconfiguring the connections to physical registers has two fundamental benefits: At first, more physical registers can be exploited than available in the architecture in order to avoid spilling of register values to memory. In case of an architecture containing multiple processors with separate register banks like the QuadroCore, a processor can also use registers of other processors temporarily. Furthermore, communication between processors can be accelerated by using registers through reconfiguration in a shared manner. Such behaviour is beneficial for the QuadroCore in particular, whose processors have no common register bank, but communicate using the comparably slow shared memory (see Section 6.1.1).

In this part, we present our approach for reconfiguring register banks, which was also described in the diploma thesis of Ralf Dreesen [46] and was published in [47]. After the publication, we developed the name CAPRICoRn[2], which is used throughout this thesis. For clarity, a variant of the CoBRA compiler focusing on register reconfiguration is called CAPRICoRn compiler.

Our reconfigurable register architecture has been inspired by the approach of Kiyohara et al. [97]: The physical registers are not accessed directly, but only through architectural registers, which are actually encoded as operands of instructions. The mappings from architectural to physical registers can be modified by executing special reconfiguration instructions. In order to reduce the effort in reconfiguration, we have decided to use a more coarse-grained reconfiguration based on contiguous blocks of registers. Concretely, the architectural and physical registers are partitioned into blocks of a common size. A reconfiguration instruction connects an architectural with a physical block. The size must ideally be a power of 2 and can be configured in the compiler.

CAPRICoRn consists of two phases: At first, physical registers are allocated on function level using register allocation by graph coloring [32]. In addition, CAPRICoRn aims to reduce the number of reconfiguration instructions during the second phase. Therefore, we use a heuristic to assign virtual registers that are often accessed in conjunction to physical registers of the same physical block. The heuristic is based on a static program analysis developed by us, which determines the n pair-wise different values accessed next for each program position. Finally, the second phase replaces physical registers by architectural ones and inserts appropriate reconfiguration instructions.

For simplification, reconfigurable register connections are used for all functions of a compiled program, currently. If CAPRICoRn was only applied for selected functions, a function f using reconfiguration might call a function g without reconfiguration, and vice versa. Hence, the register connections would need to be changed to identical mappings before or during a function call.

Furthermore, reconfigurable register banks are used throughout complete functions, because CAPRICoRn inserts reconfiguration instructions only where necessary. If the register pressure within a function does never exceed the number of physical registers, a reconfiguration is only performed once at the beginning. This effort is clearly negligible compared to a more complex approach using additional intra-procedural analysis and a code integration phase (see Section 3.3.2).

The presented method allows arbitrary mappings between architectural and physical blocks

[2]**C**ompiler **A**nticipated **P**rocessor **R**egister **I**nter-**Co**nnected **R**econfiguratio**n**

(see Section 10.1.3) independent of the number of processors, for simplification. We aim to extend CAPRICoRn by restricted mappings and to distinguish between local and remote accesses to physical registers (see Section 10.2) in the future.

Structure Our explananations are structured as follows: At first, Chapter 9 gives an overview of the work related to CAPRICoRn. We present both conventional methods known from textbooks as well as a couple of architectures based on reconfigurable registers. Chapter 10 presents the reconfigurable register architecture considered by CAPRICoRn as well as a more general architecture, which models existing systems more precisely. This also comprises a definition of important notions as well as a discussion of the fundamental properties and their interaction. Chapter 11 introduces the data-flow problem n-liveness that is used to compute the future register accesses for all program positions. The resulting information is used by the first phase of CAPRICoRn to allocate physical registers such that the effort in reconfiguration is minimized in the second phase (see Chapter 12). We first explain the two phases of CAPRICoRn in detail and then discuss extensions for multi-core architectures like the QuadroCore.

9. Related Work

Contents

9.1 Classical Register Allocation Techniques **IV-6**
9.1.1 Register Allocation for Expression Trees IV-6
9.1.2 Register Allocation for Basic Blocks by Lifetime Analysis IV-7
9.1.3 Register Allocation by Graph Coloring IV-7
9.2 Restricted Form of Register Reconfiguration **IV-9**
9.2.1 Register Renaming IV-9
9.2.2 Register Windowing IV-9
9.3 Reconfigurable Registers **IV-10**
9.3.1 Multiple Register Banks IV-10
9.3.2 Register Connections IV-11
9.3.3 Register Queues IV-12

The work related to CAPRICoRn[1] of reconfigurable register banks can be partitioned into three groups. At first, Section 9.1 deals with classical techniques for register allocation well known from text books. In Section 9.2, two restricted forms of register reconfiguration used in modern general-purpose microprocessors are presented. Finally, Section 9.3 discusses three processor architectures where reconfigurable register banks are used to exploit more physical registers than available in the Instruction Set Architecture (ISA). The concepts of all reconfigurable approaches are compared with CAPRICoRn.

9.1. Classical Register Allocation Techniques

Here, we present three classical approaches for allocating registers in chronological order. Section 9.1.1 explains a technique developed for expression trees, while the approach introduced in Section 9.1.2 focuses on basic blocks. Finally, a global technique based on graph coloring, which is also employed by modern compilers, is outlined in Section 9.1.3.

9.1.1. Register Allocation for Expression Trees

The algorithm by Sethi and Ullman [147] unifies code generation and register allocation for single expression trees, where each intermediate result is exactly used once. The generated code first evaluates one subtree and stores the result in a register. After the computation of the second subtree, both results are combined. In order to minimize the number of needed registers, the subtree which uses the most registers is evaluated first. If not enough registers are available, a spilling will be performed. The approach yields optimal results with respect to a certain set of registers and operations, i.e. with minimal spilling, if every best evaluation code can be arranged to be contiguous.

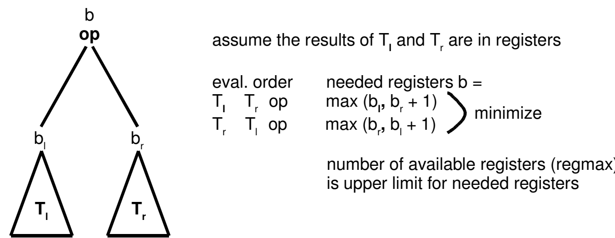


Figure 9.1.: Register allocation for expression trees

The left-hand side of Figure 9.1 shows an expression tree with two subtrees T_l and T_r, which are combined by an operand op. We assume that the results of T_l and T_r are stored in registers. The two feasible evaluation orders are shown on the right-hand side of the picture. If T_l is evaluated before T_r, the computation of T_l needs b_l registers. Then T_r is evaluated using b_r while the result of T_l is stored in an additional register. Obviously, $b = max(b_l, b_r + 1)$ registers are needed in total. A similar observation can be made for the reverse evaluation order. The register allocation will take the order which minimizes the number of registers needed in total.

[1] **C**ompiler **A**nticipated **P**rocessor **R**egister **I**nter-**Co**nnected **R**econfiguratio**n**

9.1.2. Register Allocation for Basic Blocks by Lifetime Analysis

The register allocation by Belady [12] operates on basic blocks and uses the lifetimes of values to minimize the number of needed registers. It consists of two phases: The first phase computes the lifetimes of values from the definition until the last use and constructs an interval graph. Obviously, the maximum cut of the graph corresponds to the number of registers needed for a basic block. Then the registers in the graph are allocated. In case of shortage of registers, a value is selected for spilling. Preferably, a value that already resides in memory is chosen to save a store instruction. Otherwise that value is selected which is latest used again. The latter concept was presented originally as an optimal paging technique. Clearly, it can only work, if the behaviour of a program is known in advance. The second phase allocates the registers in the instructions, while preserving the evaluation order. In contrast to the approach of Sethi and Ullman, a defined register value can be re-used multiple times. On the other hand, register values are communicated via memory at the boundaries of basic blocks.

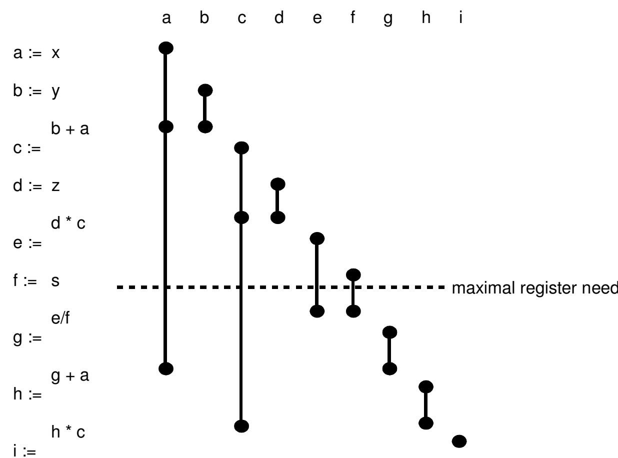


Figure 9.2.: Register allocation for basic blocks with lifetime analysis

Figure 9.2 shows the program code of a basic block with the corresponding registers accesses and lifetimes respresented by filled circles and vertical lines. Importantly, the approach distinguishes between use and definition of register values. The maximal number of registers needed can be determined by the maximal cut (drawn as a horizontal dashed line).

9.1.3. Register Allocation by Graph Coloring

Chaitin [32] presented the first global register allocation technique, which considers a whole function. A major benefit is the fact, that values are also stored in registers beyond basic blocks. Hence, this approach is employed by the most compilers used today. The fundamental idea is to reduce the allocation problem to a graph partitioning problem. The nodes of the graph correspond to the virtual registers of a program. An edge between two nodes will be added, if the life spans of the associated virtual registers overlap. In such a case, the values have to be stored in different registers. The life spans are computed by solving the DFA problem *Live Variables*.

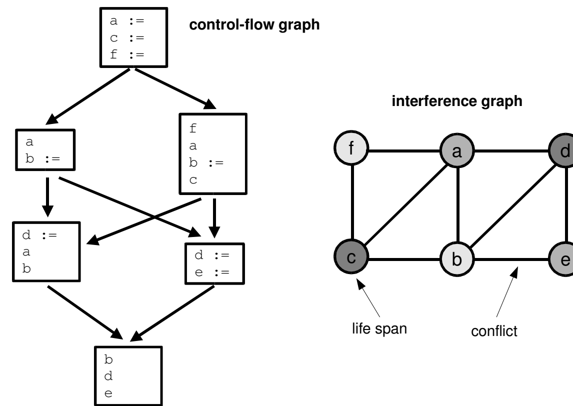
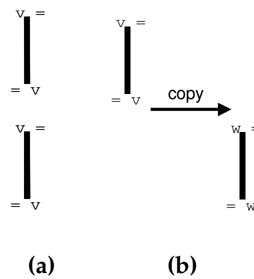


Figure 9.3.: Register allocation by graph coloring

In the following, the graph is called *interference graph* and the edges are denoted as *conflict edges*. The actual register allocation is performed by coloring the interference graph. Unfortunately, coloring a graph with the minimum number of colors is an $\mathcal{NP}$-complete problem. Therefore, practical implementations use a heuristic to color the graph with n colors, whereas n is the number of physical registers. Thereby all nodes with degree $< n$ and incident edges are removed from the graph. If the resulting graph is empty, the original graph is n-colorable by assigning colors to nodes in reverse order of elimination. Otherwise, a virtual register is selected for spilling and the corresponding node is removed from the graph. Then, the heuristic is repeated until a valid register allocation is found.



(a) **(b)**

Figure 9.4.: (a) Partitioning of a life span (b) Example of Coalescing

Briggs et al. [24] improved the approach of Chaitin in several directions: At first, registers to be spilled are not yet determined when removing nodes from the graph. Instead, a register is only spilled, if it cannot be colored, i.e. the neighbours are colored using all n colors. Secondly, a virtual register can be represented by multiple disjoint life spans, if a virtual register is re-defined (see Figure 9.4 (a)). As a consequence, the number of conflict edges can be reduced dramatically. The third improvement called *coalescing* can eliminate redundant transport instructions. If two disjoint life spans for virtual registers v and w are linked by a transport operation (see Figure 9.4 (b)), the affected life spans can be merged. By these means, the same physical register is used for both life spans and the transport instruction becomes superfluous.

9.2. Restricted Form of Register Reconfiguration

In this section, we introduce register renaming (see Section 9.2.1) and register windowing (see Section 9.2.2) as two restricted forms of register reconfiguration, which are used in common general-purpose microprocessors.

9.2.1. Register Renaming

Register renaming [161] is a technique used in RISC processors to avoid unnecessary serialization caused by anti-dependences. For instance, if the value of a register x should be written to memory, it must not be overwritten by a succeeding instruction, until the store has been completed. A naive solution is to use in-order execution where instructions are processed according to their order in a machine program. Such behaviour is suboptimal for superscalar processors trying to speed up program execution by dynamically parallelizing a sequential instruction stream.

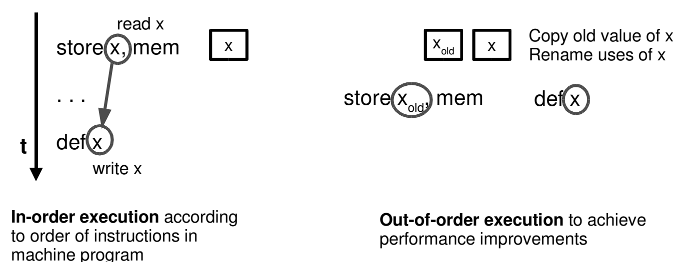


Figure 9.5.: Register renaming

Register renaming avoids this restriction by using different physical registers for one architectural register. In our example, the register name x can be bound to a physical register p_u, while the succeeding instruction writes the new value to $p_d \neq p_u$. Importantly, the physical registers are not visible outside a processor providing register renaming, while the architectural registers are defined in its ISA and used by a compiler. In modern general-purpose microprocessors, register renaming is implemented by using either a tag-indexed register file or reservation stations. The first implementation is used by the P6 microarchitecture of Intel, while the latter concept is employed by PowerPC processors, for instance.

In contrast to CAPRICoRn, register renaming does not increase the number of registers utilizable by a compiler. Instead, the life times of register values are shortened to map an architectural register to different physical registers transparently during execution.

9.2.2. Register Windowing

The comparatively high costs for function calls in register-oriented architectures have led to the development of multiple-window register files [130]. The idea is to make different sets of registers available using a sliding window as illustrated in Figure 9.6 for the Berkeley RISC architecture [129]. On the other hand, a large set of registers is needed and the circuitry

becomes more complicated. Such disadvantages were reduced by the use of multi-size windows [96, 83, 60] or shift registers [84].

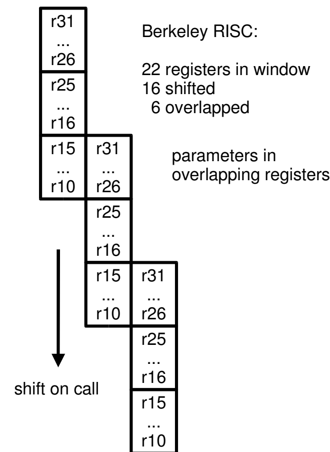


Figure 9.6.: Register windowing

Register windowing marks a special case of CAPRICoRn which might be restricted by mapping only adjacent physical blocks into architectural blocks. Hence, multiple window shift operations would be needed to activate a certain physical block. As a consequence, register windowing is not recommended for programs with interspersed accesses to many registers, but only for repeated accesses to a certain set of registers like parameters.

9.3. Reconfigurable Registers

We have found three approaches in the literature, which also employ reconfigurable register banks to exploit additional physical registers. The first two approaches enable the usage of additional physical registers by reconfiguration with contrasting granularities. While Ravindran et al. (see Section 9.3.1) only switch between register banks, Kiyohara et al. (see Section 9.3.2) offer a very fine-grained mapping per register. Smelyanskiy et al. (see Section 9.3.3) extended the latter approach by register queues to improve software-pipelining of loops. In all cases, the work concentrates on single processors and therefore ignores the communication aspect completely.

CAPRICoRn unifies the advantages of reconfiguring whole register banks or single registers by partitioning both architectural and physical registers into blocks of a common size. As a benefit, the effort in reconfiguration is reduced a lot compared to Kiyohara et al. and also avoids costly moves between register banks needed by Ravindran et al.

9.3.1. Multiple Register Banks

Ravindran et al. [139, 146] presented a single processor architecture with two register banks that are used in an exclusive manner. Each register bank consists of n homogeneous registers which can be addressed via n architectural registers. Two special instructions are provided to activate a certain bank or move data between the banks.

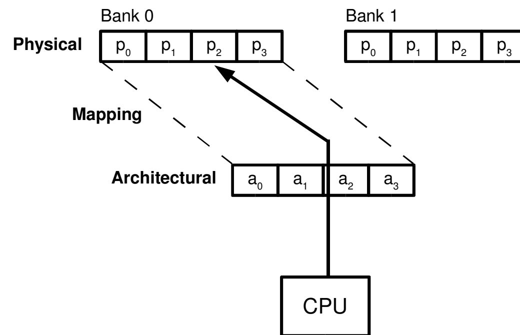


Figure 9.7.: Register architecture by Ravindran et al. [139, 146]

Their register allocation tries to minimize the number of bank swaps and inter-bank moves. It consists of two phases: Firstly, an affinity graph is constructed where the nodes correspond to the virtual registers and the edge weights to affinities. The affinities model the costs resulting from the assignment of two registers to different banks, i.e. the weighted number of bank swaps and inter-bank moves. Obviously, a graph partitioning yields the assignment of the virtual registers to the register banks. The actual register allocation is done separately for each register bank by graph coloring [32]. Finally, the bank swap and inter-bank move instructions are inserted.

Although the authors do not consider an extension of their approach to more than two register banks, this work should be straightforward due to the utilization of the affinity graph. Hence, it should merely require the generalization of the two reconfiguration instructions.

9.3.2. Register Connections

The approach of Kiyohara et al. [97] suggests a mapping of architectural registers to physical registers for *each* register. All accesses to architectural registers are forwarded to the physical registers by considering a look-up table which also differs between read/write accesses. The mappings can be modified by a special instruction.

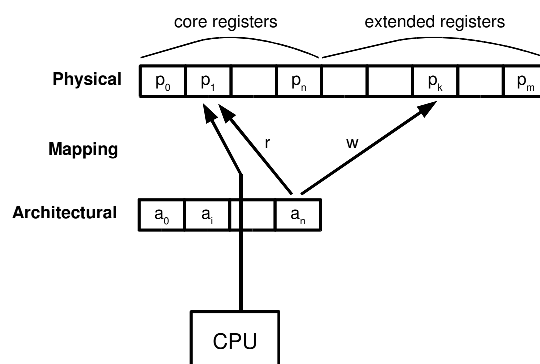


Figure 9.8.: Register architecture by Kiyohara et al. [97]

In order to reduce the number of reconfiguration instructions, two mappings can be combined into a single instruction. Furthermore, mappings can be established implicitly after a

register write. Kiyohara et al. present four different models which differ in the modification of the read and/or write mappings.

The register allocation is based on the approach of Chaitin [32]. Frequently used virtual registers are allocated to the first n physical registers to minimize the effort in reconfiguration at run-time, where n is the number of architectural registers. The approach is backward-compatible to code for the non-reconfigurable architecture with n physical registers, if the default identical mapping is always used.

The reconfiguration instructions can be implemented with zero-cycle execution latency, because no actual data movement is performed. The physical registers of instructions issued in the same cycle can be modified on-the-fly by an additional forward logic.

9.3.3. Register Queues

Smelyanskiy et al. [151] extended the previous approach by the concept of rotating register files [10, 137] in order to exploit more physical registers for software-pipelined loops. Software-pipelining [104, 2] interleaves multiple iterations of a loop to speed up its execution. On the other hand, such restructuring increases the lifetime of variables as well as the number of simultaneously live instances of a variable.

As a consequence, each instance needs to be stored in its own register and must be identified uniquely to link a use with the correct definition. Modulo Variable Expansion (MVE) is a software-only approach which assigns each variable a unique name and unrolls the loop body accordingly. It is not considered in the following, because it increases both register pressure and code size.

Rotating Register File (RRF) performs the register renaming transparently in hardware and was employed in the famous VLIW machines. The basic idea is to access the instance of a variable by its name as well as the loop iteration count. Hence, RRF does not rely on code duplication, but still requires a large number of architectural registers.

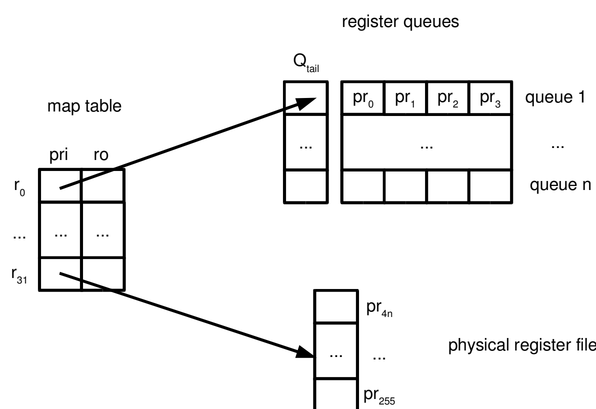


Figure 9.9.: Register architecture by Smelyanskiy et al. [151]

The approach of Smelyanskiy et al. uses a hardware-managed register renaming scheme similar to RRF and avoids the high pressure on architectural registers by the register connection technique of Kiyohara et al. The fundamental idea is to store variables with multiple

live instances in a queue and the remaining variables in conventional registers. Figure 9.9 shows the register architecture consisting of three parts: Each of the n queues has a Q_{tail} pointer to identify the current end of the queue as well as a set of registers. The remaining physical registers belong to one register file. An architected register map table associates architectural registers with physical registers and register queues. Each entry consists of a tuple identifying a physical register or a queue as well as an offset into a queue. The offset is invalid if a single physical register is targeted.

The approach is beneficial for software-pipelined loops, because it minimizes the register pressure caused by overlapping multiple loop iterations and avoids unnecessary code duplication.

10. Register Architecture

Contents

10.1 Selected Register Architecture **IV-16**
10.1.1 Terminology IV-16
10.1.2 Examples IV-17
10.1.3 Elements of Register Architecture IV-18
10.2 General Register Architecture **IV-20**
10.2.1 Permitted Mappings IV-22
10.2.2 Discrimination between Read/Write Accesses IV-23
10.2.3 Operands in Multiple Registers IV-24

The fundamental idea of CAPRICoRn[1] is to access physical registers only indirectly through architectural registers. In order to minimize the effort in reconfiguration, both physical and architectural registers are partitioned into blocks of a common size. The connections between architectural and physical blocks can be changed by a special reconfiguration instruction.

Structure This chapter consists of two basic parts. At first, Section 10.1 introduces the reconfigurable register architecture assumed by CAPRICoRn. Furthermore, we define important terms used in the following and outline examples for both single and multi-processor systems. Section 10.2 discusses optional properties of a more general register architecture, which may be supported in a future version of CAPRICoRn.

10.1. Selected Register Architecture

This section introduces our reconfigurable register architecture incrementally. At first, a terminology is presented in Section 10.1.1, which will be used frequently in the following. Section 10.1.2 illustrates the functionality of a reconfigurable register architecture using some examples. The register architecture assumed by CAPRICoRn is defined precisely in Section 10.1.3.

10.1.1. Terminology

The notions are defined by using the excerpt of a reconfigurable register architecture shown in Figure 10.1.

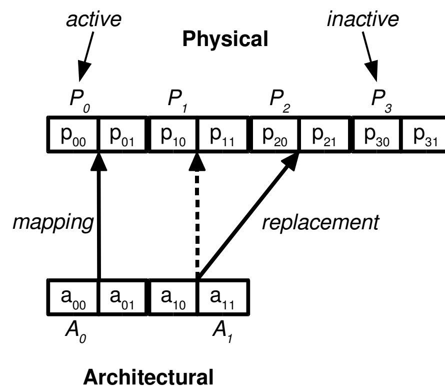


Figure 10.1.: Example of terminology of reconfigurable register architecture

If there *exists a mapping* between an architectural block A_i and a physical block P_j, the k-th architectural register a_{ik} of A_i will be *mapped* to the k-th physical register p_{jk} of P_j. We also say that A_i is *connected* with P_j. Similar conventions can be made for a_{ik} and p_{jk}.

A reconfiguration instruction *establishes a mapping* between an architectural block A and a physical block P. By these means, P is *enabled*, *activated*, or *made accessible*, while another physical block P' which was connected with A beforehand is *disabled* or *deactivated*.

[1] **C**ompiler **A**nticipated **P**rocessor **R**egister **I**nter-**Co**nnected **R**econfiguratio**n**

A physical block P is called *active* if there exists a mapping from an architectural block A to P. Otherwise, P is denoted to be *inactive* or *unmapped*. The same terms can be defined in a similar way for architectural blocks.

Let there be a mapping between an architectural block A and a physical block P. If another physical block P' is connected with A, we say that P is *replaced* by P'. But this terminology cannot be re-used in the reversed case, because different architectural blocks may be connected to a physical block.

For simplification, we introduce a short notation to describe the mappings between architectural and physical blocks:

Definition 10.1 (Mappings between architectural and physical blocks)
Let A be an architectural block and P a physical block. The predicate $\mu(A, P)$ holds if and only if there exists a mapping between A and P. Let $\mathcal{M}$ be a set of mappings. Then, $\mu(A, P)$ holds, if and only if $(A, P) \in \mathcal{M}$.

If $\mu(A, P)$, then $\mu_A(A) = P$ and $A \in \mu_P(P)$. Otherwise $\mu_A(A) = P'$ for a different physical block P' or $\mu_A(A) = \varnothing$, and $A' \in \mu_P(P)$ for a different architectural block A' or $\mu_P(P) = \varnothing$.

10.1.2. Examples

Here, we outline some exemplary instances of the register architecture for both single and multiple processors.

Figure 10.2 (a) shows a conventional architecture that addresses the physical registers $p_0, \ldots, p_3$ directly. The exemplary instruction adds the values of the physical register p_0 and p_3, and stores the result in p_0 afterwards.

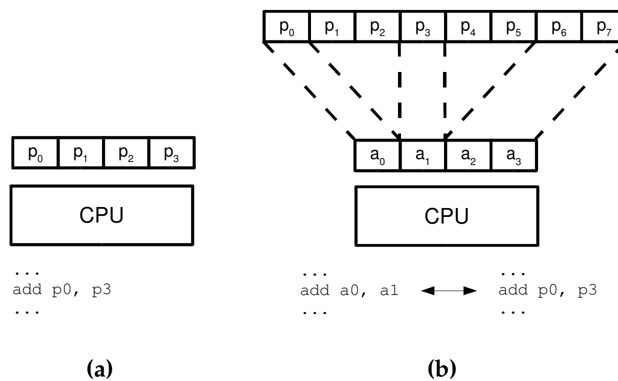


Figure 10.2.: Single processor architectures with conventional register bank (a) and a reconfigurable register bank (b)

Figure 10.2 (b) illustrates an instance of a reconfigurable register architecture that accesses the physical registers $p_0, \ldots, p_7$ via the architectural registers $a_0, \ldots, a_3$. The shown instruction writes the sum of the architectural registers a_0 and a_1 back to a_0 again. According to the mappings, the semantics is equivalent to the non-reconfigurable machine.

For completeness, an architecture with two processors is shown in Figure 10.3. Importantly, both processors share the physical registers p_2 and p_3 of the physical block P_1. By these means an efficient communication can be realized: CPU_1 stores the result of the addition in its architectural register a_0, which is connected to the physical register p_2. CPU_0 uses this result through its own architectural register a_0.

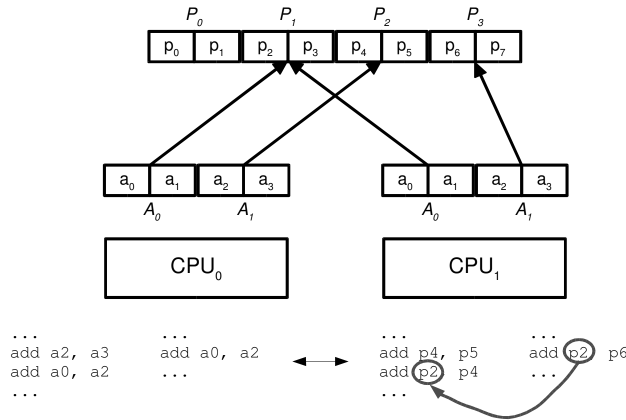


Figure 10.3.: Multi-Core architecture with reconfigurable register bank

10.1.3. Elements of Register Architecture

Figure 10.4 shows an instance of our reconfigurable register architecture. Each of the two processors owns m architectural registers to address the *global* set of n physical registers. Typically, $m = 2^x$ and $n = 2^y$ for $0 \leq x \leq y$. In contrast to Section 10.1.1, the registers are consecutively numbered. This scheme is also used in the following.

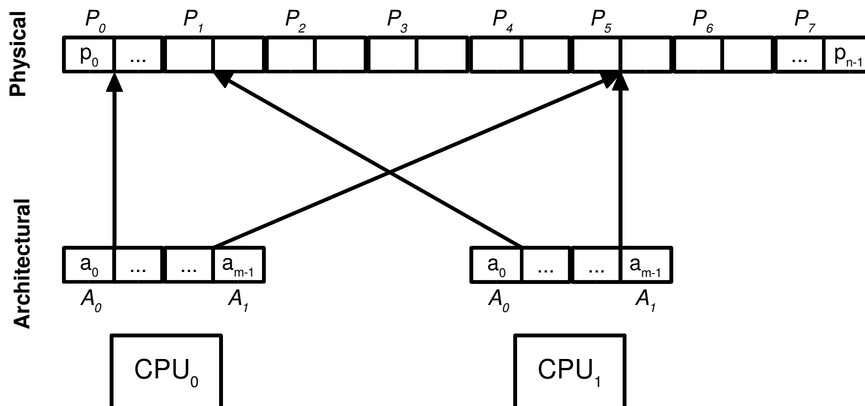


Figure 10.4.: Our reconfigurable register architecture with *one* of the feasible mappings

Both architectural and physical registers are partitioned into blocks of a common size, which must also be a power of 2. In our example, a processor has two architectural blocks, which can be mapped to four different physical blocks. The size of register blocks is neglected in this example, but discussed in the following. The arrows in the figure model the

block-wise mappings from architectural to physical registers in a feasible situation.

Concretely, instances of our register architecture consist of the following elements:

Architectural registers $\mathcal{R}_A = \{a_0, \ldots, a_{m-1}\}$ Architectural registers are representants for the associated physical registers. They do not provide memory space, but mappings.

Physical registers $\mathcal{R}_P = \{p_0, \ldots, p_{n-1}\}$ Physical registers are used to store values and are accessed indirectly via an architectural register. A physical register can be read by an arbitrary number of processors simultaneously in the first semi-cycle, while exactly one write operation is allowed in the second semi-cycle.

Physical register blocks $\mathcal{B}_P = \{P_0, \ldots, P_7\}$ The physical registers are partitioned into disjoint physical blocks of a fixed size. Each physical block can be accessed through multiple architectural blocks by different processors to use its registers in a shared manner.

Architectural register blocks $\mathcal{B}_A = \{A_0, A_1\}$ In analogy to the physical blocks, this set includes the architectural blocks of a processor. An architectural block can only be mapped to a single physical block or may be unmapped.

Processors $\mathit{CPU} = \{\mathrm{CPU}_0, \mathrm{CPU}_1\}$ The processors are uniform, i.e. have same capabilities in terms of instruction set, frequency, etc. Every processor can access each physical block via reconfiguration. The number of processors can be arbitrary.

Number of Registers per Block Existing systems based on reconfigurable registers either switch between entire register banks (see Section 9.3.1) or use a fine-grained mapping per register (see Section 9.3.2). In CAPRICoRn, registers are partitioned into blocks, whose size can be chosen statically. The influence on the performance of the register architecture is discussed in Section 13.4.

If the number of registers per block is small, very flexible mappings are enabled, but many reconfiguration instructions may be needed. On the other hand, many registers per block allow to enable multiple registers with a single reconfiguration instruction. But such strategy may require further instructions to move register values between blocks as motivated below.

Obviously, the number of blocks is closely related to the number of registers per block. If there is only one register per block, the number of blocks will be maximal. On the other hand, the maximal number of registers per block also implies a minimal number of blocks.

If the number of architectural blocks is smaller than the maximum number of operands of an instruction, there are situations where not all operands can be accessed. This may require to copy register values between physical blocks. Figure 10.5 shows an example where the three operands x, y, and z cannot be accessed directly, because only two architectural blocks are available. If there is only one architectural block (see Figure 10.6), the physical blocks can only be accessed alternately. Hence, the inter-bank copying of values may even not be possible. CAPRICoRn demands that each processor has n architectural blocks in case of a n-address arithmetics in order to avoid such problems.

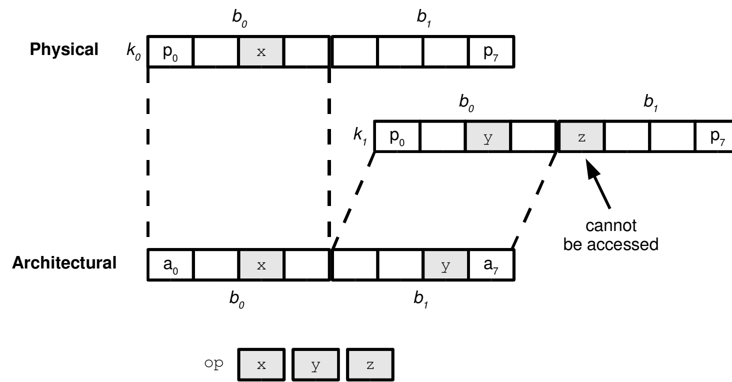


Figure 10.5.: Mapping problem in case of insufficient number of architectural blocks

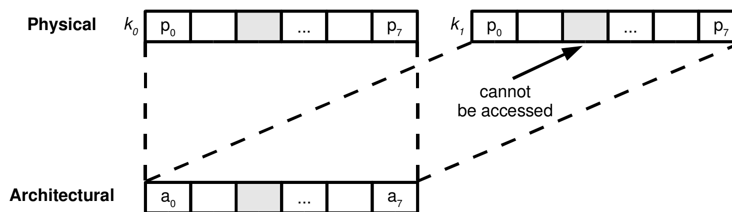


Figure 10.6.: Mapping problem in case of only one architectural block

10.2. General Register Architecture

The register architecture used by CAPRICoRn (see Section 10.1.3) neglects some properties of real existing machines. For instance, physical registers are usually partitioned into register banks, which are connected to a subset of the available processors, respectively. In the QuadroCore, each S-Core has two register banks. Without reconfiguration, each processor can only access its own registers. Instead, CAPRICoRn assumes that all physical registers are organized in one global register file connected to all processors. But such model is only precise with respect to the QuadroCore, if local and remote registers can be accessed at the same costs through reconfiguration. Currently, we have no performance data, because a hardware implementation is pending.

A more precise approach may distinguish between different latencies of local and remote accesses to registers. The compiler can first allocate the local registers for the instructions of a certain processor. If the register need is larger than the number of local registers available, registers of other processors can be used to avoid spilling. As the execution time of instructions may vary depending on the accessed registers, a re-scheduling is needed after register allocation in order to arrange operations properly.

CAPRICoRn assumes that each register can be read by all processors simultaneously, but only written once per cycle. In practice, the number of ports of a register bank limits the number of simultaneous register accesses. This constraint has to be ensured by a re-scheduling in addition to the properties mentioned above.

The mapping between architectural and physical blocks may be restricted such that only certain physical blocks can be connected to an architectural block. Furthermore, the map-

ping can distinguish between read and write accesses in order to double the number of mappings. As a drawback, the hardware implementation becomes more complicated and further reconfiguration instructions are needed.

For completeness, Figure 10.7 shows a general register architecture with the mentioned additional properties compared to the register architecture of CAPRICoRn (see Section 10.1.3). In the following, we explain the new elements and discuss their effects on the approach. Here, $\mathcal{B}_P$ refers to the physical blocks of a single bank, while the register architecture presented in Section 10.1.3 features a global set of physical registers.

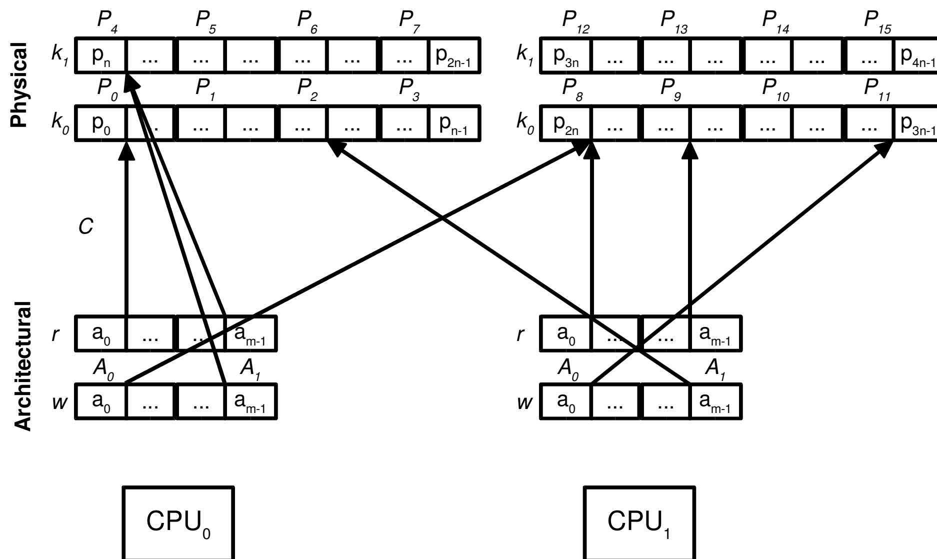


Figure 10.7.: General reconfigurable register architecture with *one* of the feasible mappings

Physical register banks $\mathcal{K} = \{k_0, k_1\}$ Each processor has at least one physical register bank. The number of registers in a physical bank may be equal to the number of architectural registers per processor.

Access mode $\mathcal{M} = \{r, w\}$ The access mode reveils, if a register is read or written by an instruction. If CPU_0 writes a value into the architectural register a_0, this value will be stored in the physical register p_{2n} (first physical register in the bank k_0 of CPU_1). A read access is forwarded to the physical register p_0 (first physical register in the bank k_0 of CPU_0).

Permitted mappings $\mathcal{C} : \mathcal{B}_A \times \mathcal{M} \times CPU \rightarrow \mathcal{B}_P \times \mathcal{K} \times CPU$ The set of all permitted mappings in the register architecture is specified by a signature $\mathcal{C}$. The associated function is *one* configuration of the reconfigurable register banks and maps an architectural block to a physical block. The register architecture in Figure 10.7 has the signature $\mathcal{C} : \mathcal{B}_A \times \mathcal{M} \times CPU \rightarrow \mathcal{B}_P \times \mathcal{K} \times CPU$. Consequently, a register block $\mathcal{B}_A$ in one of the two architectural banks $\mathcal{M}$ of a processor *CPU* is mapped to a register block $\mathcal{B}_P$ of the two physical register banks $\mathcal{K}$ of a processor *CPU*.

If the register architecture supports only a restricted set of mappings between blocks of common index within a bank, the set $\mathcal{B}_P$ must be omitted.

10.2.1. Permitted Mappings

The set of physical blocks which can be connected to an architectural block is represented by the signature $\mathcal{C}$. Here, we discuss two exemplary signatures:

Signature for Arbitrary Mappings The signature $\mathcal{C} : \mathcal{B}_A \rightarrow \mathcal{B}_P \times \mathcal{K}$ allows to map an architectural block $\mathcal{B}_A$ to an arbitrary physical block $\mathcal{B}_P \times \mathcal{K}$. In practice, the number of ports may limit the power of arbitrary mappings. CAPRICoRn also assumes non-restricted mappings, but neglects register banks and ports in addition.

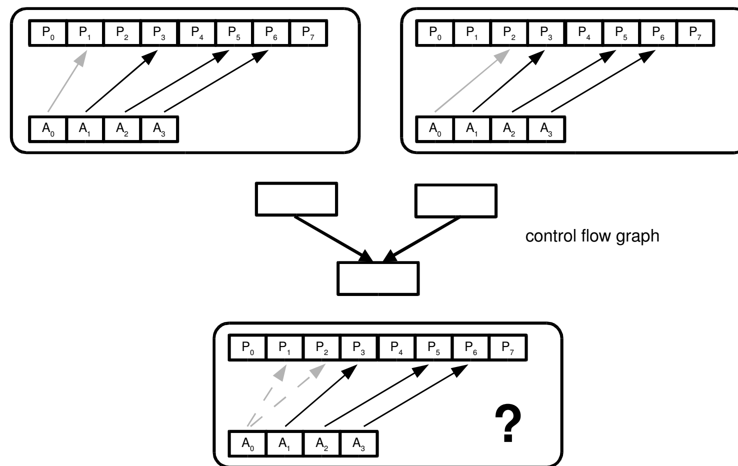


Figure 10.8.: Mapping problem in case of non-restricted mappings

Figure 10.8 motivates that placement of reconfiguration instruction is a challenging task when using arbitrary mappings in case of structured control-flow: The lower basic block has two predecessors, where the architectural block A_0 is mapped to different physical blocks. Also, a physical block may be mapped to different architectural blocks. In CAPRICoRn, such problem can be handled by two different strategies: The intra-block placement (see Section 12.2.1) does not re-use mappings from preceding basic blocks, but inserts reconfiguration instructions to enable physical blocks where needed as late as possible. Alternatively, the mappings of adjacent basic blocks can be matched such that an architectural block A is mapped to a physical block P in all predecessors of a basic block b (see Section 12.2.2). Hence, such mapping can be re-used in b in order to reduce the effort in reconfiguration.

Signature for Restricted Mappings When using the signature $\mathcal{C} : \mathcal{B}_A \rightarrow \mathcal{K}$, an architectural block $\mathcal{B}_A$ can only be mapped to a physical block with same index of a bank $\mathcal{K}$. Different register blocks with same index cannot be enabled simultaneously. In the following we say that such blocks are *in conflict*. Figure 10.9 illustrates an example: Both blocks denoted by b_0 of the register banks k_0 and k_1 can only be connected alternatively with the architectural block b_0. Hence, the two highlighted physical blocks cannot be used simultaneously.

The restriction has three major consequences: At first, two virtual registers used as operands of a common instruction might be allocated to the same physical block. The desired mapping can be established with one reconfiguration instruction. Secondly, two virtual registers

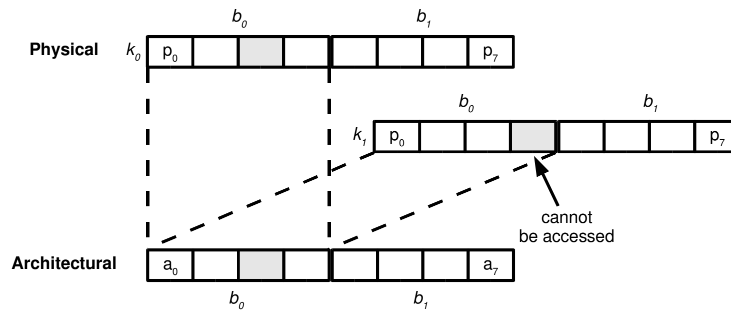


Figure 10.9.: Mapping problem in case of physical blocks with common index

could be allocated to physical blocks with different indices. In order to use them for one instruction, two physical blocks have to be enabled. Finally, the virtual registers might be allocated to physical blocks of different registers banks but with common index. Hence, the registers *cannot* be used by a single instruction, because they cannot be enabled simultaneously.

10.2.2. Discrimination between Read/Write Accesses

In general, a reconfigurable register architecture can also distinguish between read and write accesses by using two separate architectural register banks. Such additional layer of discrimination implicitly allows to address up to $2n$ operands in case of a n-address machine as Figure 10.10 illustrates: The `inc` instruction increases a register value by 1 and uses the architectural register a_1 as read/write operand. For read accesses, a_1 is connected with the physical register p_1, while write accesses operate on p_5. Hence, the original value of p_1 is preserved and the result of the increment is written to p_5, although only one register operand is encoded in the `inc` instruction. Such effect can be beneficial for 2-address machines, because a 3-address operation can be emulated and the overall runtime even might be reduced. On the other hand, this behaviour can be confusing and requires two reconfiguration instructions to establish read and write mappings for an architectural register.

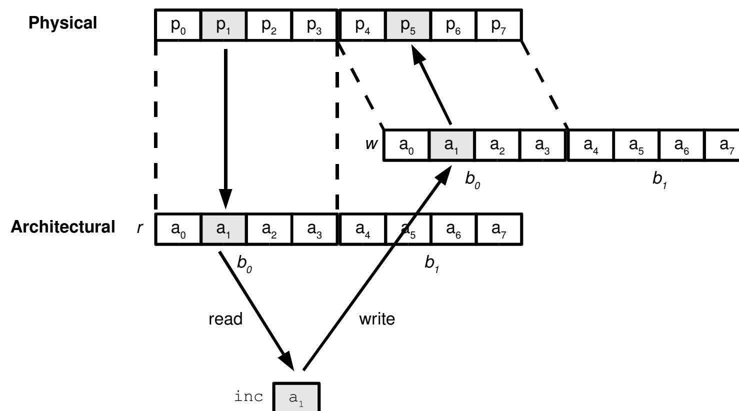


Figure 10.10.: Modified semantics of read/write operands

10.2.3. Operands in Multiple Registers

Some processors store a single operand in multiple physical registers. For example, the Intel processors provide 32-bit registers which consist of two 16-bit registers that are comprised of two 8-bit registers in turn. A compiler can support this feature by using logical registers which represent multiple physical registers and are allocated for virtual registers. If such logical registers belong to different physical blocks, multiple reconfiguration instructions are needed to access one operand as Figure 10.11 outlines. In the example, establishing such mappings is even not possible, because there are only two architectural blocks.

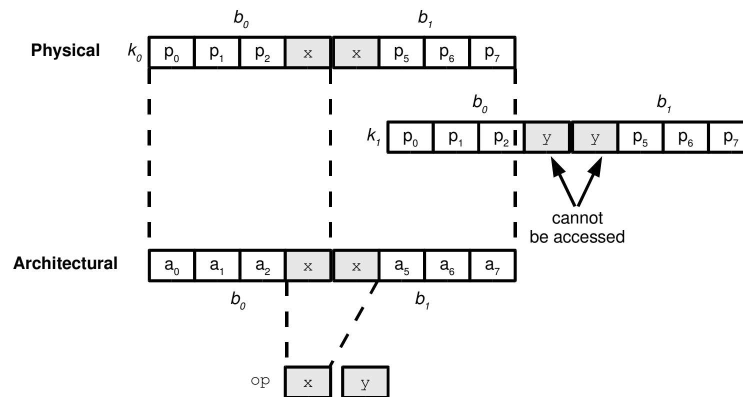


Figure 10.11.: Mapping problem in case of operands located in multiple physical blocks

11. Analysis of n-Liveness

Contents

11.1 Definition and Representation of n-Liveness **IV-26**
11.1.1 Access Trees IV-27
11.1.2 Probabilities in Access Trees IV-28
11.2 Specification of Data-Flow Problem **IV-29**
11.2.1 Tree Operations IV-30
11.2.2 Transfer and Union Function IV-31
11.2.3 Properties of Probabilities in Access Trees IV-32
11.3 Convergence of Data-Flow Analysis **IV-35**
11.3.1 Complete Access Tree IV-35
11.3.2 Partial Order of Access Trees IV-36
11.3.3 Monotonicity of Transfer Function IV-36
11.3.4 Monotonicity of Union Function IV-39
11.3.5 Convergence of Data-Flow Analysis IV-40
11.3.6 Properties of Union Function IV-41

One remarkable goal of CAPRICoRn[1] is to minimize the number of reconfigurations, which is achieved by three fundamental strategies: The physical registers of the machine are partitioned into blocks of a common size in order to avoid the expensive fine-grained reconfiguration per register. Furthermore, physical registers have to be allocated in such a way that a physical block contains mainly those registers which are often accessed close together (see Section 12.1). Last but not least, the placement of reconfiguration instructions is aimed at re-using mappings established in preceding basic blocks (see Section 12.2.2).

We have reduced the two latter tasks to computing a so-called n-liveness property for each program position, which represents the set of n pair-wise different values accessed next. During register allocation, affinities between virtual registers are determined based on the register accesses in a function and the mentioned property (see Section 12.1). The affinities represent the estimated number of reconfiguration instructions which are prevented when virtual registers accessed in conjunction are assigned to the same physical block. The register allocation of CAPRICoRn considers the affinities as a secondary criterion during coloring.

The inter-block strategy for placing reconfiguration code determines those physical blocks which maintain active until the beginning of succeeding basic blocks by analyzing the accesses. Such information is used to prevent additional reconfiguration instructions to activate those physical blocks (see Section 12.2.2). Our replacement heuristic unmaps that physical block which is *latest used again* (see Section 12.1.1), if no free architectural block is available.

Throughout this thesis, n-liveness is considered for accesses to virtual registers and physical blocks. In general, the property can be computed for any kind of values which may be accessed in a structured program.

Structure Section 11.1 first defines the n-liveness property precisely and introduces access trees as an intuitive representation. The n-liveness property can be computed for each program position by a Data-Flow Analysis (DFA), which is presented in Section 11.2. Finally, Section 11.3 shows the convergence of our DFA.

11.1. Definition and Representation of n-Liveness

Definition 11.1 (n-live)
Given a value v, a program position s, and a constant $n > 0$. Then, v is denoted to be n-live at s, if there is a path P starting at s where v belongs to the first n pair-wise different accessed values.

The existance of a path with the desired property can also be formulated as the union of sets: Let V_P be the set of the first n pair-wise different values which are accessed on a path P starting at s. Define V as the union of V_P for all those paths P. Then, v is denoted to be n-live at s, if $v \in V$.

[1]**C**ompiler **A**nticipated **P**rocessor **R**egister **I**nter-**Co**nnected **R**econfiguratio**n**

11.1.1. Access Trees

Obviously, the order of register accesses and the structure of control-flow is not preserved, if the n-live registers for a certain program position s are represented as a set. Hence, we decided to model the n-liveness property by a so-called access tree with depth $\leq n$, which contains all registers that are n-live at s. Basically, each path from the root to a leaf models the sequence of the first n pair-wise different values which are accessed on a path starting at s. As a consequence, control-flow is represented implicitly in the n-liveness information. Furthermore, the property of a program position preceding s can be computed efficiently by updating the access tree for s as outlined in Section 11.2.

Definition 11.2 (Access tree)
Given a set of registers $\{1, \ldots, k\}$, a program position s, and a constant $n > 0$. Let V_P be the ordered set of the first n pair-wise different values which are accessed on a path P starting at s. Define T_P as the tree representing V_P, which actually corresponds to a linear chain without any branches. Then, T is defined as the tree with root r and subtrees T_P for all those paths P. The root r is not associated with a register and has no further meaning.

If a node has multiple successors for the same register, such children are merged to a single node. The resulting access tree is called a *reduced* access tree. By these means, the total number of nodes can be decreased and therefore the complexity of applied algorithms is reduced.

Definition 11.3 (Reduced access tree)
Let T be an access tree and L_k the set of nodes on level k. Hence, L_0 corresponds to the root and L_k to the union of the k-th nodes from T_P according to Definition 11.2. Then, the reduced tree T' is defined as the union of $L_k, \forall k \in \{0, \ldots, n\}$.

Obviously, the following statements hold with respect to the above definitions:

Theorem 11.4
For all paths from the root to a leaf of an access tree, each register occurs at most once.

Theorem 11.5
The maximum depth of an access tree for the n-liveness property is n.

Theorem 11.6
A node in a reduced access tree cannot have more than one direct successor of the same register.

