RETARGETABLE COMPILER TECHNOLOGY FOR EMBEDDED SYSTEMS

Retargetable Compiler Technology for Embedded Systems

Tools and Applications

by

Rainer Leupers and Peter Marwedel
University of Dortmund

SPRINGER-SCIENCE+BUSINESS MEDIA, B.V.

A C.I.P. Catalogue record for this book is available from the Library of Congress.

ISBN 978-1-4419-4928-8 ISBN 978-1-4757-6420-8 (eBook)
DOI 10.1007/978-1-4757-6420-8

Printed on acid-free paper

To Carl and
Veronika, Malte,
Gesine, and Ronja

Contents

Preface

Embedded systems are information processing systems embedded in larger products. They have their own characteristics which make them different from desktop systems. For example, they have to be implemented efficiently, and the same holds for the increasing amount of embedded software. Designing an efficient, software- and processor-based system requires that optimized processors are used. Such an optimization requires a careful analysis of the design space, including a study of cost/performance tradeoffs. In order to avoid assembly language programming for such studies, compilers are needed. For analyzing the effect of design options on the performance, these compilers should be capable of generating code for all potential hardware configurations.

This is possible with *retargetable compilers.* Such compilers can be adapted to new hardware easily. Recently, many new approaches for designing such compilers have been developed. This book presents an overview and classification of these techniques. For each of the compilers, we mention key features, limitations, as well as software availability. The list of retargetable compilers covered has never been collected before. We also introduce key terms and techniques for compiler construction, explain where in the design flow retargetable compilers fit and present a history of retargetability. For people starting work on compilers for embedded systems, this book should save a significant time for finding references.

The book is self-contained and requires only fundamental knowledge in software design. It is intended to be a key reference for researchers and designers working on embedded software, compilation, and processor optimization. It can also be used by people who need to get an overview quickly, such as consultants and advisors. Please enjoy reading this book!

RAINER LEUPERS, PETER MARWEDEL

Chapter 1

INTRODUCTION

Writing this book was motivated by the growing usage of a large variety of processors in embedded systems. This trend has to be reflected in the corresponding development of still missing tools for the design of embedded systems. In order to understand the requirements for such tools, we will first of all have to look at the characteristics of the application areas that we are considering.

1. Embedded systems and their characteristics

Embedded systems can be defined as information processing systems which are integrated into larger systems. In such systems, information processing is usually not directly visible to the user, and customers will typically not buy a certain product –like a car or a mobile phone– because he or she is interested in the features of the information processing equipment. For example, customers will not buy a certain model of a mobile phone because of the brand name and the clock speed of the processor contained in the phone. They buy it because of the functionality that it provides, regardless of the electronic system supplying part of that function. The same applies to information processing in cars, trains, airplanes and smart homes, to mention some other examples of embedded systems.

In many cases, the customer does not even recognize that information processing takes place, partly because the standard interfaces for PC-like equipment –like mice, keyboards and screens– are typically not available for embedded systems. Rather, push buttons and small displays are used as user interfaces.

Tools for the design of embedded systems are very much influenced by the characteristics of embedded systems. What are those characteristics ? The following list contains some of the most relevant ones:

1 Embedded systems have to be **efficient**. One of the most notable instances of this is the need for *energy efficiency*, which applies at least to all portable embedded systems.

  Mobile phones and personal digital assistants are a very good example for this. They use orders of magnitude less energy than today's desktop computers and battery lifetime is a key argument for selling these systems.

  Energy consumption has also to be considered in the case of cars. Customers expect cars to be operational even after many weeks of not using them. Hence, the energy consumption of parked cars has to be extremely low.

  Weight is another aspect of efficiency. From looking at the mobile phone market (as well as at the laptop market) it becomes obvious that customers prefer light-weight systems.

  Silicon area is also very important, since many embedded systems have to be small and have to be fabricated at competitive prices.

  Many embedded systems -especially multimedia and communication systems- have very *high performance* requirements. These requirements have to be met with the least amount of resources.