Due to the limited depth and degree of each access tree, the number of nodes is limited. Although the tree can theoretically consist of many nodes, it is usually rather small as basic blocks typically contain accesses to multiple virtual registers. This results in linear chains without any branches in the tree. Furthermore, the degree of a branch in the tree is limited by the degree of the corresponding control-flow node. Many control-flow nodes have at most two successors.

Example Now, we explain the intuitive definition of access trees using an example. Figure 11.1 (a) shows a CFG with accesses to values $a, \ldots, e$ and a program position s. The access tree of the 4-liveness property for s can be found in Figure 11.1 (b). Obviously, $a, \ldots, d$ are 4-live at s, because they are accessed after at most four steps. As d is used by two adjacent accesses on the left path, e is also n-live at s. Figure 11.1 (c) illustrates the 3-liveness property. e is not 3-live, because it remarks the fourth or fifth pair-wise different access on the two paths, respectively. All remaining registers are 3-live.

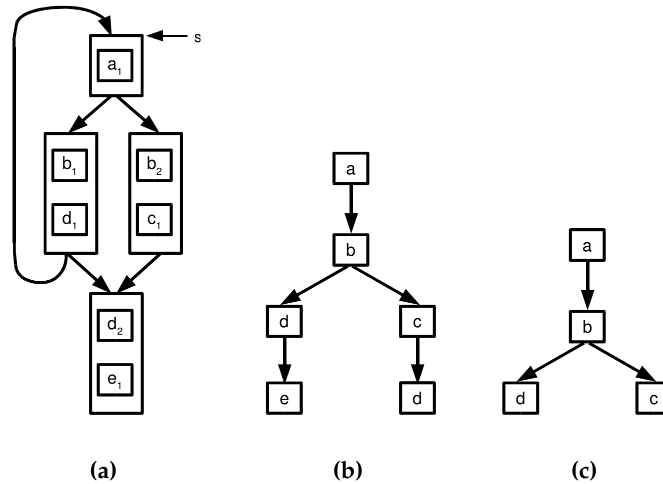


Figure 11.1.: (a) Control-flow graph (b) Access tree for 4-live registers at s (c) Access tree for 3-live registers at s

11.1.2. Probabilities in Access Trees

In order to compute the n-liveness with maximum preciseness, the branching probabilities in the CFG have to be mapped to the access tree. Thereby, we can supply quantitative information about the likelihood of register accesses occuring at runtime. Such probabilities may be assumed to be uniformly distributed or can result from a profiling. The current implementation of the CAPRICoRn system does not rely on a profiling tool, but uses uniformly distributed probabilities as a static estimation. When using probabilities, the definition of n-liveness presented above can be extended as follows:

Definition 11.7 (n-live with probability)
Given a value v, a program position s, and a constant $n > 0$. Let $p_P \leq 1$ be the probability for accessing a value $v \in V_P$ on a path P starting at s, and define p as the sum of p_P for all such P. Consequently, v is n-live at s with probability p, if $v \in V$.

Example Figure 11.2 (a) shows the CFG from Figure 11.1 (a) with uniformly distributed probabilities. The access tree of the 4-liveness property for s is illustrated in Figure 11.1 (b). Clearly, a, b, and d are 4-live with probability 1, because they are accessed on each path. As

c is only accessed on the right-hand path, it is 4-live with probability $1/2$. Importantly, e is 4-live with probability $1/4$ only, although the access is reached on both paths, because it is the fifth pair-wise different access when using the right-hand path. Basically, the access tree representing the 4-liveness property includes a further access of a with probability $1/4$ (border drawn with dashed line). In this example, the access is negligible, because there is already a node for a with probability 1. On the other hand, it demonstrates that the loop is executed with probability $1/4$ when assuming uniform probabilities.

Figure 11.1 (c) illustrates the 3-liveness property. Like in the example from Figure 11.1 (c), a, b, and c are 3-live, but not e. d is 3-live with probability $1/2$ only, because the access in the lower basic block is not considered anymore.

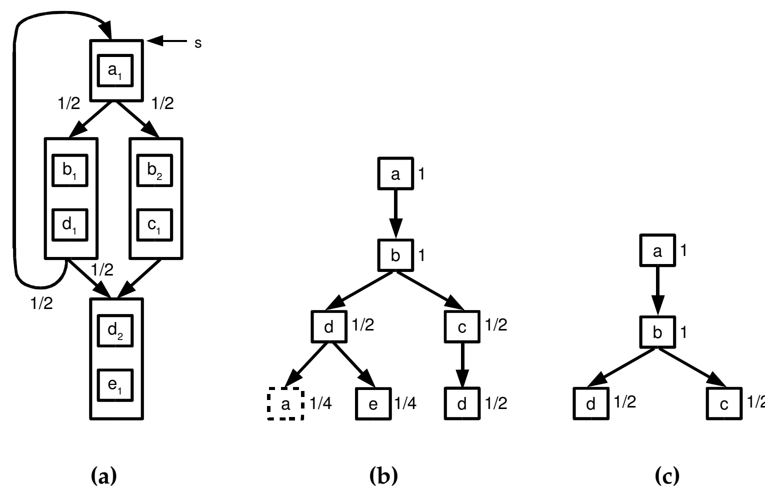


Figure 11.2.: (a) Control-flow graph (b) Access tree for 4-live registers at s (c) Access tree for 3-live registers at s

Execution Frequency of Basic Blocks In the following, we often need to model the iteration count $\omega(b)$ of a basic block b. If profiling data is available, such information can be computed from the branching probabilities. As the CAPRICoRn compiler assumes uniformly distributed probabilities, the execution frequency is based on a heuristic which considers the nesting depth of a basic block. Concretely, a fixed iteration count c is maintained for each loop. If n loops are nested, the innermost blocks are assumed to be executed c^n times.

11.2. Specification of Data-Flow Problem

The set of n-live registers at a program position can be represented as an access tree. Such tree is annotated with the branching probabilities of the CFG in order to model the control-flow structure and likelihood of register accesses best. In this section, we describe a data-flow problem to compute access trees for each position in a program.

Access trees are propagated against the control-flow, because the n-liveness property depends on succeeding register accesses (see Definition 11.1). As the definition is based on the existance of a path, access trees are unified at the boundaries of the basic blocks (join function). Hence, the data-flow problem for the n-liveness is a backward-union problem, The DFA computes a minimal fixpoint which demands an initialization of all properties with $\perp$ and a monotonic transfer function.

Section 11.2.1 introduces four basic operations to manipulate access trees, which are employed by the transfer and union functions outlined in Section 11.2.2. Finally, Section 11.2.3 demonstrates important properties of probabilities in access trees that are needed to prove the convergence of the DFA in Section 11.3.

11.2.1. Tree Operations

The transfer and union functions (see next subsection) manipulate the access trees using four basic operations:

- prepending a list of nodes to an access tree
- removing nodes from an access tree
- combining two access trees
- reducing an access tree

Here, we define the above operations precisely by using the examples shown in Figure 11.3 to Figure 11.6.

Removing Nodes The transfer function removes superfluous nodes and nodes with depth greater n from an access tree (see Section 11.2.2). Removing a node means to replace it by its successors, if such nodes exist (see Figure 11.3). In the example, a node labeled c is replaced by its children $s_1, \dots s_n$. As a result, the father of the removed node can have equal children which implies a reduction. If $b = s_1$, for instance, both nodes will be merged by a reduction. Obviously, removing nodes from an access tree does not influence the probabilities of the remaining nodes.

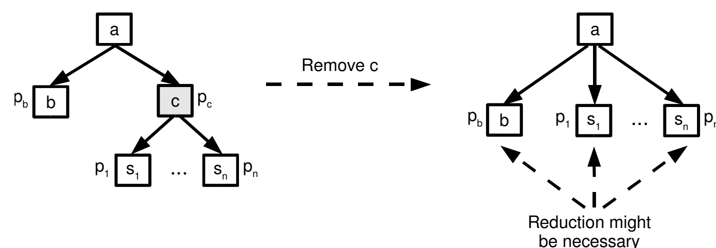


Figure 11.3.: Removing a node from the tree

Combining Trees The union function combines multiple access trees (see Section 11.2.2). The combination of two trees (see Figure 11.4) unifies the artifical nodes and then reduces the resulting tree to avoid redundant children. Thereby, the probabilities are scaled according to the semantics of the probabilities. In the CAPRICoRn compiler, all probabilities are multiplied with $\frac{1}{2}$. Consequently, the combination of n trees demands a scaling of the probabilities by $\frac{1}{n}$.

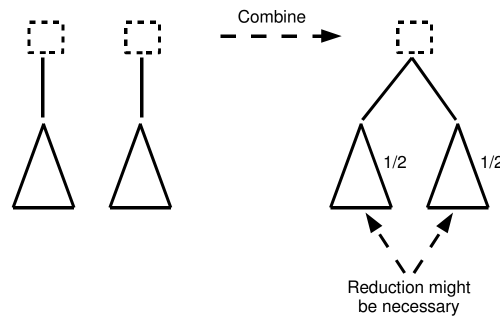


Figure 11.4.: Combination of two trees

Reducing Trees If a node has two equal successors, these will be merged by a reduction (see Figure 11.5) and their probabilities will be added. As a conseqence, the sets of their children will be merged which might require further reduction.

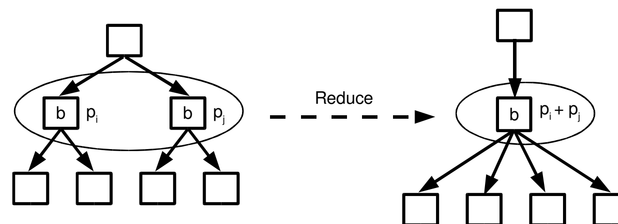


Figure 11.5.: Reduction of two common child nodes

Prepending Nodes Prepending a list of nodes to an access tree can be realized by a successive prepending of single nodes to the tree. A node is inserted between the artifical root and its children, whereas its probability is set to 1. Furthermore, all nodes of the same register are removed from the tree beforehand (see Figure 11.6).

11.2.2. Transfer and Union Function

The transfer function maps the OUT property of a basic block b to its IN property. Figure 11.7 illustrates the behaviour of our transfer function operating on access trees.

Initially, the list of the first accesses to registers in b is prepended to the OUT tree. Such list is called *GEN list* in the following and describes the influence of a basic block to the propagated property of n-liveness. If the n-live registers are computed with respect to the

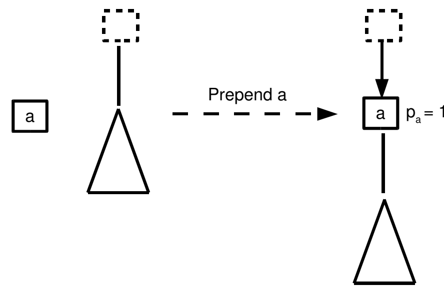


Figure 11.6.: Prepending a node

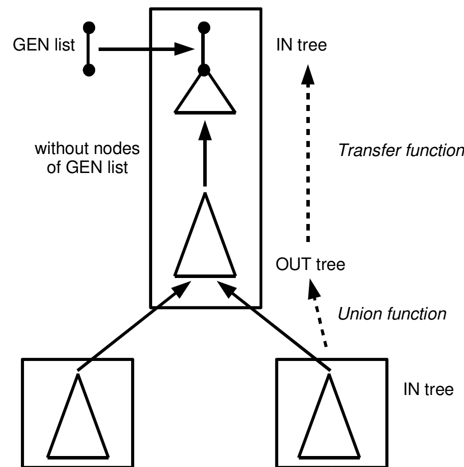


Figure 11.7.: Propagation of access trees

beginning of b, the probability of all register accesses in the GEN list is 1, because branches are only allowed at the end of a basic block.

In order to avoid redundant nodes on a path from the root to a leaf, those nodes of the OUT tree which also occur in the GEN list are removed. Additionally, all nodes with a depth greater than n are eliminated. After a reduction, the resulting IN tree fulfills the Theorem 11.4 to Theorem 11.6.

The union function computes the OUT property of a basic block by combining the IN properties of its successors. Two trees are combined by merging their roots and reducing the resulting tree to avoid redundant children.

11.2.3. Properties of Probabilities in Access Trees

Here, we define a special class of access trees and prove three important theorems about probabilities in such trees. These are needed to demonstrate the convergence of our DFA in the next section.

Definition 11.8 (Fully occupied access tree)
Let T be an access tree for the n-liveness property at a program position s. If the sum of all probabilities in T equals n, T is denoted a *fully occupied* access tree. An access tree is *not* fully

occupied, if there exists a path from s to the end of a program, where less than n different registers are accessed. A fully occupied tree should not be mistaken for a *complete* tree.

Theorem 11.9
The probability p_v of a node v in an access tree T is not smaller than the sum S_v of the probabilities of its children. If the tree is fully occupied, then $p_v = S_v$. This statement also holds for the root.

$$\forall v : p_v \geq \left(\sum_{s \in \text{succ}(v)} p_s \right) = S_v$$

Proof: It is shown that the statement holds for the empty tree and after each tree operation. As an access tree is constructed by applying the tree operations repeatedly on the empty tree, the statement is true for all access trees then.

Empty tree The empty tree only consists of the artifical root and has no further children. Hence $1 = p_v > S_v = 0$, i.e. p_v is not smaller than S_v. The empty tree is clearly not fully occupied.

Remove node Let u be the father of v. By definition, it holds $p_v \geq S_v$ and $p_u \geq S_u$ before removing v. If v is removed, it will be replaced by its children. Hence, $p_u \geq S_u + \underbrace{S_v - p_v}_{\leq 0} \geq S'_u$ after removing v, with $S'_u = \sum_{s \in \text{succ}(u)} p_s$ after a reduction. If the original tree is fully occupied, just replace all $\geq$ and $\leq$ symbols by $=$.

Combine trees During the combination of two trees, the probabilities of all nodes are multiplied with the same constant factor f. It follows:

$$p_v \geq \sum_{s \in \text{succ}(v)} p_s \Rightarrow f * p_v \geq \sum_{s \in \text{succ}(v)} f * p_s$$

Reduce trees Let the statement be true for an access tree containing two subtrees with roots x and y, respectively, which are to be reduced. We show that the statement still holds after a reduction:

$$\begin{aligned} & \left(p_x \geq \sum_{s \in \text{succ}(x)} p_s \right) \wedge \left(p_y \geq \sum_{s \in \text{succ}(y)} p_s \right) \\ \Rightarrow \quad & p_x + p_y \geq \left(\sum_{s \in \text{succ}(x)} p_s \right) + \left(\sum_{s \in \text{succ}(y)} p_s \right) \\ \Rightarrow \quad & p_x + p_y \geq \left(\sum_{s \in \text{succ}(x) \cup \text{succ}(y)} p_s \right) \end{aligned}$$

Prepend node The artifical root r of a tree and the prepended node a always have probability 1.

$$p_r \geq \sum_{s \in \mathrm{succ}(r)} p_s \wedge p_r = p_a \Rightarrow p_a \geq \sum_{s \in \mathrm{succ}(a)} p_s$$

Theorem 11.10
The sum L_d of all probabilities at a level d of an access tree is not greater than 1. It holds $L_d = 1$, if the tree is fully occupied.

$$\forall d : L_d = \left(\sum_{x, depth(x)=d} p_x \right) \leq 1$$

Proof: Let v be a node from level d. Theorem 11.9 implies that $p_v \geq \sum_{s \in \mathrm{succ}(v)} p_s = S_v$. Hence, $L_d \geq \sum_{x, depth(x)=d} \left(\sum_{s \in \mathrm{succ}(x)} p_s \right) = L_{d+1}$, i.e. the sum of all probabilities for level $d+1$ is not greater than the corresponding sum for level d. As the probability of the root is 1, the statement holds for all levels.

Theorem 11.11
The sum R_r of probabilities of the nodes for a register r is not greater than 1.

$$\forall r : R_r = \left(\sum_{x, reg(x)=r} p_x \right) \leq 1$$

Proof: Let $R_{r,v}$ be defined similar to R_r, but be restricted to the subtree with root v.

The statement is proven by induction over the depth $d \geq 1$ of subtrees t' of an access tree T, whereas the leafs of t' must also be leafs of T.

At first, we show that the statement is true for $d = 1$. Let l be a leaf of an access tree. Obviously it holds:

$$R_{r,l} = \begin{cases} p_l & reg(l) = r \\ 0 & otherwise \end{cases}$$

which implies $R_{r,l} \leq p_l \leq 1$.

Now we prove that the statement is also fulfilled for subtrees with depth $d > 1$ and root v. If $reg(v) = r$, then $R_{r,v} = p_v \leq 1$. Otherwise, we can conclude:

$$\begin{aligned} & p_v \geq S_v = \sum_{s \in succ(v)} p_s \;\wedge\; p_s \geq R_{r,s} \;\wedge\; R_{r,v} = \sum_{s \in succ(v)} R_{r,s} \\ \Rightarrow\; & R_{r,v} = \sum_{s \in succ(v)} R_{r,s} \leq \sum_{s \in succ(v)} p_s = S_v \leq p_v \\ \Rightarrow\; & R_{r,v} \leq p_v \end{aligned}$$

11.3. Convergence of Data-Flow Analysis

In order to ensure the convergence of a DFA, two requirements need to be fulfilled in general. Firstly, the transfer function must be monotonically increasing, i.e.

$$a \subseteq b \Rightarrow f(a) \subseteq f(b)$$

Secondly, the union function must have the behaviour of a join operation (see Section 11.2). This condition is *not* fulfilled by our data-flow problem as discussed in Section 11.3.6. However, we show that the union function is monotonic.

The monotonicity of the transfer and union functions can be exploited to prove that the DFA converges (see Section 11.3.5).

11.3.1. Complete Access Tree

Before, we introduce a special class of access trees, which is used to simplify the proofs and to define the partial order of access trees.

The complete access tree is a k-ary tree with depth n, whereas k corresponds to the number of registers considered by the data-flow analysis. In contrast to the access trees considered beforehand, some nodes may have probability 0, i.e. the nodes that are not accessed. Hence, an access tree is always a subtree of the complete access tree. Using this representation, all access trees for k registers with depth n have the same structure and differ only in the probabilities of nodes.

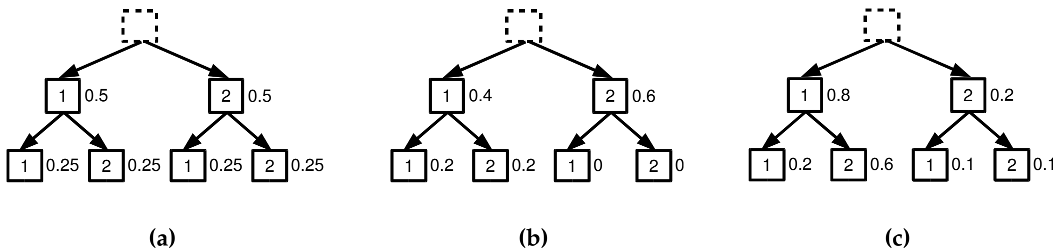


Figure 11.8.: Examples of complete access trees

Figure 11.8 shows three exemplary complete access trees for the registers $\{1, 2\}$ and $n = 2$. As the following proofs demand a unique identification of nodes, we introduce an appropriate convention:

Definition 11.12 (Position of node)
Let v be a node of a complete access tree T. The position of v is determined by the simple path P from the root of T to v and has the form $x = (\nu_1, \ldots, \nu_d)$. ν_d corresponds to the node v and d is the depth of v in T. The set of all node positions is denoted as $\Pi := \{\{\nu_1, \ldots, \nu_n\}^i \mid i \leq n\}$.

In terms of Figure 11.8, the direct successors of the root have the positions (1), (2) and the indirect successors the positions $(1,1)$, $(1,2)$, $(2,1)$, and $(2,2)$.

11.3.2. Partial Order of Access Trees

Here, the partial order of complete access trees is introduced in order to use it in the following proofs. The definition is based on the probabilities of nodes.

A complete access tree a is *smaller* than a complete access tree b, if the probability of each node position $x \in \Pi$ is not greater than the probability of the corresponding position in b. Let $p_a(x)$ be the probability of the node at position x in the tree a.

$$a \subseteq b \Leftrightarrow \forall x \in \Pi : p_a(x) \leq p_b(x)$$

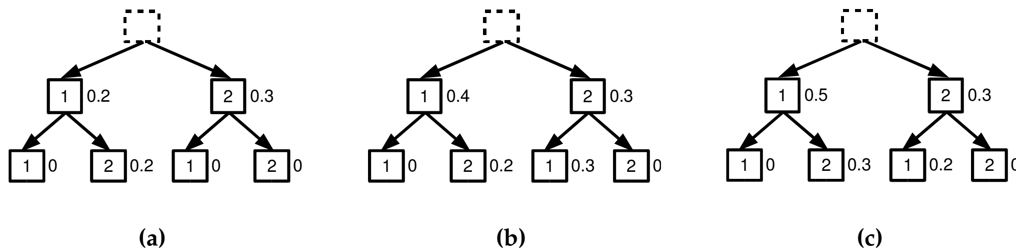


Figure 11.9.: Partial order of complete access trees

Figure 11.9 shows three complete access trees for the register set {1,2} and $n = 2$. The probabilities are printed on the right hand side of the nodes. The tree (a) is smaller than the trees (b) and (c) according to the above definition. But tree (b) is not smaller than the tree (c), because the position $(2,1)$ has the probabilities 0.3 and 0.2 in (b) and (c), respectively. Hence, it does not hold $p_b(2,1) \leq p_c(2,1)$. On the other hand, the tree (b) is also not greater than the tree (c), because the probabilities of positions (1) and $(1,2)$ in tree (b) are smaller.

11.3.3. Monotonicity of Transfer Function

Now we show that the transfer function f is monotonically increasing, i.e.

$$a \subseteq b \Rightarrow f(a) \subseteq f(b)$$

At first, we introduce two functions to model important properties of the transfer function needed by our proof. Section 11.2.2 described the behaviour of this function: It maps an OUT tree to an IN tree by prepending the GEN list to the OUT′ tree. The latter tree is constructed from the OUT tree by removing the registers of the GEN list (see Section 11.2.2).

Figure 11.10 characterizes an IN tree: The upper part, which is constructed from the GEN list, is constant, because the GEN list is also constant. All nodes in the GEN list have probability 1, as the represented accesses occur within a common basic block. The OUT′ tree can

be modelled by a complete tree: The black nodes correspond to the nodes of the GEN list and therefore have probability 0. The white nodes are the remaining nodes and have a non-zero probability.

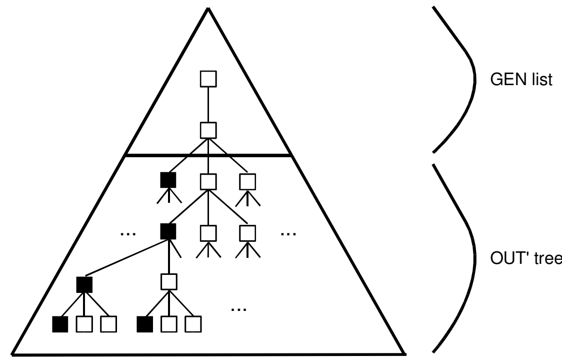


Figure 11.10.: Example of an IN tree

del function In order to estimate the relation between the probabilities of the white nodes and the probabilities in the OUT tree, the *del* function is introduced. It defines how the registers of a tree T are moved, if the registers of the GEN list are removed from T. Such registers are denoted as GEN registers in the following. Concretely, the *del* function maps a list of parent nodes representing the position of a node to a list without the GEN registers. The function is only defined for positions of nodes *not* included in the GEN list:

$$\begin{aligned} del(\epsilon) &= \epsilon \\ del(\nu_1, \nu_2, \ldots, \nu_d) &= \begin{cases} del(\nu_2, \ldots, \nu_d) & \nu_1 \in GEN \\ \nu_1 \circ del(\nu_2, \ldots, \nu_d) & otherwise \end{cases} \end{aligned}$$

Let us consider the example in Figure 11.11 which assumes that the GEN list consists of register 1 only. Hence, all nodes for this register are removed from the tree. Please re-call that register 1 may occur multiple times, because the nodes of a complete access tree can have probability 0. Consequently, the node at position $(1, 2)$ is unified with the node at position (2) and $del(1, 2) = (2)$. The node at position (2) is mapped to itself, i.e. $del(2) = (2)$, because its position is not influenced by the operation. The *del* function is not defined for the node at position $(2, 1)$, because it is removed.

We can conclude that the definition of the *del* function only depends on the GEN list. As the GEN list for a certain basic block is constant, the corresponding *del* function is also constant.

reg function Now we introduce the *reg* function which yields the register for a given position and is defined as follows:

$$reg(\nu_1, \nu_2, \ldots, \nu_d) = \nu_d$$

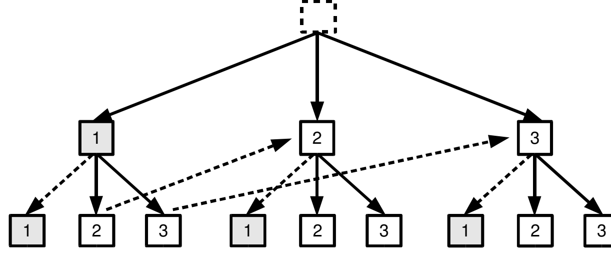


Figure 11.11.: Removing of GEN registers of complete access tree

Furthermore, let $D(x)$ be the set consisting of all positions which are mapped to x by the *del* function, but without those positions where nodes are removed:

$$D(x) := \{y \mid del(y) = x \;\; \forall y \in \Pi \wedge reg(y) \notin GEN\}$$

As the definitions of the *del* and *reg* functions are constant, the definition of $D(x)$ is also constant. Hence, a unique set of positions $D(x)$ is mapped to the position x by the *del* function. For the example of Figure 11.11, $D(2) = \{(1,2),(2)\}$. Consequently, the probability of a node at position x in the OUT′ tree is equal to the sum of probabilities at the positions $D(x)$ in the OUT tree:

$$p_{OUT'}(x) = \sum_{y \in D(x)} p_{OUT}(y)$$

Remarkably, we have an equation to compute the probability of a node in the OUT′ tree *and* the probability depends on a unique set of nodes from the OUT tree.

Monotonicity of Transfer Function Let OUT1 and OUT2 be two complete trees with

$$OUT1 \subseteq OUT2$$

Clearly, the tree OUT1 is smaller than the tree OUT2. The definition of the partial order implies that

$$\forall x : p_{OUT1}(x) \leq p_{OUT2}(x)$$

Now we can conclude that the probability of a node x in the OUT1′ tree is smaller than for the OUT2′ tree:

$$\begin{aligned} & p_{OUT1'}(x) = \left(\sum_{y \in D(x)} p_{OUT1}(y) \right) \leq \left(\sum_{y \in D(x)} p_{OUT2}(y) \right) = p_{OUT2'}(x) \\ \Rightarrow \quad & p_{OUT1'}(x) \leq p_{OUT2'}(x) \end{aligned}$$

The above inequality also holds for a node $x \in GEN$, because its probability is 0 in both trees. As the nodes of the GEN list are constant, a similar inequality can also be shown for the IN1 and IN2 trees:

$$\forall x : p_{IN1}(x) \leq p_{IN2}(x)$$

The definition of the partial order implies:

$$IN1 \subseteq IN2$$

As a summary, we have shown the monotonicity of the transfer function:

$$OUT1 \subseteq OUT2 \Rightarrow IN1 \subseteq IN2$$

11.3.4. Monotonicity of Union Function

As mentioned above, the union function does not fulfill the requirements for a join function (see Section 11.3.6). Here, we show that the union function is monotonic. This is an important cornerstone for the convergence of the data-flow analysis proven in Section 11.3.5. The properties of the union function are discussed again in Section 11.3.6.

In the following, we say that a tree has been *increased* or *decreased*, if the probabilities of its nodes have been increased or decreased, respectively. In case of non-modified probabilities, a tree is denoted to be *preserved*. The monotonicity is defined by

$$a \leq a' \wedge b \leq b' \Rightarrow a \cup b \leq a' \cup b'$$

We refer to the definition of combining access trees (see Section 11.2.1) and the properties of probabilities (see Section 11.2.3). The probability at a position x in the OUT tree corresponds to the average of probabilities at x in the trees $IN_1, \ldots, IN_m$:

$$p_{OUT}(x) = \frac{\sum_{i=0}^{m} p_{IN_i}(x)}{m}$$

Hence, the OUT tree increases with rising IN trees. Now we show that the OUT tree is only increased, if all IN trees are preserved or increased:

$$(\forall i \leq m : IN1_i \subseteq IN2_i) \Rightarrow (OUT1 \subseteq OUT2)$$

The proof considers propositions about the probabilities of nodes corresponding to the properties of trees:

$$\begin{aligned}
& \forall i \leq m : IN1_i \subseteq IN2_i \\
\Rightarrow\ & \forall i \leq m, x \in \Pi : p_{IN1_i}(x) \leq p_{IN2_i}(x) \\
\Rightarrow\ & \forall x \in \Pi : \frac{\sum_{i \leq m} p_{IN1_i}(x)}{m} \leq \frac{\sum_{i \leq m} p_{IN2_i}(x)}{m} \\
\Rightarrow\ & p_{OUT1}(x) \leq p_{OUT2}(x) \\
\Rightarrow\ & OUT1 \subseteq OUT2
\end{aligned}$$

11.3.5. Convergence of Data-Flow Analysis

Up to now, we have demonstrated that the transfer and union functions are monotonic. Now it will be shown that the IN and OUT properties are monotonically increasing with respect to the number of iterations.

At the beginning of the DFA, all probabilities in the IN and OUT trees are initialized with 0. Clearly, these probabilities can only increase during the first iteration.

As the OUT trees have been increased monotonically, the IN trees are also increasing monotonically because of the monotonicity of the transfer function. The same holds for the OUT trees due to the monotonicity of the union function.

The convergence of the probabilities in the IN and OUT trees follows from the observation that the probabilities are bounded by 1 and the trees are monotonically increasing. This also implies the convergence of the DFA.

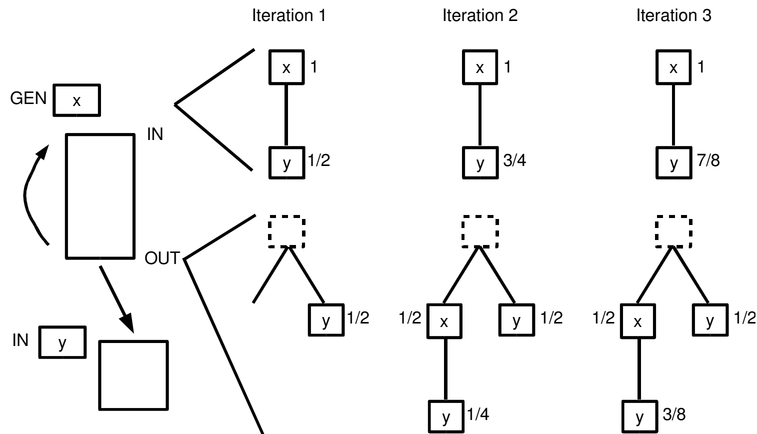


Figure 11.12.: Probabilities of IN and OUT trees during multiple DFA iterations

However, it can happen that the fixpoint is not reached after a finite number of iterations. For instance, if the CFG contains a loop and one probability behaves like the sequence $1 - 2^{-k}$, the probability will never reach the limit 1 (see Figure 11.12). But as the value set of probabilities is countable due to the usage of data types with finite space, the DFA will converge in practice after a finite number of steps.

11.3.6. Properties of Union Function

Our union function f does not fulfill the requirements for a join function (see Section 11.2). For instance, the result is not always greater than the operands due to the computation of the average over all probabilities. This requirement can be fulfilled, if a sum is computed instead of the average. On the other hand, such union function f' does not compute the supreme of two operands, whereas the supreme is the smallest element of the set, which is not smaller than the two operands. A function of the latter type is denoted as f''. Clearly, f'' idempotent, i.e. $f''(x) = f''(f''(x))$, and f' does not bear this property.

Furthermore, the set of feasible access trees does not have a top element, which needs to be greater than all feasible access trees. Such tree must be a complete tree with a probability of 1 for all nodes. Obviously, this access tree is not permitted, because Theorem 11.4 is violated.

12. Register Allocation

Contents

12.1 Allocation of Physical Registers **IV-45**
12.1.1 Replacement Strategy IV-46
12.1.2 Definition of Affinity IV-47
12.1.3 Affinity to Physical Blocks IV-48
12.1.4 Construction of Affinity Graph IV-50
12.1.5 Improvement of Allocation IV-52
12.2 Reconfiguration **IV-54**
12.2.1 Intra-Block Reconfiguration IV-54
12.2.2 Inter-Block Reconfiguration IV-55
12.2.3 Inter-Procedural Reconfiguration IV-58
12.3 Extensions for Multi-Cores **IV-59**
12.3.1 Reconfiguration IV-61
12.3.2 Re-Scheduling IV-62

CAPRICoRn[1] consists of two phases (see Figure 12.1) and replaces the conventional phase for allocating registers in typical compilers. In the first phase, the physical registers of the machine are assigned to the virtual registers in each function. Such physical registers cannot be accessed directly in our register architecture (see Chapter 10). Hence, the second phase replaces the physical registers by architectural ones and inserts reconfiguration instructions to establish proper mappings between architectural and physical blocks.

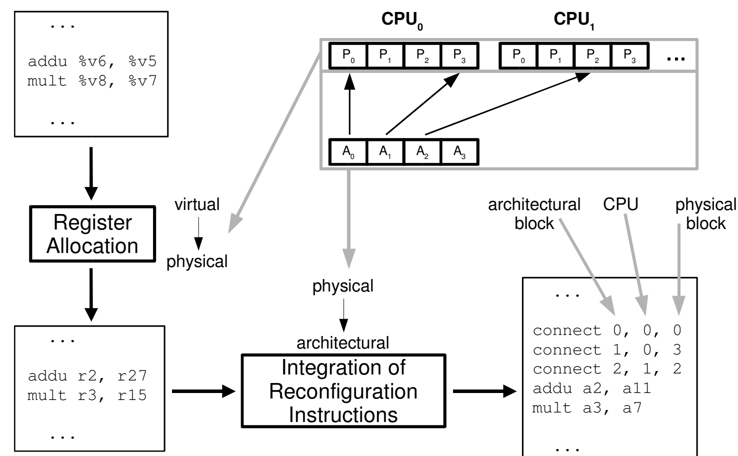


Figure 12.1.: Structure of register allocation

Importantly, the second phase depends heavily on the distribution of register values to physical blocks decided by the first phase. Let us assume that two virtual registers are often accessed close together, but both are assigned to physical registers of *different* physical blocks. Then two reconfiguration instructions would be needed to make both virtual registers accessible. If such case occurs for many pairs of register values or is even part of a frequently executed loop body, high penalty costs will be caused during execution.

Consequently, an estimate of anticipated reconfiguration costs is used, to partition the virtual registers into the physical blocks. The computation is based on the DFA of n-liveness presented in Chapter 11, which can determine the n pair-wise different values accessed next for each program position.

Structure In this chapter, we explain the characterized two-pass register allocation in detail. The first phase is outlined in Section 12.1, while the second pass is covered by Section 12.2. For simplification, we concentrate on single processors in the first place. Last but not least, Section 12.3 discusses some extensions of CAPRICoRn for multi-core architectures like the QuadroCore. The following paraphrases use implicitly the terminology introduced in Section 10.1.1.

[1]**C**ompiler **A**nticipated **P**rocessor **R**egister **I**nter-**Co**nnected **R**econfiguratio**n**

12.1. Allocation of Physical Registers

The register allocation of CAPRICoRn is based on global register allocation by graph coloring [32] (see Section 9.1.3). Additionally, it incorporates several improvements to reduce spill code and to avoid conflict edges in the interference graph presented by Briggs et al. [24].

We use a special heuristic which tries to allocate the physical registers in such a way that a physical block contains preferably those registers which are often used in conjunction. By these means, only one reconfiguration instruction is needed to access all of them. Obviously, this minimizes the number of reconfiguration instructions which are inserted after the register allocation.

Affinity The decision, which virtual registers are assigned to the same physical block, is based on the number of avoided reconfiguration instructions. Such effort in reconfiguration is determined from information about future register accesses for all positions in a given function, which can be computed by using our DFA of n-liveness (see Chapter 11). In the following, the number of prevented reconfiguration instructions is denoted as the *affinity* between both virtual registers. The affinities between all pairs of virtual registers are modelled as edge weights of a so-called *affinity graph*. Its nodes correspond to the virtual registers of an abstract machine program. An affinity graph is used during graph coloring as a secondary criterion for selecting a physical register.

Affinity Graph Figure 12.2 shows a small exemplary affinity graph for the virtual registers $v_0, \ldots, v_3$ as well as a physical register bank with 8 physical registers divided into 4 physical blocks. If v_0 is assigned to p_0 and v_2 to p_1, 3 reconfiguration instructions are avoided, because the affinity of v_0, v_2 is 3 and p_0, p_1 are located in the same physical block. Obviously, such a decision is part of an optimal assignment: If v_0, v_2 and v_1, v_3 are assigned to common physical blocks each, only one additional reconfiguration instruction will be needed.

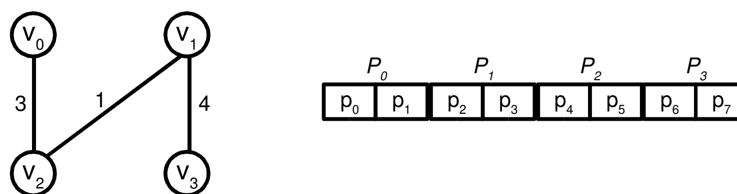


Figure 12.2.: Example of affinity graph

Structure This section concentrates on the computation of the affinities and their usage by the graph coloring method. At first, we present the replacement strategy for physical blocks in Section 12.1.1. Furthermore we prove an important theorem which demonstrates when a physical block is definitely *not* replaced. Such property is used in Section 12.1.2 to define the affinity in terms of register accesses. Using this definition, affinities can be determined by our analysis of n-liveness (see Chapter 11). Afterwards, the definition is extended to handle register accesses in loops properly. Section 12.1.3 argues that affinities must also be computed between virtual registers and physical blocks to model the situations

during the allocation process properly. The algorithm to compute the affinities and the actual construction of affinity graphs is explained in Section 12.1.4. Finally, Section 12.1.5 motivates challenges in the interpretation of affinities during graph coloring. We present a heuristic to balance the partitioning of virtual registers into physical blocks.

12.1.1. Replacement Strategy

If a physical block P must be activated at a program position s, the compiler has to select an architectural block which should be mapped to the physical block. An obvious decision is to use an unmapped architectural block, if possible. Otherwise, an architectural block A must be chosen that is already mapped to a physical block $\mu_A(A) = P'$. The selection of an architectural block A is determined by the *replacement strategy* and depends on its associated physical block P'.

CAPRICoRn replaces the physical block which is *latest used again* (LUA). Let s be a program position and P a physical block which is latest used again after passing s. By default, P will be replaced according to LUA. But if there exists another physical block P' which is not used anymore, P' is regarded to be accessed later than P. Assume that there are multiple paths starting at s and $\mathcal{P}$ is the set of physical blocks that are latest used again on such paths, respectively. In this case, an arbitrary block in $\mathcal{P}$ is chosen. A similar strategy is used if there are multiple physical blocks $\mathcal{P}'$ which are not accessed anymore after passing s.

Originally, this strategy was presented by Belady [12] as an optimal paging algorithm for virtual memory management and is also used for local register allocation in basic blocks (see Section 9.1.2). The strategy by Belady also defines that a physical block is made accessible as late as possible, i.e. right before the next access of the physical block.

In order to define the affinity in terms of register accesses (see Section 12.1.2), it must be known, if a physical block P might be replaced between two accesses of P. The following theorem expresses when a physical block is definitely *not* replaced by the LUA strategy:

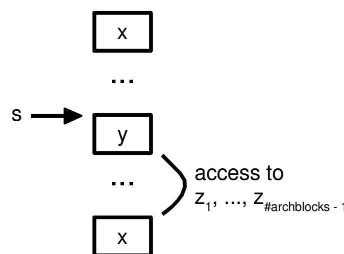


Figure 12.3.: Situation considered by proof of Theorem 12.1

Theorem 12.1
According to LUA, a physical block x will not be replaced between two accesses of x, if less than $|\mathcal{B}_A|$ pairwise different blocks are accessed in between.

Proof: This theorem is proven by contradiction. Let x be replaced by y at a position s between the two accesses to x, as shown in Figure 12.3. Then all architectural blocks must be mapped at the position s. Furthermore, let the other architectural blocks be mapped

to $z_1, \ldots, z_{|\mathcal{B}_A|-1}$. Since x is replaced, the physical blocks $z_1, \ldots, z_{|\mathcal{B}_A|-1}$ must be accessed between s and the second access of x. All in all, $|\mathcal{B}_A|$ physical blocks need to be accessed between the two accesses to x, in order to replace x. This is a contradiction to the condition that less than $|\mathcal{B}_A|$ other physical blocks are accessed. □

12.1.2. Definition of Affinity

The affinity between two virtual registers x and y corresponds to the estimated number of reconfiguration instructions which are avoided if x and y are assigned to the same physical block. Here we introduce an equivalent definition based on the register accesses in a given function and the employed replacement strategy (see Section 12.1.1). By these means, the affinities can be computed from the results of our analysis of n-liveness (see Chapter 11). The definition of the affinity is based on unordered pairs of accesses:

Definition 12.2 (Access pair)
Let x and y be two virtual registers and let x_i and y_j uniquely identify all occurrences of x and y in access operations within the code of a function, respectively. The unordered pair $\{x_i, y_j\}$ is called an access pair if there is an execution path from x_i to y_j or from y_j to x_i and x_i and y_j have the following properties:

1. The number of pair-wise different virtual registers which are accessed between x_i and y_j is less than $|\mathcal{B}_A|$.
2. x and y are not accessed between x_i and y_j.

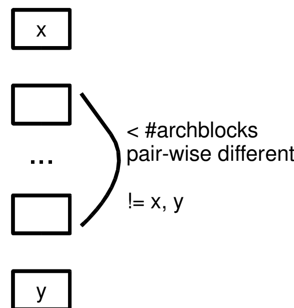


Figure 12.4.: Definition of access pair

Definition 12.3 (Affinity of virtual registers)
Let x and y be two virtual registers. Then the affinity $\alpha(x, y)$ between x and y is defined as the number of access pairs $\{x_i, y_j\}$.

Assume that the probabilities of register accesses are considered and y is $|\mathcal{B}_A|$-live at x_i with probability p_{ij}. Then the affinity between x and y is set to the sum of p_{ij} for all access pairs $\{x_i, y_j\}$.

Equivalence of Definitions We argue that Definition 12.3 corresponds to our intuitive formulation of the affinity modelling the number of avoided reconfiguration instructions (see beginning of Section 12.1). Let x and y be two virtual registers and $\{x_i, y_j\}$ be an access pair (see Figure 12.4). Obviously, less than $|\mathcal{B}_A|$ other physical blocks are accessed between x_i and y_j, because less than $|\mathcal{B}_A|$ virtual registers are accessed according to Definition 12.2. If x and y are assigned to the same physical block, P cannot be replaced between x_i and y_j according to Theorem 12.1, i.e. no further reconfiguration is needed to access y_j. If x and y are assigned to different physical blocks, it *might* be necessary to insert a reconfiguration instruction between x_i and y_j.

Obviously, the first condition of Definition 12.2 can be reduced to computing the $|\mathcal{B}_A|$ pair-wise different virtual registers accessed after each access v_i, i.e. the set of $|\mathcal{B}_A|$-live registers (see Chapter 11). The second requirement ensures that each prevented reconfiguration instruction is counted only once. Hence, we avoid an over-estimation of the affinities. Further information are given in Section 12.1.4, which also presents the algorithm to compute the affinities.

Estimation of Loop Iteration Counts In order to reduce the effort in reconfiguration at run-time, reconfiguration instructions should not be inserted into frequently executed basic blocks, if possible. Up to now, the affinities only model the number of prevented reconfiguration instructions if each basic block is executed once. This is called the *static* affinity α_s. For maximum preciseness, a *dynamic* affinity α_d is used which corresponds to $\alpha_s * \omega(b)$, where $\omega(b)$ is the iteration count of a basic block (see Section 11.1.2). Consequently, reconfiguration within a frequently executed loops becomes very costly in order to compute a mapping of virtual register to physical blocks with minimum reconfiguration costs.

12.1.3. Affinity to Physical Blocks

During the register allocation process, some of the virtual registers have already been assigned to physical registers. Hence, the affinity analysis associates those virtual registers with the corresponding physical blocks. This is similar to replacing the nodes of the virtual registers by nodes of the related physical blocks in the affinity graph.

Our goal is to allocate the registers in such a way that each physical block contains mainly those registers with a high number of contemporary accesses. Obviously, re-computing the affinities everytime after the allocation of a physical register improves the expressiveness of the affinity graph. Thereby, physical blocks and virtual registers are treated interchangeably. The affinity between a virtual register v and a physical block P represents the number of avoided reconfiguration instructions, if v is assigned to P. For simplification, we still assume that affinity graphs only consist of virtual registers in the other sections.

Example Figure 12.5 (a) shows an affinity graph whose virtual registers have not been allocated yet. If the virtual registers c, d, and e are assigned to the physical block P, we get the affinity graph in Figure 12.5 (b). c, d, and e have been replaced by P, while a and f now have non-zero affinities to P instead of c, d, e. For simplification, the new affinities are not shown.

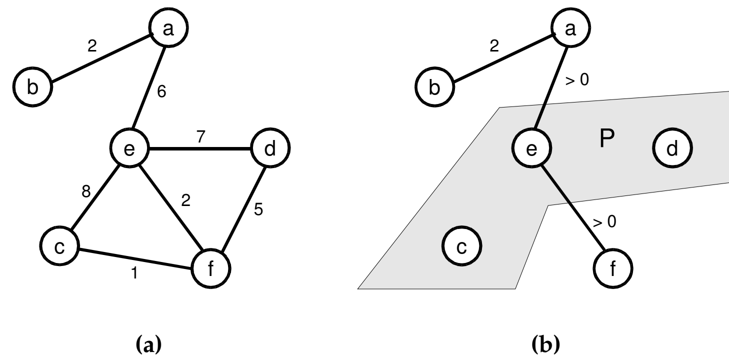


Figure 12.5.: Association of virtual registers with physical blocks in affinity graph

New Access Pairs The definition of the affinity based on access pairs (see Definition 12.3) does not make any assumptions, which virtual registers are assigned to a common physical block. Consequently, the access pairs must be computed pessimistically, i.e. all virtual registers are assumed to be mapped to pair-wise different physical blocks, if no register has been allocated yet. This implies that two accesses x_i, y_j only form an access pair, if less than $\mathcal{B}_A$ *virtual registers* are accessed between them. But if some virtual registers have been assigned to common physical blocks, new access pairs can emerge as demonstrated in the following.

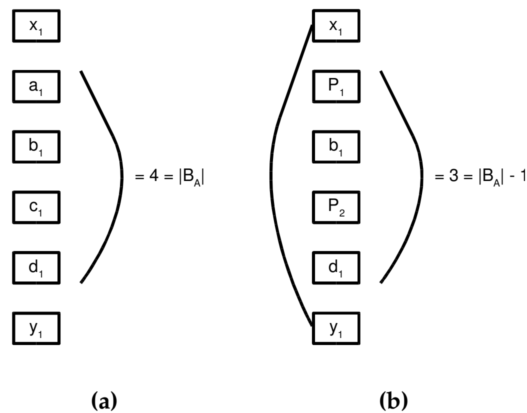


Figure 12.6.: Appearance of new access pairs

The two accesses x_1 and y_1 in Figure 12.6 (a) do not form an access pair, because $\mathcal{B}_A$ pair-wise different virtual registers are accessed in between. If the virtual registers a and c are assigned to physical registers of the physical block P, $\{x_1, y_1\}$ becomes a new access pair, because less than $\mathcal{B}_A$ physical blocks are accessed between x_1 and y_1. As a benefit, the affinity between the register values x and y can be estimated more precisely. On the other hand, the result of the allocation process depends on the order in which the virtual registers are colored.

12.1.4. Construction of Affinity Graph

The edges of an affinity graph are weighted with the affinities between virtual registers. In order to compute the desired affinities, all access pairs need to be determined for each pair of virtual registers (see Definition 12.3). Importantly, the two requirements of Definition 12.2 have to be met.

Our analysis of n-liveness (see Chapter 11) can determine the $|\mathcal{B}_A|$ pair-wise different virtual registers which are accessed next after each access v_i. This corresponds to the first condition of Definition 12.2. Let x_i and y_j be accesses of virtual registers x and y. Concretely, less than $|\mathcal{B}_A|$ pair-wise different virtual registers are accessed between x_i and y_j, if y is $|\mathcal{B}_A|$-live at x_i or x is $|\mathcal{B}_A|$-live at y_j, respectively. The representation of the n-liveness property by access trees (see Section 11.1.1) preserves information about the order of register accesses. If both x and y are *not* accessed between x_i and y_j, the second condition is fulfilled, also. Consequently, $\{x_i, y_j\}$ is an access pair.

Figure 12.7 shows several examples for access pairs of virtual registers x and y. Two accesses are connected by a line, if they form an access pair. In (a) and (b), $\{x_i, y_j\}$ are access pairs, because x is n-live at y_j or y is n-live at x_i, respectively. Hence, we get two access pairs $\{y_j, x_i\}$ and $\{x_i, y_k\}$ in (c). On the other hand, $\{x_i, y_k\}$ do not form access pairs in (d) and (e), because y_j occurs between them, respectively. In case of (d), y is a predecessor of x in the access tree for y_k and therefore y must be accessed between y_k and x_i. Linking x_i and y_k in (e) is avoided, because a given access x_i is only connected with the first access of y succeeding x_i.

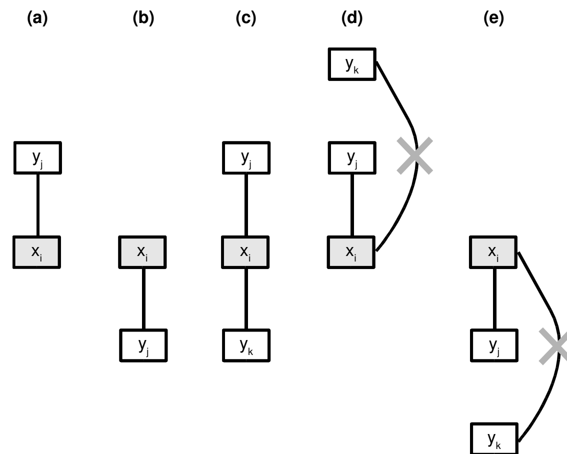


Figure 12.7.: Examples of access pairs

Algorithm In the following, we explain the algorithm to compute the affinities using n-liveness information, whereas $n = \mathcal{B}_A$ (see Algorithm 12.1). The fundamental idea is to traverse the instructions of a basic block in backward order according to our DFA for computing the n-liveness property (see Section 11.2). Let o be a register operand of an instruction at a program position s. Then, $\{o, r\}$ is an access pair for all registers r which are (i) n-live at s and (ii) are not successors of o in the correponding access tree. Hence, the affinity between o and such r will be incremented.

At the beginning of the algorithm, the n-liveness analysis computes the n-live virtual registers for the start and end of each basic block (l. 1). Then the algorithm iterates over all basic blocks (ll. 2-15) and handles the instructions in reverse order (ll. 4-14). For each basic block, the n-liveness information is initialized with the access tree at the end of that block (l. 3). The access tree for each program position in a basic block is determined by successive prepending of the found register operands (l. 12).

Lines 5-13 consider the operands o of an instruction at a program position s. Given a fixed o, lines 6-11 iterate over the n-live registers r according their order with respect to s. For each r, the affinity between o and r is increased (l. 10) by the execution frequency of a basic block (see Section 12.1.2). If the algorithm encounters the register o, it proceeds with the next access of the current instruction in order to ensure the second condition of Definition 12.2.

Require: Machine instructions of a function
Ensure: Affinities between all virtual registers

1: analyze n-liveness
2: **for all** basic blocks b **do**
3: $nlive_regs = OUT(b)$
4: **for all** instructions i in b **do** ▷ last to first
5: **for all** operands o of i **do**
6: **for all** $r \in nlive_regs$ **do** ▷ original order
7: **if** $r == o$ **then** ▷ ensure second requirement of Definition 12.2
8: break ▷ leave loop
9: **end if**
10: $\alpha(o, r) \mathrel{+}= \omega(b)$ ▷ increase affinity between o and r
11: **end for**
12: $nlive_regs = \{o\} \cup nlive_regs$ ▷ o is n-live now
13: **end for**
14: **end for**
15: **end for**

Algorithm 12.1: Algorithm for the computation of the affinities

Ordering of Accesses Up to now, we assumed implicitly *totally ordered* sequences of register accesses. Typically, instructions have multiple operands and the read or write accesses are mostly performed simultaneously, respectively. Hence, the accesses within an instruction are only partially ordered and the replacement strategy cannot determine the latest access.

In order to enable the feasibility of LUA, we decided to re-use the order of the register operands in the instructions. An alternative would be to use the temporal order of register accesses as the first criterion and the order of the register operands as the second criterion. Obviously, this compromise should yield a more precise modelling in case of a VLIW schedule for multiple processors.

For completeness, we consider an example which outlines the effects of the total order on the replacement of physical blocks. Figure 12.8 (a) shows a sequence of instructions, where each operand corresponds to a physical block and the annotated list represents the currently active blocks. Depending on the order of x and y in the serialization, we get the total orders

from Figure 12.8 (b) and Figure 12.8 (c). Consequently, an access of block d implies that block y is replaced in (b) or block x in (c), respectively. In both cases, a reconfiguration is needed before the last instruction.

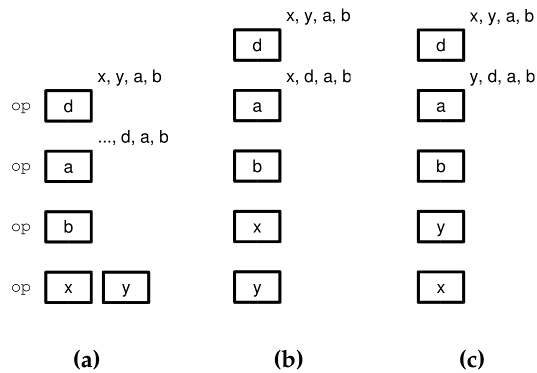


Figure 12.8.: Relation between replacement and total order of accesses

12.1.5. Improvement of Allocation

A proper heuristic for allocating physical registers based on the affinity graph must fulfill two important issues: At first, physical blocks with a large number of unused registers should be favoured. Otherwise, the physical registers would be allocated according to the order of their blocks. Concretely, the register allocation would start with assigning physical registers of P_0 and proceeds with P_1, when all registers of P_0 have been allocated and so on. Simultaneously, the heuristic should minimize the number of used physical blocks instead of distributing register values to all available blocks.

For the sake of contradiction, assume a naive strategy which selects that physical register p for a virtual register v, where the affinity between v and the physical block P containing p is maximal. Obviously, such heuristic would not fulfill the first goal, because the affinity of a virtual register v and a physical block P without any allocated register would be 0. As a solution, the affinity could be divided by the number of allocated registers of a physical block. But this would not achieve the second goal, because the register allocation would try to allocate a similar number of registers of each physical block.

Rating Function We decided to use a powerful heuristic which considers both the number of free registers in a physical block as well as the affinity to those virtual registers, which have not been allocated yet. In the following, the rating function of our heuristic is explained incrementally.

Let α_P be the original affinity between a virtual register v and a physical block P. Define α_{max} as the maximum affinity between v and P. For each access v_i, there can be at most two access pairs for v_i and the physical block P. Hence, the maximum affinity is

$$\alpha_{max}(x) = 2 * \sum_i \omega(v_i)$$

α_{sum} is the sum of affinities between the virtual register v and all remaining non-allocated virtual registers. Further, let f be the number of free physical registers in P. Last but not least, $c \geq 0$ is a constant to control the rating. For simplification, we assume $c = 1$ at first in order to outline the fundamental idea of the heuristic.

Now, the rating function is defined as:

$$rate = \alpha_{max} - \frac{\alpha_{max} - \alpha_P}{1 + f * \alpha_{sum} * c}$$

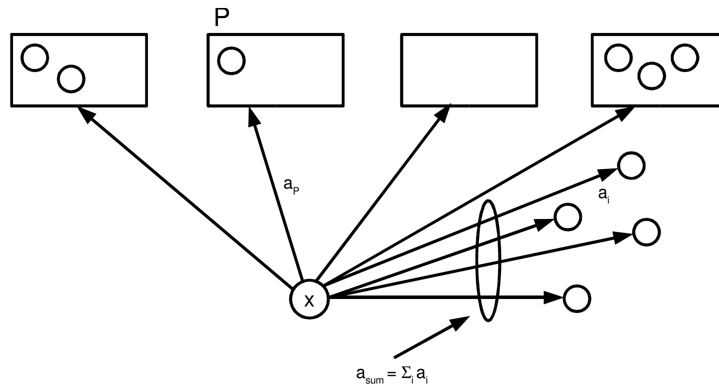


Figure 12.9.: Weighting of empty physical blocks to achieve an uniform distribution

Obviously, the rating increases with rising f or α_{sum} towards the upper bound α_{max}. The value of the rating function corresponds to the lower bound α_P for $f = 0$ or $\alpha_{sum} = 0$. Hence, the rating function ensures that the affinity is included in the interval $[\alpha_P, \alpha_{max}]$.

Let us focus on the parameter f first and neglect α_{sum}. For simplification, assume two physical blocks P and P' with equal values for α_P, α_{max}, and α_{sum}. Furthermore, let P have many unused registers, while f is small for P'. With respect to P, the value of the fraction will become quite small and therefore the rating will have a large value. Instead, the rating function will yield a small value for P', because the fraction becomes large. Hence, P will be favoured over P'.

The intention of α_{sum} is to attenuate the effect of dividing through f in order to minimize the number of used physical blocks. Assume that the affinity of v to those virtual registers V, which have not been allocated yet, is large. Hence, v and V should be stored in a common physical block in order to minimize the effort in reconfiguration. Consequently, P is preferred as mentioned above. But if α_{sum} is small, a physical register of P' may be assigned to v in order to reduce the total count of used physical blocks. This behaviour is modelled by weighting f with α_{sum} in the rating function. Note that α_{sum} is independent of any physical block and the value of the fraction is inversely proportional to the α_{sum}.

Now, we focus on the c parameter which is used to influence the combination of f and α_{sum}. For $c = 0$, the rating corresponds to the original affinity between v and P. For large

c, the rating mainly depends on the number of free registers and the affinities of the non-allocated virtual registers.

12.2. Reconfiguration

The register allocation of CAPRICoRn (see Section 12.1) replaces the virtual registers of each function by physical registers of the machine. Such physical registers cannot be accessed directly in our register architecture (see Chapter 10). Here we describe the second phase of CAPRICoRn that maps the physical registers to architectural ones and inserts appropriate reconfiguration instructions, while aiming at minimizing the reconfiguration overhead. The quality of generated code also depends on the first phase, which allocates registers of a common physical block for two virtual registers that are often accessed close together.

For simplification, Section 12.2.1 first outlines the placement of reconfiguration code for single basic blocks. Obviously, this strategy cannot achieve a cost-minimal placement of reconfiguration instructions with respect to whole functions in general. Section 12.2.2 presents an extension of this method which tries to minimize the effort in reconfiguration by re-using mappings established in preceding basic blocks. Finally, Section 12.2.3 discusses the effects of function calls on the mappings.

12.2.1. Intra-Block Reconfiguration

The fundamental idea of the algorithm is to keep track of the mapping function μ_A while iterating over all instructions of a basic block. At the beginning of a basic block, the mapping function rbμ_A is pessimistically assumed to be undefined for all architectural blocks, i.e. $\forall A : \mu_A(A) = \bot$ holds.

If an instruction i accesses a register of an inactive physical block P, a reconfiguration instruction to activate P is inserted directly before i. Such reconfiguration establishes a mapping $\mu_A(A) = P$, where A was either unmapped or mapped to a physical block $P' \neq P$ that is latest used again. Importantly, mappings are established *as late as possible*.

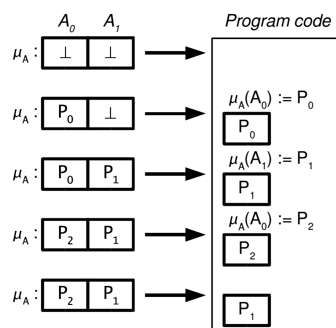


Figure 12.10.: Integration of reconfiguration instructions and replacement of physical registers by architectural registers

Example In the following, we consider a small example to illustrate the functionality of our algorithm: Figure 12.10 shows the code of a basic block on the right hand side with accesses to the physical blocks P_0, P_1, P_2. On the left hand side, the current mapping function μ_A is indicated for a register architecture with two architectural blocks A_0, A_1.

Since the architectural blocks are assumed to be unmapped at the beginning, a reconfiguration instruction $\mu_A(A_0) = P_0$ needs to be inserted at the very beginning of the basic block, to activate P_0. The same applies for the second reconfiguration instruction prior to the access of P_1.

The third reconfiguration instruction needs to replace a mapping, since both architectural blocks are already mapped to physical blocks. According to the LUA strategy, the mapping $\mu_A(A_0) = P_0$ is replaced, as P_0 is latest used again (see Section 12.1.1).

12.2.2. Inter-Block Reconfiguration

If no mappings can be assumed at the beginning of a basic block, *at least one* reconfiguration instruction must be added for *each* accessed physical block. Obviously, many reconfiguration instructions can be saved, if mappings established in preceding basic blocks are re-used. In the first place, such mappings result from accesses to physical blocks in predecessors. Additionally, reconfiguration instructions can be moved explicitly to directly preceding basic blocks, if beneficial. As a result, reconfiguration instructions are moved out of loops into a block dominating the loop.

Here we present a powerful placement strategy, which consists of two phases: At first, the mappings at the beginning and end of a basic block are determined such that the estimated reconfiguration costs are minimized. The second phase inserts reconfiguration instructions for each basic block separately using a similar approach as outlined in Section 12.2.1 and the mappings computed in the first phase. In the following, we use the terminology introduced in Section 10.1.1.

12.2.2.1. First Phase

IN and OUT Mappings From now on, we refer implicitly to direct predecessors and successors of a basic block if no further information are given. A physical block P can only be assumed to be active at the beginning of a basic block, if $A \in \mu_P(P)$ for the *same* architectural block A at the end of *all* preceding basic blocks (see Figure 12.11). This requires the computation of so-called IN and OUT mappings between all basic blocks. The IN mapping $\mu_{IN\,b}$ of a basic block b contains all configurations $\mu(A, P)$ which can be assumed at the beginning of b. Accordingly, the OUT mapping $\mu_{OUT\,b}$ specifies the mappings that *must* be provided at the end of b for its successors.

In order to assume a mapping $(A, P) \in \mu_{IN\,b}$ for a basic block b, it must hold $(A, P) \in \mu_{OUT\,p}$ for all predecessors p of b. If A is mapped to different physical blocks in μ_{OUT} of some predecessors, the mapping of A is undefined ($\perp$) in $\mu_{IN\,b}$. This can be expressed formally as:

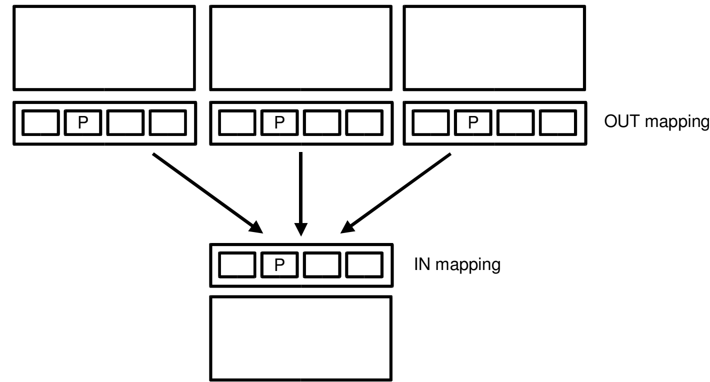


Figure 12.11.: Determination of mappings between basic blocks

$$\mu_{IN_b} = \bigcap_{p \in pred(b)} \mu_{OUT_p}$$

where $\cap$ is defined as

$$\mu_x(A) \cap \mu_y(A) = \begin{cases} P & \text{for } \mu_x(A) = \mu_y(A) = P \\ \bot & \text{else} \end{cases}$$

Selection of Physical Blocks The definition of μ_{IN_b} aims to reduce the number of reconfiguration instructions at the beginning of a basic block b by re-using mappings established in preceding basic blocks p. Obviously, the n-liveness analysis (see Chapter 11) can also be applied to determine the set of physical blocks used in the future with respect to a certain program position. If a physical block P is $|\mathcal{B}_A|$-live at the beginning of a basic block b, a mapping $\mu_{IN_b}(A) = P$ may be defined, if possible. Accordingly, mappings $\mu_{OUT_p}(A) = P$ are established for all predecessors p of b. The $|\mathcal{B}_A|$-liveness guarantees that P will not be replaced from the beginning of b up to the first access of P (see Theorem 12.1). Hence, a reconfiguration instruction is definitely avoided in b.

Effect on Predecessors On the other hand, a mapping $\mu_{OUT_p}(A) = P$ for a predecessor p of b can introduce an *additional* reconfiguration instruction in p. In order to predict, wether a reconfiguration instruction must be added to ensure $\mu_{OUT_p}(A) = P$, the maximum cut of life spans between the last access of P and the end of p is regarded. Thereby, $(A, P) \in \mu_{IN_p}$ or $\mu(A, P) \in \mu_{OUT_p}$ are handled as the first or last accesses to P in p, respectively. If the maximum cut is smaller than $|\mathcal{B}_A|$, the physical block P will persist until the end of p according to LUA (see Section 12.1.1). Thus, no additional reconfiguration instruction is required for p. Consequently, $\mu_{IN_b}(A) = P$ may be defined, if the total overhead in reconfiguration is minimized.

Estimation of Costs The disadvantage of an additional reconfiguration instruction in a predecessor is expressed in terms of costs. The costs of a reconfiguration instruction in a

block b are given by its execution frequency $\omega(b)$ (see Section 12.1.2). Hence, the costs for a mapping $\mu_{IN\,b}(A) = P$ correspond to the sum of costs of additional reconfiguration instructions in the preceding basic blocks. If these costs are lower than the costs of a reconfiguration in b, $\mu_{IN\,b}(A) = P$ is defined.

Computation of IN and OUT Mappings The first phase is realized as a topological traversal of the the CFG, which computes $\mu_{IN\,b}$ for a basic block b and $\mu_{OUT\,p}$ for its predecessors p. Such information is utilized by the second phase to place reconfiguration code according to Section 12.2.1.

12.2.2.2. Second Phase

In the second phase, reconfiguration instructions are placed into each basic block separately. Here, we outline only the differences to the approach presented in Section 12.2.1. The set of mappings which are available at the beginning of a basic block b is initialized with $\mu_{IN\,b}$. If $(A, P) \in \mu_{IN\,b}$ for a physical block P, no reconfiguration instruction needs to be inserted to activate P at the beginning of a basic block b.

If a physical block P must be made accessible in a basic block b, our heuristic is aimed at selecting an architectural block A such that $\mu(A, P) \in \mu_{OUT\,b}$ as a secondary goal. The primary criterion is still based on the LUA strategy (see Section 12.1.1). Remember that $(A, P) \in \mu_{OUT\,b}$ is regarded as the last access of P in a basic block b. At the end of b, all non-active mappings $(A, P) \in \mu_{OUT\,b}$ are established. For instance, there may be a physical block P which is not accessed in b and $(A, P) \notin \mu_{IN\,b}$. Hence, P needs to be activated explicitly in b to provide $\mu(A, P)$ for its successors.

Selection of Architectural Blocks The secondary criteron mentioned above may replace physical blocks $\mathcal{U}$ which are used again in the same basic block b. Thereby, $(A, P) \in \mu_{OUT\,b}$ is *not* regarded as a feasible access of P. Hence, establishing mappings demanded by $\mu_{OUT\,b}$ could be counter-productive to minimizing the effort in reconfiguration. As a consequence, such heuristic is only applied for physical blocks $\hat{\mathcal{U}} = \mathcal{B}_P \setminus \mathcal{U}$.

Let $\mathcal{F}$ be the set of architectural blocks A, which are mapped to physical blocks $P \in \hat{\mathcal{U}}$ or are unmapped. If $\mathcal{F}$ is not empty, the selection of an architectural block A to activate a physical block P operates according to the following order as a secondary criterion:

1. If $(A, P) \in \mu_{OUT\,b}$ and $A \in \mathcal{F}$, A will be selected.
   Hence, the heuristic first tries to select an architectural block A which should be mapped to P according to $\mu_{OUT\,b}$. As A is not mapped to a physical block P' that is used again in b, no additional reconfiguration instruction is needed to replace P later. Importantly, P will not be replaced until the end of b according to LUA, because $(A, P) \in \mu_{OUT\,b}$ is treated as the last access of P in b.

2. If $(A, P') \notin \mu_{OUT\,b}$ for any physical block P' and $A \in \mathcal{F}$, A will be selected.
   In this case, A is irrelevant for the OUT mapping and connecting P to A does not imply penalty costs also. P may be replaced later according to LUA, because $(A, P) \notin \mu_{OUT\,b}$.

3. Otherwise, an arbitrary $A \in \mathcal{F}$ with $(A, P') \in \mu_{OUT b}$ for a physical block $P' \neq P$ will be selected.
Here, a reconfiguration instruction will be needed to establish the mapping demanded by $\mu_{OUT b}$ later. Please note that P' is not accessed again in b because of $A \in \mathcal{F}$ and the mapping to P' is only required by successors of b.

12.2.3. Inter-Procedural Reconfiguration

Up to now, we considered only intra-procedural placement of reconfiguration instructions. In case of multi-procedure programs, many challenges such as the access of special registers as well as the scope of mappings between register blocks need to be considered.

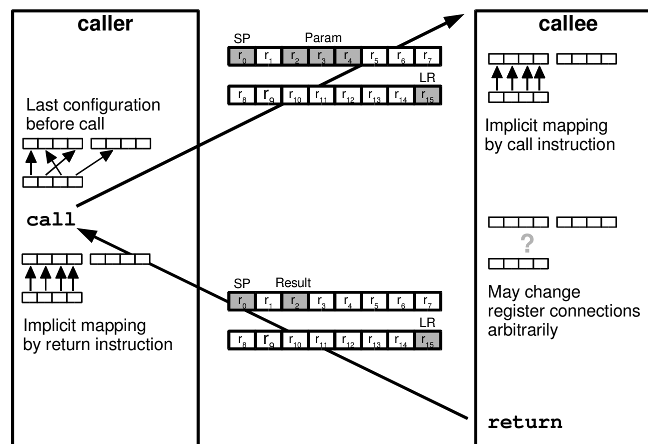


Figure 12.12.: Reconfiguration between functions

Special Registers Obviously, using the same set of physical registers for parameters, stack pointer, and return address as in the non-reconfigurable case simplifies the register allocation. This agreement enables hybrid machine programs consisting of functions either using or not using register reconfiguration. But mapping these special register values (permanently) into certain architectural registers is neither necessary nor beneficial, because it does not matter in general which architectural register is used for an access. Furthermore, a frequently used special register like the stack pointer would be permanently active without any visible benefit. Consequently, the compiler does not need to handle these registers in a special way.

Implicit Register Accesses In most microprocessors, some machine instructions such as function calls access special registers like the link register implicitly. As those implicit accesses to special registers are not encoded in an instruction, no operands can be replaced by architectural registers. A naive solution would be to establish an identical mapping between a physical register p_i and the corresponding architectural register a_i before the access. However, this would require major changes of the affinity analysis which only considers the register accesses encoded as operands. Consequently, implicit accesses to special registers are directly forwarded to the physical registers for simplification.

Mapping Convention The mapping convention specifies the mappings which can be assumed at the beginning of a function or after returning from a function call. In principle, a caller could provide mappings needed by a called function and vice versa. By these means, reconfiguration instructions could be moved out of frequently executed functions into their callers. For instance, if a function g is called iteratively by a function f, reconfiguration instructions may be placed before the corresponding loop in f, instead of the beginning of g.

But such strategy would demand costly inter-procedural analysis which can be performed only in a quite restricted way at compile-time. Furthermore, the CAPRICoRn approach has been developed for single functions and would be complicated unnecessarily.

Our Approach The CAPRICoRn compiler supports two different strategies: The first mode assumes that the mappings are in an undefined state at the beginning of a function or after returning from a function call. Consequently, the called or calling functions have to reconfigure the register banks according to their own purposes and cannot trust in different functions, respectively. The benefit of this solution is the simplification of the calling convention and its implementation in the compiler.

The second mode features implicit reconfiguration of the register banks during a function call or backward branch, respectively. This idea is a good compromise between providing certain mappings to save reconfiguration instructions as well as exploiting the delay slots of branch instructions. Especially, the access to special registers such as function parameters can be accelerated significantly.

Currently, such implicit reconfiguration establishes identical mappings for each processor. Assume n processors with p physical blocks and a architectural blocks, whereas the physical blocks are shared among all processors (see Section 10.1). Then the architectural block A_i of a processor is connected with the physical block P_i of the first processor, for $a \leq p$. If $a > p$, the architectural blocks $A_i, i \in \{0, \ldots, p-1\}$ are mapped to the physical blocks P_i of the first processor, while $A_i, i \in \{p, \ldots, 2 * p - 1\}$ are connected with the physical blocks P_{i-p} of the second processors, etc.

12.3. Extensions for Multi-Cores

Up to here, CAPRICoRn has been discussed for a single-processor architecture. This section outlines how the method can be extended for a VLIW machine or a multi-core like the QuadroCore. In the following, we refer to multi-core architectures for simplification and use the terms *processor*, *core*, or *functional unit* interchangeably.

Multi-Core Architectures Figure 12.13 shows an exemplary multi-core architecture with reconfigurable registers. It consists of a shared physical register file that can be accessed uniformly by every processor. Each core has its own set of architectural registers to establish its mapping independently.

Read accesses are performed in the first half of a cycle, while write accesses occur only in

the second half. The combination of this property with the presence of reconfigurable register banks enables the generation of very dense schedules with aggressive re-use of registers as outlined later.

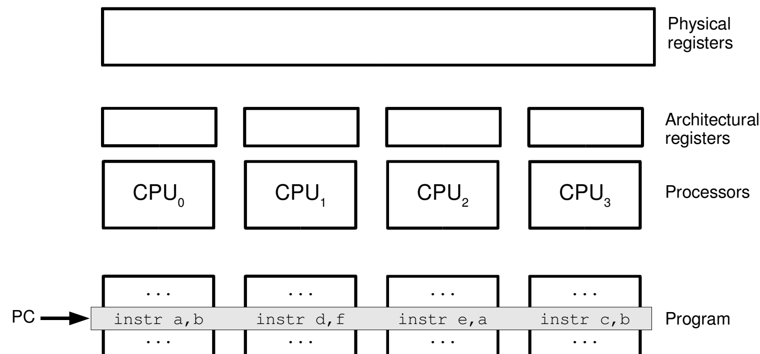


Figure 12.13.: Multi-Core architecture

Alternatively, Section 10.2 discusses a more general register model where the physical registers are partitioned into banks of the processors. This model also enables to distinguish between local and remote accesses to registers. Furthermore, the number of ports of a register bank can be restricted which bounds the number of simultaneous accesses. An overview of the required extensions to the register allocation and the re-scheduling is given in the referred section.

In this section, a homogenous architecture is considered in order to simplify our explanations. Furthermore, we refer to a machine with synchronous cores implicitly, unless no different statement is made. When using an asynchronous execution model, barrier instructions can be inserted by the re-scheduling phase (see Section 4.2.3) to avoid conflicts when multiple processors access physical registers concurrently.

Register Allocation As the processors access a shared register file, a single conflict graph has to be constructed to consider all life spans simultaneously. Such conflict graph is used to allocate the physical registers for all processors together. Parallel instructions are treated as a single VLIW instruction, whereby the conventional algorithms for single processors can be applied.

The affinity analysis is performed for each core separately, because every processor has a dedicated architectural register bank. Hence, we defined that register accesses by different processors can never be access pairs (see Section 12.1.2). As the coloring is performed for all processors together, a single affinity graph is constructed. Thereby, the affinity between two virtual registers x and y is computed separately for each processor and summed up afterwards.

Structure The remaining part of this section is structured as follows: In Section 12.3.1, the placement of reconfiguration code is outlined for a multi-core. We demonstrate by using an example that inserting reconfiguration instruction may be impossible due to the mentioned density of schedules and discuss several solutions. Section 12.3.2 deals with the re-

scheduling of code after using CAPRICoRn.

12.3.1. Reconfiguration

As each processor has an own set of architectural registers, reconfiguration instructions can be inserted for each processor separately using the approaches presented in Section 12.2. Consequently, `nop` instructions must be added to the other processors in order to preserve the consistency of the schedule.

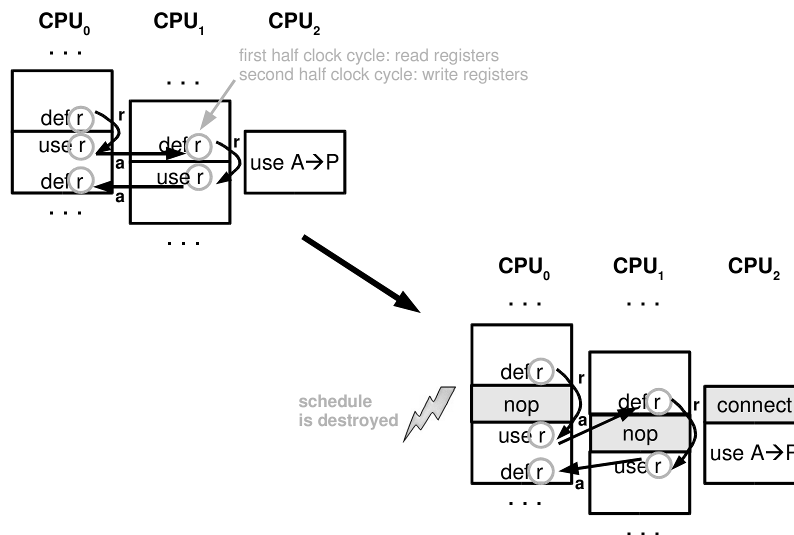


Figure 12.14.: Infeasible placement of reconfiguration instructions

But the frequent re-use of physical registers can lead to very dense schedules, where any modification may harm the semantics as illustrated by Figure 12.14. The original schedule is correct, because registers are read and written in different halfs of a cycle as defined at the beginning of this section. Now assume that a reconfiguration instruction which maps an architectural block A to a physical block P is inserted before the use of such a mapping on processor 2. As a result, `nop` instructions need to be added between the instructions on processors 0 and 1, respectively. Obviously, such modification destroys the schedule, because the value x is overwritten by processor 1 before processor 0 has read it.

Surprisingly, such phenomenon of unmodifiable schedules has not been described yet in literature to our best knowledge. Conventional VLIW compilers must have faced similar challenges, if schedules needed to be altered after register allocation.

In the CAPRICoRn compiler, a situation as presented above is avoided by extending a life span at its final use u by one-half clock cycle. Hence, a definition d on another processor which occurs one-half clock cycle later than u will access a different register than u.

Figure 12.15 illustrates the effect of this strategy: When a register allocation is performed for the schedule in the upper left corner, the same register r is assigned to the virtual registers x and y (upper right). If the life span of x is extended as described above (lower left), it overlaps with the life span of y starting at the definition. Hence, different registers are allocated for x and y (lower right). With respect to the example in Section 12.2, the anti-dependences

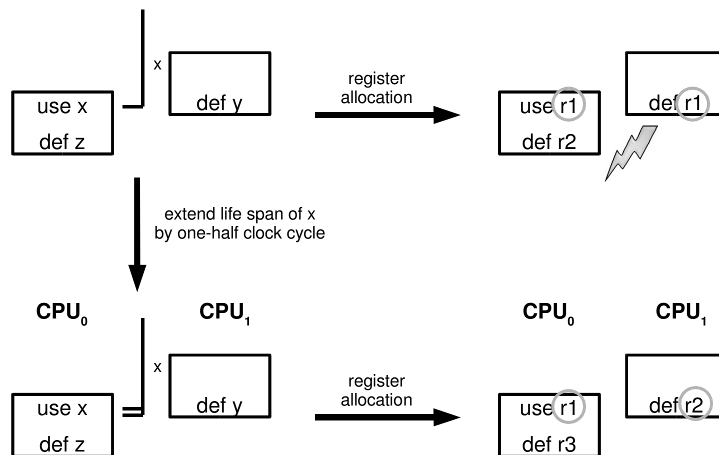


Figure 12.15.: Extension of life span by one-half clock cycle

between processors 0 and 1 vanish, when applying this strategy. As a drawback of this technique, the register pressure is increased. This issue is discussed in Section 12.3.2.2.

Alternatively, the actual coloring and the placement of reconfiguration instructions could be executed iteratively. The iteration might start with the best register allocation followed by the insertion of reconfiguration instructions and the re-scheduling. If changes to a schedule are not possible, the register allocation can be performed again with worse re-use of registers until a proper solution is found.

12.3.2. Re-Scheduling

Section 6.2.3 motivates the need for a re-scheduling phase after the register allocation with respect to the QuadroCore. When using CAPRICoRn, reconfiguration instructions are inserted separately for each processor (see Section 12.3.1). The resulting empty slots on the remaining processors are filled with `nop` instructions. Consequently, the overhead in register reconfiguration may be larger than the speedup achieved by avoiding spilling and using the shared registers for a fast communication.

In this section, we focus on re-scheduling of code generated for machines with reconfigurable register banks. At first, we explain the basic concepts of constructing a DDG for such code (see Section 12.3.2.1). The aggressive re-use of registers may cause cyclic dependences which need to be handled by the scheduler. Section 12.3.2.2 discusses two approaches.

If the CAPRICoRn compiler targets an architecture with multiple, asynchronous processors, the re-scheduling phase must also place barrier instructions as outlined in Section 6.2. Inserting synchronization code may face similar problems as adding instructions to reconfigure the register connections (see Section 12.3.1). Hence, such challenges can be tackled as described there.

12.3.2.1. Construction of Data-Dependence Graph

Machine code produced by CAPRICoRn for machines with reconfigurable register banks has two outstanding properties: Physical registers are only accessed indirectly through architectural registers. The mappings between architectural and physical blocks are modified by executing special reconfiguration instructions. Both features have to be considered when re-scheduling such machine code after the register allocation. Our re-scheduling phase (see Section 4.2.3) performs a complete scheduling based on a DDG for the current schedule.

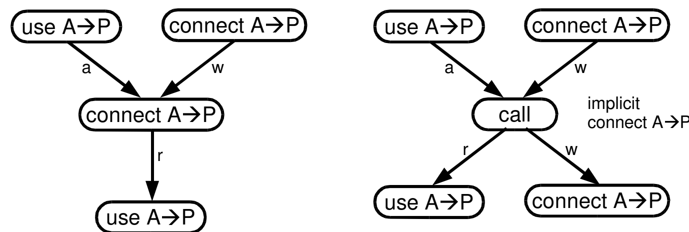


Figure 12.16.: Construction of data-dependence graph

The DDG models the indirect usage of physical registers as well as reconfiguration of block mappings. Figure 12.16 illustrates the fundamental ideas. For simplification, the examples do not distinguish between different architectural blocks A and physical blocks P. A reconfiguration instructions which establishes a mapping between an architectural block A and a physical block P is denoted as $A \to P$. Other instructions are called normal instructions.

At first, we ignore function calls temporarily and focus on the left-hand side of the picture. Clearly, normal instructions can be regarded as uses of mappings between architectural blocks A and physical blocks P. Importantly, this comprises both uses *and* definitions of *physical* registers. From the perspective of CAPRICoRn, definitions correspond to reconfiguring the block mappings. A reconfiguration $A \to P$ is always anti-dependent on a normal instruction using a mapping for A, because A will point a different physical block P' after the reconfiguration. Two reconfigurations for an architectural block A are write-dependent on each other. Finally, a normal instruction using a mapping for A is data-dependent on a modification of such mapping.

Calling a function g from a function f may alter the register connections established by f. Hence, the effects of function calls on the block mappings have to be taken into account (see right-hand side of Figure 12.16). As the re-scheduling is performed for single functions, we just have to consider function calls. Basically, a function call destroys mappings of the caller and can also define new mappings by implicit reconfiguration (see Section 12.2.3). Like a reconfiguration, a function call is anti-dependent on normal instructions as well as write-dependent on prior reconfigurations. Last but not least, succeeding instructions using block mappings are data-dependent on function calls.

12.3.2.2. Scheduling with Cyclic Dependences

As motivated above, the reconfiguration of registers enables very dense schedules by aggressively re-using physical registers. Obviously, this behaviour can cause cyclic dependences between instructions which need to be respected during re-scheduling.

Surprisingly, we have not found any approach which can deal with such requirement, although re-scheduling code for VLIW machine should handle similar constraints. In terms of the QuadroCore, the problem only occurs when using reconfigurable register banks. Without reconfiguration, register values are transported by special instructions between the processors (see Section 6.1).

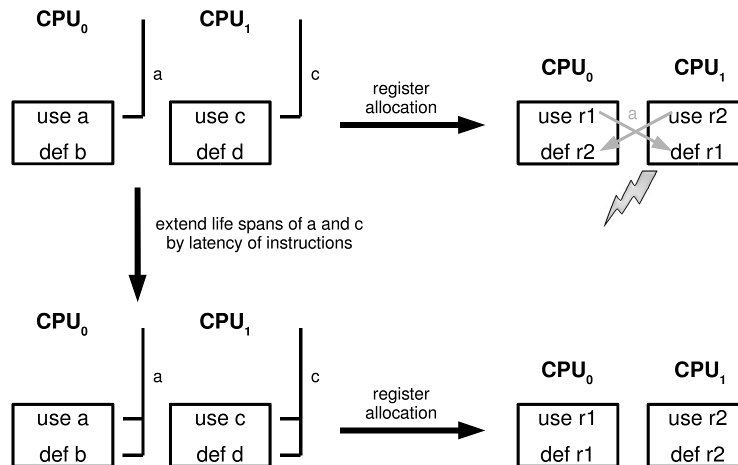


Figure 12.17.: Extension of life span by latency of instruction

The current prototype of the CAPRICoRn compiler avoids cyclic dependences by extending life spans at their final use u by the latency of the instruction corresponding to u. The fundamental idea has already been described in Section 12.3.1). Figure 12.17 motivates the utilization of the latency. If the registers are allocated without lenghtening the life spans accordingly (upper left corner), the two instructions are anti-dependent on each other. Such cyclic dependences are avoided when applying the mentioned extension of life spans. Extending the life spans by the latency is crucial to ensure that use and definition on different processors access different registers.

Unfornately, the number of needed registers may be increased significantly which impedes the benefits of reconfiguring the register banks. This situation can be improved in two different ways: At first, the compiler may extend the life spans only where necessary by analyzing the register accesses precisely.

A much better strategy is to augment the capabilities of the re-scheduler in order to handle cyclic dependences properly: The scheduler could identify parts of a schedule exhibiting the mentioned constraints. This can be realized by determining the Strongly Connected Components (SCC) of the DDG. During the re-scheduling, the arrangement of the operations of each SCC must be preserved, while the remaining parts may be re-ordered.

Part V.

Evaluation and Conclusion

13. Evaluation

Contents

13.1 Simulation of QuadroCore **V-5**
13.1.1 Specification and Implementation V-5
13.1.2 VLIW/Barrier Reconfiguration V-7
13.1.3 Description of Benchmarks V-9
13.2 Parallelizing Compiler **V-10**
13.2.1 Scheduling and Optimization V-12
13.2.2 Synchronization V-20
13.2.3 Processor Partitioning V-30
13.2.4 Communication V-33
13.3 SIMD/MIMD Reconfiguration **V-38**
13.3.1 Execution Time V-39
13.3.2 Ratio of SIMD/MIMD Execution V-42
13.3.3 Code Size V-46
13.3.4 Power Consumption V-47
13.3.5 Costs of Reconfiguration V-50
13.3.6 Costs of Communication V-50
13.4 Reconfigurable Register Banks **V-51**
13.4.1 Execution Time V-53
13.4.2 Code Size V-56
13.4.3 Power Consumption V-58
13.4.4 Costs of Reconfiguration V-59
13.5 Combination of SIMD/MIMD and Register Bank Reconfiguration **V-60**
13.5.1 Execution Time V-61
13.5.2 Ratio of SIMD/MIMD Execution V-61
13.5.3 Code Size V-62
13.5.4 Power Consumption V-63
13.5.5 Costs of Reconfiguration V-65

This chapter discusses the evaluation of the CoBRA[1] approach, which has been characterized in Chapter 3. At first, we present the edge conditions of our experiments in Section 13.1. This comprises information about the simulation of the QuadroCore as well as an overview of the considered benchmark programs. The entire evaluation is split into four sections: Section 13.2 analyzes the parallelizing compiler for the QuadroCore (see Part II) with respect to execution time, code size, communication, synchronization, and data partitioning. The results of the SIMD/MIMD reconfiguration (see Part III) are appraised in Section 13.3, while Section 13.4 presents the evaluation of the register reconfiguration (see Part IV). Finally, the effect of both reconfiguration dimensions is studied in Section 13.5.

Overview of Results Our experiments have been conducted using excerpts from practical audio and video applications. Parallelizing a sequential program for the QuadroCore yields speedups of up to factor 4, as expected. The relative run-time costs for communication of at most 20.2% (about 6% in average) accounts for a reconfiguration of the registers. A further argument for that reconfiguration dimension is the unbalanced register pressure on the processors of the multi-core due to the parallelization for some benchmarks.

The SIMD/MIMD reconfiguration has demonstrated its effectiveness in improving the resource efficiency in terms of execution time, code size, and energy consumption. More than 50% of the execution of some benchmarks are performed in SIMD mode. The utilization of a single SIMD code stream obtains a reduction of the code size by up to 22.66%. Similarly, the number of instruction memory accesses decreases by up to 46.08%, while the switching activity of instruction fetch and program counter can be reduced by up to 53.32% and 44.69%, respectively. Even where the observed speedups are rather small, the savings in code and power constitute for an application in a resource-constrained device. With respect to a mobile phone, the comparatively small memory is utilized best and the battery runtime is maximized. When a short execution time is preferred over small code size and low energy consumption, vectorizing for multiple code streams can bring additional speedups of up to 25%. For instance, aggregation network access nodes transcoding large amounts of video and audio data demand maximum performance while code size and power consumption play a minor role. The relative overhead for reconfiguring the machine at run-time is negligible and does not exceed 1.5% in most cases (maximum 2.5%). Our evaluation has shown that the overhead of the SIMD/MIMD reconfiguration is mainly caused by transport code between scalar and vector registers. In some cases, the dynamic number of `mov` instructions is even increased by factor 3.7 compared to a vectorization generating multiple code streams. On the other hand, a single code stream helps in reducing the waiting times due to processor synchronization by up to factor 2.

The register reconfiguration yields speedups of up to 27.04% by reducing the overhead of inter-processor communication. For benchmarks with an unbalanced high need in registers for certain processors, the execution is even accelerated by more than factor 10. Such improvement may be lower for other machines, because spilling can represent a severe bottleneck of the QuadroCore as mentioned below. Again, the register reconfiguration can optimize the resource efficiency in terms of code size and energy consumption, also. For benchmarks with a high communication overhead but no or modest spill costs, the code size is reduced by up to 15.01%. Avoiding spill code even obtains savings of 88.39% and 66.54% in

[1]**Co**mpiler-**D**riven **D**ynamic **R**econfiguration of **A**rchitectural Variants (merge two *D*s to a *B*)

code size and stack accesses, respectively. The relative overhead of the register reconfiguration is only 1.38% in average. An architecture with four architectural blocks per processor with 4 registers each is an optimal choice with respect to our benchmarks.

The joint application of both reconfiguration dimensions has been conducted using a few benchmarks with low parallelism, because the CoBRA compiler is still in a prototypical stage. Nevertheless, we have shown that the two techniques consort excellent with each other. The average speedup compared to a MIMD scheduling with preceding loop unrolling is about 35%, while the maximum acceleration observed is even 45.4%. Importantly, each technique enables to improve the results of the other dimension significantly. Utilizing a single SIMD code stream reduces the code size by up to 16.61%, while the three indicators of energy consumption exhibit savings of 33-34%. The total overhead of reconfiguration corresponds to the costs of each technique on its own mentioned above. We believe that technical improvements to the CoBRA compiler can demonstrate similar results for larger applications with a higher degree of parallelism also.

Our evaluation has also shown that the QuadroCore has two main bottlenecks: At first, the waiting times at barriers can dominate the execution of programs and therefore hide effects of applied techniques like reconfiguration. The current prototype of the CoBRA compiler tries to minimize the number of barriers, but does not optimize for short waiting times. For a practical implementation, the compiler must estimate the waiting times more precisely, or a completely different execution model like VLIW or Explicitly Parallel Instruction Computing (EPIC) may be used.

Secondly, spilling of registers to the stack memory can be very expensive due to the small offset range of the S-Core memory instructions. Long sequences of increment/decrement operations need to be inserted before and after the actual spill instructions.

13.1. Simulation of QuadroCore

The fundamental structure and properties of the QuadroCore have been introduced in Section 4.1. Here, we present some details about our machine and introduce the studied benchmarks. At first, Section 13.1.1 outlines the most important instructions or groups of instructions of the QuadroCore. Furthermore, some key data about the hardware implementation are given. Section 13.1.2 motivates a synchronous execution mode of the QuadroCore in order to customize it to code with fine-grained parallelism. By default, we utilize a reconfiguration between synchronous and asynchronous execution. Finally, Section 13.1.3 presents the benchmarks used in the following sections.

13.1.1. Specification and Implementation

Specification Most arithmetic/logical instructions of the QuadroCore are performed in a single clock cycle. The sole exceptions are multiplication (18 cycles) and division (35 or 34 cycles for signed or unsigned, resp.) operations. Branch and call instructions consume about 2 or 3 cycles.

A common set of memory instructions is utilized to access both local and external mem-

ory. Three different types of load/store instructions are provided for 8, 16, or 32-bit width, respectively. Accessing the local memory always takes 3 clock cycles. An access to the external memory is managed by a bus arbitration, which works in a round-robin fashion. As the waiting time for the bus is 3 clock cycles, the costs of accessing the external memory vary between 6 and 15 cycles. Obviously, the waiting time can be at most 12 clock cycles, when a processor is stalled until the three other processors have finished accessing the external memory.

Inter-processor communication is realized by a dedicated register bank (see Section 6.1.1), which can be accessed by the `cldw` and `cstw` instructions in two clock cycles. The single-cycle `crsync` instruction (see Section 4.2.4) copies the result of a comparison to the compare registers of all processors.

Synchronization among a subset of the processors is achieved via the `barrier` instruction (see Section 6.2.2), which can be performed in a single clock cycle. Hence, the minimum waiting time at a barrier is one cycle, if all processors have reached the barrier.

Both SIMD/MIMD reconfiguration and register reconfiguration are controlled by two single-cycle instructions called `execmode` and `connect`, respectively. In SIMD mode, wide memory accesses are performed by two special instructions `wideload` and `widestore`, which take 6 clock cycles each. They operate on a dedicated vector register, which can be read/written by two further instructions `readsreg` and `writesreg`. Both operations need two clock cycles to transfer a single register value between entry c of the vector register and a register of processor c.

Last but not least, the implementation of spilling registers needs to be mentioned, because it has influenced the results of our evaluation significantly. In principle, spilling a register is realized by writing its content to the stack frame of a processor, which is stored in its local memory. Unfortunately, address offsets of S-Core memory instructions can be encoded with only 5 bits. Hence, a register cannot be spilled with a single memory instruction using the stack pointer as a base address in many situations. Instead, our compiler increases the stack pointer appropriately before such memory operation and decreases it afterwards. This behaviour may result in a large number of additional `addi/subi` instructions, which increase execution time and code size further.

As an improvement, the spill variables may be stored at the beginning of the stack frame to minimize the offsets to the stack pointer. Then, the compilation process must be split into two different phases: At first, the backend is executed until the register allocation has been finished and the number of spill variables is known. In the second phase, such spill variables are placed at the beginning of a stack frame, followed by the remaining local data structures.

Alternatively, the compiler may even estimate the number of accesses to spill variables and local data structures. Hence, the stack data can be arranged such that variables which are accessed frequently get a lower stack offset.

Implementation In principle, the evaluation can be based on three different implementations: At first, we have a cycle-accurate software simulator which is generated partly from a machine specification (see Section 4.2.2). Furthermore, the evaluation can be performed using a hardware simulator based on a synthesized model of the QuadroCore. Such model can

be mapped to an FPGA based prototyping environment [93] for rapid evaluation of large benchmarks on hardware. More information about the hardware implementation can be found in [87].

We decided to conduct the majority of the evaluations which considers the execution of benchmark programs by using the software simulator. This enabled the automatic evaluation of a large number of benchmarks with varying compiler parameters. Additionally, the current hardware prototype does not provide a reconfigurable register architecture according to our CAPRICoRn approach, because each S-Core processor can only switch between its two banks. Hence, a processor cannot access the physical registers of another processor by reconfiguration, but register values still need to be transported via the slow shared memory (see Section 6.1). Furthermore, such restricted switching of whole register banks implies the usage of a single architectural block per processor with CAPRICoRn. As the S-Core is a 2-address machine, at least two architectural blocks are required to access all operands of an instruction in general, if no inter-bank copy operation is provided (see Section 10.1.3). Instead, our software simulator supports variable-sized architectural and physical blocks in order to avoid such additional copy operation and to evaluate the influence on the number of blocks on the overall performance. A processor can access the registers of *all* processors within a single cycle by reconfiguration to speedup inter-processor communication.

Area and Performance Estimation The synthesized prototype was used to estimate properties of the hardware implementation. Table 13.1 shows the variations in terms of maximum operating frequency (the clock period), area, total dynamic power, and mW/MHz, comparing the original multi-core architecture with the reconfigurable implementation. The architecture was synthesized in UMC 130nm standard cell technology using the design compiler by Synopsys. Obviously, the synthesized architecture shows an increase of about 10% of area. The maximum operating frequency of the system is altered by about 5%. The reduction in the dynamic power calculations is attributed to the reduction in operating frequency of the reconfigurable multiprocessor, as can be seen by the fact that the mW/MHz ratio stays constant for both architectures.

Architecture	Clock Period (ns)	Area (sq mm)	Total Dynamic Power (mW)	mW / MHz
original	4.74	0.77	42	0.2
reconfigurable	5.00	0.85	40	0.2

Table 13.1.: Standard Cell Synthesis Reports

13.1.2. VLIW/Barrier Reconfiguration

By default, the processors of the QuadroCore operate independently and synchronize using barriers only where necessary. In the following, such execution is denoted as *barrier mode*. Section 4.2.4 motivates that our compiler exploits fine-grained parallelism on basic block level. Hence, the overhead of synchronization will harm the benefits of an asynchronous execution, if the number of inter-processor dependences is large. As a consequence, we decided to augment our architecture by a *VLIW mode*. Synchronous execution also matches the usage of a VLIW model by the parallelization phase of the CoBRA compiler (see Chapter 4).

Basically, such VLIW mode would require synchronizing the execution of the processors according to a global clock. Assume that a processor x needs c_r cycles for executing an instruction, but the compiler has assumed $c_p < c_r$ cycles. Then, the other processors would need to be stalled for $c_r - c_p$ cycles to ensure that the schedule is executed as computed by the compiler. Similarly, processor x needs to be stalled for $c_p - c_r$ cycles, if $c_p > c_r$, i.e. the compiler made a worse assumption.

Simplified Lock-Step Execution Typical VLIW machines have a single instruction decoder, which handles the large instruction words and offers the required properties. As the QuadroCore has been developed for asynchronous execution originally, its processors have separate decoders. Hence, our hardware prototype could not be extended to stall processors, when a processor needs more time than assumed by the compiler. As a consequence, the functionality has also not been implemented in the software simulator to model the capabilities of the hardware implementation precisely. Instead, we decided to realize lock-step execution by disabling the early exit property of instructions like multiplications or divisions.

External Memory Accesses But unfortunately, the number of cycles needed to access the external memory could not be fixed to a certain value, because the access is managed by a bus arbitration (see Section 4.1). Such problem can be avoided easily in the software simulator, if an external access is assumed to take the same number of cycles as an access to a local memory. On the other hand, the evaluation would be quite imprecise and too optimistic compared to the real hardware using this simplification.

In order to use lock-step execution as often as possible when running a program on the hardware, our compiler applies a static analysis to determine the external memory accesses (see Section 4.1). All basic blocks with external memory accesses are executed in barrier mode, while the remaining blocks can be operated safely in VLIW mode. If the accessed memory cannot be determined statically, an external access must be assumed to avoid problems because of the simplified lock-step execution. Please note that the actual scheduling treats such case as a local access to assume a smaller number of clock cycles.

Switching the multi-core from barrier to VLIW mode is achieved by synchronizing all processors. Switching to an asynchronous execution does not require any reconfiguration, because the processors can just continue their execution independently and synchronize explicitly, if necessary.

Reconfiguration of Synchronization Paradigm Consequently, the CoBRA compiler offers a very simple reconfiguration of the synchronization model based on an analysis of the memory accesses. As the current implementation is mainly a by-product of the simplified realization of the lock-step execution mentioned above, we abstained from presenting this opportunity in reconfiguration as a full-fledged part of the CoBRA approach.

In the future, both execution modes can be integrated into our CoBRA approach as two variants, if a more sophisticated method for deciding between the two paradigms is employed. For instance, the compiler may consider the number of inter-processor dependences

to decide between barrier and VLIW mode. Alternatively, it can generate code for both variants and select the best result by a code integration. In contrast to the mentioned variation of the granularity (see Section 1.3), reconfiguration would clearly be needed to switch between synchronous and asynchronous execution. But if the scheduler of the CoBRA compiler exploits fine-grained parallelism only, the barrier mode will probably be outperformed by the VLIW mode in most situations.

13.1.3. Description of Benchmarks

The evaluation has been performed using several medium-sized benchmark programs which can occur in practical audio and video applications. These computational blocks constitute typical transcoding algorithms for aggregation network access nodes.

Parallelism Classes In order to expose enough ILP to the compiler, the benchmarks have been adapted as follows: The data structures were duplicated by factor 2 and 4, while the corresponding computations were copied on basic block level. Such manual adaption is legal, because it can be realized by an automatic transformation of realistic computations. For instance, a function operating on a two-dimensional array may be mapped to separate computations using one-dimensional arrays. By these means, a well-balanced partitioning of both data and operations is enabled using the concepts presented in Chapter 5. In the following, we use the term *class of parallelism* and annotate the names of actual instances of benchmarks with numbers 1, 2, or 4.

Local and Global Data Furthermore, two versions of each benchmark using local or global data structures are maintained, respectively. Local data is stored on the stack of a processor p and can only be accessed by instructions executed on p. Global data is mapped to the external memory, which can be accessed by all processors (see Section 4.1). From now on, we distinguish between such two kinds of benchmarks by using the terms `local` and `global/static`.

Presentation of Benchmarks As a consequence, there are six instances of each benchmark to evaluate different degrees of parallelism and data management. Table 13.2 lists all benchmarks and gives a short description. In order to guarantee a realistic evaluation, the computational part is repeated several times to compensate the time spent in initialization, where necessary.

Usage of Benchmarks The following sections discuss the results of our evaluation using the mentioned benchmarks. The class 4 benchmarks have been favoured for the evaluation of the parallelizing compiler and the SIMD/MIMD reconfiguration, because they achieve the greatest speedup compared to a single processor. In some cases, instances with less parallelism have been chosen, where the highly parallel versions do not work at all. As the register reconfiguration for multiple processors still is in a prototypical stage, we mainly use the class 1 benchmarks. Furthermore, we focus on the speedup compared to a parallelization

Name	Description
convolution	Computes the discrete convolution of a 50 element array with a 16 element array. Hand-crafted.
fftlike	Represents the variable access pattern of a FFT with two arrays of 16 elements each. Hand-crafted.
fir	FIR filter based on [106].
iir	IIR filter based on [106].
imageedit	Applies 3x3 filter to 10x10 image and cuts of margins. Hand-crafted.
median	Computes the median of each n consecutive elements in an array of 50 elements. The computation is executed simultaneously for n = 1, 2, 3, . . . , 16 in a single pass. Hand-crafted.
mm	Multiplies two $n \times n$ matrices, $n \in \{4, 16\}$. Hand-crafted.
poly	Evaluates a polynomial of degree 16 with variable coefficients. Hand-crafted.
sharpening	Sharpening algorithm for images with dimension of 10x10 pixels. Hand-crafted.
vectormuladd	Multiply-accumulate on vectors of 256 elements. Hand-crafted.
vmm	Multiplies an 16×16 matrix with an 16 element vector. Hand-crafted.

Table 13.2.: Description of benchmarks

without register reconfiguration, while the speedup to a single processor is only studied for selected benchmarks.

For some benchmarks such as `sharpening`, there exist only instances with global data structures, which are initialized with constant data. As our simulator has no loader, initialization of global data is performed at the beginning of a program. If a program contains local data structures with static initializers, the initialization will be performed twice: At first, a temporary global memory is defined, which is copied to the local data structure when entering the function. Hence, the instances with local data structures have been neglected.

13.2. Parallelizing Compiler

In this section, we evaluate the parallelizing compiler presented in Part II with respect to four main issues. At first, the benefit of scheduling and optimization on the performance is explored in Section 13.2.1. Section 13.2.2 compares the VLIW/Barrier reconfiguration with the pure barrier mode (see Section 13.1.2) in terms of execution time, code size, and overhead of synchronization. Our data partitioning method (see Section 5.2) is evaluated in Section 13.2.3. Last but not least, Section 13.2.4 deals with inter-processor communication (see Section 6.1).

As this thesis concentrates on SIMD/MIMD and register reconfiguration, we just give an overview of the results and study selected benchmarks only where necessary. The three succeeding sections discussing the evaluation of the two reconfiguration dimensions analyze most benchmarks used there in detail.

Optimization In the following, we often use the notation of Table 13.3 to refer to different optimization strategies. The classical optimizations comprise algebraic transformations,

strength reduction, constant folding and propagation, copy propagation, dead-code elimination, for instance. Loop unrolling is performed according to Section 8.3.2. Duplication of induction variables creates multiple independent copies of the variables and the accessing code in order to reduce the communication costs. Our data partitioning method (see Section 5.2) does not model induction variables as part of an affinity graph. Otherwise, additional edges between two apparently independent sets of variables can occur such that the final result of the partitioning is poor. If the duplication is disabled, induction variables are assigned to the processors by a heuristic.

Strategy	Classical optimizations	Duplication of induction variables	Loop unrolling
opt0			
opt1	×		
opt2	×	×	
opt3	×		×
opt4	×	×	×

Table 13.3.: Optimization strategies

Scheduling We decided to use the As Late As Possible (ALAP) heuristic for our list scheduler, because it is known to achieve a good parallelization for a modest register pressure. Instead, ASAP arranges operations as early as possible without considering the need in registers. A high register pressure leads to additional spill code, which can be very expensive for the S-Core (see Section 13.1.1). Unfortunately, even ALAP has neglected the register pressure in many cases. In particular, the evaluation of the class 4 benchmarks has shown a significant performance loss when using loop unrolling. Such phenomenon can also be observed for `sharpening_static4`, even if loop unrolling is not applied beforehand.

We extended the ALAP heuristic by a secondary criterion temporarily, which favours operations where many life spans end, without visible success. Applying ASAP does not help, anyway, because it increases the register pressure much more. We aim for evaluating more sophisticated scheduling heuristics in the future, which consider the register need more carefully. Whenever spill code is discussed in the following, the reader may refer to Section 13.1.1 for additional information.

13.2.1. Scheduling and Optimization

Here, we discuss the results of the parallelizing compiler using VLIW/Barrier reconfiguration (see Section 13.1.2) for all benchmarks described in Section 13.1.3. The effects of parallelization and optimization are discussed separately. For lack of space, we focus on speedups and neglect the absolute values.

Speedup by Parallelization Figure 13.1 to Figure 13.6 illustrate the speedup of a parallelization compared to a single processor. Thereby, we consider all parallelism classes mentioned in Section 13.1.3 as well as the optimization strategies listed by Table 13.3. In the best case, the execution is accelerated by factor 3.82 (`fir16_local4` with opt2), due to optimization and duplication of induction variables to reduce the communication costs. With respect to the class 4 benchmarks, the average speedup with opt2 is factor 2.47, while class 2 and 1 achieve only 1.7 and 1.1, respectively, because of the lower degree of parallelism. The execution of many class 1 benchmarks is even slower than using a single processor due to poor parallelization as well as additional code for communication and synchronization.

As expected, the instances with local data achieve a greater speedup than those with global data, where the bus arbitration of the external memory represents a bottleneck. Only `fft-like16_local4` has a slightly lower speedup than its global counterpart due to infrequent, but costly spilling. The stack offsets of the spill variables are larger because of the local data in the stack frames (see Section 13.1.1). Without any optimizations (opt0), the number of `addi` instructions of the local instance is increased by 8.2%, while it even has factor 13 more `subi` operations. This big difference is justified by the high number of `addi` instructions in the machine code resulting from code selection. Furthermore, the optimized placement of spill code tries to reduce the number of `addi/subi` instructions.

Unfortunately, loop unrolling (opt3 and opt4) degrades the performance dramatically in many cases, although the parallelization should benefit from the exposed parallelism. With respect to some benchmarks like `convolution16_local4` or `fir16_local4`, multiple processors are even slower than a single core when using loop unrolling. This performance loss is caused by expensive spill code due to the aggressive scheduler neglecting register pressure (see beginning of Section 13.2). In general, the slow-down can be observed for two kinds of benchmarks: Unrolling the loops of highly parallel programs uncovers even more parallelism such that the list scheduler produces code with a very large need in registers. When a benchmark has local datastructures, spill code becomes more costly due to the increased stack offsets compared to instances with global data.

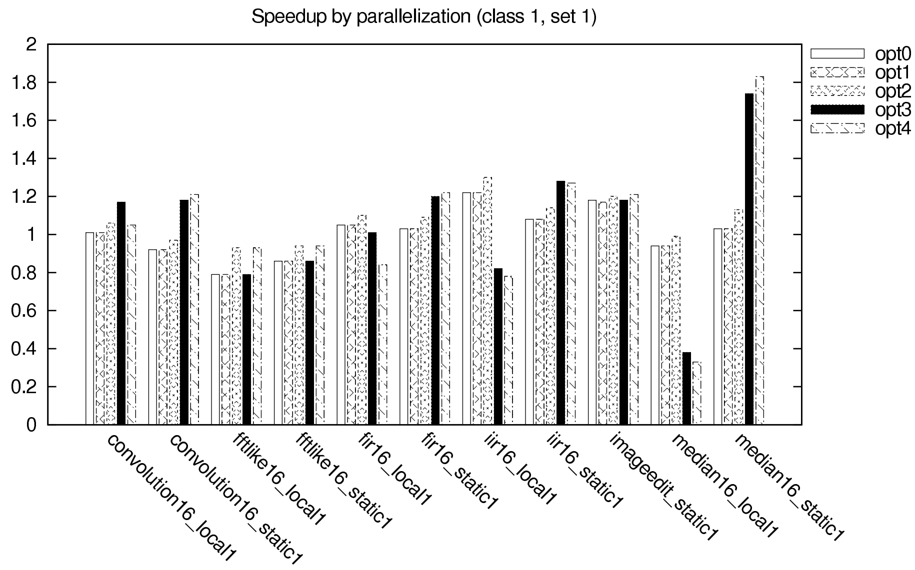


Figure 13.1.: Speedup by parallelization (class 1, set 1)

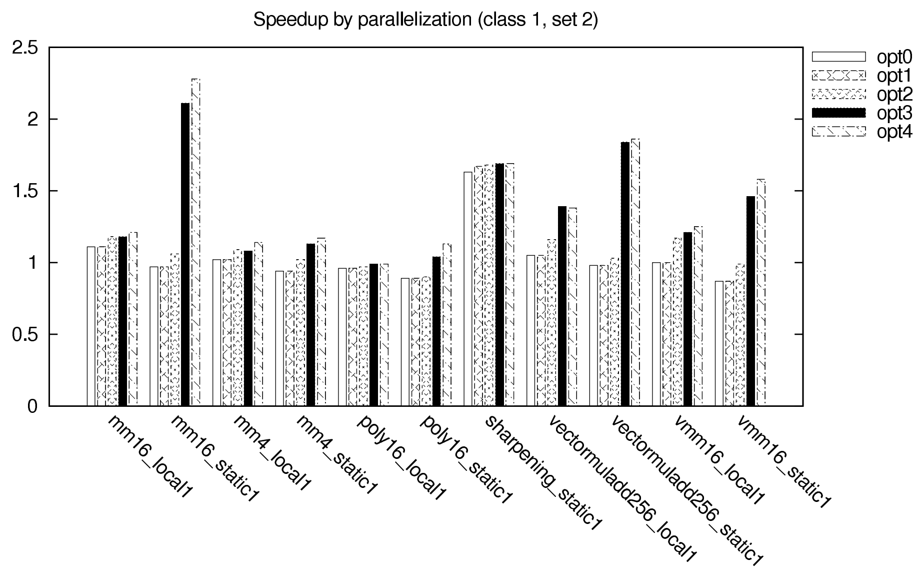


Figure 13.2.: Speedup by parallelization (class 1, set 2)

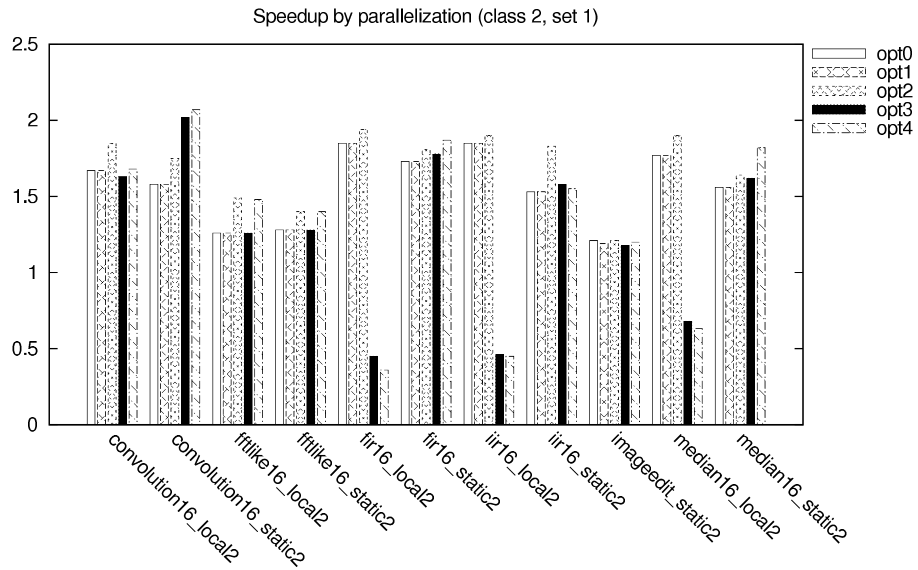


Figure 13.3.: Speedup by parallelization (class 2, set 1)

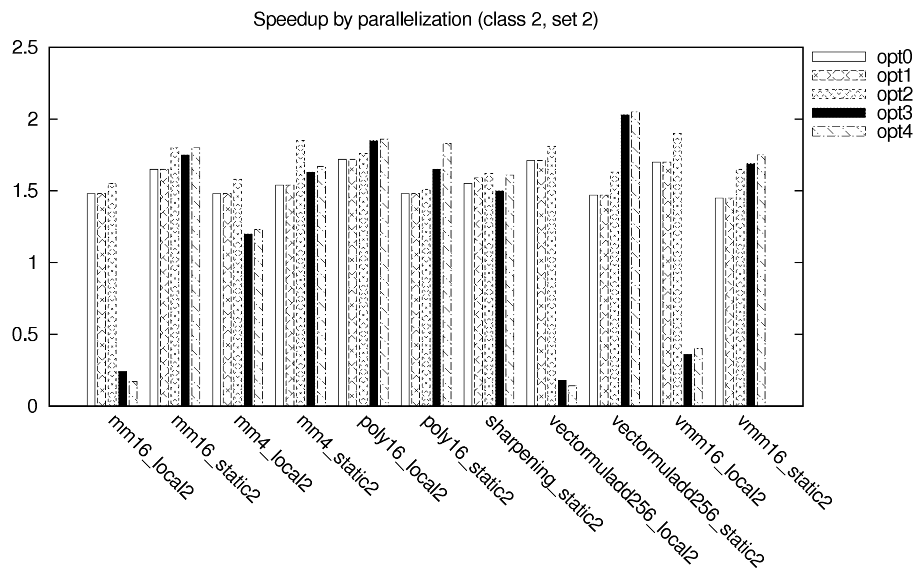


Figure 13.4.: Speedup by parallelization (class 2, set 2)

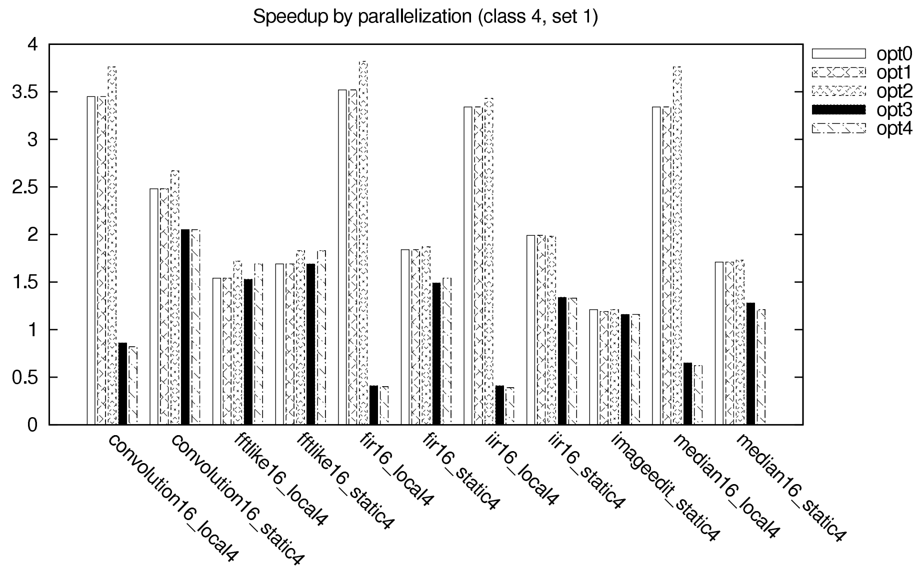


Figure 13.5.: Speedup by parallelization (class 4, set 1)

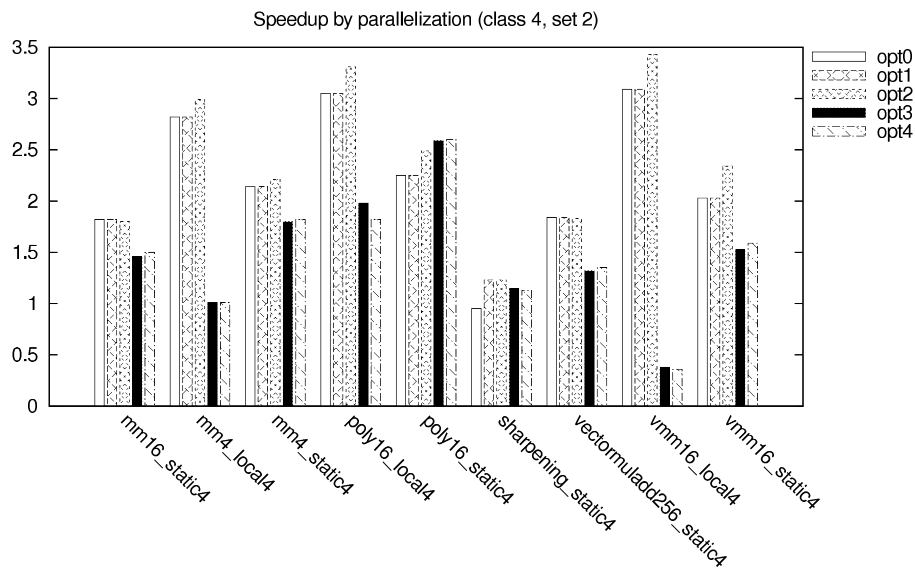


Figure 13.6.: Speedup by parallelization (class 4, set 2)

Speedup by Optimization In the following, the influence of the three optimization strategies (see Table 13.3) is studied more in detail. Figure 13.7 to Figure 13.12 visualize the results. Please note that the improvement is measured in % from now on. Actually, classical optimization (classical) refers to the speedup of opt1 over opt0. In order to evaluate the effect of the duplication of induction variables (indvar), we have compared opt2 and opt1. The difference between opt4 and opt2 represents the influence of loop unrolling (unroll) on the performance.

Classical optimization is beneficial in rare cases only, because the used benchmarks seem

to have only few opportunities for typical compiler optimizations. Solely `sharpening_static4` is accelerated by 28.75% due to a decreased register pressure. As a result, the number of stack accesses decreases by 18.5%, the number of `addi/subi` instructions to increment/decrement the stack pointer for spilling by 35% even. The number of waiting cycles at barriers is reduced by 42.7%. However, the speedup of parallelization compared to a single processor is only factor 1.23 with opt1.

For some benchmarks, classical optimization even causes a performance deterioration: The execution of `imageedit_static4` is slowed down by 1.05%, because the number of waiting cycles increases by 2.6%. The discussion of the results of the two reconfiguration dimensions will demonstrate that the dynamic effect of waiting at barriers can have a significant impact on the performance.

Duplication of induction variables obtains positive speedups of up to 15.32% in most cases. The average improvement for the class 4 benchmarks is 5.8%, while the class 2 and class 1 instances show a mean speedup of 8.35% and 7.28%, respectively. `vmm16_static4` is accelerated by 15.32%, because the relative overhead of the communication is reduced from 8.2% to 5.8%. Concretely, the number of `cstw` and `cldw` instructions decreases by 24.4% and 49.2%, respectively. On the other hand, `mm16_static4` has the largest degradation of 1.22%, because the number of waiting cycles increases by 10%.

With respect to the class 4 benchmarks, the combination of classical optimizations and duplication of induction variables yields an average improvement of 7.19% This speedup may be even much higher, if the need for classical optimizations would have been larger. The class 2 and 1 instances of the used benchmarks have a mean speedup of 8.41% and 7.35%, respectively.

Loop unrolling causes dramatic performance losses in most cases, because the register pressure is raised a lot due to an extremely aggressive parallelization (see beginning of Section 13.2). As the need in registers increases according to the quality of parallelization, the class 4 benchmarks suffer from an average deterioration of 39.31% even. For the class 2 instances, the degradation is reduced to 21.93%. The poor parallelization of the class 1 benchmarks even promotes a speedup by loop unrolling of 15.1%. However, the overall speedup compared to a single processor is only factor 1.24 with opt4 (class 2: 1.32).

The deterioration for the instances with local data (68.53%) is much greater than using the corresponding versions with global data structures (19.85%). The obvious reason is the higher overhead of spilling due to larger stack offsets (see Section 13.1.1).

To summarize, the execution of the class 4 benchmarks can be accelerated enormously by parallelization if loop unrolling is disabled. We believe that a better scheduling heuristic will avoid such significant increase of need in registers when loop unrolling is applied beforehand. From now on, we mostly focus on the relative speedup compared to a normal parallelization to demonstrate that certain techniques like reconfiguration work in principle. For a general application in practice, the improvement of the scheduler is crucial.

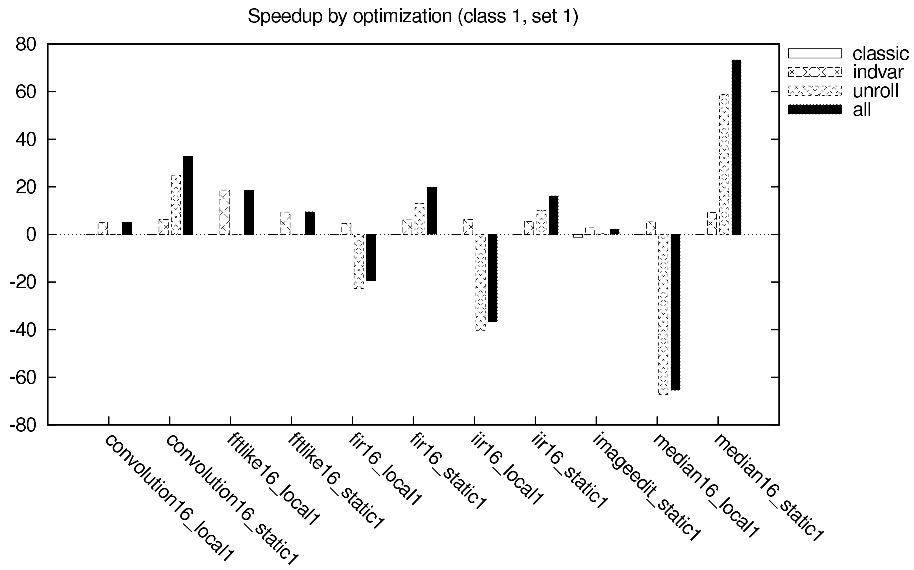


Figure 13.7.: Speedup by optimization (class 1, set 1)

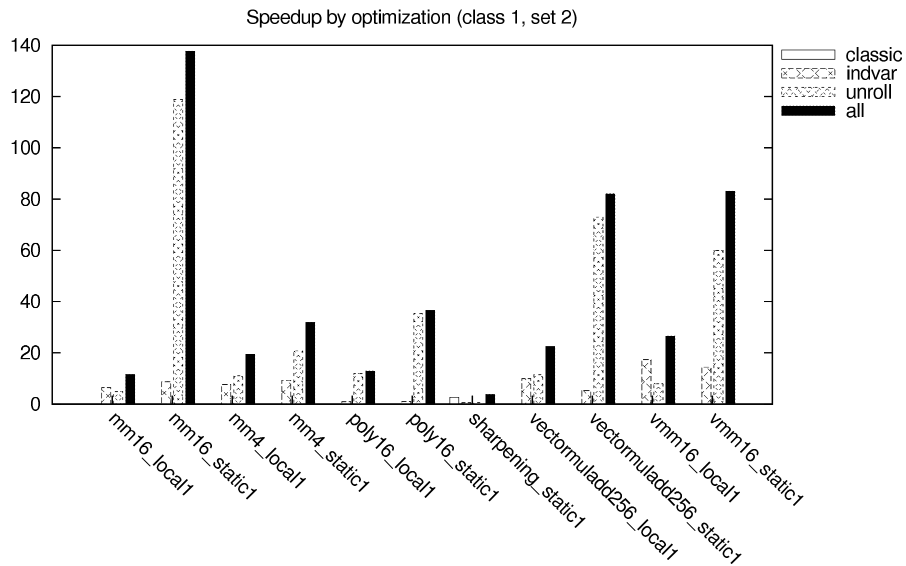


Figure 13.8.: Speedup by optimization (class 1, set 2)

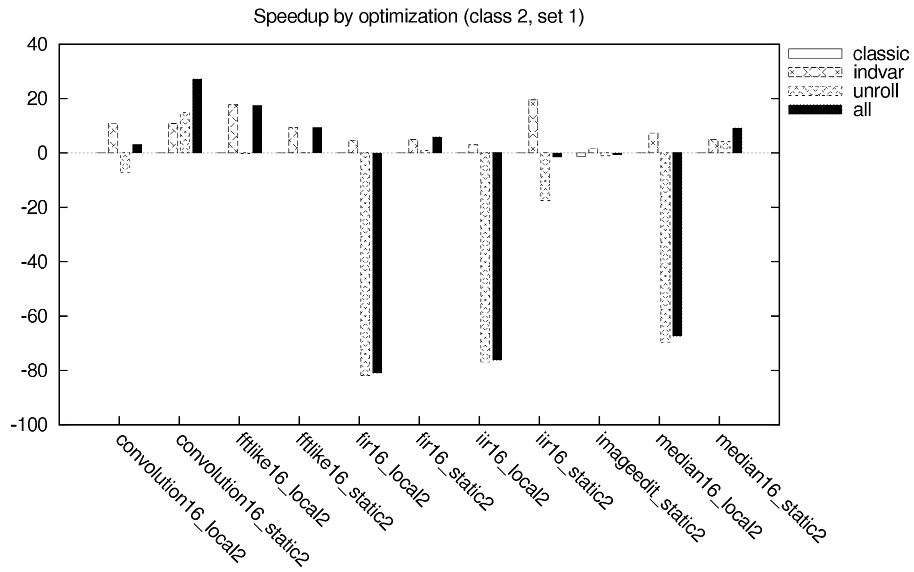


Figure 13.9.: Speedup by optimization (class 2, set 1)

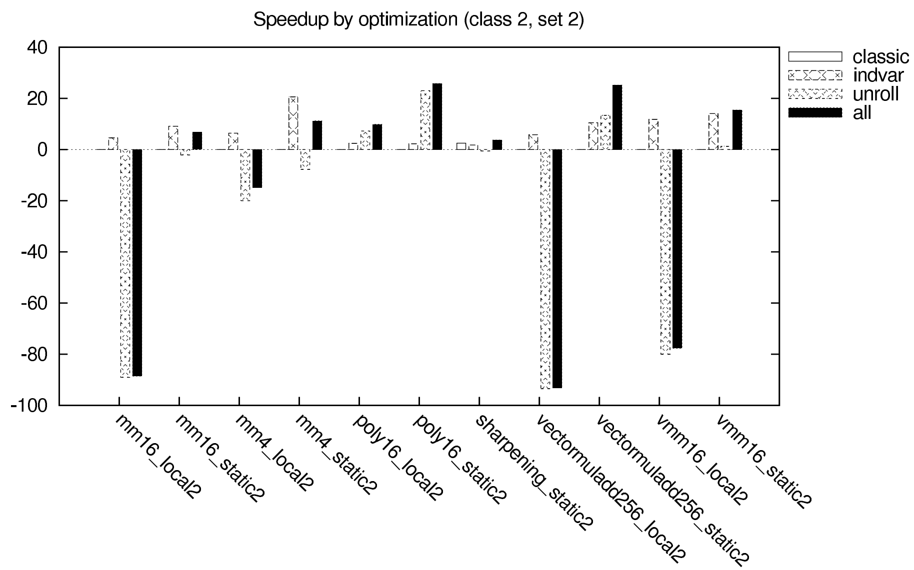


Figure 13.10.: Speedup by optimization (class 2, set 2)

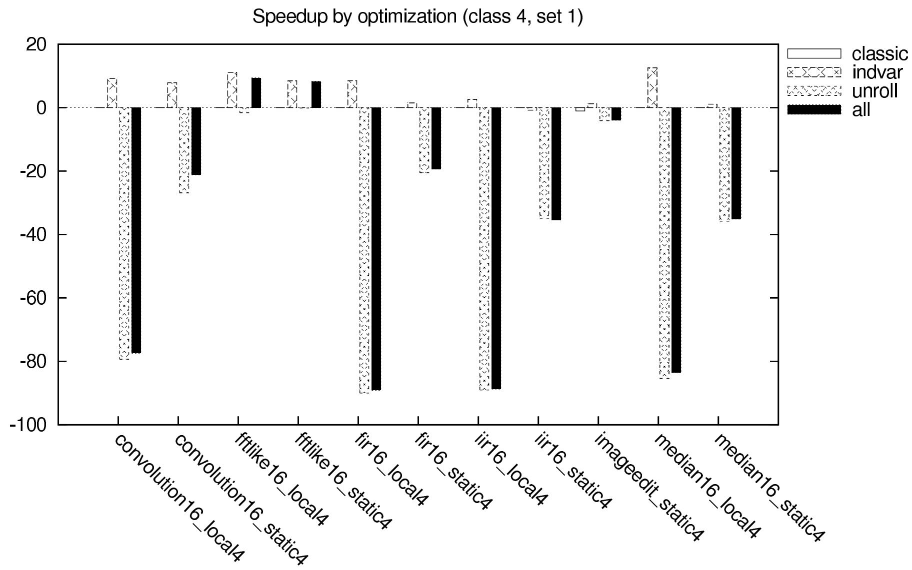


Figure 13.11.: Speedup by optimization (class 4, set 1)

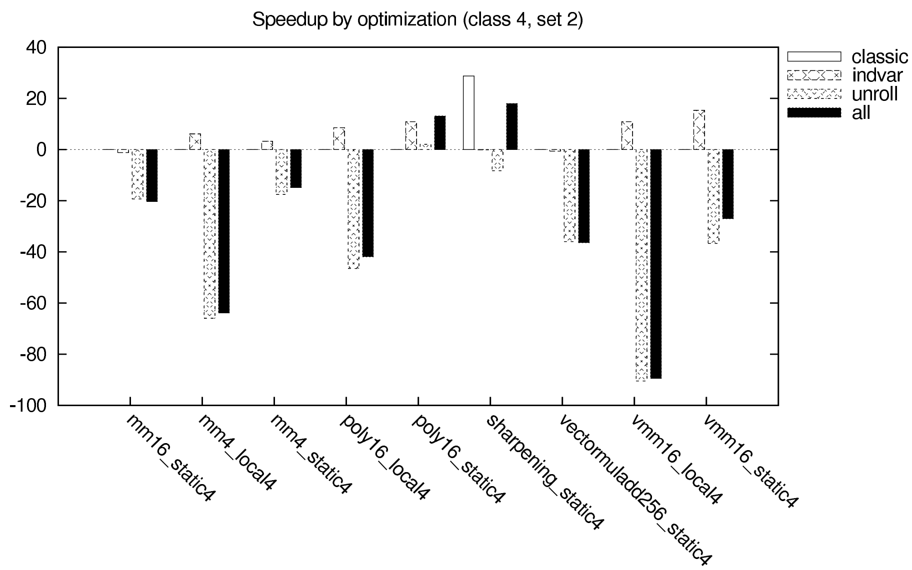


Figure 13.12.: Speedup by optimization (class 4, set 2)

13.2.2. Synchronization

The scheduler of the CoBRA compiler only exploits fine-grained parallelism within basic blocks. Hence, an asynchronous execution (barrier mode) may suffer from the overhead for synchronization in many cases. As a consequence, the CoBRA compiler parallelizes code for the VLIW/Barrier reconfiguration (see Section 13.1.2) by default. Lock-step execution (VLIW mode) is chosen for a basic block, when it does not contain any external memory accesses whose latency is not known statically due to the bus arbitration. In this section, we compare the VLIW/Barrier reconfiguration with the pure barrier mode in terms of execution time, code size and overhead of synchronization.

Execution Time Figure 13.13 to Figure 13.18 show the speedup of the VLIW/Barrier reconfiguration over the barrier mode for all parallelism classes (see Section 13.1.3) and optimization strategies (see Table 13.3).

As expected, the highest speedups (up to 68.31%) are achieved for the class 1 benchmarks, because the low parallelism is more suited for a VLIW execution instead of explicit synchronization through barriers. In average, we get speedups of 5.51-10.41% for the different optimization strategies. For example, a speedup of 68.31% is obtained for `mm16_static1` due to a reduction of the waiting time by 73.3%. `sharpening_static1` is accelerated by 31.44% (without any optimizations, opt0), because the waiting cycles are reduced from 66167 to 37649 (43.1%), while only 1143 `nop` instructions are needed in VLIW mode. On the other hand, additional barriers are required to synchronize the processors before switching to VLIW mode. Hence, `poly16_static1` executes 4.11% more barrier instructions when using the VLIW/Barrier reconfiguration and has 7% more waiting cycles.

The more parallel class 2 and especially class 4 instances of the benchmarks are more suited for an independent processing of operations with rare synchronization using barriers. Here, the mean improvements are only 2.4-5.99% and -1.8-3.72%, respectively. The class 4 benchmarks even suffer from performance losses of up to 35.08% with the VLIW/Barrier reconfiguration compared to an asynchronous execution. For instance, `sharpening_static4` with opt0 has 198516 waiting cycles when using the reconfiguration and only 141512 with the pure barrier mode (increase by 28.7%). Similar observation can be made for other benchmarks.

In the succeding sections dealing with SIMD/MIMD and register reconfiguration, we always refer to the VLIW/Barrier reconfiguration, because it yields a non-zero average speedup in most cases. Furthermore, many evaluations use the class 1 benchmarks where the reconfiguration clearly outperforms the pure barrier mode.

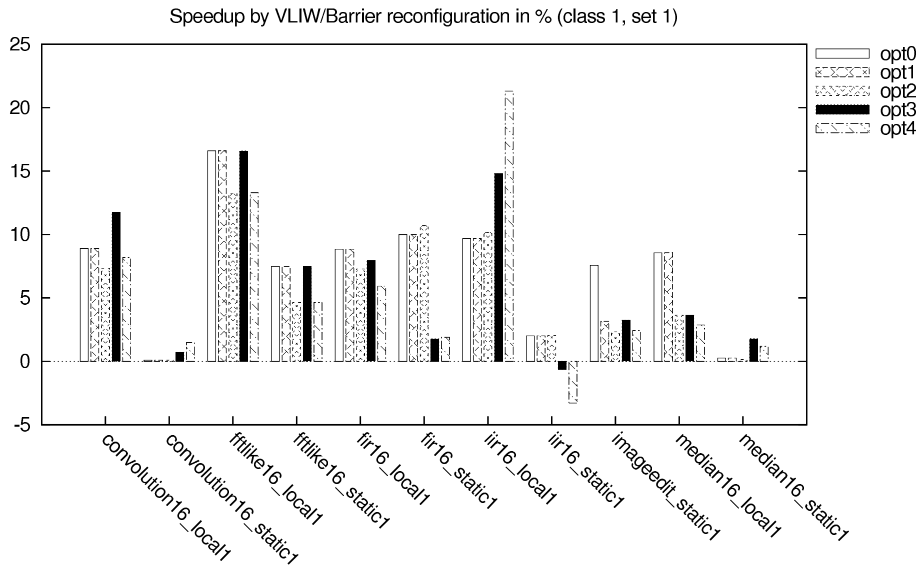


Figure 13.13.: Speedup by VLIW/Barrier reconfiguration in % (class 1, set 1)

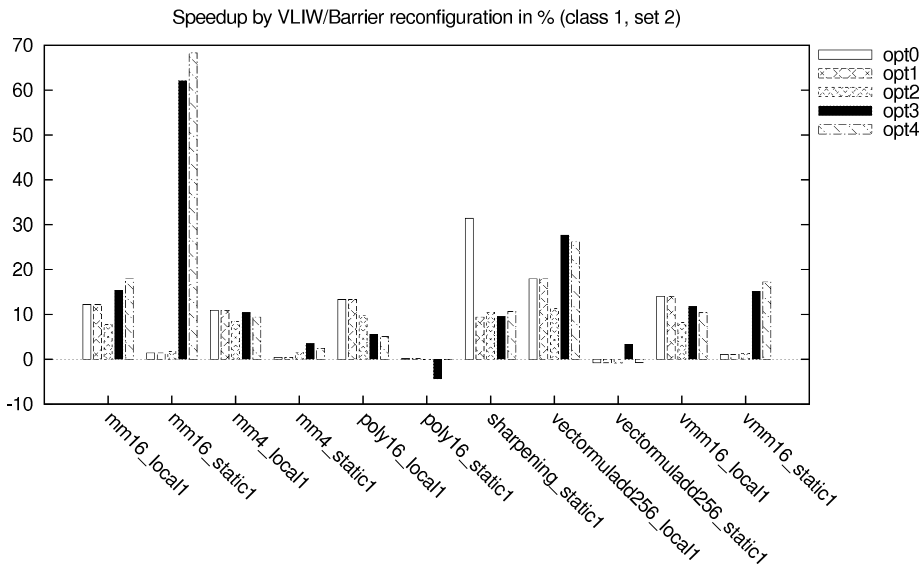


Figure 13.14.: Speedup by VLIW/Barrier reconfiguration in % (class 1, set 2)

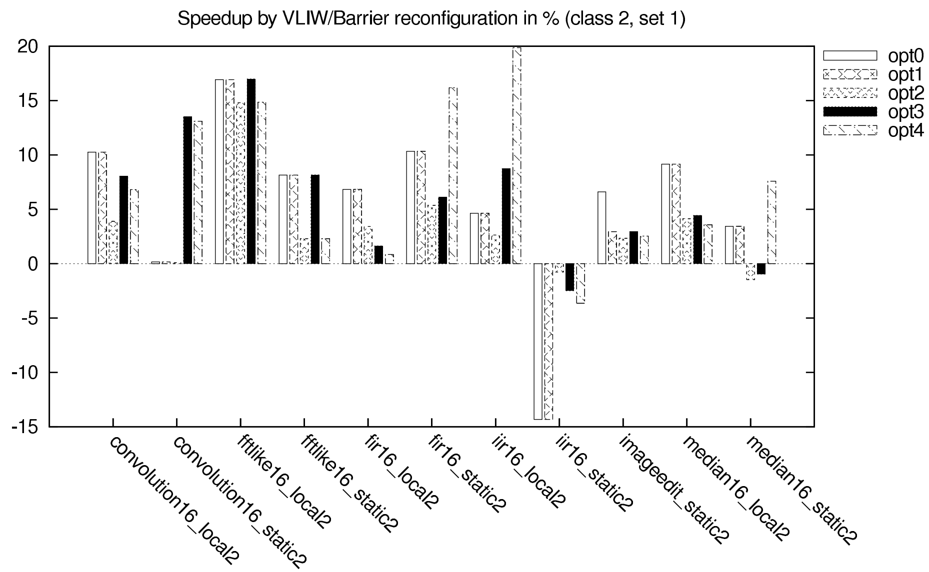


Figure 13.15.: Speedup by VLIW/Barrier reconfiguration in % (class 2, set 1)

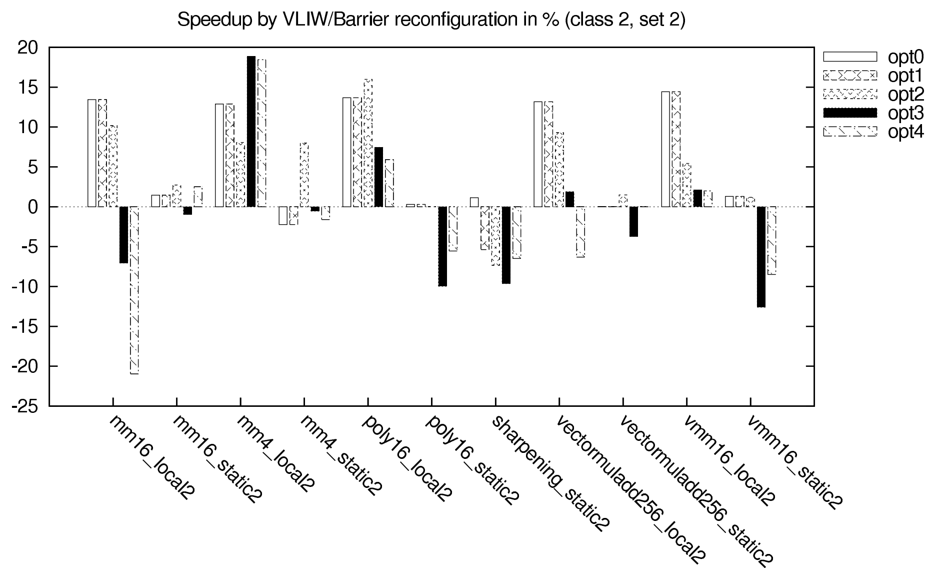


Figure 13.16.: Speedup by VLIW/Barrier reconfiguration in % (class 2, set 2)

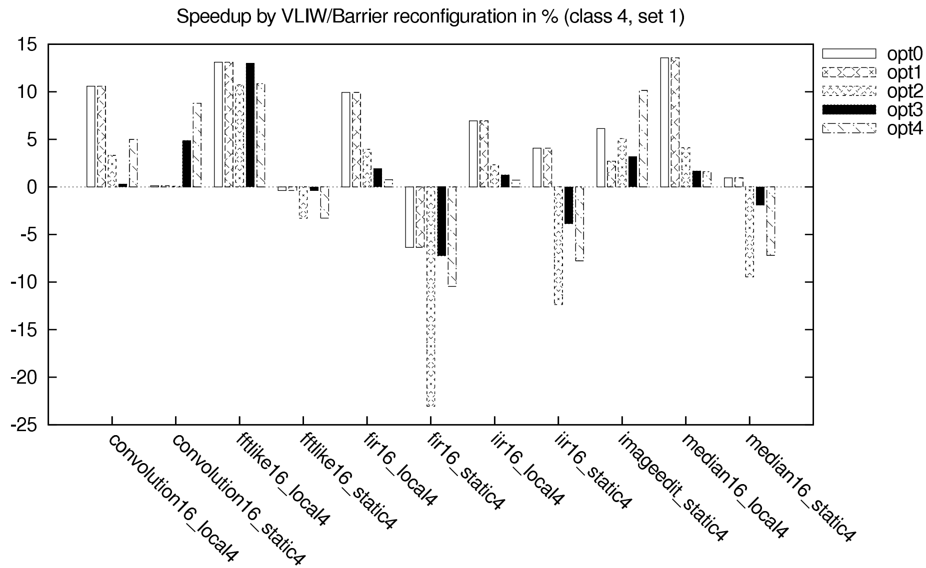


Figure 13.17.: Speedup by VLIW/Barrier reconfiguration in % (class 4, set 1)

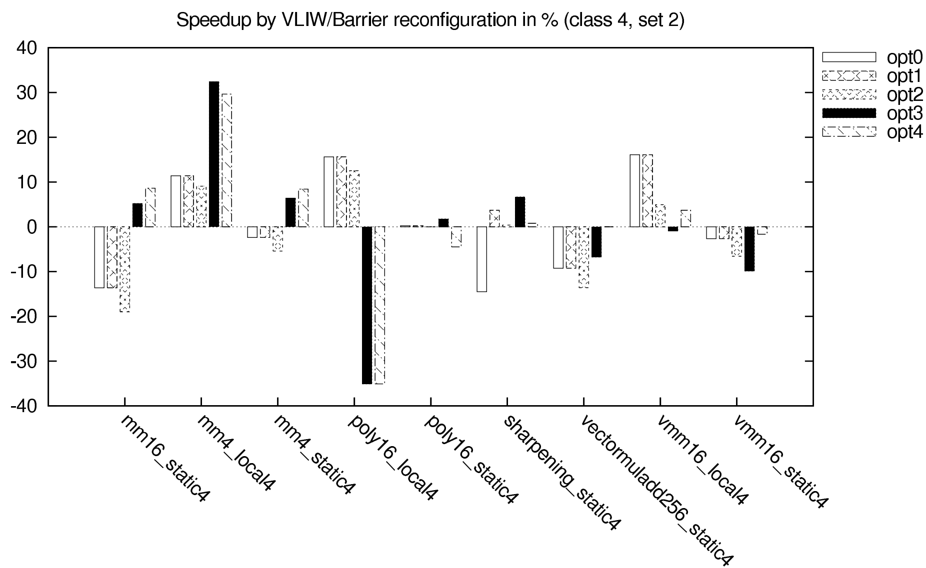


Figure 13.18.: Speedup by VLIW/Barrier reconfiguration in % (class 4, set 2)

Code Size The speedup of using the VLIW mode for pieces of code with fine-grained parallelism comes at a certain cost in code size, because additional `nop` instructions are needed to fill empty slots. Their number depends on the degree of parallelization. On the other hand, a barrier mode requires additional synchronization instructions based on the number of inter-processor dependences (see Section 6.2).

The increase of the code size by the VLIW/Barrier reconfiguration compared to the barrier mode is visualized by Figure 13.19 to Figure 13.24.

The code size of the highly parallel class 4 benchmarks is only raised by 13.52% in average,

because most slots of the schedule are filled with real instructions. The class 2 and class 1 instances require more `nop` instructions due to less parallelism and exhibit 29.62% and 33.59% relative costs, respectively. A significant increase is observed for `median16_local1` (with all optimizations, opt4), where the code of VLIW/Barrier contains 2736 `nop` instructions, but only 12 `barrier` operations less than the barrier mode (decreased from 16 to 4). For `mm16_local1` with opt4 again, we even get a reduction in code size by 19.6%, because the barrier mode suffers from almost factor 10 more barrier instructions.

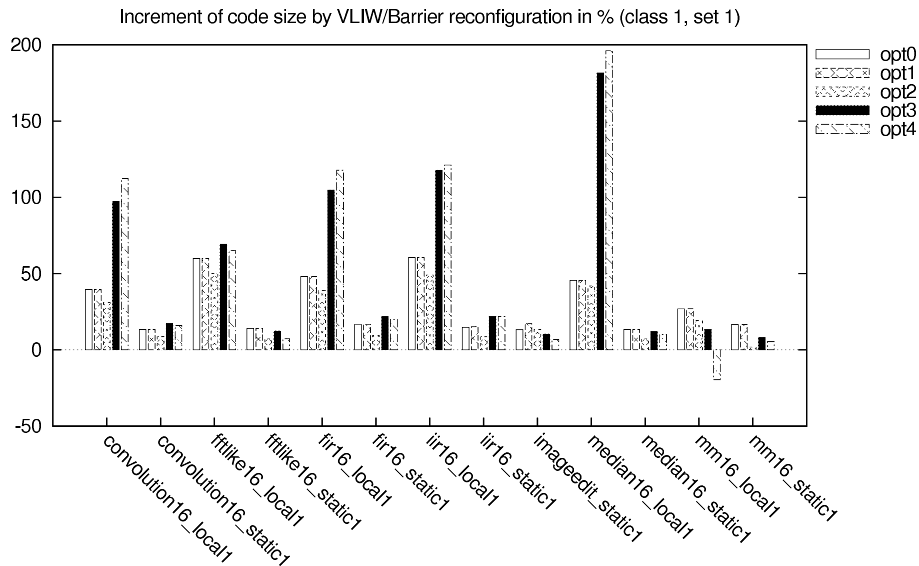


Figure 13.19.: Increment of code size by VLIW/Barrier reconfiguration in % (class 1, set 1)

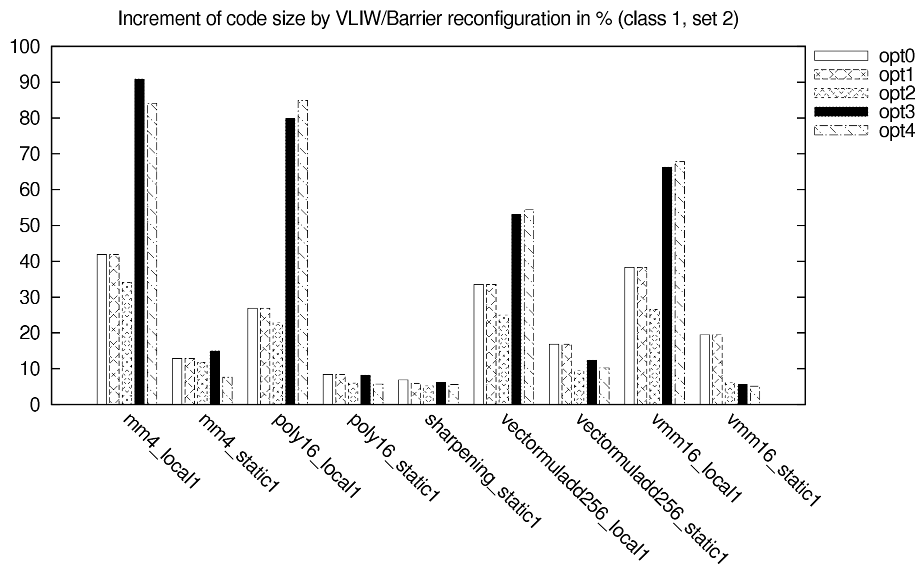


Figure 13.20.: Increment of code size by VLIW/Barrier reconfiguration in % (class 1, set 2)

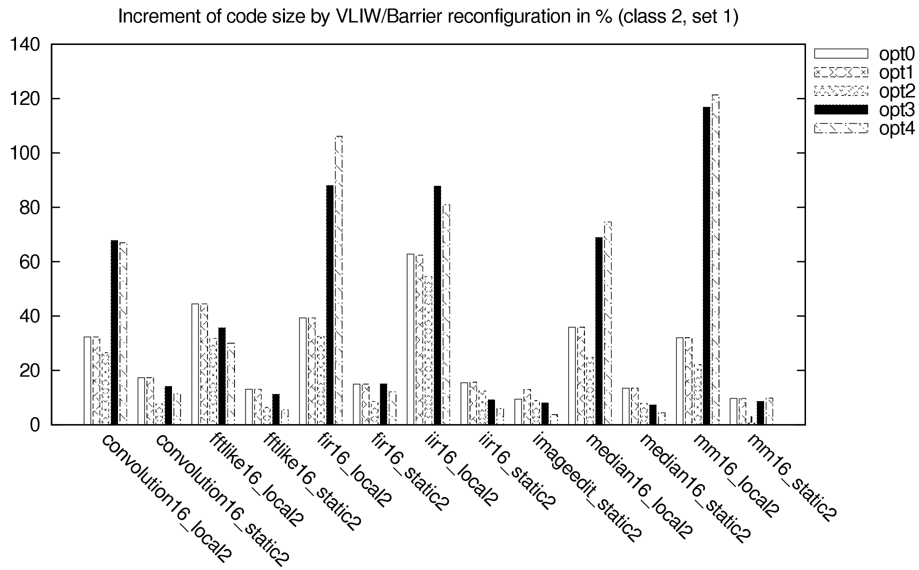


Figure 13.21.: Increment of code size by VLIW/Barrier reconfiguration in % (class 2, set 1)

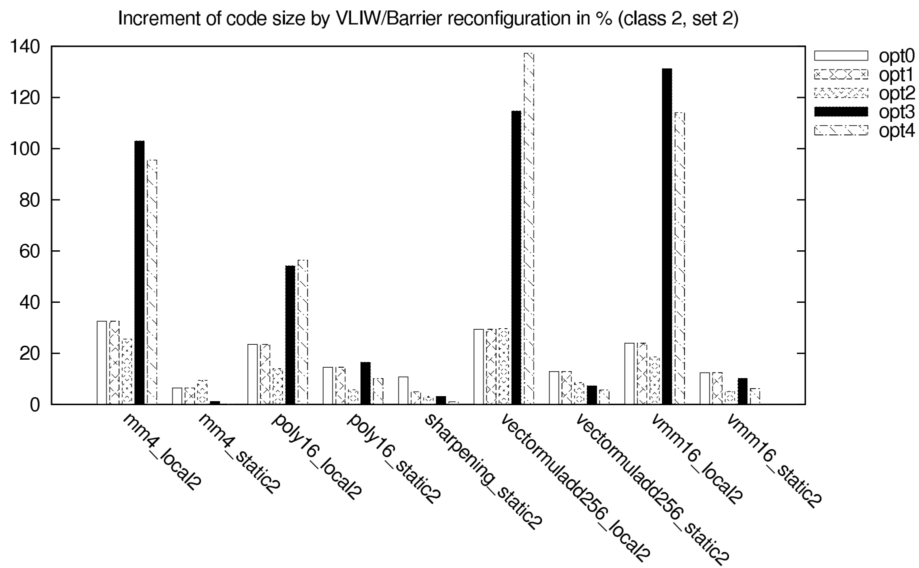


Figure 13.22.: Increment of code size by VLIW/Barrier reconfiguration in % (class 2, set 2)

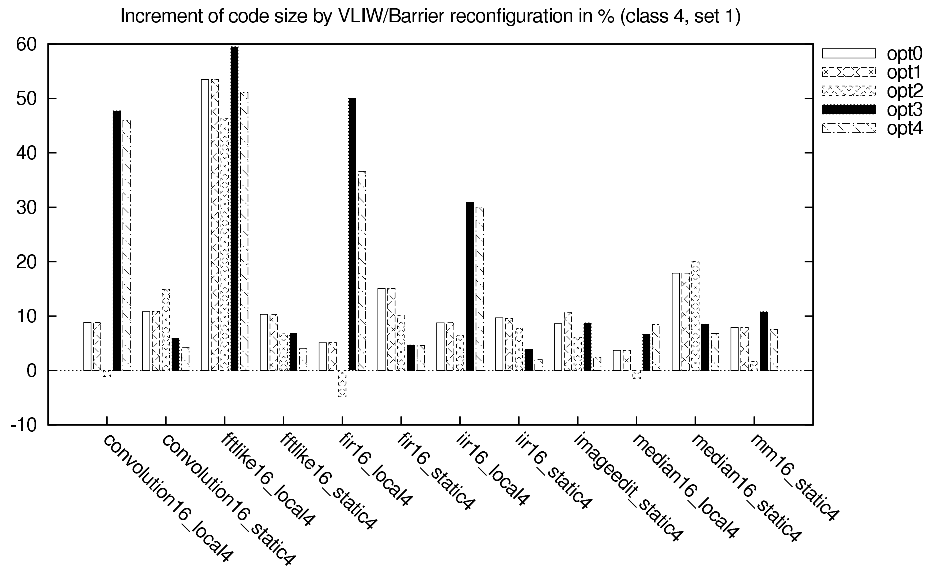


Figure 13.23.: Increment of code size by VLIW/Barrier reconfiguration in % (class 4, set 1)

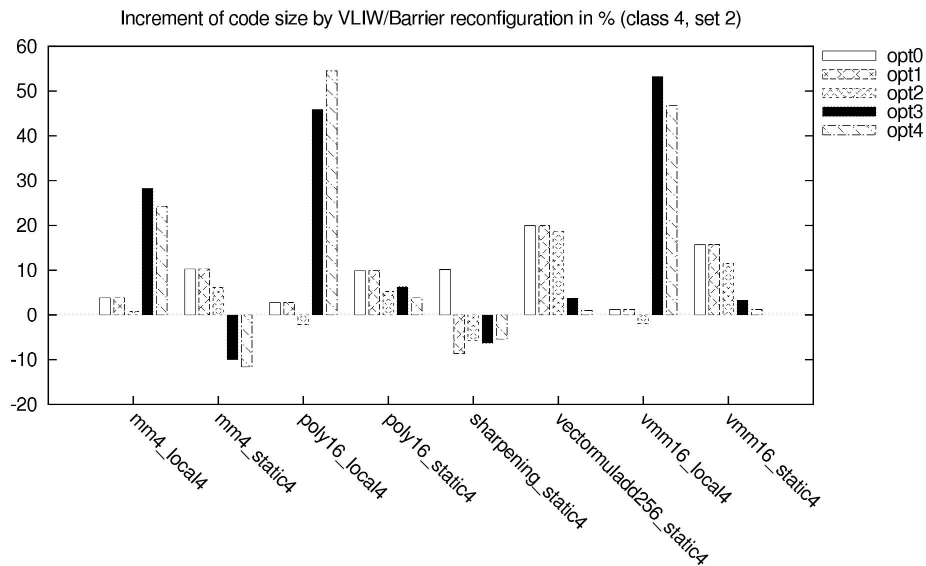


Figure 13.24.: Increment of code size by VLIW/Barrier reconfiguration in % (class 4, set 2)

Synchronization When using VLIW/Barrier reconfiguration, barriers are only required for pieces of code using barrier mode and when switching from barrier to VLIW mode, because the processors need to be synchronized. Hence, the overhead of synchronization will be reduced in most situations, unless the program toggles between both modes frequently.

Figure 13.25 to Figure 13.30 give an overview of the reduction of waiting cycles compared to the barrier mode.

The average reduction is greatest when using the class 1 benchmarks (51.61%), because the low parallelism is not suited for explicit synchronization. This correlates with the measured speedups compared to an asynchronous execution. The mean attenuation for the class 2 and class 4 instances is 47.39% and 29.43%, respectively. Furthermore, the benchmarks with local data structures benefit at most due to our heuristic which selects the VLIW mode for basic blocks without external memory accesses (see Section 13.1.2). There, the overhead of synchronization is reduced to almost 0.

With regards to some benchmarks accessing *global data structures*, the number of waiting cycles increases dramatically due to a high overhead for switching from barrier to VLIW mode. For instance, `mm16_static4` with opt2 spends 125.75% more time for waiting at barriers with the VLIW/Barrier reconfiguration compared to a pure barrier mode. The inner loop body is executed in barrier mode, because it contains external memory accesses of the matrix elements, while the loop control employs the VLIW mode. As the execution toggles 4629 times between both modes, many additional barriers are needed to provide the VLIW/Barrier reconfiguration. In the future, the VLIW/Barrier reconfiguration should be extended by a code integration which takes the effort in synchronization into account.

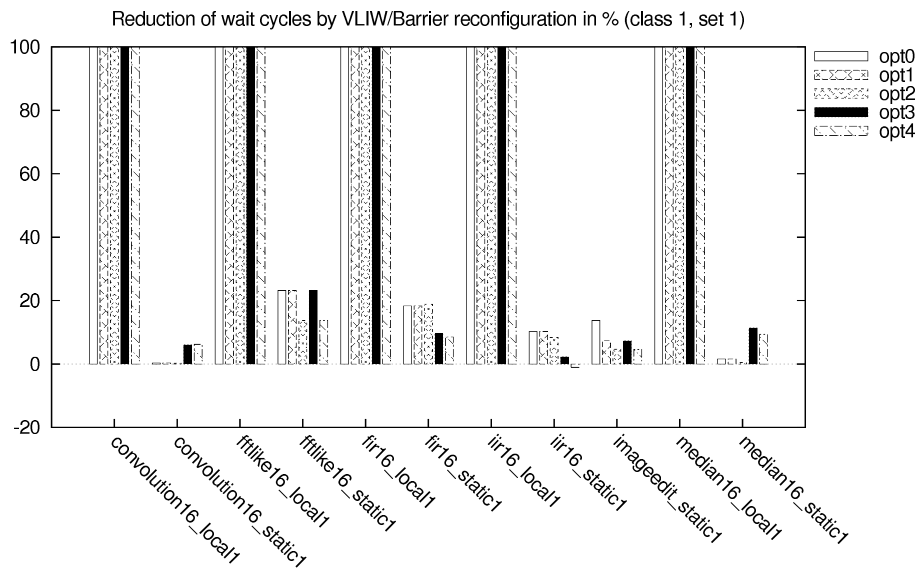


Figure 13.25.: Reduction of wait cycles by VLIW/Barrier reconfiguration in % (class 1, set 1)

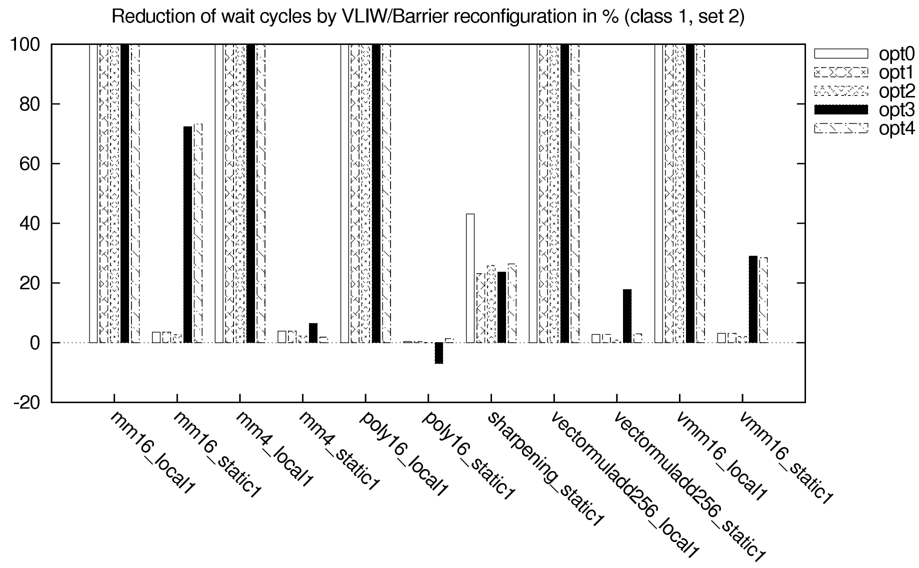


Figure 13.26.: Reduction of wait cycles by VLIW/Barrier reconfiguration in % (class 1, set 2)

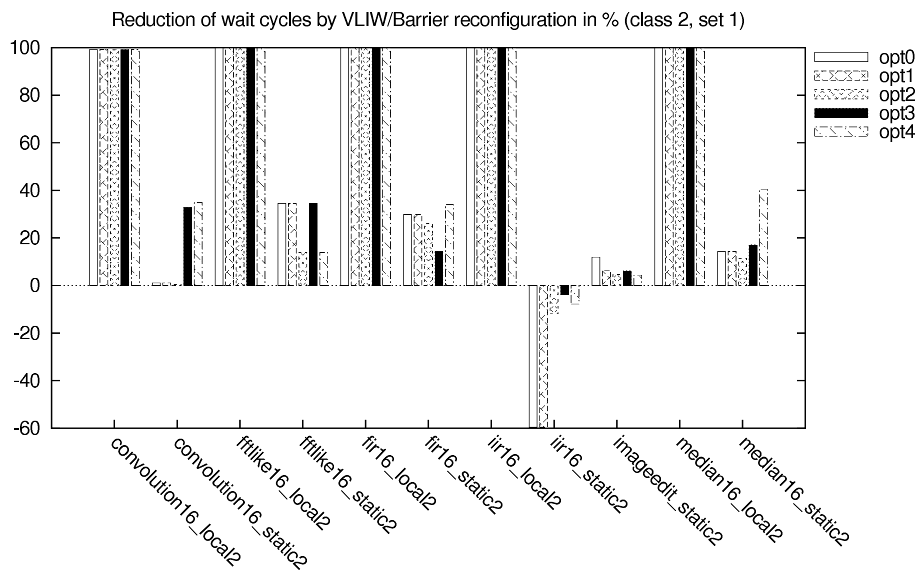


Figure 13.27.: Reduction of wait cycles by VLIW/Barrier reconfiguration in % (class 2, set 1)

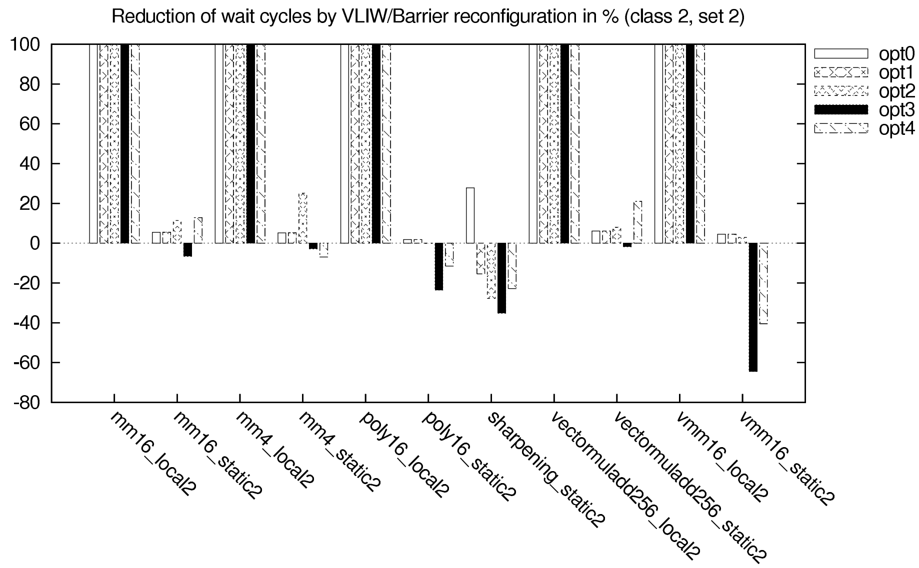


Figure 13.28.: Reduction of wait cycles by VLIW/Barrier reconfiguration in % (class 2, set 2)

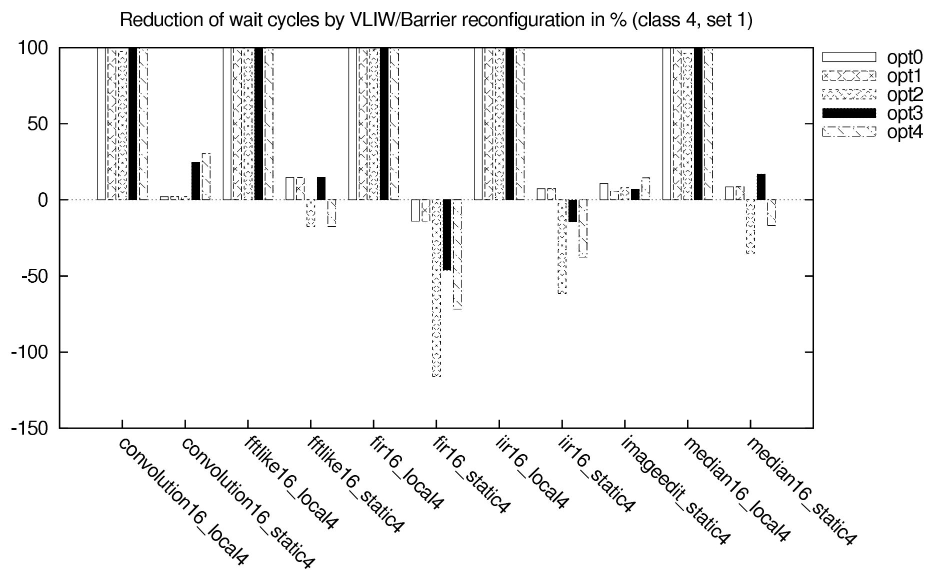


Figure 13.29.: Reduction of wait cycles by VLIW/Barrier reconfiguration in % (class 4, set 1)

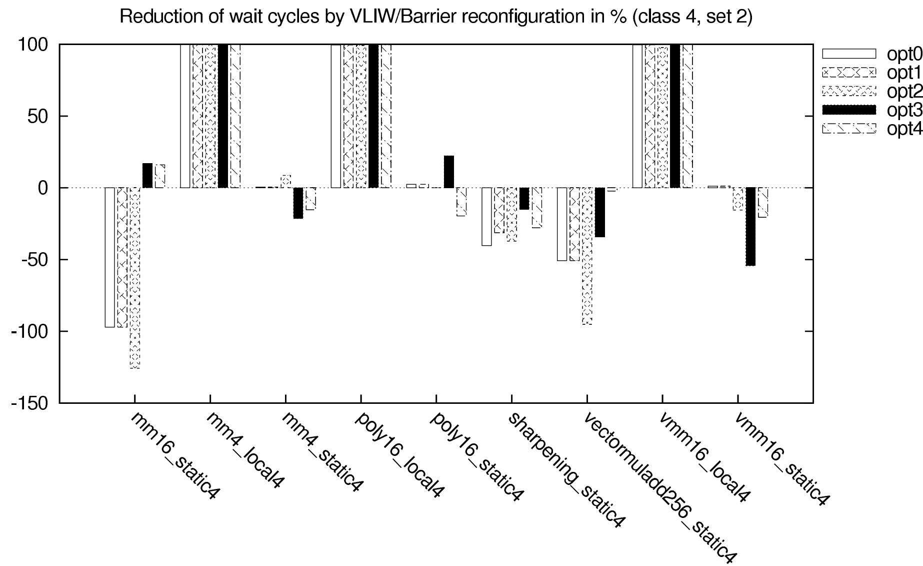


Figure 13.30.: Reduction of wait cycles by VLIW/Barrier reconfiguration in % (class 4, set 2)

13.2.3. Processor Partitioning

In this section, we evaluate our data partitioning method (see Section 5.2). Allocation of functional units is based on the BUG algorithm by Ellis [50].

Optimal Partitioning Our approach constructs a Variable Affinity Graph (VAG) for the variables of a function which is partitioned using the graph partitioning tool set METIS [94]. Furthermore, it aims for determining the optimal number of processors by using a heuristic described in Section 5.2.3. Concretely, the ratio r between the sum of affinities of the cut edges and all edges is computed as a number of the interval $[0, 1]$. With respect to a given partitioning P using p processors, P will be rejected if r is greater or equal than a pre-defined constant $\sigma \in [0, 1]$. Then a new partitioning is computed with $p - 1$ processors until a proper partitioning is found or $p = 1$.

We have evaluated all benchmarks with all optimization strategies (see beginning of Section 13.2) for $\sigma \in [0.05, 0.1, 0.15, \ldots, 0.95, 1.0]$ to determine its optimal value. Interestingly, the execution time is independent of σ for most of the benchmarks. Only in rare cases, our results imply that σ should always be greater or equal than 0.7. This is surprising at the first glance, because the optimal values for σ should be included within an interval $[a, b]$, with $0 < a < b < 1$. Our results signal that a partitioning will be accepted although there are many cut edges.

An intuitive argument refers to the parallelism classes: The class 4 benchmarks were created by duplicating data structures and corresponding code by factor 4, such that the resulting parts can be executed independent of each other. Consequently, the affinity graph decomposes naturally into 4 parts.

When using the class 1 instances of our benchmarks, the speedup compared to a single

processor is quite small due to the low parallelism. Please remember that the CoBRA compiler allocates physical registers of a processor p for instructions executed on p. For a given instruction, p depends on the allocation of functional units, while the data partitioning is only used indirectly. Consequently, different results of the data partitioning may lead to the same executable program at last.

For simplification, we do not present the detailed results here. All evaluation data presented in this thesis are based on $\sigma = 0.7$.

Automatic and Manual Partitioning Our data partitioning method could not be compared with other approaches from literature, because no further implementations were available. Instead, we compared it with a manual partitioning of local stack data, which is supported by the CoBRA compiler. Thereby, the user adds the prefix `local_C__` to the name of a variable, if it should be stored on processor C.

Figure 13.31 to Figure 13.33 illustrate the speedups achieved with our VAG approach compared to a manual assignment. For simplification, only those benchmarks are shown where actual changes could be observed. Please note that the results for instances with global data structures are independent of the two partitioning methods. First of all, we can conclude that our approach achieves comparable or even better results than a manual input in many cases.

With respect to benchmarks with low parallelism (class 1), the largest speedups are achieved when loop unrolling is applied (opt3/opt4). Apparently, the code generation relying on manual data partitioning produces much more spill code due to a higher register pressure caused by the list scheduler. For instance, `vectormuladd256_local1` is 1154.76% faster when using our partitioning method. Manual partitioning implies a bad parallelization such that more than factor 22 additional `nop` instructions are needed. The register pressure on the first processor raises from 12 to 20 (66%). Hence, the number of stack accesses are increased by about factor 2. The frequency of `addi` instructions increases from 2895 to 88589 (factor 30.6), the number of `subi` instructions from 14 to 85709 even.

Without loop unrolling, the speedups are much lower due to the smaller differences in register pressure (see beginning of Section 13.2), but still greater 0 in most cases. The maximum speedup is reached by `iir16_local1` for opt2 with 26.12%. For `fftlike16_local1`, we have a degradation of 4.89%, because the parallelization is slightly worse than the result based on a manual assignment of local stack variables.

Unfortunately, the automatic data partitioning achieves bad results for highly parallel benchmarks (class 4), because it suffers from imprecise static assumptions. For instance, `fftlike16_local4` has factor 4.14 more `nop` instructions with opt0 when using our approach. While the user recognizes four independent computations, the VAG approach overestimates additional control code needed to select between different computations. The variables used by the control code have a medium affinity to the main variables of the FFT computation. Hence, the VAG is partitioned into two instead of four parts. In this case, a more precise computation of affinities by prior profiling would probably help to achieve similar results like the manual partitioning.

With respect to `mm4_local4`, our approach computes the same partitioning as the manual

assignment concerning local stack data. But the assignment of further temporary variables which are introduced during loop unrolling is permuted. This leads to a slightly different parallelization with higher register pressure, which implies 7.4% additional stack memory accesses. The number of `addi` and `subi` instructions increases by even 49.9% and 51.5%, respectively.

To summarize, our VAG approach behaves as good as a manual assignment in average and clearly outperforms it for benchmarks with a low degree of parallelism. A human user may only take affinities between the most important data structures into account, while an automatic method can consider all variables including temporary ones used by the compiler. Exploiting more precise information by a profiling tool will probably improve the results for highly parallel benchmarks where balanced register pressure is crucial for a good performance.

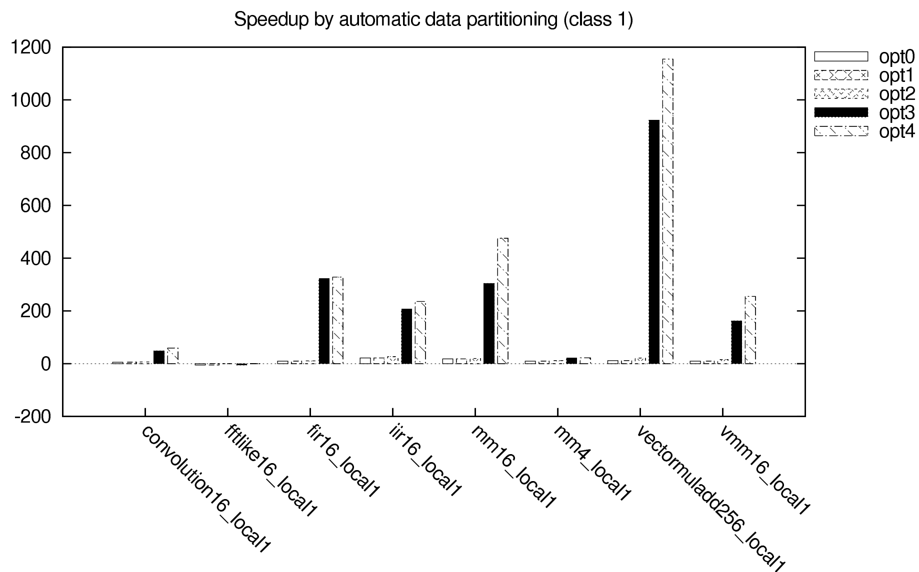


Figure 13.31.: Speedup by automatic data partitioning (class 1)

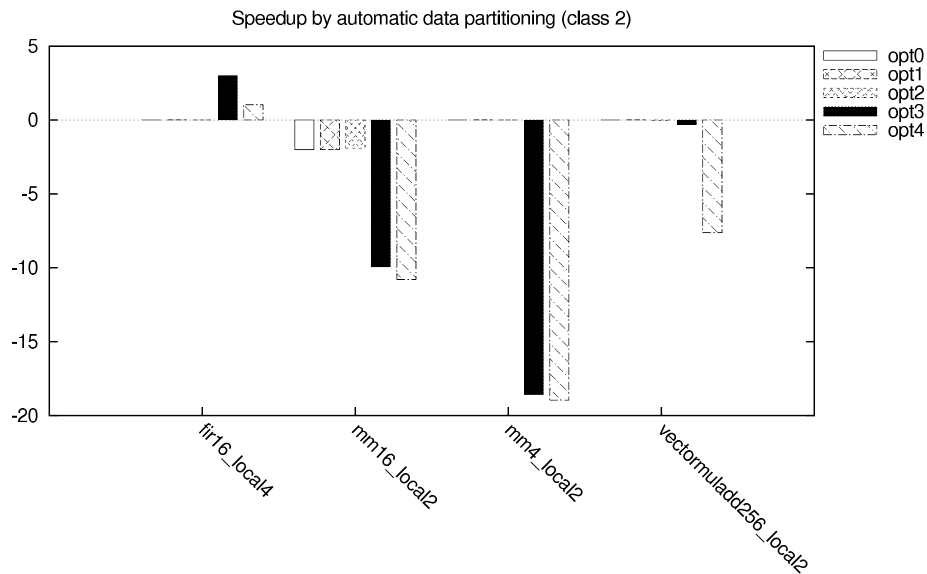


Figure 13.32.: Speedup by automatic data partitioning (class 2)

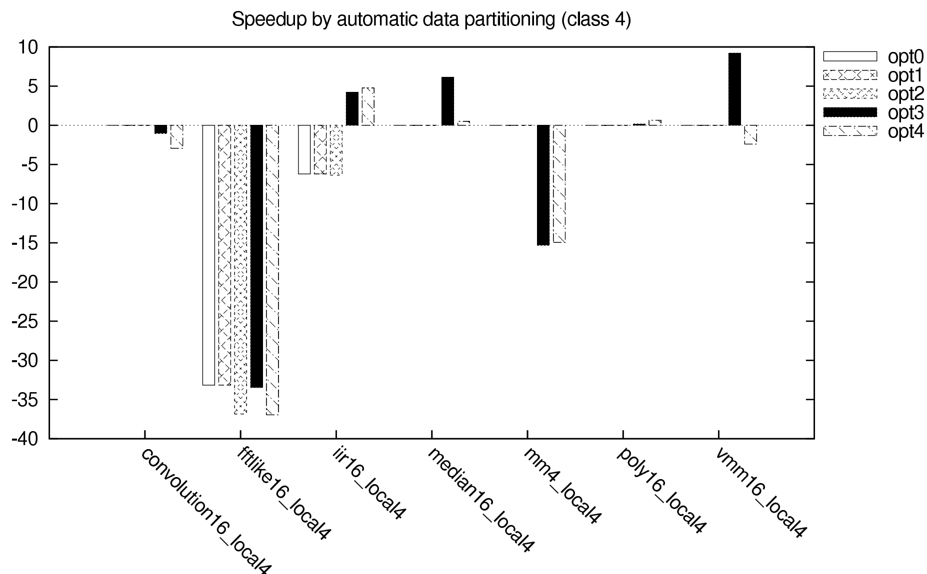


Figure 13.33.: Speedup by automatic data partitioning (class 4)

13.2.4. Communication

The CoBRA compiler parallelizes a sequential program to accelerate its execution. Parallelization is bought with additional communication between the processors. The communication mechanism of the QuadroCore can be used to transport register values. Global memory data is stored in the external memory which can be accessed by all processors. Local data structures can only be accessed from that processor whose stack contains the data.

At first, we give an overview of the communication costs in terms of the relative number

of instruction cycles spent for `cstw` and `cldw` operations. Then, we evaluate three strategies for placing communication code. For further information about the underlying concepts the reader may refer to Section 6.1.2.

Costs of Communication Figure 13.34 to Figure 13.39 show the relative overhead of communication for the three parallelism classes. In average, communication implies costs of 6.58%, 6.71%, and 6.51% for the three parallelism classes (see Section 13.1.3), respectively, which represents a modest effort. But for some benchmarks like `vectormuladd256_local1` and `mm4_local4`, about 20% of the instructions cycles executed on the processors of QuadroCore are spent in communication.

Interestingly, loop unrolling has different effects with respect to the parallelism classes: When using benchmarks with low parallelism (class 1), the relative communication costs increase in average if loop unrolling is applied. An obvious reason is an optimistic parallelization of the unrolled loop bodies, which distributes dependent instructions on different processors. On the contrary, the communication costs of highly parallel benchmarks (class 4) are even reduced in many cases when unrolling loops before parallelization. Apparently, the communication costs are hidden by the enormous overhead of spilling due to an aggressive parallelization of unrolled loop bodies (see Section 13.2.1).

Relative costs of up to 20.2% constitute the expectation that communication is a bottleneck for the parallelization of many benchmarks. This motivates the utilization of shared registers by a reconfiguration of register banks, which is evaluated in Section 13.4.

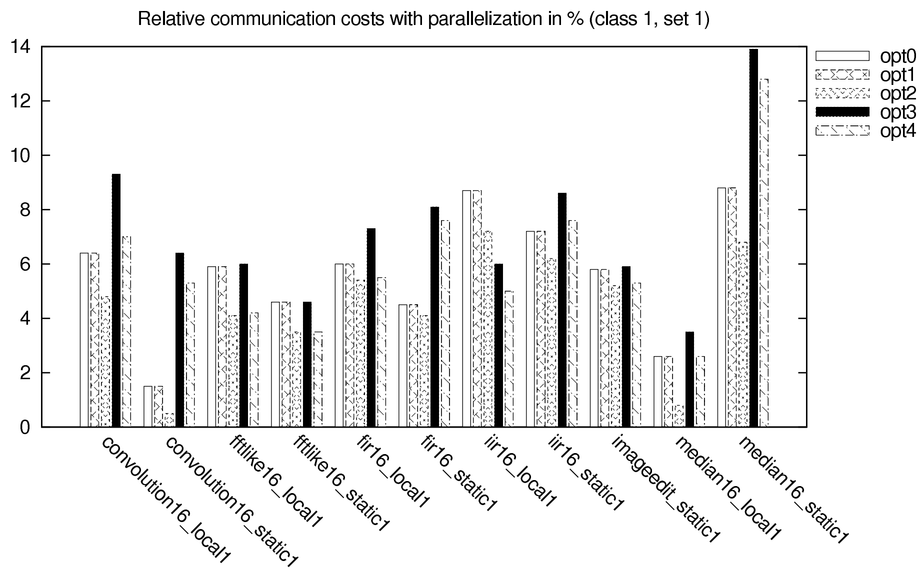


Figure 13.34.: Relative communication costs with parallelization in % (class 1, set 1)

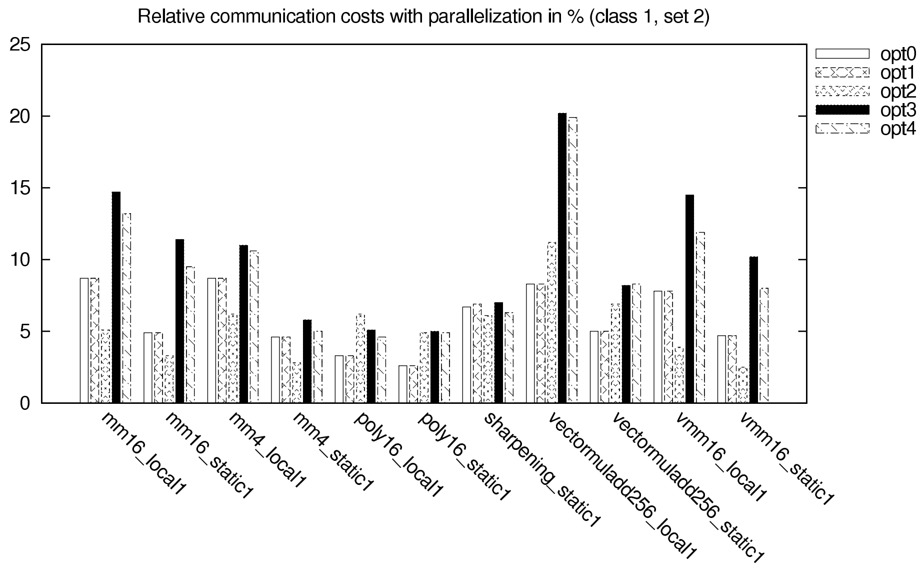


Figure 13.35.: Relative communication costs with parallelization in % (class 1, set 2)

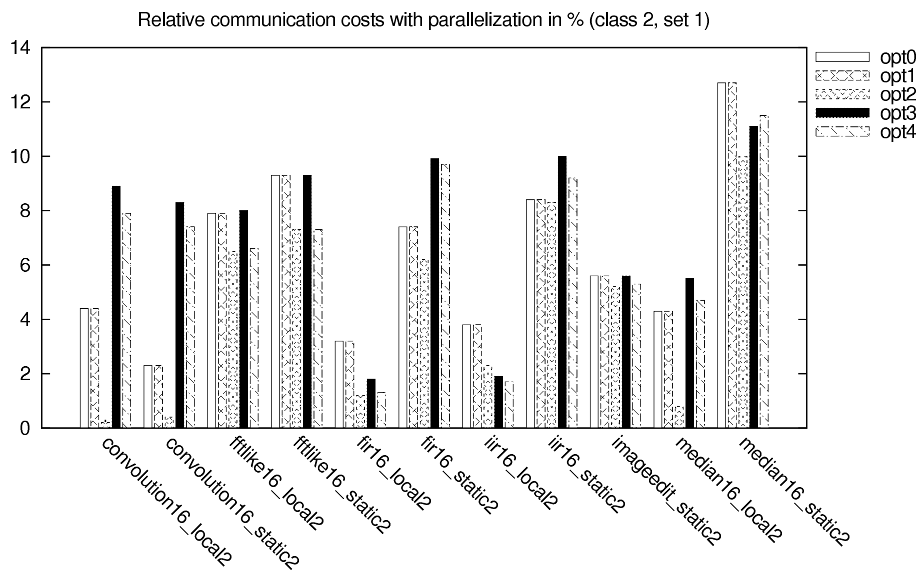


Figure 13.36.: Relative communication costs with parallelization in % (class 2, set 1)

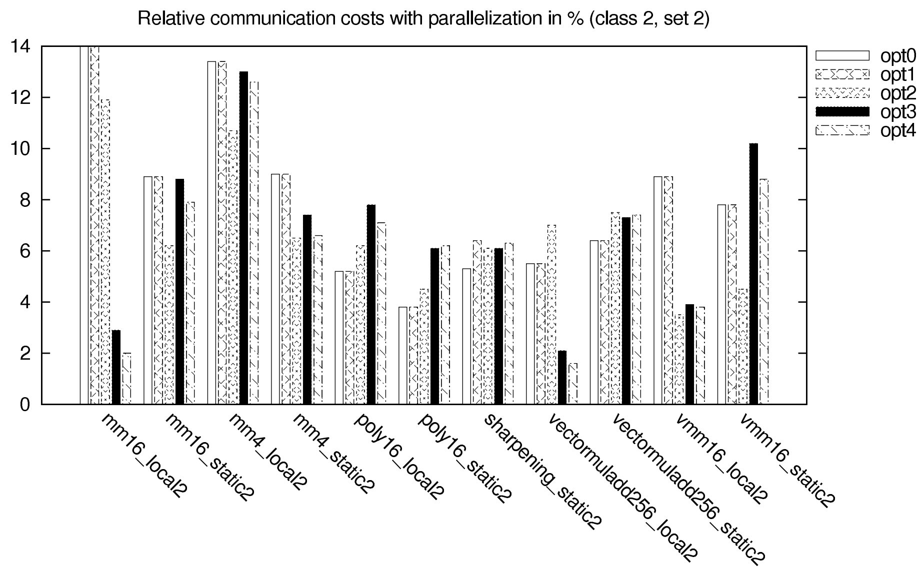


Figure 13.37.: Relative communication costs with parallelization in % (class 2, set 2)

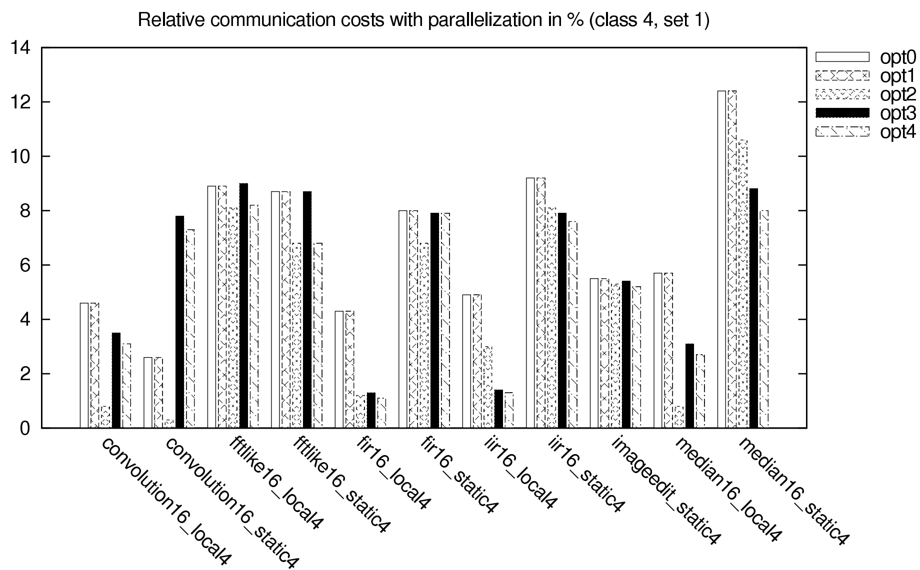


Figure 13.38.: Relative communication costs with parallelization in % (class 4, set 1)

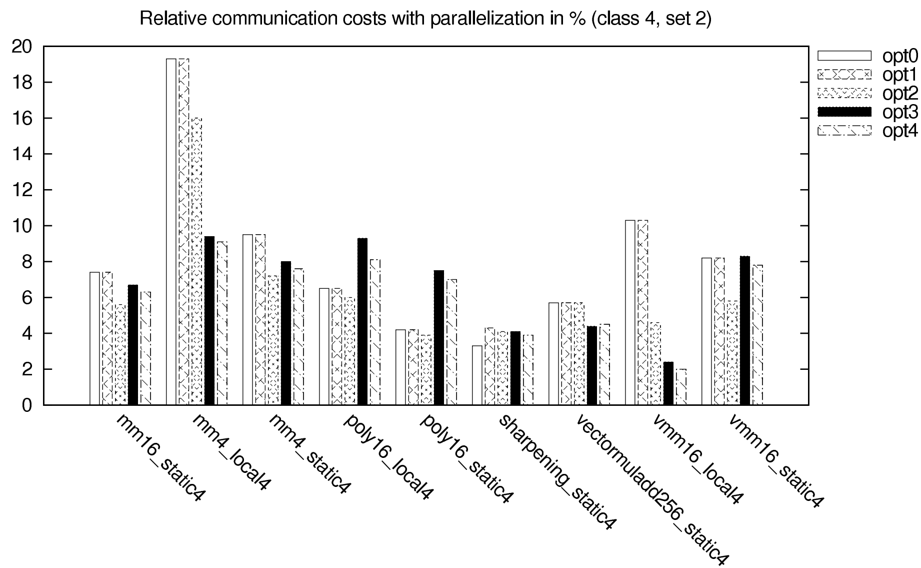


Figure 13.39.: Relative communication costs with parallelization in % (class 4, set 2)

Placement of Communication Code Communication code can be placed using three different strategies (Section 6.1.1) by the CoBRA compiler: *Directly After Definition* inserts communication code after a defining operation in the same basic block. Such strategy is suboptimal, because it does not aim for moving communication code out of loops. In principle, communication code can be placed into a basic block which lies on all paths from a definition to its uses. The second heuristic *Common Cheap Basic Blocks* selects a block with such property and lowest execution frequency. In addition, the third strategy *Merge Definitions* identifies definitions for the same register which are executed on a common processor in order to copy a value only once.

Interestingly, the two latter approaches have only achieved minimal speedups for few benchmarks. Hence, we consider two articifial programs tailored to this evaluation instead, in order to demonstrate the feasibility of our concepts. Their control-flow structure is shown in Figure 13.40 (a) and (b), respectively.

Figure 13.41 shows the execution times for the benchmarks when applying the three placement strategies. The first program is accelerated by 19% through *Common Cheap Basic Blocks*, while the execution time of the second one is reduced by 19.6% with *Merge Definitions*. As a consequence, the two strategies work in principle. In the following, we will use the naive strategy *Directly After Definition*, because the other ones do not yield nameable improvements for our benchmarks.

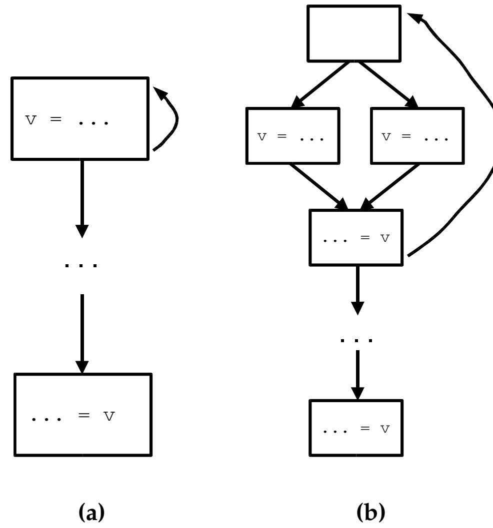


Figure 13.40.: (a) Benchmark for strategy *Common Cheap Basic Blocks* (b) Benchmark for strategy *Merge Definitions*

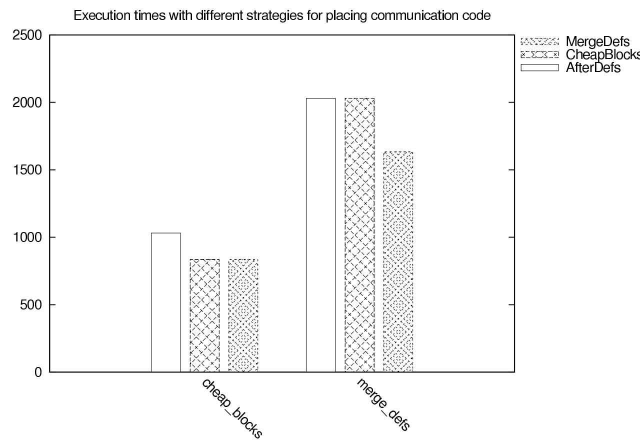


Figure 13.41.: Execution times with different strategies for placing communication code

13.3. SIMD/MIMD Reconfiguration

This section discusses the evaluation of the SIMD/MIMD reconfiguration (see Part III) in terms of execution time, ratio of SIMD execution, code size, energy consumption and overhead of reconfiguration. Finally, the reconfiguration of register banks is motivated by the communication costs. The SIMD/MIMD reconfiguration has been implemented in the CHARISMA[2] module (see Chapter 8) of our CoBRA compiler.

The real SIMD mode uses a single code stream in order to reduce the code size of a pro-

[2]**C**ompiler **H**andles **A**rchitectural **R**econfiguration **I**ntegrating **S**IMD **M**IMD **A**utomatically

gram. As the SIMD instructions are decoded by a single processor, the power dissipation of executing a program can be decreased also. Both benefits are bought by additional reconfiguration instructions to switch the QuadroCore between SIMD and MIMD mode.

Furthermore, we have evaluated the SIMD/MIMD reconfiguration by using the pseudo SIMD mode, which is motivated in Section 8.2. Vectorization is applied transparently like a normal VLIW scheduling technique to produce multiple code streams which are executed in a MIMD manner. In the following, the switching between pseudo SIMD and MIMD mode is often denoted as pseudo SIMD/MIMD reconfiguration. Whenever it can be derived from the context, we just use the term pseudo SIMD for simplification.

Obviously, no reconfiguration is required to switch between pseudo SIMD and MIMD. Also, no additional code to transport data between vector and scalar registers is needed. Hence, the speedups measured with pseudo SIMD can be regarded as an optimistic estimation of the performance improvements observed with the real SIMD mode. As the pseudo SIMD mode relies on multiple code streams, it is also considered to determine the savings in code size and energy consumption of using a single code stream in the real SIMD mode.

Edge Conditions and Notation All benchmark programs have been compiled using classical optimization and loop unrolling. The latter transformation is needed by the SLP vectorizer (see Section 8.3). As memory data used in SIMD mode must be stored in the external memory of the QuadroCore (see Section 8.2.2), only the instances with global data structures are studied. Although the CoBRA compiler is able to transform local to global variables if necessary, this would yield similar results like using programs with global data directly. Please note that the SLP approach may also yield MIMD code, if a given piece of code cannot be vectorized completely (see Section 8.3.3). Importantly, branches are executed in MIMD mode with the current prototype for simplification (see Section 8.2.3).

We consider the following alternatives: The term *MIMD* refers to the results of our parallelizing compiler (see Section 13.2). Code generation using SLP is identified by *SIMD*. *LCI* denotes a local code integration strategy (see Section 3.3.2), which selects the best result of scheduling and vectorization based on the estimated execution time.

13.3.1. Execution Time

Here, the speedup by scheduling and vectorization compared to a single processor is discussed. We give a short overview first and then study the results for most benchmarks in detail. Thereby, we consider further metrics like register pressure, spill costs, transport code, overhead of synchronization and communication etc. For more information about the edge conditions of our experiments and the notation used in the following, the reader may refer to the above paragraph.

The speedup observed by the real SIMD/MIMD reconfiguration is shown in Figure 13.42. The speedups of some benchmarks are comparatively small due to expensive spill code, which is caused by the parallelization of unrolled loop bodies. Section 13.2.1 discusses the effect of loop unrolling on the performance of scheduled code more in detail. Without loop unrolling, the CoBRA compiler has achieved much better speedups which correlate to the number of employed processors. On the other hand, a single SIMD code stream compen-

sates low speedups by significant reductions of code size (see Section 13.3.3) and energy consumption (see Section 13.3.4).

For some benchmarks like `vectormuladd256_static1` and `vmm16_static2`, the SLP vectorizer (*SIMD*) yields nameable speedups (up to factor 2.24) over a pure list scheduling (*MIMD*). But it also faces performance losses (up to 43.5%) caused by larger waiting times at barriers and additional transport code between scalar and vector registers, when concerning `fftlike16_static2` or `poly16_static4`, for instance. The synchronization time seems to be the bottleneck of QuadroCore, because the current prototype of the CoBRA compiler does not optimize for short waiting times. Instead, it tries to minimize the number of barriers which are placed during re-scheduling (see Section 6.2.3) and by a heuristic approach (see Section 6.2.5).

The local code integration (*LCI*) can combine the best results of both scheduling and vectorization to an executable program which is faster than deciding for a single technique (`median16_static1`, `vectormuladd256_static1`, ...). On the other hand, its restricted view in terms of both CFG and backend phases may lead to suboptimal results (see above). For `imageedit_static2` and `sharpening_static2`, *LCI* is a bit worse than *SIMD*. *LCI* and *SIMD* are slower than MIMD for benchmarks like `fftlike16_static2`, `fir16_-static2` etc.

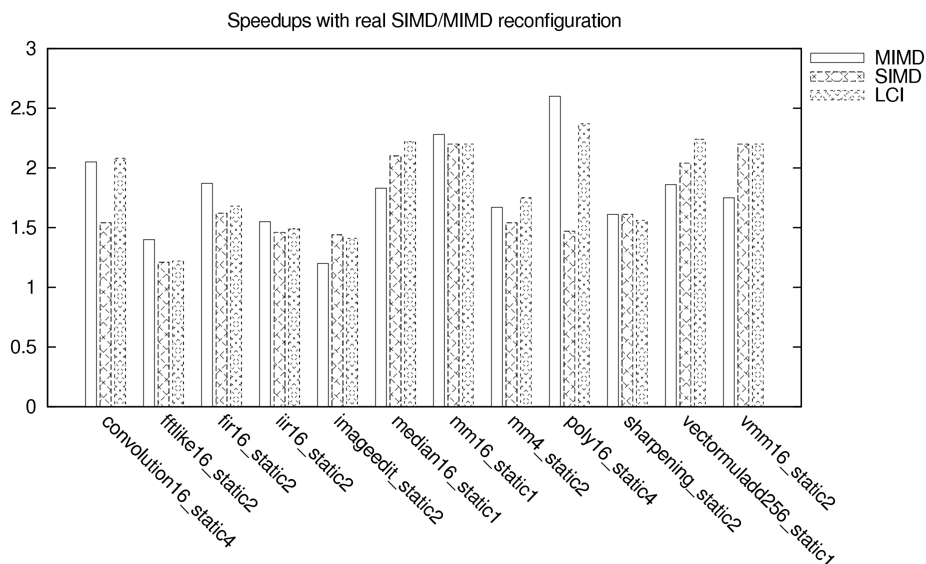


Figure 13.42.: Speedups with real SIMD/MIMD reconfiguration

In order to evaluate the reconfiguration globally, we have compared its results with the pseudo SIMD/MIMD reconfiguration (see beginning of Section 13.3). Figure 13.43 shows the degradation of the execution time of the real SIMD/MIMD reconfiguration compared to the pseudo reconfiguration. For completeness, Figure 13.44 illustrates the speedup of using the pseudo SIMD/MIMD reconfiguration. Only half of the cases exhibit a deterioration compared to pseudo SIMD of 8.56% in average. In the worst case, the execution of `poly16_-static4` is slowed down by 24.7%. Even if a performance loss is observed, the real SIMD mode can optimize for a small code size and low power dissipation.

Despite of the overhead in reconfiguration, using real SIMD also yields mean performance improvements of 7.02%. `median16_static1` is even accelerated by 13.97% over pseudo SIMD. Such improvements are justified by a lower run-time overhead of synchronization and communication. Apparently, pseudo SIMD suffers from the fact that the result from vectorization is treated as a MIMD schedule and may degenerate in contrast to using a single code stream in with real SIMD. Similar observerations can be made for `vectormuladd256_static1` and `vmm16_static2`.

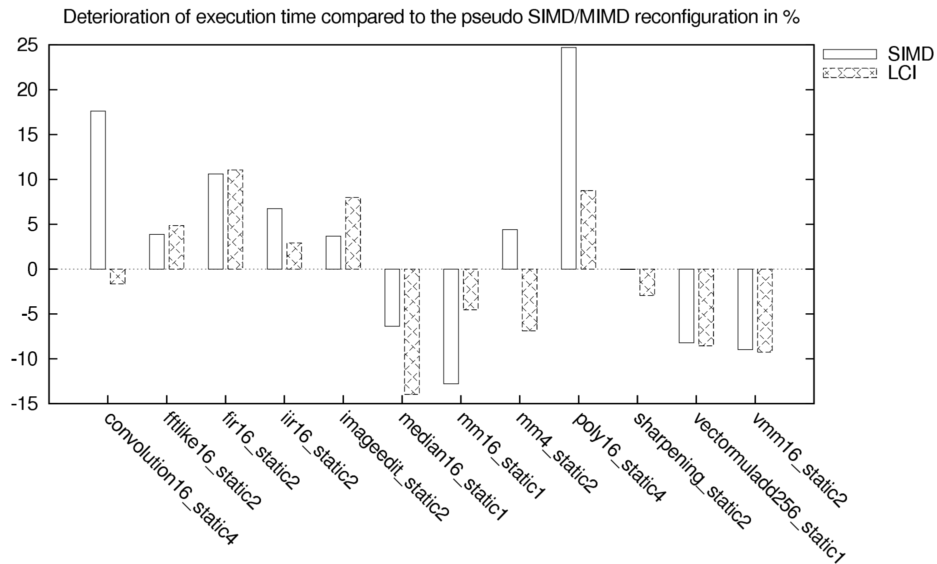


Figure 13.43.: Deterioration of execution time compared to the pseudo SIMD/MIMD reconfiguration in %

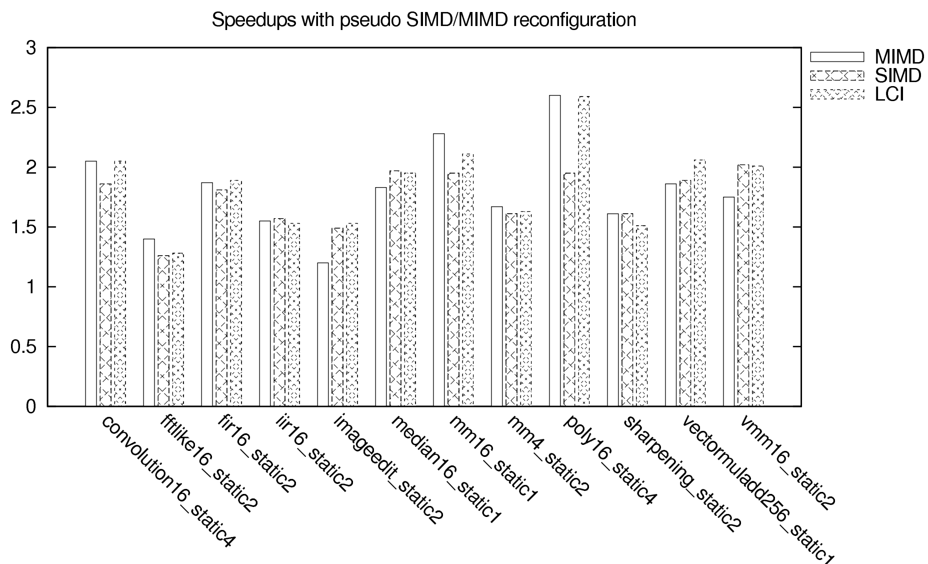


Figure 13.44.: Speedups with pseudo SIMD/MIMD reconfiguration

Discussion of Selected Benchmarks In the following, we outline selected benchmarks in detail. For a short overview of the results, the above summary should be sufficient. By default, we refer to the *real* SIMD/MIMD reconfiguration.

With respect to the pseudo reconfiguration, `convolution16_static4` is 17.64% slower when using *SIMD*, because of additional transport code between vector and scalar registers. The number of `mov` instructions changes from 6785 to 14585 (115%). Furthermore, the register pressure increases by up to 26% for the second processor, while the number of stack accesses is raised by 78.5%. On the other hand, using *LCI* even yields a small speedup of 1.65% over the pseudo reconfiguration, because *LCI* has 36.7% less waiting cycles than *SIMD*. The number of stack accesses has been reduced by 39.2% (register pressure with *SIMD* is up to 55% higher). As a result, *LCI* is slightly faster than *MIMD*.

`fftlike16_static2` in pure MIMD mode yields a higher speedup than *SIMD*, which needs 4.3% more waiting cycles. A similar observation can be made for *LCI*. The number of waiting cycles for `fir16_static2` is 7.4% higher with *SIMD* compared to *MIMD*. On the other hand, the waiting cycles for `imageedit_static2` are reduced by 36.9% when using *SIMD* instead of *MIMD* which motivates a performance improvement of 19.7%.

`median16_static1` with real SIMD and *LCI* outperforms the pseudo reconfiguration by 13.97%, because the number of waiting cycles is reduced by 49.3% (relative costs increased from 27.5% to 17.5%). *LCI* is 21.5% faster than *MIMD* which executes 83.3% more waiting cycles.

Similarly, real SIMD achieves a performance improvement of 12.79% (*SIMD*) over the pseudo reconfiguration for `mm16_static1`, because the number of waiting cycles increased by 83.7% (relatively: 42%) and relative overhead of communication changed from 7.5% to 9.7%.

A performance loss of even 24.7% compared to the pseudo reconfiguration is observed for `poly16_static4` with *SIMD* due a raise of the number of transport instructions by factor 3.7. Importantly, a higher register pressure implies an increase of stack accesses from 106 to 3347. But the local code integration combines the results of both scheduling and vectorization such that performance is improved significantly, although still slower than using the MIMD mode only. *SIMD* executes factor 2.67 more waiting cycles and the register pressure is up to 53.8% higher compared to *LCI*.

For `vectormuladd256_static1` with *SIMD*, the relative overhead of communication and synchronization is 3.8% and 10.1% smaller when using real instead of pseudo SIMD, respectively. *LCI* outperforms *MIMD* by 25.7%, because the number of waiting cycles is about 62.4% smaller, and it is also faster than *SIMD* with a similar argument.

The number of waiting cycles of `vmm16_static2` decreases by 38.7% with real SIMD. *SIMD* and *LCI* are 26.1% faster than a MIMD mode, because the waiting time at barriers is reduced by 64.7%.

13.3.2. Ratio of SIMD/MIMD Execution

This section evaluates the ratio of execution cycles in SIMD and MIMD mode. Figure 13.45 shows the results of the reconfiguration using the real SIMD mode, while the degradation

to the pseudo SIMD/MIMD reconfiguration is illustrated by Figure 13.46. The ratio values observed when using the pseudo reconfiguration are given by Figure 13.47.

First of all, the CHARISMA module can vectorize a significant part of most benchmarks such that up to 47.4% (`vectormuladd256_static1` with *SIMD*) of the execution is done in SIMD mode. In the majority of the cases, the fraction of instruction cycles executed in SIMD mode is independent of using *SIMD* or *LCI* for a specific benchmark. `vectormuladd256_static1` and `iir16_static2` have a lower SIMD ratio when using the local code integration, because it favoured the results of the scheduling phase to create a faster executable program (see Section 13.3.1).

The deterioration of the SIMD ratio compared to the pseudo SIMD/MIMD reconfiguration is clearly greater than 0 in most cases due to additional transport code between scalar and vector registers (see Section 8.4.2), which is executed in MIMD mode. The exorbitant high differences observed for some benchmarks like `imageedit_static2` or `poly16_static4` are caused by a low SIMD ratio with pseudo SIMD, that is decreased by more than 50% when using real SIMD.

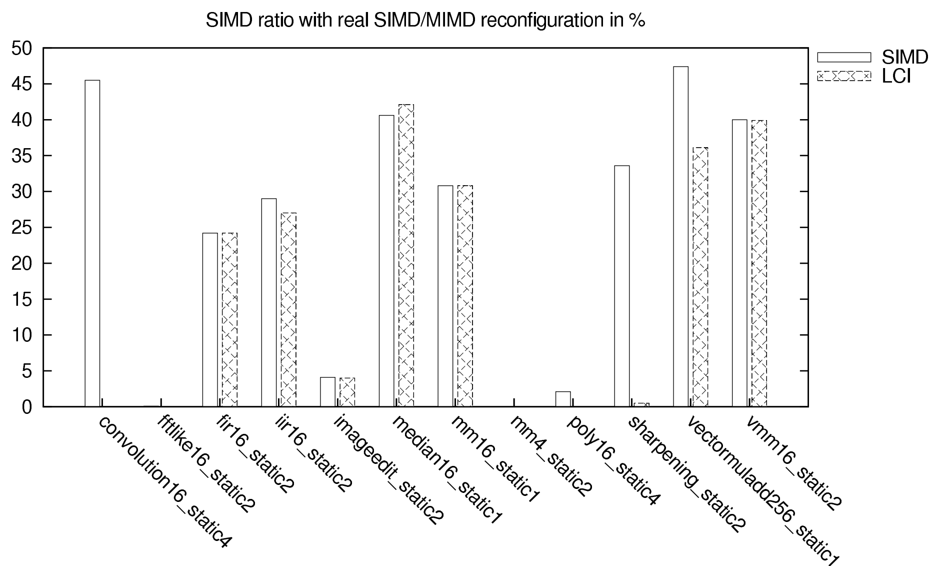


Figure 13.45.: SIMD ratio with real SIMD/MIMD reconfiguration in %

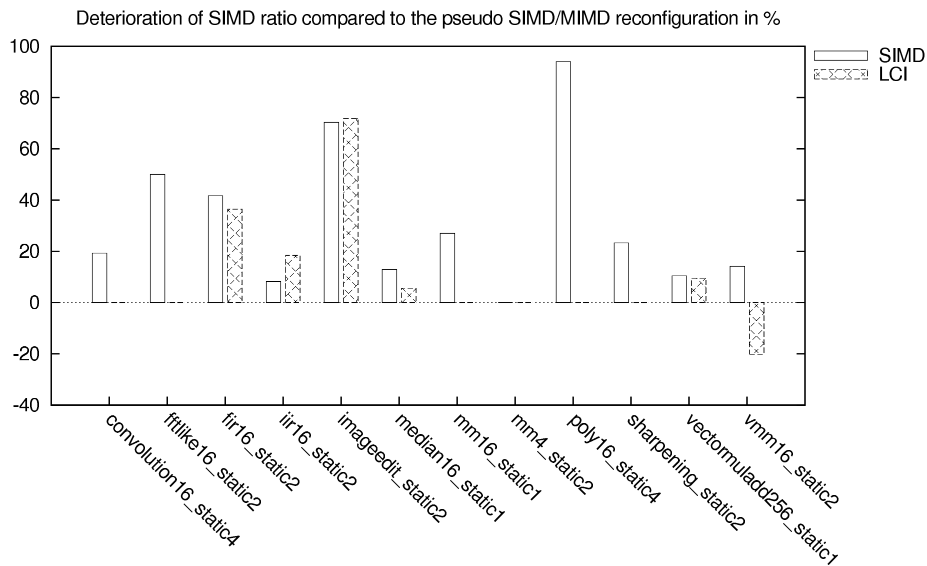


Figure 13.46.: Deterioration of SIMD ratio compared to the pseudo SIMD/MIMD reconfiguration in %

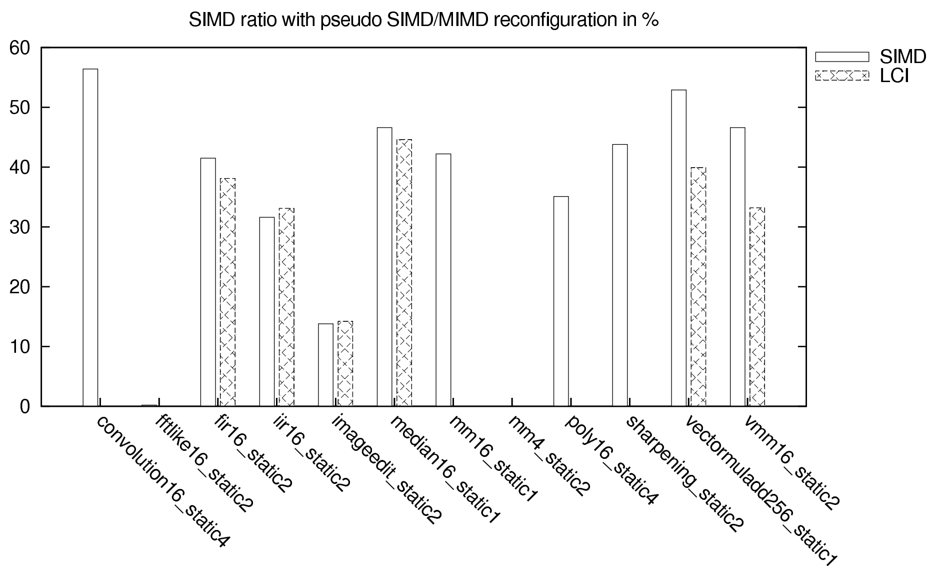


Figure 13.47.: SIMD ratio with pseudo SIMD/MIMD reconfiguration in %

Discussion of Selected Benchmarks In order to justify the obtained results, selected benchmarks are discussed more in detail now. Again, the reader may skip the explanations if the above overview is sufficient.

45.5% of the execution of `convolution16_static4` is performed in the real SIMD mode, when the SLP vectorizer is used for code generation (*SIMD*). The local code integration (*LCI*) always prefers the results of the scheduling phase and therefore yields not SIMD for this benchmark. *SIMD* with the pseudo reconfiguration even features a SIMD ratio of 56.4%,

because it safes 115% transport code needed for real SIMD. The functionality of the local code integration is proven again for `iir16_static2`, where the SIMD ratio of *LCI* is reduced by 2% compared to *SIMD*, while the execution time is 1.8% shorter.

Surprisingly, a diminutive ratio of SIMD is found for `fftlike16_static2`, although the source code should be well-suited for vectorization. Our inquiry has shown that the module for computing adjacent memory accesses (Section 8.3.1) does only recognize adjacency for pairs of memory accesses here. As the CHARISMA module only vectorizes four adjacent memory instruction, a lot of vectorizable code is executed in MIMD instead. Please note that the degradation to pseudo SIMD only amounts to 50% because of the small ratio observed with the real SIMD mode.

`fir16_static2` has 24.2% SIMD for both *SIMD* and *LCI* when using the real SIMD/MIMD reconfiguration. The deterioration to pseudo SIMD is caused by an increment of the transport instruction by 62%.

`imageedit_static2` exhibits a deterioration of about 70% compared to pseudo SIMD, because the number of transport instructions increases by 43.7% and the SIMD ratio is about 4% only when applying real SIMD.

Interestingly, `median16_static1` has slightly more SIMD and a shorter execution time, if *LCI* is used instead of *SIMD*. Such phenomenon is caused by a reduction of the waiting time at barriers by 22.3%, which only occurs in MIMD mode.

The degradation of the SIMD ratio for `mm16_static1` when applying *SIMD* is caused by an increase of the transport instructions by 57%. *LCI* does not yield any SIMD code for pseudo SIMD, because the result of the scheduling is always better. This is surprising at the first glance, because the same observation should have been made for real SIMD. But the CoBRA compiler inserts auxiliary add immediate instructions (`addi`) with constant 0 in order to vectorize address code completely when real SIMD is used. Hence, the dependent memory instructions can be executed in SIMD mode also (see Section 8.2.2). Such `addi` instructions were not generated by the code selection originally, because they are rather superfluous. The presence of 2-dimensional data structures in `mm16_static1` and loop unrolling motivates the comparatively high number of `addi` instructions such that the dependence graphs of pseudo and real SIMD differ significantly.

No vectorizable constructs are found in `mm4_static2`, because adjacency between the memory accesses in the innermost loop body is not recognized. Interestingly, such a phenomenon only occurs for 4x4 matrices, because loops are unrolled by factor 4 also. In this case, slightly different address computations are generated which are not handled by the adjacency module currently. Fortunately, this problem has not been observed for the other benchmarks.

For `poly16_static4`, a high degradation of the SIMD ratio from 35.1% to 2.1% can be observed, because the number of transport instructions increases by 274%. Furthermore, the overall waiting time at barriers raises by 20.3%.

`sharpening_static2` with real SIMD is a negative example of the local code integration: The SIMD ratio decreases from 33.6% to 0.5%, while a small performance loss is caused by 10% more relative communication costs, which cannot be estimated at code integration time.

`vmm16_static2` demonstrates the interaction of execution time, SIMD ratio, waiting time,

and spill costs in an interesting manner: Both *SIMD* and *LCI* achieve a faster execution with real SIMD in contrast to pseudo SIMD, while they behave conversely with respect to the SIMD ratio. For *LCI*, its value is 6.7 percentage points larger when using real SIMD, because the waiting time is reduced by 38.7% compared to pseudo SIMD. Hence, less MIMD cycles imply both a shorter execution time and a larger SIMD ratio. But for *SIMD*, the ratio is smaller when using real SIMD, although the waiting time of pseudo SIMD is still higher. Here, the register pressure for pseudo SIMD is slightly larger than with real SIMD, such that more spill code is added in SIMD mode. As a consequence, pseudo SIMD is slower and has a greater SIMD ratio, simultaneously.

13.3.3. Code Size

As the real SIMD mode relies on a single code stream (see Section 8.2), the overall code size can be reduced significantly. The vectorization of the CoBRA compiler can also be used to produce code for the pseudo SIMD mode, which consists of multiple streams. Hence, the reduction in code size has been determined by comparing the size of executable programs produced by the real and the pseudo SIMD/MIMD reconfiguration. Obviously, computing a reduction by considering the code size of a pure MIMD scheduling does not work in practice, because scheduling and vectorization compute totally different parallelizations. Reconfiguration instructions and transport code between scalar and vector registers attenuate the benefit of using a single code stream. Additionally, the instructions of the QuadroCore have non-uniform latencies such that the execution of many operations with a high latency may yield a large SIMD ratio, but a comparatively small reduction in code size. Last but not least, some instructions of the QuadroCore like branches, calls, and the operation to the load address of a global variable expect additional data after the instruction word, such that the observed results are influenced further.

Figure 13.48 illustrates the reduction of the code size. In average, the code size is reduced by 8.6%. Significant savings are achieved for `median16_static1`, `mm16_static1`, and `vmm16_static2`, which also correlate to the measured fractions of SIMD cycles. When code generation is performed by the vectorizer only (*SIMD*), a reduction of 22.66% can be observed for `vectormuladd256_static1`. Only in rare cases like `mm4_static2` and `sharpening_static2`, the code size even increases by up to 5.37% due to additional synchronization code. In the following, we consider further selected benchmarks in detail.

Discussion of Selected Benchmarks The code size for `convolution16_static4` has been reduced by 15.61% with *SIMD*, while it obtains a SIMD ratio of even 45.5%. As the local code integration prefers the result of the scheduling phase, the code size even increases by 1.13% compared to the pseudo reconfiguration.

For `fftlike16_static2`, both strategies yield a similar ratio of SIMD execution, but a different reduction. Apparently, the code sizes with real SIMD are quite similar, but *SIMD* has a larger code size than *LCI* for pseudo SIMD due to more communication and barrier instructions. The argument also applies for `fir16_static2` and `iir16_static2`. Concerning `imageedit_static2`, *SIMD* even has a nameable smaller code size than *LCI* with real SIMD.

The code sizes of mm4_static2 increase by 5.37% and 1.1%, because no SIMD code exists and the number of barriers increases by 57.1% for *SIMD*.

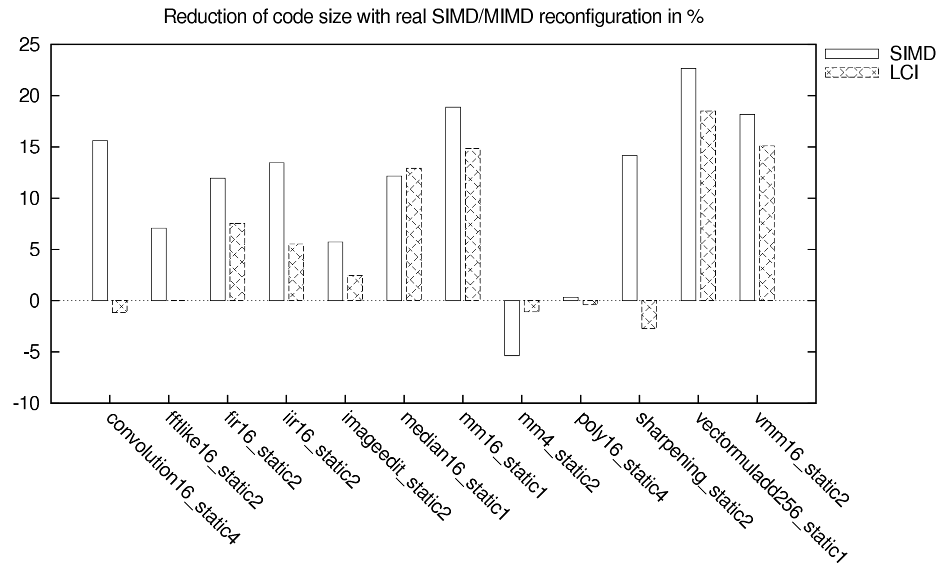


Figure 13.48.: Reduction of code size with real SIMD/MIMD reconfiguration in %

13.3.4. Power Consumption

In the real SIMD mode, instructions are decoded by the first processor only and forwarded to the ALUs of all processors. Hence, the remaining decoders can be turned off to safe power. Unfortunately, an instruction decoder only consumes a negligible part of the energy needed by the S-Core processors employed in the QuadroCore. However, energy is also saved, because no access of the instruction memory needs to be performed. Furthermore, the switching activity is reduced, which is given as the number of switches between 0 and 1 on each wire. According to [117], the switching activity is a key parameter that models the costs of charging and discharging different load capacitances, which are the most important source of energy dissipation in digital CMOS circuits.

As a consequence, we decided to model the energy reduction of the real SIMD mode in terms of instruction memory accesses and switching activity of instruction fetch and program counter. Such parameters are also influenced by the actual program, the instruction set of the machine, and the bit patterns of instructions. Nevertheless, they are a good hint for the power consumption of executing a given program. Hence, a benchmark with about 30-40% SIMD should also exhibit a significant reduction in energy dissipation.

Figure 13.49 to Figure 13.51 illustrate the savings. As done for the code size (see Section 13.3.3), the reduction of the power consumption has been determined by comparing the real SIMD/MIMD reconfiguration with the pseudo reconfiguration. Again, significant reductions can be observed which mostly correlate to the savings in code size. Worth mentioning are fir16_static2, median16_static1, mm16_static1, vectormuladd256_-static1, and vmm16_static2. Only poly16_static4 suffers from a dramatic increment of the power dissipation by up to 50.85% due to a high overhead of the transport between

vector and scalar registers. In average, the benefits for the three criteria amount to 15.19%, 14.12%, and 13.73%, respectively.

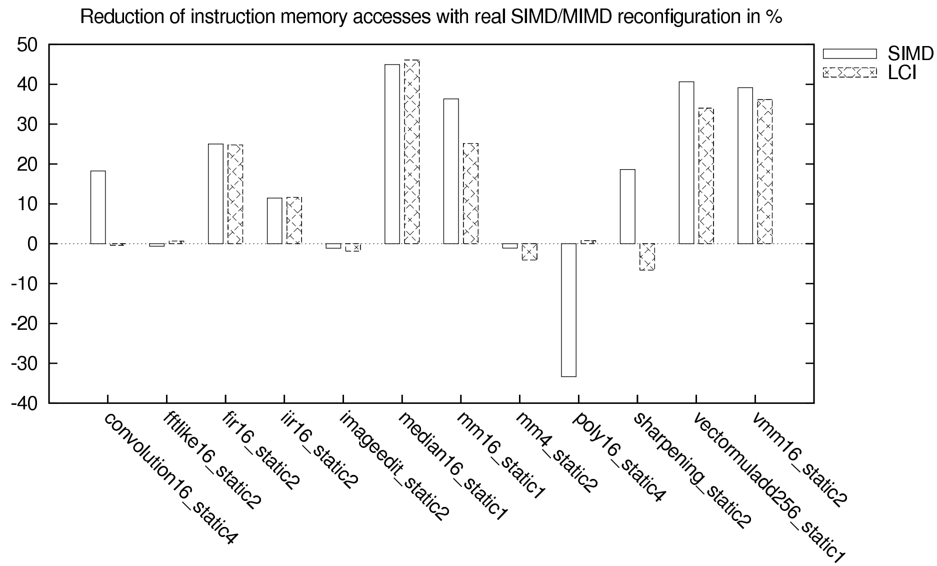


Figure 13.49.: Reduction of instruction memory accesses with real SIMD/MIMD reconfiguration in %

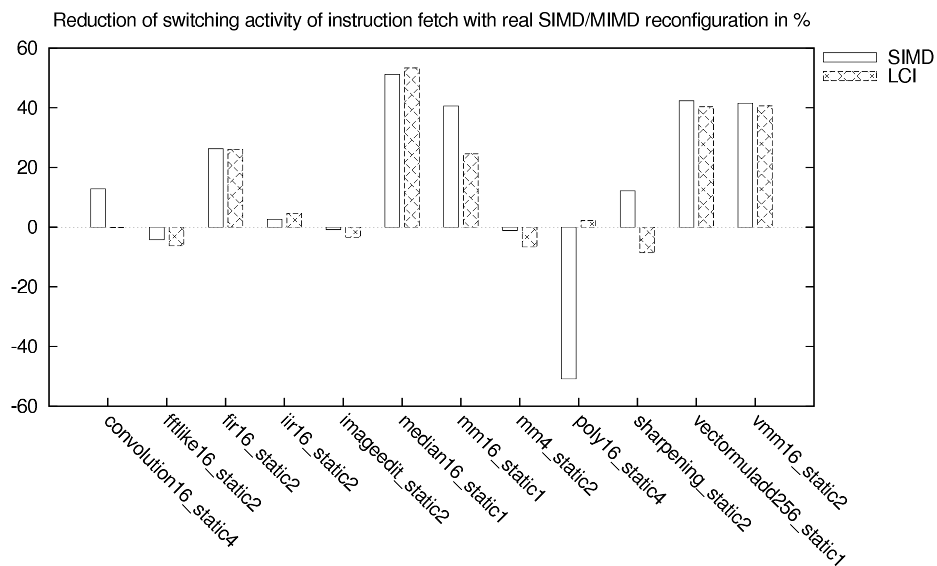


Figure 13.50.: Reduction of switching activity of instruction fetch with real SIMD/MIMD reconfiguration in %

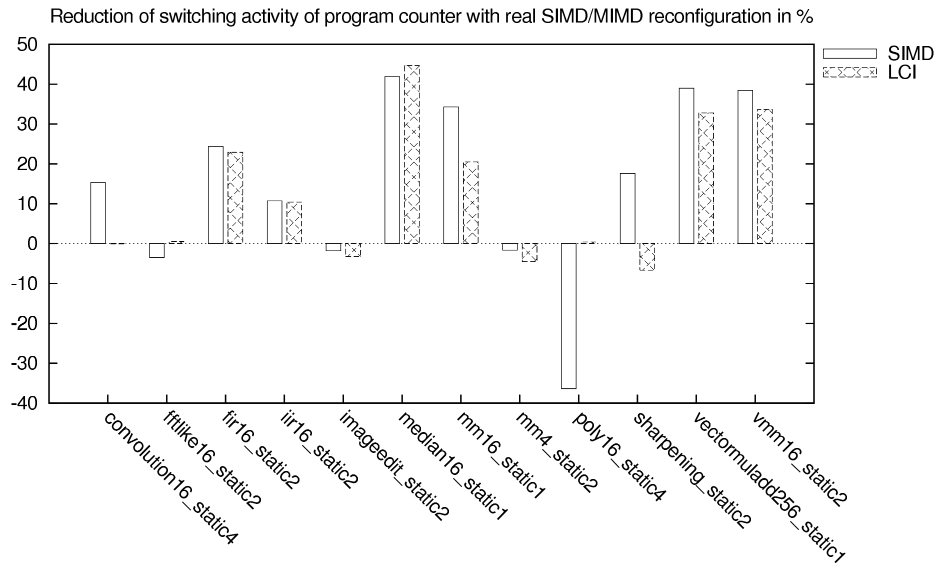


Figure 13.51.: Reduction of switching activity of program counter with real SIMD/MIMD reconfiguration in %

Discussion of Selected Benchmarks The results of `convolution16_static4` correlate with the reductions in code size. Significant savings between about 12-18% are achieved when code is generated by the SLP vectorizer exclusively (*SIMD*), which yields a SIMD ratio 45.5%. Instead, the local code integration (*LCI*) implies a small increase in both dynamic power and code size.

`fftlike16_static2` with real SIMD exhibits partly a slight increase of the energy costs according to the deterioration of the execution time compared to pseudo SIMD (see Figure 13.43). Interestingly, the code size is reduced for *SIMD* or remains the same for *LCI*, while the reduction of instruction memory accesses behaves conversely. The code size is a static parameter, while the number of accesses of the instruction memory depends on the actual execution of a program. For `imageedit_static2`, even all energy parameters are raised while there are savings in code size.

Concerning `fir16_static2`, *SIMD* and *LCI* achieve a similar reduction of the accesses to the instruction memory, while there is a nameable difference between the savings in code size. With respect to both real and pseudo SIMD, the number of instruction memory accesses with *SIMD* is larger than for *LCI* by similar factors. Hence, those factors also implies the similarity of the reduction values. On the other hand, the savings in switching activity are slightly lower with *LCI*, because they are also influenced by the choice of bitpatterns for instructions. Similar observations can be made for `iir16_static2`.

`mm4_static2` features a smaller increase of the dynamic power than code size, because the number of barrier instructions in its program code has raised more than the execution frequency.

The significant increment of the energy consumption for `poly16_static4` correlates with the deterioration of the execution time compared to the pseudo SIMD/MIMD reconfiguration (see Figure 13.43). The high number of transport instructions requires many instruc-

tion fetches and implies a high switching activity, because transport and other operations are loaded alternately. As such transport code is executed very often, it mainly affects dynamic values instead of code size.

13.3.5. Costs of Reconfiguration

The costs for reconfiguring the QuadroCore are given as the number of cycles spent to switch between SIMD and MIMD execution. As the `execmode` instruction (see Section 13.1.1) can be executed in a single cycle, the effort in the actual reconfiguration of the machine is rather small. Figure 13.52 gives an overview of the relative costs, which are at most 2.5% for `vectormuladd256_static1` with *SIMD* and less than 1.5% in many cases. The discussion of the speedup gained by the SIMD/MIMD reconfiguration (see Section 13.3.1) has shown that the total overhead is mainly caused by transport code between scalar and vector registers. Furthermore, the waiting time at barriers is a severe bottleneck that influences many effects of the reconfiguration.

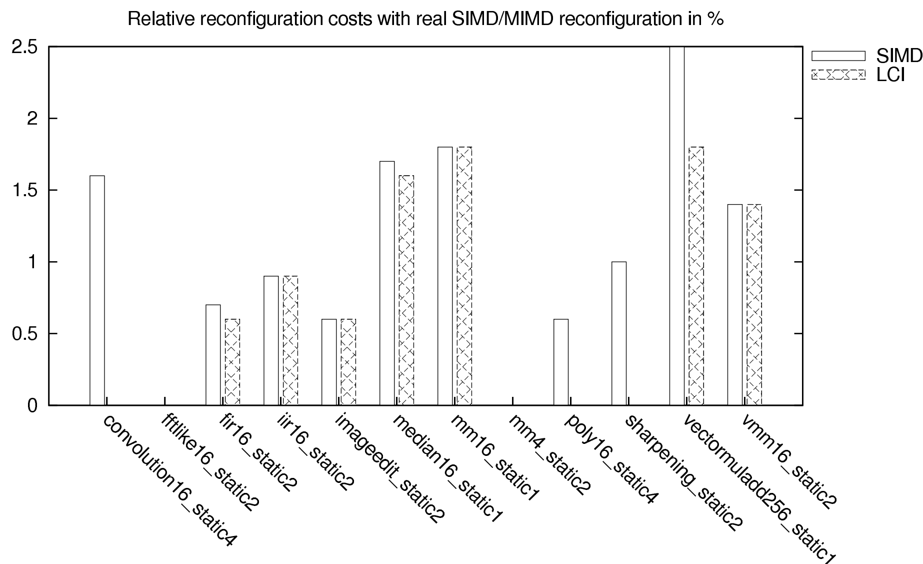


Figure 13.52.: Relative reconfiguration costs with real SIMD/MIMD reconfiguration in %

13.3.6. Costs of Communication

According to our results, the SIMD/MIMD reconfiguration is beneficial in terms of execution time, code size, and energy consumption, while the reconfiguration costs are negligible. On the other hand, the overhead of communication is still comparatively large with up to 12.8% as illustrated by Figure 13.53. Such costs can be reduced by a reconfiguration of the register banks. In Section 13.5, we will show that the combination of both reconfiguration dimensions yields a further speedup over the SIMD/MIMD reconfiguration.

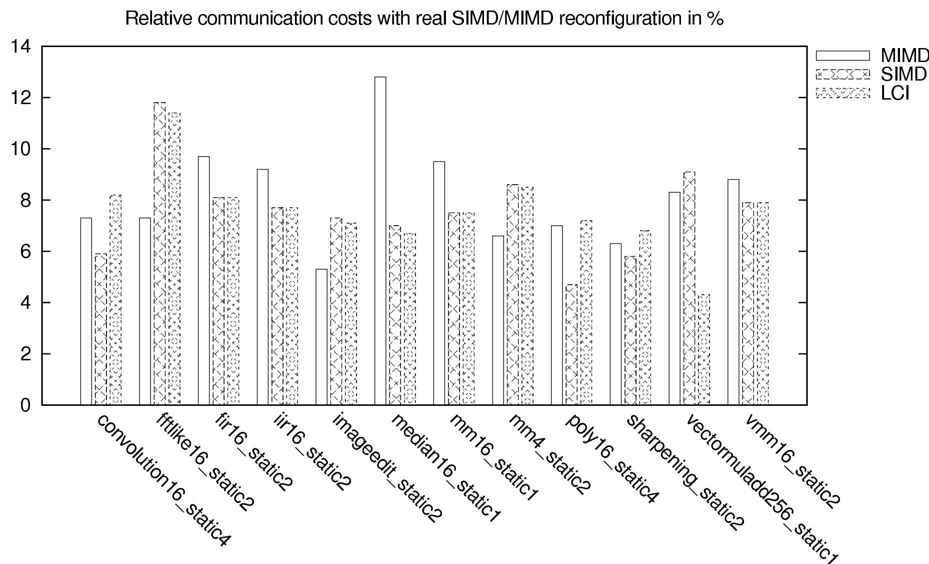


Figure 13.53.: Relative communication costs with real SIMD/MIMD reconfiguration in %

13.4. Reconfigurable Register Banks

Reconfiguring the connections to the registers (see Part IV) has two benefits: At first, the number of memory accesses can be reduced by utilizing more physical registers than are addressable in a certain architecture. Hence, a single processor can be augmented by further physical registers without changing the encoding of instructions or even enlarging the size of instruction words. For instance, the S-Core processor used in the QuadroCore has two register banks with 16 entries each, which can only be used alternatively by default (see Section 4.1). In a multi-core like the QuadroCore, a processor can borrow registers from another processor temporarily to avoid spilling. Last but not least, multiple processors can communicate very efficiently by using physical registers in a shared manner. Originally, communication in the QuadroCore is based on a dedicated shared register bank (see Section 6.1.1), where writing followed by immediate reading already takes 4 clock cycles. When using the external memory for communication, the delay is even much longer (12 to 30 clock cycles).

In this section, we discuss the evaluation of the register reconfiguration, which has been implemented in the CAPRICoRn[3] module (see Chapter 12) of the CoBRA compiler. For simplification, we concentrate on a machine consisting of multiple cores like the QuadroCore. The results for a single processor have been described in [47] and are only characterized shortly here. The remaining parts of this section are structured as follows: At first, the speedups are compared to the results of the parallelizing compiler for different optimization strategies in Section 13.4.1. Section 13.4.2 discusses the changings in code size influenced by additional reconfiguration instructions as well as saved communication and spill code. The reduction of the energy consumption in terms of stack accesses is considered in Sec-

[3]**C**ompiler **A**nticipated **P**rocessor **R**egister **I**nter-**Co**nnected **R**econfiguratio**n**

tion 13.4.3. Finally, Section 13.4.4 demonstrates that the reconfiguration costs are negligible.

Edge Conditions Concerning multiple processors, the implementation of the register reconfiguration is still in a prototypical stage (see Section 12.3). In particular, the aggressive re-use of physical registers by different processors can lead to cyclic dependences as well as dense schedules which do not allow insertion of reconfiguration or barrier instructions. Currently, such situations are simply avoided by extending the life spans at their final uses, if needed. As a drawback, the register pressure can be increased unnecessarily if two life spans overlap due to the extension only. Hence, our modification may interfere with the register reconfiguration aimed at reducing memory accesses and execution time. Using the current prototype of the CoBRA compiler, mainly those instances of our benchmarks (see Section 13.1.3) with a low to medium degree of parallelism can be compiled and simulated properly. Consequently, our evaluation concentrates on the execution times compared to a parallelization without register reconfiguration and neglects the total speedup over a single processor.

Due to the artificially increased register pressure, spilling may be even necessary when using register reconfiguration. With the current prototype of CAPRICoRn, a register of processor p is spilled by p, in principle, to balance the overhead among the cores. If no physical register is available for a life span during the coloring phase already, its virtual register is marked to be stored in memory. Such registers are spilled by the processor with the smallest stack frame to minimize expensive code for address computations (see Section 13.1.1).

Single Processor In [47], we have evaluated the register reconfiguration for a single S-Core processor with 16 architectural and 32 physical registers. Here, we give an overview of the main results and conclude some decisions for the evaluation with multiple processors. The reader may refer to the paper for further questions about the experiments.

Speedups of up to 50% have been obtained by a significant reduction of the overhead in spilling registers. The evaluation has been based on selected benchmarks with an inherent high register pressure, which was achieved by transforming arrays into a set of register variables. When using the CoBRA compiler for the QuadroCore, such a precondition is already generated by compiler optimizations like loop unrolling, copy propagation, and CSE. Software pipelining causes a great demand in registers, but is not considered here. Transport code between scalar and vector registers inserted by the CHARISMA module (see Section 8.4.1) can increase register pressure compared to a normal MIMD scheduling. On the other hand, programs with a high overhead of communication (see Section 13.2.4 and Section 13.3.6) but low to medium need in register pressure can also benefit from the register reconfiguration.

The evaluation for a single processor studied register architectures with block sizes 1, 2, 4, 8, such that the overall number of architectural registers is always 16. In general, using 4 architectural blocks with 4 registers each seems to be a good choice. Obviously, small block sizes require many reconfiguration instructions to activate all physical blocks containing needed register values. This effect becomes visible for larger programs, in particular, where much reconfiguration code is needed between different computations and around each function call, for instance. The approach by Kiyohara (see Section 9.3.2) constitutes an

extreme case with one reconfiguration instruction per access.

But a small number of architectural blocks may also be suboptimal: For instance, a 4x4 matrix multiplication accesses rows and columns of 4 registers repeatedly, which fit into single blocks naturally. When using an architecture with two architectural blocks of size 8, a row and a column or two rows/columns would probably be stored together. In both cases, more reconfiguration instructions are needed compared to an architecture with 4 architectural blocks in order to activate the blocks containing the operands of a single row/column multiplication.

If the number of architectural blocks is even smaller than the number of register operands of an instruction, special copy operations are needed to transport values between the blocks (see Section 10.1.3). For instance, Ravindran (see Section 9.3.1) developed a single processor architecture with two banks that can be used alternatively only. Some benchmarks like `convolution`, `median`, and `poly` access registers repeatedly in the same order. There, two architectural blocks are even slightly better than the architecture with block size 4.

The affinity analysis presented in Section 12.1 aims for reducing the overhead of reconfiguration by allocating registers of a common physical block for virtual registers often used in conjunction. Our evaluation in [47] has shown that the execution time can be decreased dramatically compared to a random assignment. The reconfiguration costs can be reduced further by two techniques: The inter-block placement of reconfiguration instructions re-uses mappings established in preceding blocks (see Section 12.2.2). Implicit reconfiguration during function calls and backward branches (see Section 12.2.3) can provide an initial set of mappings, which may be suited to the needs of the caller or callee, respectively. Consequently, the results discussed in the following are based on the mentioned techniques.

13.4.1. Execution Time

In this section, we consider the speedup by register reconfiguration compared to a normal parallelization with and without preceding loop unrolling. This corresponds to the optimization strategies opt0 and opt3 introduced at the beginning of Section 13.2. Duplication of induction variables is neglected, because it increases the number of live registers and duplicates code. Hence, it can be beneficial only when register reconfiguration is disabled and communication must be performed using a slow dedicated register bank or a shared memory. As classical optimization techniques only yield negligible speedups for our benchmarks, they have been disabled, too.

According to the results observed for a single processor, affinity analysis, inter-block placement, and implicit connections are always enabled. Also, we focus on register architectures with 2, 4, and 8 architectural blocks per processor and assume that each processor has 16 physical registers. The latter agreement should promote that processors borrow registers from each other in case of a high register pressure. In the following, the three architectures are identified by *aX_rY*, where X is the number of architectural blocks and Y is the block size.

The speedups are illustrated by Figure 13.54 and Figure 13.55. Without loop unrolling, *a2_r8* and *a4_r4* are slightly better than *a8_r2*. When loops are unrolled before parallelization, *a4_r4* is clearly better than the other architectures. As a consequence, an architecture with 4

architectural blocks of size 4 each should be a good choice for a practical application, again.

Without loop unrolling, register reconfiguration yields average speedups of 13.15%, 13.29%, and 12.12% for the three register architectures, respectively. In the best case, `mm4_local2` exhibits a speedup of 27.04% when using two architectural blocks. Provided that loop unrolling is applied beforehand, we even get performance improvements of up to 1035.3% for `vectormuladd256_local2` due to an enormous reduction of spill code. A slight deterioration is observed for `vectormuladd256_static1` due to a significant increase of the waiting time at barriers. The savings in spilling are very large for the QuadroCore, in particular, because additional `addi/subi` instructions are needed to increment/decrement the stack pointer when an offset value is too large for direct encoding in a memory instruction. In order to get more realistic results, the spilling capabilities of the S-Core processors should be improved in the future. Hence, we do not present average speedups here, which are very high as well.

Interestingly, the instances with global data exhibit lower speedups than those with local data structures. This seems to be a side effect of the comparatively low parallelism in most of the benchmarks. The overhead of synchronization between accesses to the external memory is enormously high, while no barriers are required for the local benchmarks. With respect to our benchmarks, the processors wait up to 64.5% of the execution time at barriers (`convolution16_static1`). As mentioned for the results of the vectorization (see Section 13.3), external memory access and synchronization seem to be important bottlenecks of the QuadroCore.

In the following, we outline selected benchmarks in detail. The above characterization of the results should be sufficient for a short overview.

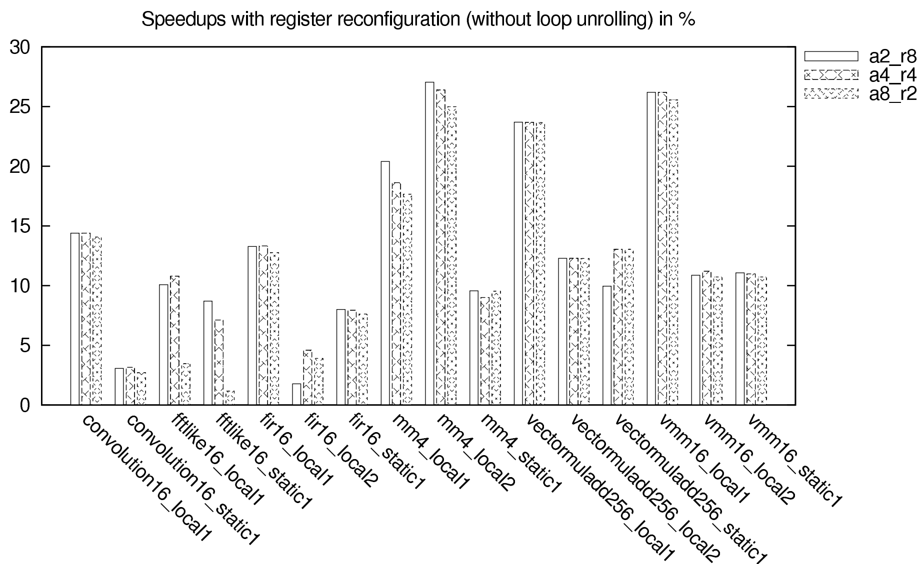


Figure 13.54.: Speedups with register reconfiguration (without loop unrolling) in %

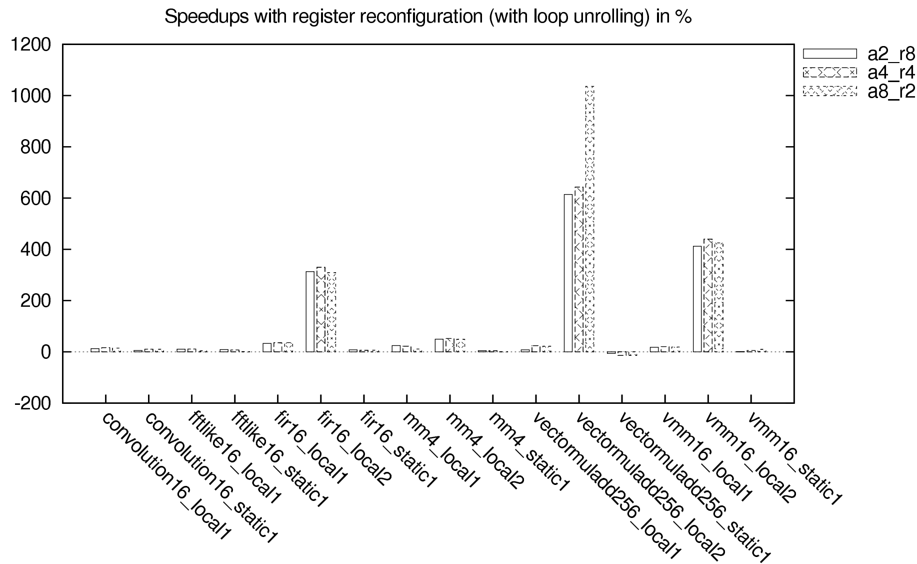


Figure 13.55.: Speedups with register reconfiguration (with loop unrolling) in %

Discussion of Selected Benchmarks (Without Loop Unrolling) Without register reconfiguration, the relative communication costs of the studied benchmarks vary between 3.2% and 13.4%. If loop unrolling is not applied beforehand, the register pressure on each processor is lower than the number of physical registers such that no spilling is required. Hence, register reconfiguration can be used to reduce the communication costs by utilizing some physical registers in a shared manner.

A speedup of more than 14% is achieved for `convolution16_local1` due to a reduction of the communication costs. Without register reconfiguration, 3536 communication operations are executed (relative costs: 6.4%), while only 113 `connect` instructions are needed when register reconfiguration is enabled (*a2_r8*).

a8_r2 achieves comparatively small or even negative speedups for `fftlike16_static1`, because it executes about factor 2.12 more `connect` instructions than *a4_r4*. Consequently, the number of `nop` instructions used to fill empty slots within the reconfiguration code raises by 9.7%. A similar observation can be made for `fftlike16_local1`.

When considering two instances of a benchmark with different degrees of parallelism, the relative overhead of communication is directly propertional to the speedup by reconfiguring the register banks in three of four cases. For instance, `fir16_local1` has speedups of about 13% (relative communication: 6%), while `fir16_local2` only exhibits up to 4.59% (relative communication: 3.2%). The same applies for `vectormuladd256_localX` and `vmm16_localX`. On the other hand, `mm4_local2` achieves speedups of up to 27.04%, but `mm4_local1` only up to 20.4%. This is justified by a reduction of executed `nop` instructions by 23.7% for `mm4_local2`, while `mm4_local1` only saves 15.3% (*a2_r8*). Less `nop` instructions means a better utilization of the processors and hencefore a shorter execution time. Concerning the other three benchmarks, a greater reduction is observed for the class 1 instances. When register reconfiguration is applied for `mm4_local2`, 8853 `connect` instructions are needed compared to 25812 communication instructions (relative communication: 13.4%)

vmm16_local1 is accelerated by about 26% when reconfiguring the register banks. Again, communication only plays a minor role, while the number of nop instructions is reduced by 24.6% (*a2_r8*). 124 connect instructions are needed versus 2484 communication operations without register reconfiguration.

Discussion of Selected Benchmarks (With Loop Unrolling) When loop unrolling is applied beforehand, more parallelism is uncovered for the parallelization. On the other hand, executing four loop bodies in parallel also increases the need of physical registers. This effect can be observed for our benchmarks in particular, because the heuristic of our list scheduler neglects the register pressure. Consequently, some processors need more physical registers than available, while others still have some free registers. Such a situation is well-suited for our register reconfiguration, where a processor can borrow register temporarily to avoid spilling. As a result, we can observe speedups of up to factor 10. The improvement may be even much higher when concerning benchmarks with more parallelism.

In the following, we focus on the class 2 instances of the benchmarks, because the speedup observed for the remaining ones mainly depends on the acceleration of the inter-processor communication. For instance, the number of memory store operations is only reduced by 1.2% for convolution16_local1 due to register reconfiguration (*a4_r4*). Instead, 1720 connect instructions are needed to avoid 4383 communication instructions (relative communication: 9.3%).

With respect to fir16_local2, the maximum cut of the first two processors is 23 and 20, respectively, while the other ones only need at most 8 or 7 registers. Register reconfiguration balances the utilization of physical registers and avoids a lot of spill code such that the execution time is reduced by more than factor 3. Concretely, 66.5% less accesses of the stacks are performed, while the number of addi/subi instructions even decreases by factor 27.7 and 72.2, respectively (*a4_r4*).

Similar observations can be made for vectormuladd256_local2 and vmm16_local2, where much greater speedups are achieved. The dramatic overhead of additional code to increment/decrement the stack pointer for spilling becomes very clear for vectormuladd256_local2, where the number of subi instructions is reduced by factor 11170 (*a8_r2*).

A performance loss by register reconfiguration is only observed for vectormuladd256_static1 where the number of waiting cycles is increased by 66%. Simultaneously, 2828 communication instructions are saved at a cost of 1256 connect instructions.

13.4.2. Code Size

The discussion of the speedups (see Section 13.4.1) has shown that the register reconfiguration can reduce the overhead of spilling and communication dramatically, while the reconfiguration costs are rather small (see also Section 13.4.4). As a consequence, the code size may also decrease due to saved communication and spill code. Figure 13.56 and Figure 13.57 illustrate the reduction of the code size by register reconfiguration.

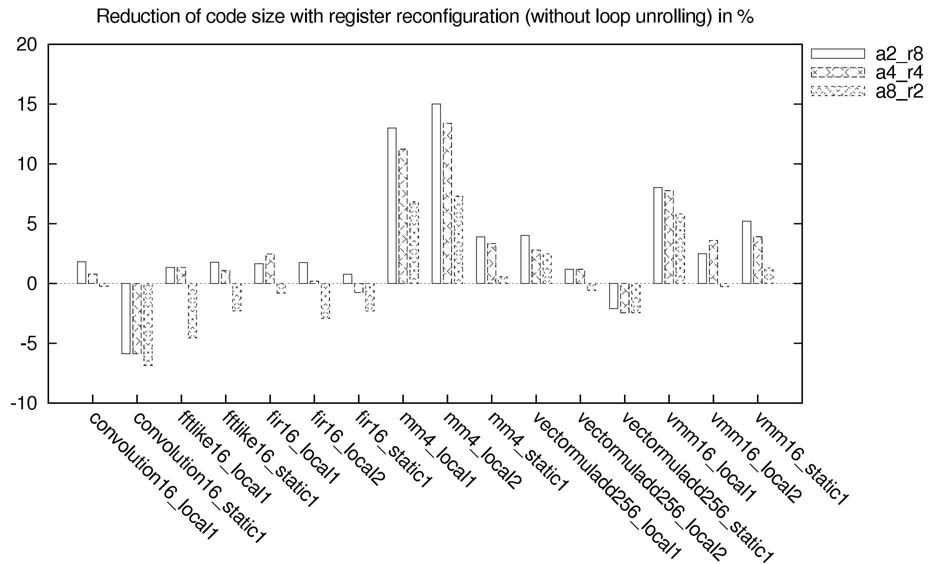


Figure 13.56.: Reduction of code size with register reconfiguration (without loop unrolling) in %

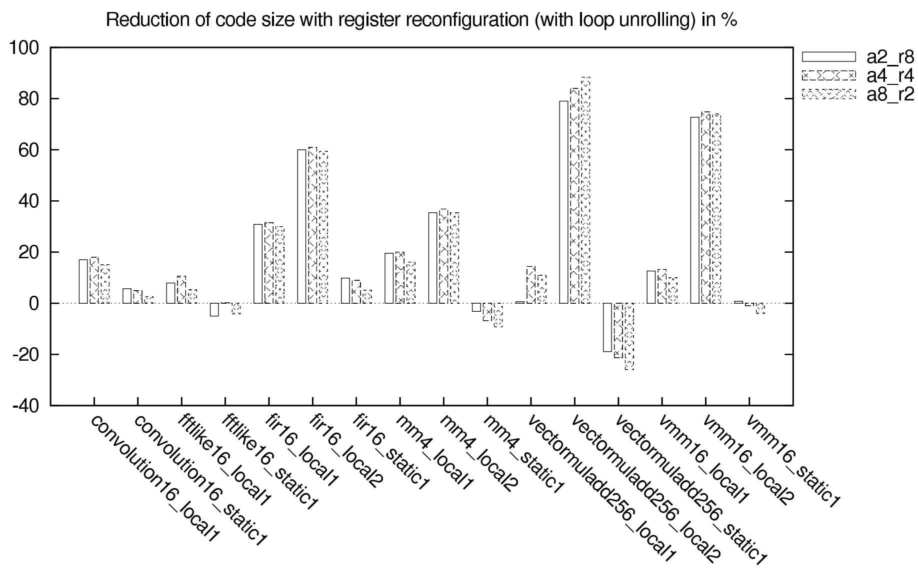


Figure 13.57.: Reduction of code size with register reconfiguration (with loop unrolling) in %

Without Loop Unrolling When loop unrolling is disabled, we only save communication instructions, but need additional `connect` instructions to reconfigure the connections between architectural and physical blocks. In average, the savings for the three register architectures are 3.37%, 2.75%, and 0.065%, respectively. The greatest reduction of 15.01% can be observed for `mm4_local2` with *a2_r8*, which also encountered the largest speedup by register reconfiguration (see Figure 13.54). Here, the number of `nop` instructions in code is reduced by 29.9%. 36 `connect` instructions are needed compared to 65 saved communica-

tion instructions.

The code size of `vectormuladd256_static1` even increases by 2.45% for *a4_r4*, because the speedup of 13.04% is mainly achieved by a reduction of the waiting time at barriers by 14.7%. Concerning the program code, 56 instead of 40 `nop` instructions are needed when register reconfiguration is enabled. This increase of 40% has a great impact, because the absolute size of the program code is quite small (1164 bytes for *a4_r4*).

In other cases such as `fir16_local1`, the increase is caused by alignment code. Some branch and call instructions or the operation to the load address of a global variable require additional 32 bit data after the instruction word. As the instructions are only 16 bit wide, alignment may be needed between the instruction itself and the supplementary data. Such phenomenon and the small absolute sizes seem to impede a significant reduction of the code size in general.

With Loop Unrolling With loop unrolling, much greater reductions of up to 88.39% for `vectormuladd256_local2` can be observed due to the huge savings in spill code (see Section 13.4.1). Concretely, the number of `addi` and `subi` instructions in code is reduced by factor 100 and 243, respectively. Such fact holds for the remaining class 2 instances of the benchmarks as well.

A negative example is `vectormuladd256_static1`, which suffers from an increase of the code size of up to 25.92% for *a8_r2*. This is justified by a raising of the `barrier` instructions from 86 to 128 (48.8%) and corresponds to the performance loss of about 14% due to longer waiting times.

We omit average values of the savings in code size, because of the enormous overhead of spilling for parallelized unrolled loop bodies (see Section 13.4.1).

13.4.3. Power Consumption

In the two previous sections, we have demonstrated that the register reconfiguration can achieve great speedups, while achieving smaller code sizes as well. A further benefit can be the reduction of the energy consumption by decreasing the overhead of spilling and communication. Spilling requires an access of the stack memory and may even imply executing many `addi/subi` instructions to update the stack pointer (see Section 13.1.1). Communication is based on a dedicated register bank by default and is performed completely using the registers in a shared manner, if register reconfiguration is enabled. Currently, power estimations for a hardware prototype are missing, because the register reconfiguration has not been implemented yet in hardware.

As the greatest savings concerning execution time and code size have been obtained by reducing spill code, we study the reduction of the energy consumption in terms of stack accesses here. Figure 13.58 shows the decrease for our benchmarks provided that loop unrolling is enabled. In most cases, we can observe reductions of up to 66.54% for `fir16_local2`, which also exhibits a speedup of more than factor 3. The same applies for the remaining class 2 benchmarks. We do not present average values here, because the parallelization of unrolled loop bodies causes a very high register pressure leading to a distortion

of the mean results.

For `fftlike16_static1`, the number of stack accesses even increases by 15% (*a2_r8*) due to comparatively small absolute values (40 to 46). Other benchmarks with global data exhibit no or slight reductions, because the register pressure is not high enough.

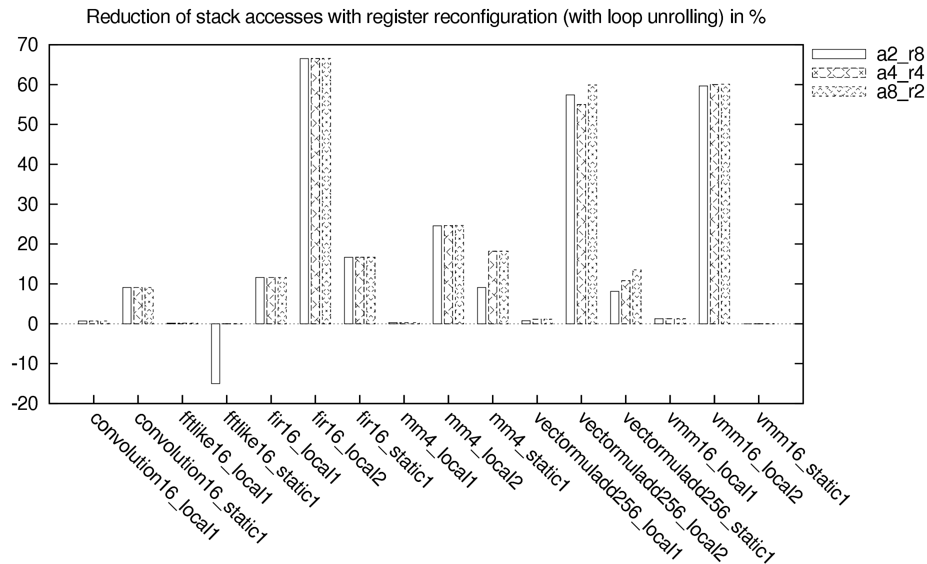


Figure 13.58.: Reduction of stack accesses with register reconfiguration (with loop unrolling) in %

13.4.4. Costs of Reconfiguration

Finally, we consider the overhead of reconfiguring the connections between architectural and physical blocks. The execution of the `connect` instruction (see Section 13.1.1) needs a single clock cycle only. For comparison, an access to the dedicated register bank used for communication takes 2 cycles already. Spill code needs at least 3 cycles for accessing the local memory of a processor and may be much higher due to address computation.

Figure 13.59 illustrates the reconfiguration costs provided that loop unrolling is disabled. The costs with preceding loop unrolling are visualized by Figure 13.60.

Without loop unrolling, the relative overhead is less than 1% for most benchmarks. `fftlike16` and `mm4` spent up to 5.1% of the total execution time of all processors in reconfiguration, but also achieve speedups of up to 27.04%. Hence, the reconfiguration costs are clearly compensated by a reduction of the communication costs.

When loop unrolling is enabled, the overhead in register reconfiguration is slightly larger. Avoiding a lot of spill code may require to update the connections between architectural and physical registers more frequently. The maximum relative costs of 6.8% can be observed for `mm4_local1` with *a8_r2*, where a performance improvement of more than 10% has been obtained, however.

Consequently, the overhead of reconfiguration can be neglected or is counterbalanced by an improved resource efficiency. The waiting time at barriers and the access of the external

memory is a much greater bottleneck that can hide the benefits of reconfiguration.

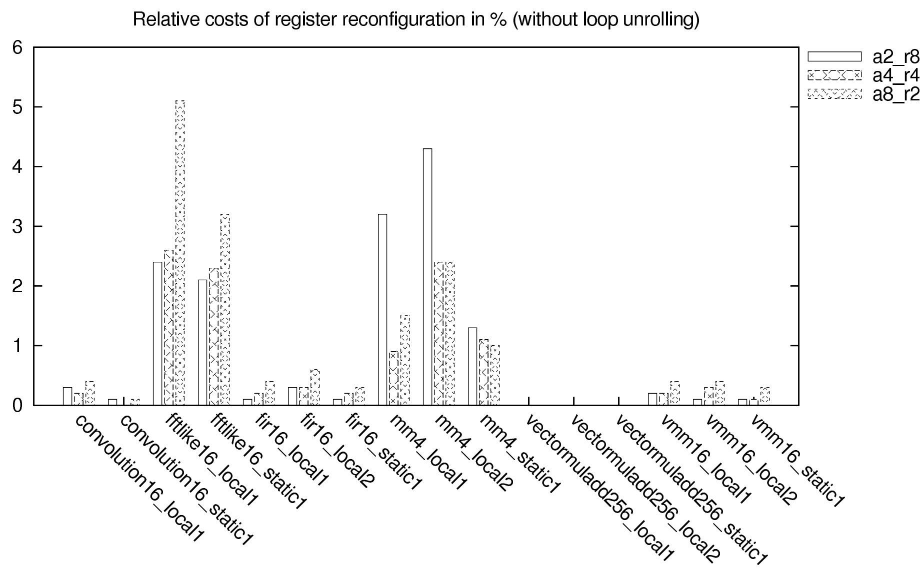


Figure 13.59.: Relative costs of register reconfiguration in % (without loop unrolling)

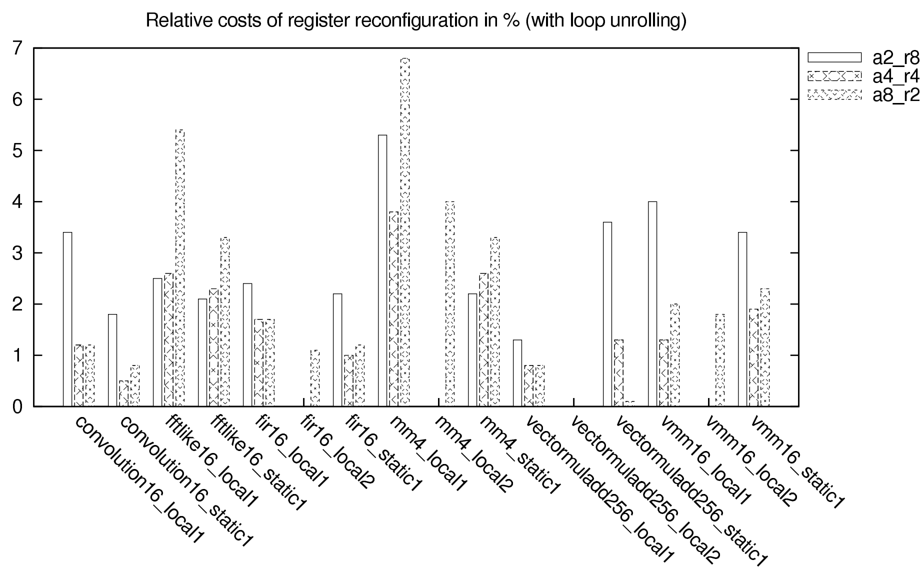


Figure 13.60.: Relative costs of register reconfiguration in % (with loop unrolling)

13.5. Combination of SIMD/MIMD and Register Bank Reconfiguration

Up to now, we have studied the results of the two reconfiguration dimensions separately. In both cases, the resource efficiency can be improved significantly in terms of execution time, code size, and energy consumption. Section 13.3.6 motivated to augment a SIMD/MIMD

execution by a reconfiguration of the register banks in order to reduce the overhead of communication. Furthermore, spill code can be avoided by assigning the physical registers of the QuadroCore to the processors on a need basis. On the other hand, programs which benefit from reconfiguring the connections to the registers may also contain regular structures well-suited for a SIMD execution. As a consequence, this section studies the combination of both reconfiguration dimensions.

The edge conditions for our measurements derive naturally from the intersection of the prerequisites of our preceding evaluations. Concretely, we focus on benchmarks with global data structures, because wide memory accesses in SIMD mode must refer to the external memory of the QuadroCore. Loop unrolling is always enabled as a mandatory requirement of our vectorizer. We focus on the local code integration (*LCI*), which has outperformed code generated by the SLP vectorizer in most cases (see Section 13.3.1).

As the application of both reconfiguration dimensions is a novel feature of the CoBRA compiler, we consider a rather small number of benchmarks with low degree of parallelism. Hence, the total speedup over a single processor may not be as high as expected. Consequently, we concentrate on the speedup compared to a MIMD scheduling with prior loop unrolling. Section 13.2.1 has demonstrated that the parallelization of benchmarks with inherently high parallelism can obtain speedups of up to factor 4 over a single processor. In [87], we have shown that it can accelerate the program execution by more than factor 4, because the well-balanced register need may avoid much spill code in contrast to a single processor. Provided that the CoBRA compiler will become technically mature, the two reconfiguration dimensions may outperform aggressively parallelized code further. In the following, we give an overview about the results, but do not discuss them in detail as done above.

13.5.1. Execution Time

Figure 13.61 illustrates the speedup of SIMD/MIMD reconfiguration and/or register reconfiguration over a pure MIMD scheduling. The application of both reconfiguration dimensions always yields speedups over a single dimension or a normal parallelization. In the best case, `fir16_static1` is accelerated by even 45.4% for *LCI-a8_r2*. Importantly, the deterioration of the execution time of `vectormuladd256_static1` for the register reconfiguration is compensated by the SIMD/MIMD reconfiguration. In the other cases, the register reconfiguration improves the results of the SIMD/MIMD reconfiguration significantly.

The average speedups of both reconfiguration dimensions for the three register architectures are 35.61%, 34.18%, and 36.75%, respectively. As the results only vary slightly, an architecture with 4 architectural blocks per processor containing 4 registers each seems to be a good choice. The same observation has been made for the register reconfiguration on its own (see Section 13.4.1). The evaluation of a larger set of benchmarks with higher degree of parallelism is probably necessary to get representative data.

13.5.2. Ratio of SIMD/MIMD Execution

For completeness, Figure 13.62 shows the ratio of cycles executed in SIMD mode. As we have selected benchmarks which benefit from a vectorization, the fractions range from 16%

to 38%. We skip a detailed discussion for simplification and refer to Section 13.3.2.

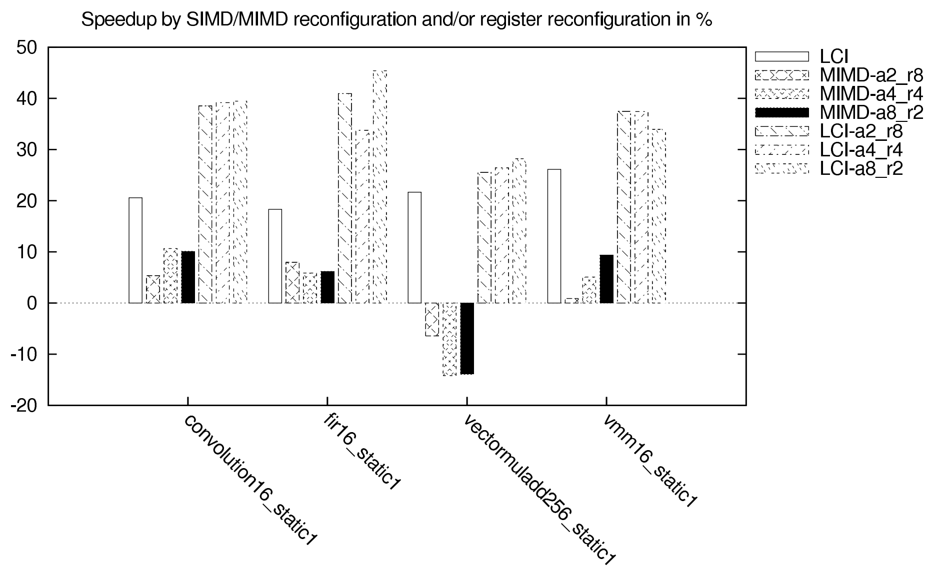


Figure 13.61.: Speedup by SIMD/MIMD reconfiguration and/or register reconfiguration in %

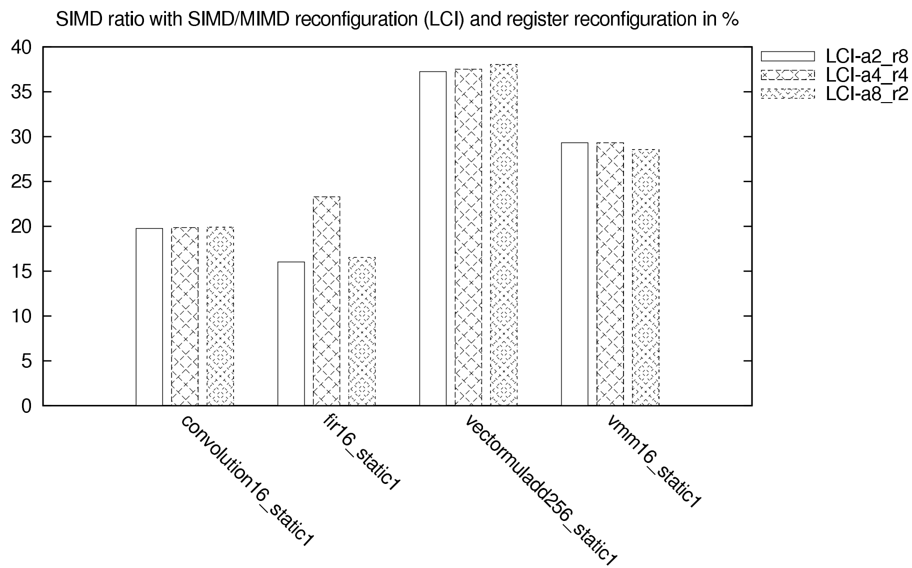


Figure 13.62.: SIMD ratio with SIMD/MIMD reconfiguration (LCI) and register reconfiguration in %

13.5.3. Code Size

Like in Section 13.3.3, the reduction in code size is computed by comparing the size of code produced by using the real and the pseudo SIMD/MIMD reconfiguration, respectively. The obtained results are illustrated by Figure 13.63.

Using a single SIMD code stream achieves savings of up to 16.61% for `vectormuladd256_-static1`. This reduction is comparable to the result of the SIMD/MIMD reconfiguration on its own and demonstrates that the benefits of the vectorization are preserved by the register reconfiguration.

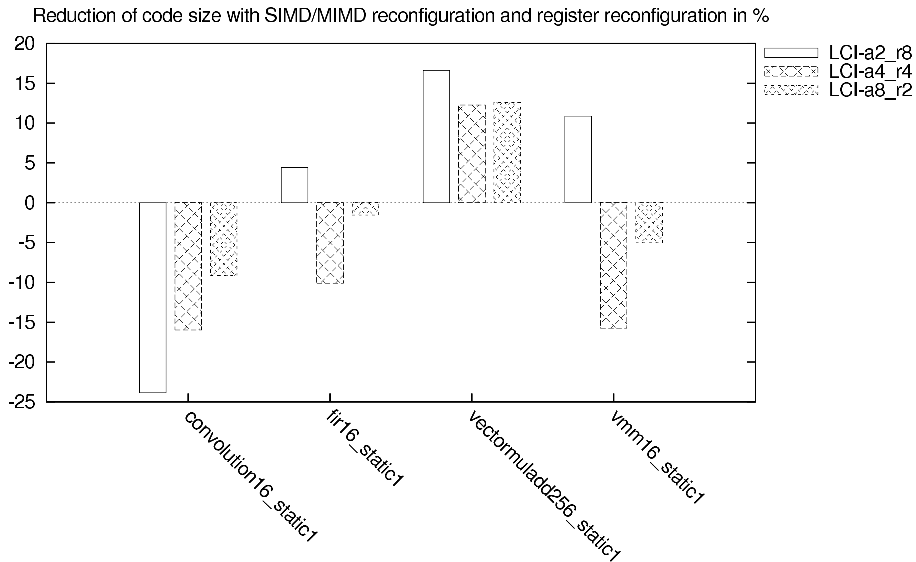


Figure 13.63.: Reduction of code size with SIMD/MIMD reconfiguration and register reconfiguration in %

In the other cases, the code size is not reduced or even increases by up to 23.86% for `convolution16_static1` with *a2_r8*. The machine code of the affected benchmarks contains up to factor 4 more `nop` instructions compared to pseudo SIMD. This may be caused by a poor parallelization due to the low parallelism or a high number of empty slots near additional instructions like transport or spill code.

13.5.4. Power Consumption

The reduction of the energy consumption has been evaluated in terms of accesses to the instruction memory and the switching activities of instruction fetch and program counter. Further information can be found in Section 13.3.4. The savings are illustrated by Figure 13.64 to Figure 13.66. We achieve reductions of up to 33-34% for all three issues, which correlate best to the SIMD ratios. A detailed discussion is omitted.

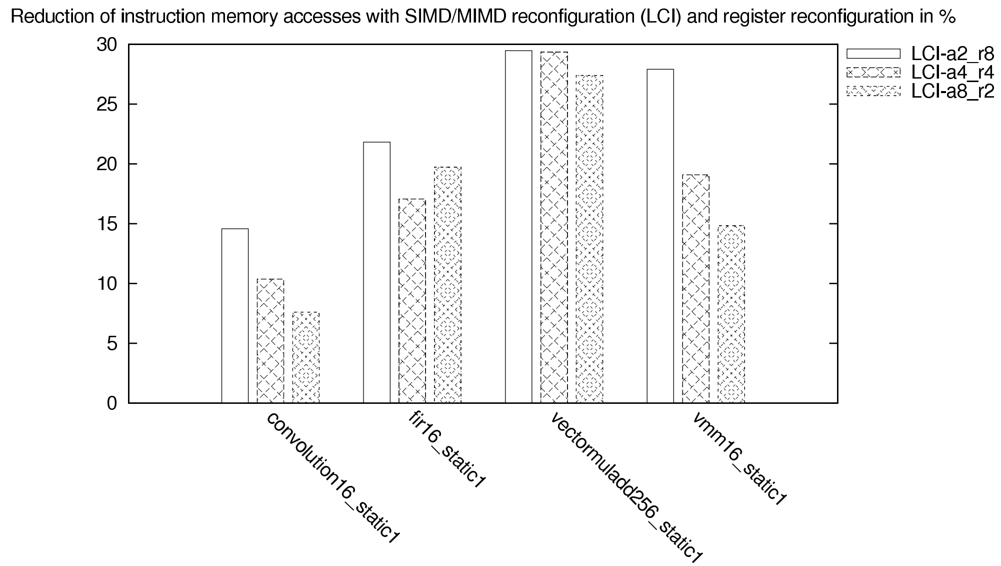


Figure 13.64.: Reduction of instruction memory accesses with SIMD/MIMD reconfiguration (LCI) and register reconfiguration in %

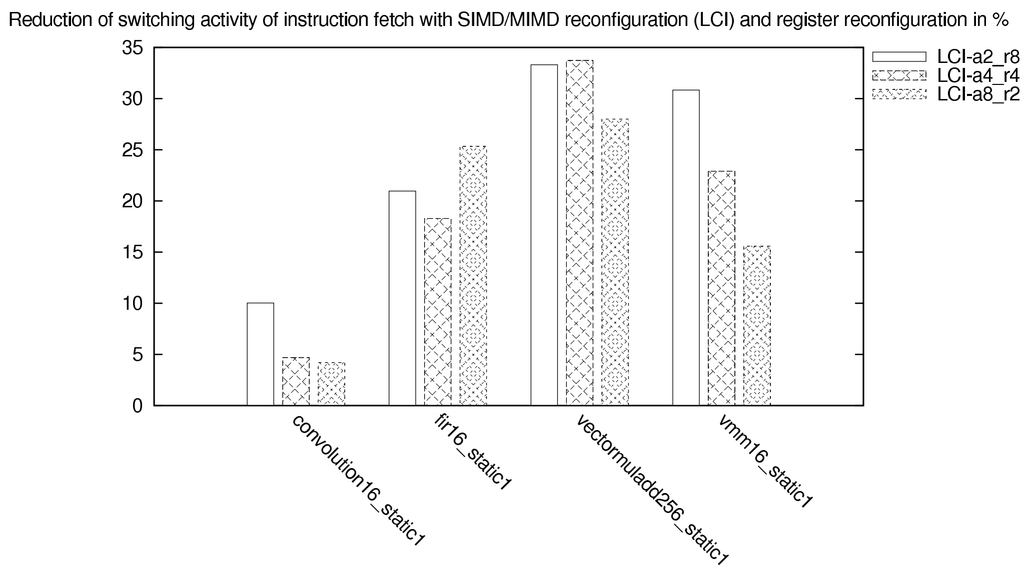


Figure 13.65.: Reduction of switching activity of instruction fetch with SIMD/MIMD reconfiguration (LCI) and register reconfiguration in %

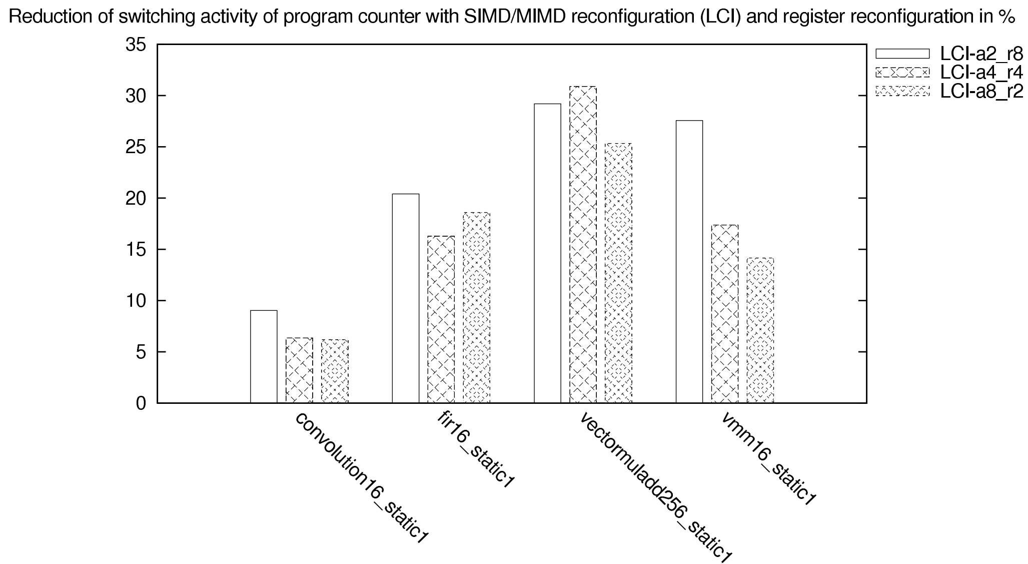


Figure 13.66.: Reduction of switching activity of program counter with SIMD/MIMD reconfiguration (LCI) and register reconfiguration in %

13.5.5. Costs of Reconfiguration

The overhead of the SIMD/MIMD reconfiguration is negligible again (see Figure 13.67) and only amounts up to 2.55%. Such overhead is clearly compensated by the significant speed-ups (see Figure 13.61). The costs of reconfiguring the register connections (see Figure 13.68) is slightly larger and smaller than 5% in most cases. Only for `vmm16_static1`, the costs reach 8.2% with *a8_r2* due to the higher number of architectural blocks.

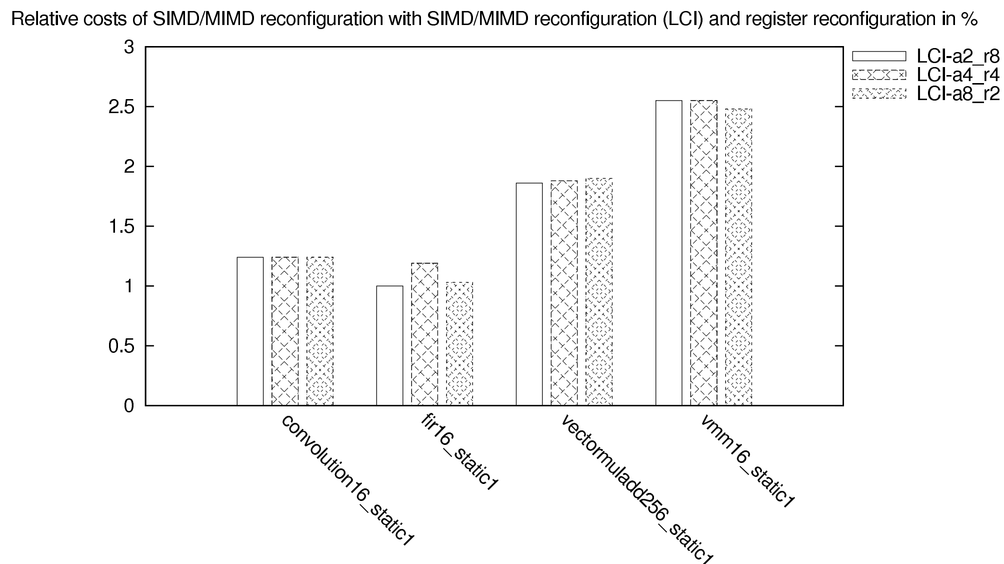


Figure 13.67.: Relative costs of SIMD/MIMD reconfiguration with SIMD/MIMD reconfiguration (LCI) and register reconfiguration in %

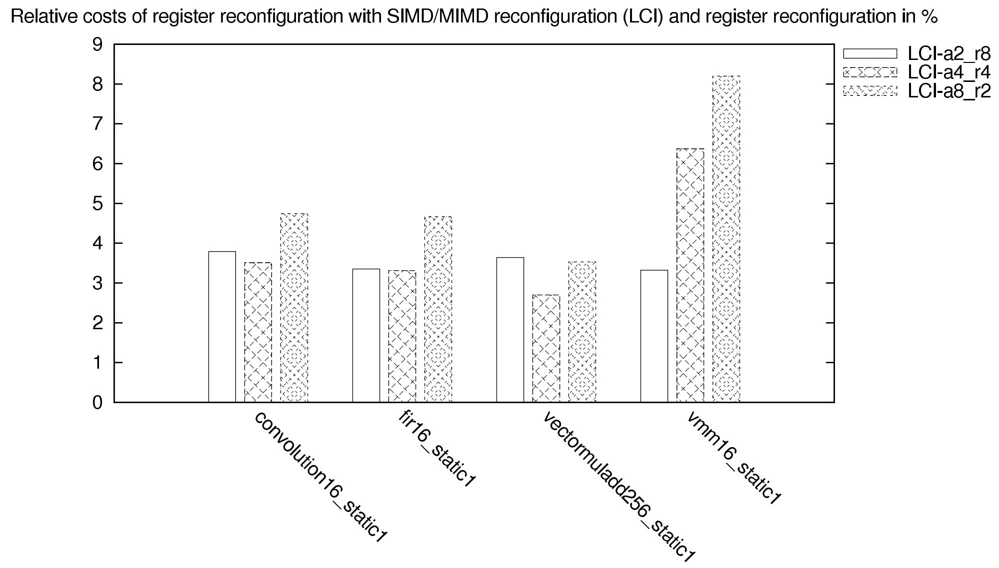


Figure 13.68.: Relative costs of register reconfiguration with SIMD/MIMD reconfiguration (LCI) and register reconfiguration in %

14. Conclusion and Future Work

Contents

14.1 Conclusion **V-68**
14.2 Future Work **V-71**

14.1. Conclusion

In this thesis, we have presented a holistic hardware/software approach called CoBRA[1] for the reconfiguration of processors. The reconfigurable machine provides a set of modes, called *reconfigurable architectural variants* or briefly *variants*. Such variants are known to the compiler and are supported by appropriate program analysis techniques. The CoBRA compiler determines the best variant for each piece of code with respect to static estimations or profiling data. Programs can be optimized for a fast execution, small code size, or low power dissipation. Reconfiguration between sections using different variants is performed by executing special instructions at run-time.

Five promising reconfiguration dimensions supported by the CoBRA approach have been suggested in Section 3.1. We have focused on two opportunities: If program code includes both regular and non-regular structures, the machine can switch between SIMD and MIMD execution in order to achieve best results. Reconfiguring the connections to the register banks enables a dynamic assignment of physical registers to the processors according to the current need. A fast communication can be established when multiple processors use a common set of registers as a shared resource.

Benefits The CoBRA compiler encapsulates much expert knowledge about optimizing code generation and reconfigurable variants in order to enable a transparent development process based on HLL programming. Time-to-market and debugging effort can be minimized at a lower risk and higher profit. The design space of machine configurations is scaled down significantly by using a manageable set of variants, which have turned out to be beneficial for the targeted application domains. Hence, the CoBRA compiler can focus on those problems which can be solved efficiently using well-known program analysis techniques, while irrelevant solutions are neglected in the first place. In contrast to fine-grained reconfigurable devices, the CoBRA approach is based on common processors augmented by reconfigurable interconnections between fixed, coarse-grained components. For instance, a SIMD mode can be achieved by connecting the ALUs of all processors with the decoder of the first processor, while the remaining decoders are turned off. A single code stream is decoded by the first processor and executed on all cores with different data. Reconfigurable connections between ALUs and register banks enable a flexible assignment of registers to processors. As a consequence, reconfiguration for the purpose of CoBRA can be performed at modest costs.

SIMD/MIMD Reconfiguration Reconfiguration between SIMD and MIMD execution is implemented in the CHARISMA[2] module of the CoBRA compiler. In principle, it applies both scheduling and vectorization techniques on given parts of a function. The current implementation of the CoBRA compiler only provides parallelization on basic block level, for simplification. Finally, the best result for each basic block is selected based on the estimated execution time.

The CHARISMA vectorizer utilizes the novel SLP approach by Larsen et al. [107] for vec-

[1] **Co**mpiler-**D**riven **D**ynamic **R**econfiguration of **A**rchitectural Variants (merge two *D*s to a *B*)
[2] **C**ompiler **H**andles **A**rchitectural **R**econfiguration **I**ntegrating **S**IMD **M**IMD **A**utomatically

torization. It targets regular parallelism in sequential code, while vector parallelism is exploited by unrolling loops. The fundamental idea of the approach is to determine adjacent memory accesses. Our vectorizer computes adjacency by an original method which is based on the idea of CSE (see Section 8.3.1). Further vectorizable constructs are found by inspecting the def-use/use-def chains of the operands. The final phase of the SLP approach schedules the operations of a basic block. Cyclic dependences between vectorized instructions are resolved by generating MIMD instructions. Our SLP scheduler aims for maximizing contiguous pieces of code using a single execution mode and inserts reconfiguration code where needed.

As the processors of the QuadroCore have disjoint register banks, we decided that vector registers should only exist conceptually. The entries of the vector register i correspond to the i-th registers of the processors (see Section 8.2.1) to minimize changes to the existing architecture. Hence, an instruction accessing vector register i is executed by each processor using its register i. This decision enables the usage of a single code stream in SIMD mode, but requires that the register allocation takes this special property into account. The fundamental idea of our approach is to determine virtual vector registers by combining the virtual registers of the vectorized instructions (see Section 8.4). The vector registers are initialized with the scalar registers before switching from MIMD to SIMD execution, and vice versa. We have developed a heuristic placement algorithm which tries to minimize the number of needed transport instructions. The resulting code serves as an input for register allocation by graph coloring [32].

Register Reconfiguration In our reconfigurable register architecture (see Chapter 10), physical registers are accessed indirectly through architectural registers. The effort in reconfiguration is reduced by partitioning the registers into blocks of a common size. Connections are established between architectural and physical blocks by executing a special reconfiguration instruction. The register allocation is performed in two phases (see Chapter 12) within the CAPRICoRn[3] module of the CoBRA compiler: At first, physical registers are allocated using graph coloring. The second phase inserts reconfiguration instructions to map a physical block containing a needed register into the architectural block of a processor.

In order to reduce the number of reconfiguration instructions, the first phase aims for allocating registers of a single physical block for values that are often accessed close together. Concretely, the affinities between virtual registers v_1, v_2 are computed which model the number of reconfiguration which are avoided when v_1 and v_2 are assigned to the same physical block. Such affinities are used during coloring as a secondary criterion. Our inter-block placement strategy tries to re-use mappings established in preceding basic blocks (see Section 12.2.2). Both optimizations are based on an original data-flow analysis called n-liveness (see Chapter 11), which can compute the n pair-wise different values accessed after a certain program position. Obviously, such information can be used to decide if a physical block is replaced between two accesses of a virtual register or until the beginning of the next basic block.

For simplification, we have assumed an unlimited number of read ports to the register banks, but a register can only be written by at most one processor per cycle. Read accesses

[3]**C**ompiler **A**nticipated **P**rocessor **R**egister **I**nter-**Co**nnected **R**econfiguratio**n**

are performed in the first half of a cycle, while write accesses occur only in the second half. A processor can access an arbitrary physical register by mapping the corresponding physical block to one of its architectural blocks. Hence, the CAPRICoRn module treats the QuadroCore as a machine with a global set of physical registers, but indirect access via architectural frames.

Further Variants During the development of the CoBRA compiler, we have identified further reconfiguration variants with respect to parallelization. In the pseudo SIMD mode, multiple code streams resulting from a vectorization are executed in a MIMD manner (see Section 8.2). Hence, the overhead of reconfiguration and data re-organization is avoided. On the other hand, using a single code stream in the real SIMD mode enables to reduce the code size and the power consumption. Such properties have also been observed for the results of the evaluation, which is outlined in Chapter 13. Deciding between real and pseudo SIMD may be realized by passing the optimization goal like short runtime or small code size as a command line option to the compiler.

Coarse-grained parallelism is suited for an asynchronous execution with explicit barrier synchronization, while fine-grained parallelism is supported ideally by a lock-step operation. In Section 13.1.2, we have proposed a reconfiguration between both paradigms called VLIW/Barrier reconfiguration in order to adapt to the granularity of parallelism found in a given piece of code. The decision can be based on the number of inter-processor dependences.

Parallelizing Compiler In addition, several techniques concerning automatic parallelization for homogenous multi-cores have been developed. Our data partitioning method can distribute local data to the processors based on an affinity graph (see Chapter 5). Currently, allocation of functional units is still based on the BUG algorithm by Ellis [50] (see Section 5.1). Hencefore, we have proposed a holistic processor partitioning method using affinity graphs. Such approach can be extended to code adaption at load-time or run-time of a program based on compiler annotations. Inter-processor communication between the processors of the QuadroCore utilizes a dedicated register bank. The CoBRA compiler inserts communication code according to the results of the parallelization by using several placement strategies (see Section 6.1). Synchronization between the processors is achieved by barriers which are also added automatically during the re-scheduling phase (see Section 6.2).

Results of Evaluation The CoBRA approach has been evaluated for the two studied reconfiguration dimensions using a cycle-accurate software simulator of the QuadroCore. The evaluation has been based on several medium-sized benchmarks which can occur in practical audio and video applications and represent typical transcoding algorithms for aggregation network access nodes. The results are characterized at the beginning of Chapter 13. A synthesized model was used to estimate properties of the hardware implementation in terms of area and performance. With respect to the non-reconfigurable QuadroCore, the synthesized architecture exhibits an increase of about 10% of area, while the maximum operating frequency of the system decreases by about 5%.

14.2. Future Work

The CoBRA approach has a number of benefits over existing reconfigurable architectures and tools: efficient compilation utilizing a multitude of program analysis and code generation techniques, transparent programming using a HLL, fast reconfiguration of a target machine as well as support of arbitrary application domains. Our evaluation has demonstrated its practical impact by considering the SIMD/MIMD and the register reconfiguration. In the following, we outline several directions of future work.

Reconfiguration Dimensions/Variants On a methodical level, CoBRA can be extended by further reconfiguration dimensions (see Section 3.1). For instance, the topology of the multi-core can be reconfigured to a pipeline organization in order to adapt to repeated computations including several dependent tasks such as encryption, compression, or error correction.

Additionally, a clear procedure for the integration of new dimensions and variants needs to be developed. We believe that this requires to review the relation of *dimension* and *variant*: Within the dimension *parallelization*, reconfiguration selects among multiple execution modes using different kinds of parallelism. The *register access* dimension has two different levels: At first, a machine can be switched between different types of mapping configurations like identical, restricted, or arbitrary. The second level actually allows reconfiguring the register connections if supported by the first level. As a consequence, a future version of CoBRA should provide a scheme to classify new opportunities for reconfiguration in order to enable an efficient integration into the CoBRA compiler.

Compiler-Driven and Compiler-Supported Reconfiguration In Section 3.2, we have introduced two application scenarios for the CoBRA approach: The compiler-driven reconfiguration assumes that the target machine is known precisely at compile-time. Hence, parallelization and code generation for selected reconfiguration variants can be done by the compiler. If the target machines vary in the number of processors or some processors are broken, scheduling and decisions for reconfiguration have to be delayed to the load-time of a program. Hardware defects may even force a dynamic adaption of program code at run-time.

The current implementation of CoBRA concentrates on the compiler-driven scenario. Section 5.3.2 suggests a concept for parallelization at load-time which is supported by code annotations computed at compile-time. The proposed approach extends our processor partitioning method using affinity graphs. But preparing the selection of reconfiguration variants and corresponding code generation will probably require a lot of further research. In Section 3.2.3, we have motivated that the division of CoBRA tasks between compile-time and load-time is a non-trivial challenge and probably needs to be done for each reconfiguration dimension on its own.

Practical Implementation In principle, the selection of a certain variant for a given piece of code can be based on program analysis, which suggests that a decision is made before code generation. A practical implementation can also apply several techniques and determine the

best variant by evaluating the results according to different criteria like execution time, code size, and power consumption. Such procedure has been used for the SIMD/MIMD reconfiguration also. On the other hand, we have experienced that a well-structured and modular compiler implementation is crucial to realize this strategy. The challenges will become very apparent, when many code generation techniques with different contexts such as basic block, loop, or trace are applied. Then, a schedule for a loop may correspond to multiple schedules for the basic blocks of the loops. Additional glue code is needed for software-pipelined or vectorized loops in order to combine them with the remaining code of a function. The current implementation of CoBRA avoids a lot of these challenges by a parallelization on basic block level.

Code Integration The current implementation of the SIMD/MIMD reconfiguration just selects the schedule with the shorter estimated runtime for each basic block. Reconfiguration instructions between basic blocks are not needed, because branches are always executed in MIMD mode (see Section 8.2.3). In the future, the selection heuristic should take a larger context of a function as well as further properties like code size and power consumption into account (see Section 3.3.2). We believe that the placement of reconfiguration instructions needs to be integrated into the selection task in order to achieve best results.

Furthermore, the effect of succeeding phases like placement of communication or synchronization code and register allocation may also be taken into account to get more precise estimations. Clearly, a code integration has a much better decision base if it is performed as late as possible after the mentioned phases. On the other hand, this requires a well-organized compiler structure and implementation, again. For maximum preciseness, feedback-driven compilation can be used to guide the code integration based on results obtained in a prior run.

In order to produce better results or to customize the CoBRA approach to a certain need, the compiler should expect the optimization goal as an input. For instance, the user may prefer a fast execution, but neglects code size or power consumption, when targeting a high-performance machine. With respect to resource-constraint devices, small code size and low energy dissipation are more important than execution time, obviously.

Parallelization Future extensions can also focus on the two studied reconfiguration dimensions themselves: Currently, the implementation of the dimension *parallelization* only distinguishes between SIMD and MIMD execution using all processors of the QuadroCore, as well as a single processor. In the future, we may also consider a subset of processors when parallelism is scarce or low energy consumption is favoured over a fast execution. Higher speedups can be achieved by exploiting more coarse-grained parallelism on loop or task level. For instance, software pipelining or classical vectorization techniques parallelize whole loops automatically. Instead, a special programming model is needed to identify threads for concurrent execution. The compiler may also take user information given as directives in the source code or command line options into account to guide the code generation process.

Register Access The current implementation of CAPRICoRn assumes arbitrary mappings between architectural and physical blocks. Probably, such model can only realized with a high routing overhead in hardware and therefore may decrease performance. In the future, the dimension *register access* should also consider restricted mappings to compare such paradigm with the current prototype. Furthermore, local and remote accesses to physical registers may have different latencies in a real hardware implementation. The read and write ports can restrict the number of parallel register accesses at run-time. Both issues demand a proper arrangement of operations after register allocation to ensure a correct execution. Different mappings for read and write access (see Section 10.2.2) are only interesting on a methodical level, but probably not feasible in practice.

Section 12.3 has demonstrated that the aggressive re-use of registers driven by a register reconfiguration can lead to cyclic dependences which cannot be handled by our re-scheduler. Additionally, the insertion of further instructions for reconfiguration or synchronization may be infeasible. Currently, CAPRICoRn avoids such situations by an extension of life spans, which increases the register pressure unintentionally. In the future, the re-scheduling phase should recognize cyclic dependences by identifying Strongly Connected Components (SCC) in the Data Dependence Graph (DDG). During scheduling, SCCs can be represented as a single node to avoid re-ordering of operations. A solution for the second challenge is still pending.

Parallelizing Compiler In the future, we aim for a holistic processor partitioning method which can handle both data and operations jointly using affinity graphs. Alternatively, other equivalent techniques from literature like [36] may be studied.

The re-scheduling phase of our compiler can insert synchronization code on-the-fly and tries to minize the number of barriers. However, the evaluation has shown that the waiting time at barriers can have a significant impact on the execution time. A future version should also estimate the overhead in waiting to derive a better arrangement of operations. Alternatively, the fuzzy barrier approach by Gupta [68] mentioned in Section 6.2.1 may be used to utilize the waiting time at barriers to execute code not related to the barrier.

Part VI.

Appendix

List of Figures

1.1	Reconfiguration of variants	4
1.2	Space of reconfiguration variants	6
2.1	Trade-off between programmable and fixed function hardware	I-7
2.2	Structure of FPGA	I-9
2.3	Three design flows for algorithm implementation on reconfigurable systems	I-14
2.4	Levels of coupling in a reconfigurable system	I-19
3.1	Reconfiguration of variants	I-33
3.2	Space of reconfiguration variants	I-34
3.3	Reconfiguration variants of CoBRA	I-34
3.4	Alternative usage of reconfigurable register banks for software pipelining	I-37
3.5	Example of jointly using SIMD mode and reconfigurable registers	I-37
3.6	Structure of system (compiler-driven reconfiguration)	I-39
3.7	Structure of system (compiler-supported reconfiguration)	I-42
3.8	Structure of compiler (in principle, single decision point)	I-43
3.9	Structure of compiler (in principle, multiple decision points)	I-44
3.10	Structure of compiler (for dimension *parallelization*)	I-45
3.11	Model of code integration	I-46
4.1	System architecture based on embedded multi-cores	II-6
4.2	Structure of QuadroCore	II-7
4.3	Processing of very large instructions words in a VLIW machine	II-8
4.4	Differences between VLIW machine and superscalar processor	II-9
4.5	Scheduling model and integration of pseudo unit instructions	II-10
4.6	Structure of parallelizing compiler backend	II-11
4.7	Duplication of virtual registers	II-12
4.8	Exchanging of compare register	II-13
5.1	Comparison of VLIW machine and S-Core based multi-core	II-16
5.2	Example of variable affinity graph	II-20
5.3	Computation of affinities between variables	II-20
5.4	Example of computing the optimal number of partitions	II-22
5.5	Example of partitioning of function parameters	II-24
5.6	Example of distribution of function parameters	II-24
5.7	Matching of variable and parameter partitioning	II-24
5.8	Overview of variable/parameter partitioning	II-25
5.9	Number of target processors for matching	II-26
5.10	Partitioning of variables and instructions	II-27

5.11 Preparation of load-time scheduling at compile-time II-28
5.12 Load-time scheduling by compiler annotations II-29
5.13 Partitioning of annotation graph II-29

6.1 Communication of register values II-33
6.2 Motivating example of placement of copy code II-34
6.3 Correctness of communication after definition II-35
6.4 Result of placement strategy *Directly After Definition* for Figure 6.2 . . . . . . II-35
6.5 Infeasible communication directly before use II-36
6.6 Result of placement strategy *Common Cheap Basic Blocks* for Figure 6.2 . . . . II-37
6.7 Result of placement strategy *Merge Definitions* for Figure 6.2 . . . . II-37
6.8 Ensuring remote dependences by VLIW machine and multi-core II-40
6.9 Example of barrier synchronization for the QuadroCore II-40
6.10 Barrier insertion by list scheduling II-42
6.11 Need for barriers between basic blocks II-43
6.12 Need for barriers between functions II-43
6.13 Global barrier to avoid overwriting of entries in communication buffer . . . II-44
6.14 Global barrier to respect memory dependences II-45

7.1 Vector machine III-7
7.2 Overview of classical vectorization III-7
7.3 Vector pointers for computation of inner product III-12

8.1 Structure of compiler backend with CHARISMA component III-16
8.2 Structure of CHARISMA III-17
8.3 SIMD/MIMD modes of QuadroCore III-18
8.4 Functionality of original MIMD mode III-19
8.5 Functionality of SIMD mode III-19
8.6 Memory access in SIMD mode III-19
8.7 Concept of non-adjacent memory access with fixed offsets III-21
8.8 Concept of non-adjacent memory access with arbitrary offsets III-21
8.9 Branch in SIMD code III-22
8.10 Vectorization based on adjacent memory references III-23
8.11 Loop unrolling transforms vector parallelism into SLP III-24
8.12 Overview of SLP algorithm III-24
8.13 Data-flow problem *Common subexpressions* III-25
8.14 Intermediate trees of address computations for indexed memory access . . . III-26
8.15 Annotation of intermediate nodes with information about memory access . III-27
8.16 Computation of adjacence information from annotations III-27
8.17 Basic scheme of loop unrolling III-28
8.18 Benefits of loop unrolling for vectorization technique III-29
8.19 Aligned and unaligned accesses III-29
8.20 Example of vectorization with SLP III-31
8.21 Identification of adjacent memory references and creation of pairs III-31
8.22 Traversal of def-use/use-def chains to find further pairs III-32
8.23 Combination of pairs to groups III-33
8.24 Resolving of inter-group cycles by splitting nodes III-33

8.25 Schedule for example from Figure 8.24 . . . . . . III-34
8.26 Conceptual vector register III-34
8.27 Virtual vector registers III-35
8.28 Idea of Placement Algorithm III-37
8.29 Definition of virtual vector and scalar registers III-38
8.30 Definition of overlapping virtual vector registers III-38
8.31 Overlapping registers and structured control-flow III-39
8.32 Spilling areas for vector registers III-39

9.1 Register allocation for expression trees IV-6
9.2 Register allocation for basic blocks with lifetime analysis IV-7
9.3 Register allocation by graph coloring IV-8
9.4 Improvements to register allocation by graph coloring IV-8
9.5 Register renaming IV-9
9.6 Register windowing IV-10
9.7 Register architecture by Ravindran et al. IV-11
9.8 Register architecture by Kiyohara et al. IV-11
9.9 Register architecture by Smelyanskiy et al. IV-12

10.1 Example of terminology of reconfigurable register architecture IV-16
10.2 Single processor architectures with conventional or reconfigurable register banks IV-17
10.3 Multi-Core architecture with reconfigurable register bank IV-18
10.4 Our reconfigurable register architecture with *one* of the feasible mappings . IV-18
10.5 Mapping problem in case of insufficient number of architectural blocks . . . IV-20
10.6 Mapping problem in case of only one architectural block IV-20
10.7 General reconfigurable register architecture with *one* of the feasible mappings IV-21
10.8 Mapping problem in case of non-restricted mappings IV-22
10.9 Mapping problem in case of physical blocks with common index IV-23
10.10 Modified semantics of read/write operands IV-23
10.11 Mapping problem in case of operands located in multiple physical blocks . IV-24

11.1 n-liveness property IV-28
11.2 n-liveness property with probabilities IV-29
11.3 Removing a node from the tree IV-30
11.4 Combination of two trees IV-31
11.5 Reduction of two common child nodes IV-31
11.6 Prepending a node IV-32
11.7 Propagation of access trees IV-32
11.8 Examples of complete access trees IV-35
11.9 Partial order of complete access trees IV-36
11.10 Example of an IN tree IV-37
11.11 Removing of GEN registers of complete access tree IV-38
11.12 Probabilities of IN and OUT trees during multiple DFA iterations IV-40

12.1 Structure of register allocation IV-44
12.2 Example of affinity graph IV-45
12.3 Situation considered by proof of Theorem 12.1 . . . . . . IV-46

12.4 Definition of access pair IV-47
12.5 Association of virtual registers with physical blocks in affinity graph IV-49
12.6 Appearance of new access pairs IV-49
12.7 Examples of access pairs IV-50
12.8 Relation between replacement and total order of accesses IV-52
12.9 Weighting of empty physical blocks to achieve an uniform distribution . . . IV-53
12.10 Integration of reconfiguration instructions and replacement of physical registers by architectural registers IV-54
12.11 Determination of mappings between basic blocks IV-56
12.12 Reconfiguration between functions IV-58
12.13 Multi-Core architecture IV-60
12.14 Infeasible placement of reconfiguration instructions IV-61
12.15 Extension of life span by one-half clock cycle IV-62
12.16 Construction of data-dependence graph IV-63
12.17 Extension of life span by latency of instruction IV-64

13.1 Speedup by parallelization (class 1, set 1) V-13
13.2 Speedup by parallelization (class 1, set 2) V-13
13.3 Speedup by parallelization (class 2, set 1) V-14
13.4 Speedup by parallelization (class 2, set 2) V-14
13.5 Speedup by parallelization (class 4, set 1) V-15
13.6 Speedup by parallelization (class 4, set 2) V-15
13.7 Speedup by optimization (class 1, set 1) V-17
13.8 Speedup by optimization (class 1, set 2) V-17
13.9 Speedup by optimization (class 2, set 1) V-18
13.10 Speedup by optimization (class 2, set 2) V-18
13.11 Speedup by optimization (class 4, set 1) V-19
13.12 Speedup by optimization (class 4, set 2) V-19
13.13 Speedup by VLIW/Barrier reconfiguration in % (class 1, set 1) V-21
13.14 Speedup by VLIW/Barrier reconfiguration in % (class 1, set 2) V-21
13.15 Speedup by VLIW/Barrier reconfiguration in % (class 2, set 1) V-22
13.16 Speedup by VLIW/Barrier reconfiguration in % (class 2, set 2) V-22
13.17 Speedup by VLIW/Barrier reconfiguration in % (class 4, set 1) V-23
13.18 Speedup by VLIW/Barrier reconfiguration in % (class 4, set 2) V-23
13.19 Increment of code size by VLIW/Barrier reconfiguration in % (class 1, set 1) V-24
13.20 Increment of code size by VLIW/Barrier reconfiguration in % (class 1, set 2) V-24
13.21 Increment of code size by VLIW/Barrier reconfiguration in % (class 2, set 1) V-25
13.22 Increment of code size by VLIW/Barrier reconfiguration in % (class 2, set 2) V-25
13.23 Increment of code size by VLIW/Barrier reconfiguration in % (class 4, set 1) V-26
13.24 Increment of code size by VLIW/Barrier reconfiguration in % (class 4, set 2) V-26
13.25 Reduction of wait cycles by VLIW/Barrier reconfiguration in % (class 1, set 1) V-27
13.26 Reduction of wait cycles by VLIW/Barrier reconfiguration in % (class 1, set 2) V-28
13.27 Reduction of wait cycles by VLIW/Barrier reconfiguration in % (class 2, set 1) V-28
13.28 Reduction of wait cycles by VLIW/Barrier reconfiguration in % (class 2, set 2) V-29
13.29 Reduction of wait cycles by VLIW/Barrier reconfiguration in % (class 4, set 1) V-29
13.30 Reduction of wait cycles by VLIW/Barrier reconfiguration in % (class 4, set 2) V-30
13.31 Speedup by automatic data partitioning (class 1) V-32

13.32 Speedup by automatic data partitioning (class 2) V-33
13.33 Speedup by automatic data partitioning (class 4) V-33
13.34 Relative communication costs with parallelization in % (class 1, set 1) V-34
13.35 Relative communication costs with parallelization in % (class 1, set 2) V-35
13.36 Relative communication costs with parallelization in % (class 2, set 1) V-35
13.37 Relative communication costs with parallelization in % (class 2, set 2) V-36
13.38 Relative communication costs with parallelization in % (class 4, set 1) V-36
13.39 Relative communication costs with parallelization in % (class 4, set 2) V-37
13.40 Artificial benchmarks to evaluate placement strategies V-38
13.41 Execution times with different strategies for placing communication code . V-38
13.42 Speedups with real SIMD/MIMD reconfiguration V-40
13.43 Deterioration of execution time compared to the pseudo SIMD/MIMD reconfiguration in % V-41
13.44 Speedups with pseudo SIMD/MIMD reconfiguration V-41
13.45 SIMD ratio with real SIMD/MIMD reconfiguration in % V-43
13.46 Deterioration of SIMD ratio compared to the pseudo SIMD/MIMD reconfiguration in % V-44
13.47 SIMD ratio with pseudo SIMD/MIMD reconfiguration in % V-44
13.48 Reduction of code size with real SIMD/MIMD reconfiguration in % V-47
13.49 Reduction of instruction memory accesses with real SIMD/MIMD reconfiguration in % V-48
13.50 Reduction of switching activity of instruction fetch with real SIMD/MIMD reconfiguration in % V-48
13.51 Reduction of switching activity of program counter with real SIMD/MIMD reconfiguration in % V-49
13.52 Relative reconfiguration costs with real SIMD/MIMD reconfiguration in % V-50
13.53 Relative communication costs with real SIMD/MIMD reconfiguration in % V-51
13.54 Speedups with register reconfiguration (without loop unrolling) in % V-54
13.55 Speedups with register reconfiguration (with loop unrolling) in % V-55
13.56 Reduction of code size with register reconfiguration (without loop unrolling) in % V-57
13.57 Reduction of code size with register reconfiguration (with loop unrolling) in % V-57
13.58 Reduction of stack accesses with register reconfiguration (with loop unrolling) in % V-59
13.59 Relative costs of register reconfiguration in % (without loop unrolling) . . . V-60
13.60 Relative costs of register reconfiguration in % (with loop unrolling) V-60
13.61 Speedup by SIMD/MIMD reconfiguration and/or register reconfiguration in % V-62
13.62 SIMD ratio with SIMD/MIMD reconfiguration (LCI) and register reconfiguration in % V-62
13.63 Reduction of code size with SIMD/MIMD reconfiguration and register reconfiguration in % V-63
13.64 Reduction of instruction memory accesses with SIMD/MIMD reconfiguration (LCI) and register reconfiguration in % V-64
13.65 Reduction of switching activity of instruction fetch with SIMD/MIMD reconfiguration (LCI) and register reconfiguration in % V-64

13.66 Reduction of switching activity of program counter with SIMD/MIMD reconfiguration (LCI) and register reconfiguration in % V-65
13.67 Relative costs of SIMD/MIMD reconfiguration with SIMD/MIMD reconfiguration (LCI) and register reconfiguration in % V-65
13.68 Relative costs of register reconfiguration with SIMD/MIMD reconfiguration (LCI) and register reconfiguration in % V-66

List of Tables

13.1 Standard Cell Synthesis Reports V-7
13.2 Description of benchmarks V-10
13.3 Optimization strategies V-11

List of Algorithms

12.1 Algorithm for the computation of the affinities IV-51

Glossary

ALAP

As **L**ate **A**s **P**ossible

ALU

Arithmetic **L**ogical **U**nit

ANSI

American **N**ational **S**tandards **I**nstitute

ASAP

As **S**oon **A**s **P**ossible

ASIC

Application **S**pecific **I**ntegrated **C**ircuit

Basic Block

Maximal sequence of instructions that can be entered only at the first of them and exited only from the last of them.

BSP

Bulk-**S**ynchronous **P**arallel

BUG

Bottom **U**p **G**reedy

BURS

Bottom-**u**p **R**ewrite **S**ystems

CAD

Computer-**A**ided **D**esign

CALS

Code **A**nnotations for **L**oad-time **S**cheduling

CAPRICoRn

Compiler **A**nticipated **P**rocessor **R**egister **I**nter-**Co**nnected **R**econfiguratio**n**

CFB

Configurable **F**unction **B**lock

CFG

Control-**F**low-**G**raph

CGRA

Coarse-**G**rained **R**econfigurable **A**rray

CHARISMA

Compiler **H**andles **A**rchitectural **R**econfiguration **I**ntegrating **S**IMD **M**IMD **A**utomatically

CISC

Complex **I**nstruction **S**et **C**omputer

CLB

Configurable **L**ogic **B**lock

CMOS

Complementary **M**etal **O**xide **S**emiconductor

CoBRA

Compiler-**D**riven **D**ynamic **R**econfiguration of **A**rchitectural Variants

Control-Flow Graph

Represents the control structure of a function. The nodes are the basic blocks and two unique *entry* and *exit* nodes. There exists a directed edge between two nodes a and b if the control may flow from the end of a to the beginning of b.

CPLD

Complex **P**rogrammable **L**ogic **D**evice

CPU

Central **P**rocessing **U**nit

CRC

Cyclic **R**edundancy **C**heck

CSE

Common **S**ubexpression **E**limination

DDG

Data **D**ependence **G**raph

DES

Data **E**ncryption **S**tandard

DFA

Data-**F**low **A**nalysis

DFG

Data-**F**low **G**raph

DIL

Dataflow **I**ntermediate **L**anguage

DSL

Digital **S**ubscriber **L**ine

DSLAM

DSL **A**ccess **M**ultiplexer

DSP

Digital **S**ignal **P**rocessor

EEPROM

Electrically **E**rasable **P**rogrammable **R**ead **O**nly **M**emory

EPIC

Explicitly **P**arallel **I**nstruction **C**omputing

FFT

Fast **F**ourier **T**ransformation

FIFO

First **I**n **F**irst **O**ut

FIR

Finite **I**mpulse **R**esponse

FORTRAN

FORmula **TRAN**slation

FPGA

Field **P**rogrammable **G**ate **A**rray

FU

Functional **U**nit

HDL

Hardware **D**escription **L**anguage

HLL

High-**L**evel **L**anguage

IIR

Infinite **I**mpulse **R**esponse

ILP

Instruction-**L**evel **P**arallelism

ISA

Instruction **S**et **A**rchitecture

ISE

Instruction **S**et **E**xtension

LLP

Loop-**L**evel **P**arallelism

LRU

Least **R**ecently **U**sed

LUA

latest **u**sed **a**gain

LUT

Look-**U**p **T**able

DAG

Directed **A**cyclic **G**raph

MIMD

Multiple **I**nstruction **M**ultiple **D**ata

MIPS

Microprocessor without **i**nterlocked **p**ipeline **s**tages

MME

Multi**m**edia **E**xtension

MMX

Multi-**M**edia e**X**tension

MVE

Modulo **V**ariable **E**xpansion

NAPA

National **A**daptive **P**rocessing **A**rchitecture

NISC

No-**I**nstruction-**S**et Computer

NML

Native **M**apping **L**anguage

PAL

Programmable **A**rray **L**ogic

PAM

Programmable **A**ctive **M**emories

PLA

Programmable **L**ogic **A**rray

PLD

Programmable **L**ogic **D**evice

PRISC

PRogrammable **I**nstruction **S**et **C**omputers

PRISM

Processor **R**econfiguration **T**hrough **I**nstruction-**S**et **M**etamorphosis

PROM

Programmable **R**ead **O**nly **M**emory

RA

Reconfigurable **A**rray

RaPiD

Reconfigurable **P**ipelined **D**atapath

RAW

Reconfigurable **A**rchitecture **W**orkstation

REMARC

Reconfigurable **M**ultimedia **A**rray **P**rocessor

RFU

Reconfigurable **F**unctional **U**nit

RHOP

Region-based **H**ierarchical **O**peration **P**artitioning

RISC
: **R**educed **I**nstruction **S**et **C**omputer

RPU
: **R**econfigurable **P**rocessing **U**nit

RRF
: **R**otating **R**egister **F**ile

SALT
: **S**cheduling by Compiler-generated **A**nnotations at **L**oad-**T**ime

SAP
: **S**ingle **A**ssigment **P**rogram

SCC
: **S**trongly **C**onnected **C**omponent

SIMD
: **S**ingle **I**nstruction **M**ultiple **D**ata

SIMdD
: **S**ingle **I**nstruction **M**ultiple disjoint **D**ata

SIMpD
: **S**ingle **I**nstruction **M**ultiple packed **D**ata

SLP
: **S**uperword **L**evel **P**arallelism

Spill code
: Code to store and reload allocated registers temporarily.

SPLD
: **S**imple **P**rogrammable **L**ogic **D**evice

SRAM
: **S**tatic RAM

SSE
: **S**treaming **S**IMD **E**xtensions

SUIF
: **S**tanford **U**niversity **I**ntermediate **F**ormat

SWAR
: **S**IMD **W**ithin **A** **R**egister

TLP

Thread-**L**evel **P**arallelism

ULSI

Ultra **L**arge **S**cale **I**ntegration

UPSLA

Unified **P**rocessor **S**pecification **La**nguage

VAG

Variable **A**ffinity **G**raph

VHDL

Very **H**igh Speed Integrated Circuit Hardware **D**escription **L**anguage

VIS

Visual **I**nstruction **S**et

VLIW

Very **L**ong **I**nstruction **W**ords

XPP

Xtreme **P**rocessing **P**latform

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] Vicky H. Allan, Reese B. Jones, Randall M. Lee, and Stephen J. Allan. Software Pipelining. *ACM Computing Surveys*, 27(3):367–432, 1995.

[3] John Randy Allen and Ken Kennedy. PFC: A Program to Convert Fortran to Parallel Form. In K. Hwang, editor, *Supercomputers: Design and Applications*, pages 186–203. IEEE Computer Society Press, Silver Spring, MD, 1984.

[4] John Randy Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of Control Dependence to Data Dependence. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 177–189, New York, NY, USA, 1983. ACM Press.

[5] Randy Allen and Ken Kennedy. Automatic Translation of FORTRAN Programs to Vector Form. *ACM Trans. Program. Lang. Syst.*, 9(4):491–542, 1987.

[6] Peter M. Athanas and Harvey F. Silverman. Processor Reconfiguration Through Instruction-Set Metamorphosis. *Computer*, 26(3):11–18, 1993.

[7] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Comput. Surv.*, 26(4):345–420, 1994.

[8] Domenico Barretta, William Fornaciari, Mariagiovanna Sami, and Danilo Pau. SIMD Extension to VLIW Multicluster Processors for Embedded Applications. In *ICCD '02: Proceedings of the 2002 IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD'02)*, page 523, Washington, DC, USA, 2002. IEEE Computer Society.

[9] V. Baumgarte, G. Ehlers, F. May, A. Nückel, M. Vorbach, and M. Weinhardt. PACT XPP - A Self-Reconfigurable Data Processing Architecture. *J. Supercomput.*, 26(2):167–184, 2003.

[10] Gary R. Beck, David W. L. Yen, and Thomas L. Anderson. The cydra 5 minisupercomputer: Architecture and Implementation. *J. Supercomput.*, 7(1-2):143–180, 1993.

[11] Carl J. Beckmann and Constantine D. Polychronopoulos. Fast Barrier Synchronization Hardware. In *Supercomputing '90: Proceedings of the 1990 conference on Supercomputing*, pages 180–189, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[12] Laszlo A. Belady. A study of replacement algorithms for a virtualstorage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[13] P. Bertin, D. Roncin, and J. Vuillemin. Introduction to Programmable Active Memories. pages 301–309, 1989.

[14] C. Bidan, V. Issarny, T. Saridakis, and A. Zarras. A Dynamic Reconfiguration Service for CORBA. In *CDS '98: Proceedings of the International Conference on Configurable Distributed Systems*, page 35, Washington, DC, USA, 1998. IEEE Computer Society.

[15] Aart Bik, Milind Girkar, Paul Grey, and Xinmin Tian. An Auto-Vectorizing Compiler for the Intel Architecture. 2000. Submitted to the ACM Transactions on Programming Languages and Systems.

[16] Aart Bik, Milind Girkar, Paul Grey, and Xinmin Tian. Efficient Exploitation of Parallelism on Pentium III and Pentium 4 Processor-Based Systems. *Intel Technology Journal*, (Q1):9, 2001.

[17] Aart J. C. Bik, Milind Girkar, Paul M. Grey, and Xinmin Tian. Automatic detection of saturation and clipping idioms. In *Languages and Compilers for Parallel Computing, 15th Workshop, LCPC 2002*, pages 61–74, 2002.

[18] Aart J. C. Bik, Milind Girkar, Paul M. Grey, and Xinmin Tian. Automatic Intra-Register Vectorization for the Intel Architecture. *Int. J. Parallel Program.*, 30(2):65–98, 2002.

[19] Toby Bloom. *Dynamic Module Replacement in a Distributed Programming System*. Ph.D., MIT, 1983. Also as MIT LCS Tech. Report 303.

[20] Maarten Boekhold, Ireneusz Karkowski, and Henk Corporaal. Transforming and Parallelizing ANSI C Programs Using Pattern Recognition. *Lecture Notes in Computer Science*, 1593:673–682, 1999.

[21] Olaf Bonorden, Nikolaus Brüls, Dinh Khoi Le, Uwe Kastens, Friedhelm Meyer auf der Heide, Jörg-Christian Niemann, Mario Porrmann, Ulrich Rückert, Adrian Slowik, and Michael Thies. A holistic methodology for network processor design. In *Proceedings of the Workshop on High-Speed Local Networks held in conjunction with the 28th Annual IEEE Conference on Local Computer Networks (LCN2003)*, pages 583–592, October 2003.

[22] Jeremy S. Bradbury, James R. Cordy, Juergen Dingel, and Michel Wermelinger. A Survey of Self-Management in Dynamic Software Architecture Specifications. In *WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, pages 28–33, New York, NY, USA, 2004. ACM Press.

[23] Jens Braunes, Steffen Köhler, and Rainer G. Spallek. RECAST: An Evaluation Framework for Coarse-Grain Reconfigurable Architectures. In Christian Müller-Schloer, Theo Ungerer, and Bernhard Bauer, editors, *Proceedings of the Organic and Pervasive Computing - ARCS 2004*, volume 2981, pages 156–166. Springer-Verlag Heidelberg, February 2004.

[24] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Trans. Program. Lang. Syst.*, 16(3):428–455, 1994.

[25] Stephen D. Brown, Robert J. Francis, Jonathan Rose, and Zvonko G. Vranesic. *Field-Programmable Gate Arrays*. Kluwer Academic Publishers, Norwell, MA, USA, 1992.

[26] Mihai Budiu and Seth Copen Goldstein. Fast Compilation for Pipelined Reconfigurable Fabrics. In *FPGA '99: Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, pages 195–205, New York, NY, USA, 1999. ACM Press.

[27] D. Callahan and P. Havlak. Scalar expansion in PFC: Modifications for Parallelization. Technical Report Supercomputer Software Newsletter 5, Dept. of Computer Science, Rice University, October 1986.

[28] Timothy J. Callahan, John R. Hauser, and John Wawrzynek. The Garp Architecture and C Compiler. *Computer*, 33(4):62–69, 2000.

[29] Andrea Capitanio, Nikil Dutt, and Alexandru Nicolau. Partitioned Register Files for VLIWs: A Preliminary Analysis of Tradeoffs. In *MICRO 25: Proceedings of the 25th annual international symposium on Microarchitecture*, pages 292–300, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[30] Joao M. P. Cardoso and Markus Weinhardt. XPP-VC: A C Compiler with Temporal Partitioning for the PACT-XPP Architecture. In *FPL '02: Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications*, pages 864–874, London, UK, 2002. Springer-Verlag.

[31] William S. Carter, Khue Duong, Ross Freeman, Hung-Cheng Hseih, Jason Y. Ja, John E. Mahoney, Luan T. Ngo, and Shelly L. Sze. A User Programmable Reconfigurable Logic Array. In *IEEE 1986 Custom Integrated Circuits Conference*, pages 233–235, 1986.

[32] Gregory J. Chaitin. Register allocation & spilling via graph coloring. In *Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 98–101. ACM Press, 1982.

[33] G. Cheong and M. Lam. An Optimizer for Multimedia Instruction Sets. In *In Proceedings of the Second SUIF Compiler Workshop*, Stanford University, USA, August 1997.

[34] Anton V. Chichkov and Carlos Beltrán Almeida. A hardware/software partitioning algorithm for custom computing machines. In *FPL '97: Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications*, pages 274–283, London, UK, 1997. Springer-Verlag.

[35] Michael Chu, Kevin Fan, and Scott Mahlke. Region-Based Hierarchical Operation Partitioning for Multicluster Processors. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 300–311, New York, NY, USA, 2003. ACM Press.

[36] Michael L. Chu and Scott A. Mahlke. Compiler-directed Data Partitioning for Multicluster Processors. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 208–220, Washington, DC, USA, 2006. IEEE Computer Society.

[37] Robert P. Colwell, Robert P. Nix, John J. O'Donnell, David B. Papworth, and Paul K. Rodman. A VLIW Architecture for a Trace Scheduling Compiler. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems*, pages 180–192. IEEE Computer Society Press, 1987.

[38] Katherine Compton and Scott Hauck. Reconfigurable Computing: A Survey of Systems and Software. *ACM Comput. Surv.*, 34(2):171–210, 2002.

[39] D. C. Cronquist, P. Franklin, S. G. Berg, and C. Ebeling. Specifying and Compiling Applications for RaPiD. In *FCCM '98: Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, page 116, Washington, DC, USA, 1998. IEEE Computer Society.

[40] André DeHon. DPGA-Coupled Microprocessors: Commodity ICs for the Early 21st Century. In Duncan A. Buell and Kenneth L. Pocek, editors, *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 31–39, Los Alamitos, CA, 1994. IEEE Computer Society Press.

[41] Andre DeHon. Reconfigurable Architectures for General-Purpose Computing. Technical report, Cambridge, MA, USA, 1996.

[42] Derek J. DeVries. A Vectorizing SUIF Compiler: Implementation and Performance. Master's thesis, University of Toronto, June 1997.

[43] Keith Diefendorff. Pentium III = Pentium II + SSE. *Microprocessor Report*, 13(3):1,6–11, April 1999.

[44] Keith Diefendorff, Pradeep K. Dubey, Ron Hochsprung, and Hunter Scales. Altivec Extension to Powerpc Accelerates Media Processing. *IEEE Micro*, 20(2):85–95, 2000.

[45] H. Dietz, T. Schwederski, M. O'Keefe, and A. Zaafrani. Static Synchronization Beyond VLIW. In *Supercomputing '89: Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 416–425, New York, NY, USA, 1989. ACM Press.

[46] Ralf Dreesen. Registerzuteilung für Prozessor-Cluster mit dynamisch rekonfigurierbaren Registerbänken. Diploma thesis, University of Paderborn, 2006.

[47] Ralf Dreesen, Michael Hußmann, Michael Thies, and Uwe Kastens. Register Allocation for Processors with Dynamically Reconfigurable Register Banks. In *Proceedings of the 5rd Workshop on Optimizations for DSP and Embedded Systems (ODES) held in conjunction with the 5rd IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2007)*, March 2007.

[48] Carl Ebeling, Darren C. Cronquist, and Paul Franklin. RaPiD - Reconfigurable Pipelined Datapath. In *Proceedings of the 6th International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*, pages 126–135. Springer-Verlag, 1996.

[49] Alexandre E. Eichenberger, Kathryn O'Brien, Kevin O'Brien, Peng Wu, Tong Chen, Peter H. Oden, Daniel A. Prener, Janice C. Shepherd, Byoungro So, Zehra Sura, Amy Wang, Tao Zhang, Peng Zhao, and Michael Gschwind. Optimizing Compiler for the

CELL Processor. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, pages 161–172, Washington, DC, USA, 2005. IEEE Computer Society.

[50] John R. Ellis. *Bulldog: A Compiler for VLIW Architectures*. MIT Press, 1986.

[51] G. Estrin and C. R. Viswanathan. Organization of a "Fixed-Plus-Variable" Structure Computer for Computation of Eigenvalues and Eigenvectors of Real Symmetric Matrices. *J. ACM*, 9(1):41–60, 1962.

[52] Gerald Estrin, B. Bussell, R. Turn, and J. Bibb. Parallel Processing in a Restructurable Computer System. *IEEE Transactions on Electronic Computers*, EC-12(5):747–755, December 1963.

[53] Gerald Estrin and R. Turn. Automatic Assignment of Computations in a Variable Structure Computer System. *IEEE Transactions on Electronic Computers*, EC-12(5):755–773, December 1963.

[54] Dirk Fischer, Jürgen Teich, Michael Thies, and Ralph Weper. Efficient Architecture/Compiler Co-Exploration for ASIPs. In *ACM SIG Proceedings International Conference on Compilers*, Architectures and Synthesis for Embedded Systems (CASES 2002), Grenoble, France, 2002.

[55] Joseph A. Fisher. *The Optimization of Horizontal Microcode Within and Beyond Basic Blocks: An Application of Processor Scheduling with Resources*. PhD thesis, New York University, October 1979. Available from Courant Mathematics and Computing Laboratory as DOE report COO-3077-161.

[56] Joseph A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Trans. Comps., C-30 7/81*, pages 478–490, 1981.

[57] Randall J. Fisher and Henry G. Dietz. Compiling for SIMD Within a Register. In *LCPC '98: Proceedings of the 11th International Workshop on Languages and Compilers for Parallel Computing*, pages 290–304, London, UK, 1999. Springer-Verlag.

[58] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a Simple, Efficient Code-Generator Generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, September 1992.

[59] Ophir Frieder and Mark E. Segal. On dynamically updating a computer program: from concept to prototype. *J. Syst. Softw.*, 14(2):111–128, 1991.

[60] B. Furht. RISC Architectures with Multiple Overlapping Windows. In *Proc. Midcon 85*.

[61] David Galloway. The Transmogrifier C Hardware Description Language and Compiler for FPGAs. In *FCCM '95: Proceedings of the IEEE Symposium on FPGA's for Custom Computing Machines*, page 136, Washington, DC, USA, 1995. IEEE Computer Society.

[62] Maya Gokhale and Paul S. Graham. *Reconfigurable Computing: Accelerating Computation with Field-Programmable Gate Arrays*. Springer, 1992.

[63] Maya Gokhale, William Holmes, Andrew Kopser, Dick Kunze, Daniel P. Lopresti, Sara Lucas, Ronald Minnich, and Peter Olsen. SPLASH: A Reconfigurable Linear Logic Array. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages 526–532, August 1990.

[64] Maya Gokhale, William Holmes, Andrew Kopser, Sara Lucas, Ronald Minnich, Douglas Sweely, and Daniel P. Lopresti. Building and Using a Highly Parallel Programmable Logic Array. *IEEE Computer*, 24(1):81–89, 1991.

[65] Maya B. Gokhale and Janice M. Stone. NAPA C: Compiling for a Hybrid RISC/FPGA Architecture. In *FCCM '98: Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, page 126, Washington, DC, USA, 1998. IEEE Computer Society.

[66] Maya B. Gokhale, Janice M. Stone, Jeff Arnold, and Mirek Kalinowski. Stream-Oriented FPGA Computing in the Streams-C High Level Language. In *FCCM '00: Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, page 49, Washington, DC, USA, 2000. IEEE Computer Society.

[67] Seth C. Goldstein, Herman Schmit, Matthew Moe, Mihai Budiu, Srihari Cadambi, R. Reed Taylor, and Ronald Laufer. PipeRench: A Coprocessor for Streaming Multimedia Acceleration. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 28–39, May 1999.

[68] Rajiv Gupta. The Fuzzy Barrier: A Mechanism for High Speed Synchronization of Processors. In *ASPLOS-III: Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 54–63, New York, NY, USA, 1989. ACM Press.

[69] Rajiv Gupta, Michael Epstein, and Michael Whelan. The Design of a RISC based Multiprocessor Chip. In *Supercomputing '90: Proceedings of the 1990 conference on Supercomputing*, pages 920–929, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[70] Linley Gwennap. Altivec Vectorizes Powerpc. *Microprocessor Report*, 12(6), May 1998.

[71] Hwansoo Han, Chau-Wen Tseng, and Pete Keleher. Eliminating Barrier Synchronization for Compiler-Parallelized Codes on Software DSMs. *Int. J. Parallel Program.*, 26(5):591–612, 1998.

[72] Reiner Hartenstein. The Microprocessor is no more General Purpose. In *Proceedings of the International Conference on Innovative Systems in Silicon, ISIS'97*, October 1997.

[73] Reiner Hartenstein. A Decade of Reconfigurable Computing: a Visionary Retrospective. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 642–649. IEEE Press, 2001.

[74] Reiner Hartenstein. Coarse Grain Reconfigurable Architectures (embedded tutorial). In *Proceedings of the 2001 conference on Asia South Pacific design automation*, pages 564–570. ACM Press, 2001.

[75] Scott Hauck. Configuration Prefetch for Single Context Reconfigurable Coprocessors. In *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, pages 65–74. ACM Press, 1998.

[76] Scott Hauck and William D. Wilson. Runlength Compression Techniques for FPGA Configurations. In *Proceedings of the Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, page 286. IEEE Computer Society, 1999.

[77] John R. Hauser and John Wawrzynek. Garp: A MIPS Processor with a Reconfigurable Coprocessor. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 12–21, Los Alamitos, CA, 1997. IEEE Computer Society Press.

[78] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[79] Christine Ruth Hofmeister. *Dynamic Reconfiguration of Distributed Applications*. PhD thesis, College Park, MD, USA, 1993.

[80] Jan Hoogerbrugge and Lex Augusteijn. Instruction Scheduling for TriMedia. *Journal of Instruction-Level Parallelism*, 1, 1999.

[81] P. Y T Hsu and E. S. Davidson. Highly Concurrent Scalar Processing. In *ISCA '86: Proceedings of the 13th annual international symposium on Computer architecture*, pages 386–395, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[82] T. C. Hu. Parallel sequencing and assembly line problems. *Operations Research 9*, pages 841–848, 1961.

[83] Miquel Huguet and Tomás Lang. A C-Oriented Register Set Design. In *Proc. 29th Symp. on Mini and Microcomputers, Sant Feliu de Guixols, Catalonia*, pages 182–189, June 1985.

[84] Miquel Huguet and Tomás Lang. A Reduced Register File for RISC Architectures. *SIGARCH Comput. Archit. News*, 13(4):22–31, 1985.

[85] Michael Hußmann, Michael Thies, and Uwe Kastens. Parallelizing Compilation through Load-Time Scheduling for a Superscalar Processor Family. In *Proceedings of the 3rd Workshop on Optimizations for DSP and Embedded Systems (ODES) held in conjunction with the 3rd IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2005)*, March 2005.

[86] Michael Hußmann, Michael Thies, and Uwe Kastens. SALT: Efficient Load-Time Scheduling for Superscalar Processor Families Using Compiler Annotations. Technical Report tr-ri-05-263, University of Paderborn, October 2005.

[87] Michael Hußmann, Michael Thies, Uwe Kastens, Madhura Purnaprajna, Mario Porrmann, and Ulrich Rückert. Compiler-Driven Reconfiguration of Multiprocessors. In *Proceedings of the Workshop on Application Specific Processors (WASP) 2007 held in conjunction with the Embedded Systems Week, 2007 (CODES+ISSS, EMSOFT, and CASES)*, October 2007.

[88] Chameleon Systems Inc. *CS2000 Advance Product Specification*. San Jose, CA, 2000.

[89] Synopsys Inc. *CoCentric System C Compiler*. Mountain View, CA, 2000.

[90] Christian Iseli and Eduardo Sanchez. Beyond Superscalar Using FPGAs. In *Proceedings of the International Conference on Computer Design*, October 1993.

[91] Weihua Jiang, Chao Mei, Bo Huang, Jianhui Li, Jiahua Zhu, Binyu Zang, and Chuanqi Zhuh. Boosting the Performance of Multimedia Applications Using SIMD Instructions. volume 3443 of *Lecture Notes in Computer Science*, pages 59–75. Springer, 2005.

[92] Krishnan Kailas, Ashok Agrawala, and Kemal Ebcioglu. CARS: A New Code Generation Framework for Clustered ILP Processors. In *HPCA '01: Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, page 133, Washington, DC, USA, 2001. IEEE Computer Society.

[93] Heiko Kalte, Mario Porrmann, and Ulrich Rückert. A Prototyping Platform for Dynamically Reconfigurable System on Chip Designs. In *Proceedings of the IEEE Workshop Heterogeneous reconfigurable Systems on Chip (SoC)*, Hamburg, Germany, 2002.

[94] George Karypis and Vipin Kumar. Multilevel Algorithms for Multi-Constraint Graph Partitioning. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–13, Washington, DC, USA, 1998. IEEE Computer Society.

[95] Uwe Kastens. *Übersetzerbau*. Handbuch der Informatik. Oldenbourg Verlag, München, 1990.

[96] Manolis G. H. Katevenis. *Reduced Instruction Set Computer Architectures for VLSI*. PhD thesis, Univ. of California, Berkeley, October 1983.

[97] Tokuzo Kiyohara, Scott Mahlke, William Chen, Roger Bringmann, Richard Hank, Sadun Anik, and Wen-Mei Hwu. Register Connection: A New Approach to Adding Registers into Instruction Set Architectures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 247–256, New York, NY, USA, 1993. ACM Press.

[98] Steffen Köhler, Jens Braunes, Thomas Preußer, Martin Zabel, and Rainer G. Spallek. Increasing ILP of RISC Microprocessors Through Control-Flow Based Reconfiguration. In Jürgen Becker, Marco Platzner, and Serge Vernalde, editors, *Proceedings of the Field Programmable Logic and Application: 14th International Conference, FPL 2004*, volume 3203, pages 781–790. Springer-Verlag Heidelberg, September 2004.

[99] Steffen Köhler, Jens Braunes, Sergej Sawitzki, and Rainer G. Spallek. Improving Code Efficiency for Reconfigurable VLIW Processors. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 182. IEEE Computer Society, April 2002.

[100] L. Kohn, G. Maturana, M. Tremblay, A. Prabhu, and G. Zyner. The visual instruction set (VIS) in UltraSPARC. In *COMPCON '95: Proceedings of the 40th IEEE Computer Society International Conference*, page 462, Washington, DC, USA, 1995. IEEE Computer Society.

[101] Andreas Krall and Sylvain Lelait. Compilation Techniques for Multimedia Processors. *Int. J. Parallel Program.*, 28(4):347–361, 2000.

[102] Rainer Kress, Reiner W. Hartenstein, and Ulrich Nageldinger. An Operating System for Custom Computing Machines based on the Xputer Paradigm. In *FPL '97: Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications*, pages 304–313, London, UK, 1997. Springer-Verlag.

[103] Monica S. Lam. *A Systolic Array Optimizing Compiler*. Kluwer Academic Publishers, 1988.

[104] Monica S. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 318–328. ACM Press, 1988.

[105] Dominik Langen, Jörg-Christian Niemann, Mario Porrmann, Heiko Kalte, and Ulrich Rückert. Implementation of a RISC Processor Core for SoC Designs - FPGA Prototype vs. ASIC Implementation. In *Proceedings of the IEEE-Workshop: Heterogeneous reconfigurable Systems on Chip (SoC)*, Hamburg, Germany, 2002.

[106] Samuel Larsen. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. Master's thesis, Massachusetts Institute of Technology, May 2000.

[107] Samuel Larsen and Saman Amarasinghe. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 145–156, New York, NY, USA, 2000. ACM Press.

[108] Samuel Larsen, Radu Rugina, and Saman Amarasinghe. Alignment Analysis. Technical Report LCS-TM-605, Massachusetts Institute of Technology, June 2000.

[109] Samuel Larsen, Emmett Witchel, and Saman Amarasinghe. Techniques for Increasing and Detecting Memory Alignment. Technical Report LCS-TM-621, MIT/LCS, November 2001.

[110] Ronald Laufer, R. Reed Taylor, and Herman Schmit. PCI-PipeRench and the SwordAPI: A System for Stream-based Reconfigurable Computing. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 200–208, Los Alamitos, CA, 1999. IEEE Computer Society Press.

[111] Ruby B. Lee. Accelerating Multimedia with Enhanced Microprocessors. *IEEE Micro*, 15(2):22–32, 1995.

[112] Walter Lee, Rajeev Barua, Matthew Frank, Devabhaktuni Srikrishna, Jonathan Babb, Vivek Sarkar, and Saman Amarasinghe. Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 46–57. ACM Press, December 1997.

[113] Yanbing Li, Tim Callahan, Ervan Darnell, Randolph Harr, Uday Kurkure, and Jon Stockwood. Hardware-Software Co-Design of Embedded Reconfigurable Architectures. In *Proceedings of the 37th Conference on Design Automation*, pages 507–512. ACM Press, 2000.

[114] Zhiyuan Li and Scott Hauck. Don't Care Discovery for FPGA Configuration Compression. In *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, pages 91–98. ACM Press, 1999.

[115] Glenn Luecke, Waqar Haque, James Hoekstra, Howard Jespersen, and James Coyle. Evaluation of Fortran Vector Compilers and Preprocessors. *Softw. Pract. Exper.*, 21(9):891–905, 1991.

[116] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. Effective Compiler Support for Predicated Execution Using the Hyperblock. In *25th Annual International Symposium on Microarchitecture*, 1992.

[117] Radu Marculescu, Diana Marculescu, and Massoud Pedram. Probabilistic Modeling of Dependencies During Switching Activity Analysis. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 17(2):73–83, 1998.

[118] Millind Mittal, Alex Peleg, and Uri Weiser. MMX Technology Architecture Overview. *Intel Technology Journal*, (Q3):12, 1997.

[119] Takashi Miyamori and Kunle Olukotun. REMARC: Reconfigurable Multimedia Array Coprocessor. In *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, page 261. ACM Press, 1998.

[120] J. H. Moreno, V. Zyuban, U. Shvadron, F. D. Neeser, J. H. Derby, M. S. Ware, K. Kailas, A. Zaks, A. Geva, S. Ben-David, S. W. Asaad, T. W. Fox, D. Littrell, M. Biberstein, D. Naishlos, and H. Hunter. An Innovative Low-Power High-Performance Programmable Signal Processor for Digital Communications. *IBM J. Res. Dev.*, 47(2-3):299–326, 2003.

[121] Robert Morgan. *Building an Optimizing Compiler*. Butterworth-Heinemann, 1998.

[122] Motorola. MMC2001 Reference Manual, 1998.

[123] Steven S. Muchnik. *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, 1997.

[124] Dorit Naishlos, Marina Biberstein, Shay Ben-David, and Ayal Zaks. Vectorizing for a SIMdD DSP Architecture. In *CASES '03: Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 2–11, New York, NY, USA, 2003. ACM Press.

[125] Michael O'Boyle and Elena Stöhr. Compile time barrier synchronization minimization. *IEEE Trans. Parallel Distrib. Syst.*, 13(6):529–543, 2002.

[126] Emre Özer, Sanjeev Banerjia, and Thomas M. Conte. Unified Assign and Schedule: A New Approach to Scheduling for Clustered Register File Microarchitectures. In *MICRO 31: Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 308–315, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[127] David A. Padua and Michael J. Wolfe. Advanced Compiler Optimizations for Supercomputers. *Commun. ACM*, 29(12):1184–1201, 1986.

[128] Ian Page and Wayne Luk. Compiling Occam into FPGAs. In *Proc. of the First Intl. Symp. on Field Programmable Logic (FPL'91)*, 1991.

[129] David A. Patterson. Reduced Instruction Set Computers. *Commun. ACM*, 28(1):8–21, 1985.

[130] David A. Patterson and Carlo H. Sequin. A VLSI RISC. *Computer*, 15(9):8–21, September 1982.

[131] Alex Peleg and Uri Weiser. MMX Technology Extension to the Intel Architecture. *IEEE Micro*, 16(4):42–50, 1996.

[132] Dac Pham, Hans-Werner Anderson, Erwin Behnen, Mark Bolliger, Sanjay Gupta, H. Peter Hofstee, Paul Harvey, Charles R. Johns, James A. Kahle, Atsushi Kameyama, John M. Keaty, Bob Le, Sang Lee, Tuyen V. Nguyen, John G. Petrovick, Mydung Pham, Juergen Pille, Stephen D. Posluszny, Mack Riley, Joseph Verock, James D. Warnock, Steve Weitzel, and Dieter F. Wendel. Key features of the design methodology enabling a multi-core soc implementation of a first-generation CELL processor. In Fumiyasu Hirose, editor, *Proceedings of the 2006 Conference on Asia South Pacific Design Automation: ASP-DAC 2006, Yokohama, Japan, January 24-27, 2006*, pages 871–878. IEEE, 2006.

[133] Georg Piepenbrock. *Methoden des Software-Pipelining für Prozessoren mit Instruktionsparallelität*. PhD thesis, University of Paderborn, May 1995.

[134] Laura Pozzi. *Methodologies for the Design of Application-Specific Reconfigurable VLIW Processors*. PhD thesis, January 2000.

[135] Quickturn, A Cadence Company. Mercury™ design verification system technology backgrounder, 1999. Available online at http://www.quickturn.com/products/mercury_backgrounder.html.

[136] Quickturn, A Cadence Company. System realizer™, 1999. Available online at http://www.quickturn.com/products/systemrealizer.htm.

[137] B. Ramakrishna Rau, David W. L. Yen, Wei Yen, and Ross A. Towie. The Cydra 5 Departmental Supercomputer: Design Philosophies, Decisions, and Trade-Offs. *Computer*, 22(1):12–26, 28–30, 32–35, 1989.

[138] Tornlinson G. Rauscher and Ashok K. Agrawala. Dynamic Problem-Oriented Redefinition of Computer Architecture via Microprogramming. *IEEE Transactions on Computers*, 27(11):1006–1014, 1978.

[139] Rajiv A. Ravindran, Robert M. Senger, Eric D. Marsman, Ganesh S. Dasika, Matthew R. Guthaus, Scott A. Mahlke, and Richard B. Brown. Increasing the Number of Effective Registers in a Low-Power Processor Using a Windowed Register File. In *CASES '03: Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 125–136, New York, NY, USA, 2003. ACM Press.

[140] Rahul Razdan and Michael D. Smith. A High-Performance Microarchitecture with Hardware-Programmable Functional Units. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 172–180, 1994.

[141] Gang Ren, Peng Wu, and David Padua. A Preliminary Study on the Vectorization of Multimedia Applications for Multimedia Extension. In *In 16th International Workshop of Languages and Compilers for Parallel Computing*, October 2003.

[142] Mehrdad Reshadi and Daniel Gajski. A Cycle-Accurate Compilation Algorithm for Custom Pipelined Datapaths. In *CODES+ISSS '05: Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 21–26, 2005.

[143] Radu Rugina and Martin Rinard. Pointer Analysis for Multithreaded Programs. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 77–90, New York, NY, USA, 1999. ACM Press.

[144] Charlé R. Rupp, Mark Landguth, Tim Garverick, Edson Gomersall, Harry Holt, Jeffrey M. Arnold, and Maya Gokhale. The NAPA Adaptive Processing Architecture. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, page 28. IEEE Computer Society, 1998.

[145] Herman Schmit. Incremental Reconfiguration for Pipelined Applications. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 47–55, Los Alamitos, CA, 1997. IEEE Computer Society Press.

[146] Robert M. Senger, Eric D. Marsman, and Matthew R. Guthaus. Partitioning Variables across Register Windows to Reduce Spill Code in a Low-Power Processor. *IEEE Trans. Comput.*, 54(8):998–1012, 2005. Student Member-Rajiv A. Ravindran and Student Member-Ganesh S. Dasika and Member-Scott A. Mahlke and Senior Member-Richard B. Brown.

[147] Ravi Sethi and J. D. Ullman. The Generation of Optimal Code for Arithmetic Expressions. *J. ACM*, 17(4):715–728, 1970.

[148] M. Shand, P. Bertin, and J. Vuillemin. Hardware speedups in long integer multiplication. *SIGARCH Comput. Archit. News*, 19(1):106–113, 1991.

[149] M. Shand and J. E. Vuillemin. Fast implementations of RSA cryptography. In E. E. Swartzlander, M. J. Irwin, and J. Jullien, editors, *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, pages 252–259, Windsor, Canada, 1993. IEEE Computer Society Press, Los Alamitos, CA.

[150] Hartej Singh, Ming-Hau Lee, Guangming Lu, Nader Bagherzadeh, Fadi J. Kurdahi, and Eliseu M. Chaves Filho. MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications. *IEEE Trans. Comput.*, 49(5):465–481, 2000.

[151] Mikhail Smelyanskiy, Gary S. Tyson, and Edward S. Davidson. Register Queues: A New Hardware/Software Approach to Efficient Software Pipelining. In *PACT '00: Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*, page 3, Washington, DC, USA, 2000. IEEE Computer Society.

[152] Gerard J. M. Smit, Paul M. Heysters, and Bert Molenkamp. The Chameleon Project in Retrospective. In *Proceedings PROGRESS 2004 Embedded Systems Symposium, Nieuwegein, the Netherlands*, pages 181–184, October 2004.

[153] Michael John Sebastian Smith. *Application-Specific Integrated Circuits*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[154] N. Sreraman and R. Govindarajan. A Vectorizing Compiler for Multimedia Extensions. *Int. J. Parallel Program.*, 28(4):363–400, 2000.

[155] Esther Stümpel, Michael Thies, and Uwe Kastens. VLIW Compilation Techniques for Superscalar Architectures. In K. Koskimies, editor, *Proceedings 7th International Conference on Compiler Construction CC '98*, number 1383 in Lecture Notes in Computer Science. Springer Verlag, März 1998.

[156] Xinan Tang, Manning Aalsma, and Raymond Jou. A Compiler Directed Approach to Hiding Configuration Latency in Chameleon Processors. In *FPL '00: Proceedings of the The Roadmap to Reconfigurable Computing, 10th International Workshop on Field-Programmable Logic and Applications*, pages 29–38, London, UK, 2000. Springer-Verlag.

[157] Robert E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2), 1972.

[158] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs. *IEEE Micro*, 22(2):25–35, 2002.

[159] Andrei Terechko, Erwan Le Thénaff, and Henk Corporaal. Cluster Assignment of Global Values for Clustered VLIW Processors. In *CASES '03: Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 32–40, New York, NY, USA, 2003. ACM Press.

[160] Shreekant (Ticky) Thakkar and Tom Huff. The Internet Streaming SIMD Extensions. *Intel Technology Journal*, (Q2):8, 1999.

[161] Robert M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. pages 13–21, 1995.

[162] Marc Tremblay, J. Michael O'Connor, Venkatesh Narayanan, and Liang He. VIS Speeds New Media Processing. *IEEE Micro*, 16(4):10–20, 1996.

[163] Stephen M. Trimberger. *Field-Programmable Gate Array Technology*. Kluwer Academic Publishers, Norwell, MA, USA, 1994.

[164] Gary S. Tyson, Mikhail Smelyanskiy, and Edward S. Davidson. Evaluating the Use of Register Queues in Software Pipelined Loops. *IEEE Trans. Comput.*, 50(8):769–783, 2001.

[165] Ramachandran Vaidyanathan and Jerry L. Trahan. *Dynamic Reconfiguration: Architectures and Algorithms*. Plenum Publishing Co., 2004.

[166] Leslie G. Valiant. A Bridging Model for Parallel Computation. *Commun. ACM*, 33(8):103–111, 1990.

[167] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard. Programmable Active Memories: Reconfigurable Systems Come of Age. *IEEE Transactions on VLSI Systems*, 4(1):56–69, 1996.

[168] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring It All to Software: Raw Machines. *Computer*, 30(9):86–93, 1997.

[169] Qiang Wang and D. M. Lewis. Automated Field-Programmable Compute Accelerator Design Using Partial Evaluation. In *FCCM '97: Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines*, page 145, Washington, DC, USA, 1997. IEEE Computer Society.

[170] M. Wazlowski, L. Agarwal, T. Lee, A. Smith, E. Lam, P. Athanas, H. Silverman, and S. Ghosh. PRISM-II compiler and architecture. In Duncan A. Buell and Kenneth L. Pocek, editors, *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 9–16, Los Alamitos, CA, 1993. IEEE Computer Society Press.

[171] Markus Weinhardt and Wayne Luk. Pipeline Vectorization, book = IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, month = feb, year = 2001, pages = 234–248.

[172] Markus Weinhardt and Wayne Luk. Pipeline Vectorization for Reconfigurable Systems. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 52–62, Los Alamitos, CA, 1999. IEEE Computer Society Press.

[173] Shlomo Weiss and James E. Smith. *Power and the PowerPC*. Morgan Kaufmann Publishers, Inc., 1994.

[174] Maurice V. Wilkes. The best way to design an automatic calculating machine. pages 182–184, 1989.

[175] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer-Ann M. Anderson, Steven W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. *SIGPLAN Notices*, 29(12):31–37, 1994.

[176] Ralph Wittig and Paul Chow. OneChip: An FPGA Processor with Reconfigurable Logic. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 126–135, Los Alamitos, CA, 1996. IEEE Computer Society Press.

[177] Michael Joseph Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[178] K. Wong and Nigel Topham. OneDSP: A Unifying DSP Architecture For Systems-On-A-Chip. In *Proc. of IEEE Conf. on Acoustics, Speech and Signal Processing (ICASSP'02)*, pages 3792–3795, May 2002.

[179] Peng Wu, Alexandre E. Eichenberger, Amy Wang, and Peng Zhao. An Integrated Simdization Framework Using Virtual Vectors. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 169–178, New York, NY, USA, 2005. ACM Press.

[180] Zhi Alex Ye, Andreas Moshovos, Scott Hauck, Nagaraj Shenoy, and Prithviraj Banerjee. CHIMAERA: Integrating a Reconfigurable Unit into a High-Performance, Dynamically-Scheduled Superscalar Processor, 1999.

[181] Zhi Alex Ye, Nagaraj Shenoy, and Prithviraj Banerjee. A C Compiler for a Processor with a Reconfigurable Functional Unit. In *Proceedings of the 2000 ACM/SIGDA Eighth International Symposium on Field Programmable Gate Arrays*, pages 95–100. ACM Press, 2000.

[182] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press, New York, NY, USA, 1991.