  Progress in process technology might lead to less emphasis on the efficient use of silicon area and processor performance. However, progress in battery technology is expected to be slow. Hence, the importance of low energy consumption can be expected to increase in the foreseeable future.

2 Embedded systems are using an **increasing amount of software**. Only some peripheral components are implemented with special purpose hardware. The main reason for this is the flexibility of software, enabling a short time to market, late design changes and easy product upgrades. For most application areas, current processors are fast enough to meet performance requirements. Also, customers ask for more and more functions and these can be provided in software much easier than in special purpose hardware.

  As a result, it has been found that, for many application areas, the amount of software in embedded systems is doubling every two years [43].

3 The need of providing **instruction set compatibility**, which is a driving force in the PC-domain, exists only in cases in which a huge amount of legacy assembly language programs exist (which is sometimes the case in the car industry) and in cases in which development tools are difficult to change.

4 There is a **huge variety** of embedded systems. This is demonstrated by using the following examples:

- Information processing systems in cars are assumed to be working after disconnecting and then reconnecting the car battery. If this happens while the engine is running, there will be high-voltage transients. All designs have to guarantee that the electronic equipment within a car is fully functional after such transients [13]. It is frequently too expensive to add power regulators just for this purpose and the processors themselves have to withstand high voltages.
- Processor-based systems can be inserted into the veins of the human body in order to sense, analyze and store information about the blood [22]. Obviously, such systems have to be extremely small and power efficient. Processor performance can be low.
- For multimedia and consumer electronics applications, very high processor performance is required.

5 Embedded systems have to be **dependable**. Customers expect embedded systems to provide their functionality at all times. This is especially true for safety-critical products, but for non-safety-critical products, customer satisfaction is also a major concern.

6 The **functionality** of embedded systems is essentially **known at design time**. The flexibility of later adding significantly different applications, which is available for PCs, is not required. This has to be exploited in order to design an efficient system.

Not every embedded system will have all the above characteristics. However, we will assume that most of the characteristics discussed above will be present for the types of systems that we will talk about.

There is one immediate consequence from characteristics 1 and 2: embedded software and used processors must be efficient. In particular, they must make efficient use of energy and area.

2. Efficient hardware

The need for providing efficient processor hardware, combined with the huge variety of applications makes it impossible to work with just a small set of processors (there is no “one-size-fits-all” processor). As a result, processors optimized for embedded systems are quite common in such systems. In many cases, processors are optimized for certain application areas (such as audio and video applications) or even for certain applications. For most applications there is a matching processor.

Using this large variety of processors is possible because of the third characteristic. The reduced importance of instruction set compatibility is a pre-condition enabling the freedom of working with different optimized processors.

The following is a list of key characteristics of embedded processors:

- Embedded processors use a large amount of their circuitry for performing useful functions and hardly any amount for providing instruction set compatibility with earlier processors.
- Instruction sets are optimized for *certain application domains.* For example, instruction sets of digital signal processors (DSPs) include instructions that are optimized for digital signal processing. They contain multiply/accumulate instructions, modulo-addressing, parallel operations and special arithmetic modes [103].
- Hardware, which improves only the average execution speed but does not improve the *worst case execution time* (WCET) is frequently omitted. The reason for this is that many embedded systems have to guarantee meeting hard real-time constraints. Hence, many embedded processors come without caches.
- Hardware isolating user programs from the operating system and from other users can in many cases be omitted since the user is not expected to do any programming anyway. Therefore, memory management units and memory protection are found in only few processors.
- Energy efficiency of embedded processors (computed in terms of operations per Watt) significantly exceeds that of processors used in PCs.

In summary, optimizing processors for embedded applications is an area in which a major amount of money has been invested and in which a large amount of progress has been made.

3. Efficient software

3.1. How to specify embedded system software?

This is largely different for embedded software. Designing software for embedded systems is still not easy. Software generation techniques for embedded systems are still far from being satisfactory. Problems do already start at the specification level. How do we specify an embedded system? Many proposals have been made so far. Still, an ideal specification language meeting all requirements does not yet exist. A number of attempts for meeting the most important requirements are shown in fig. 1.1.

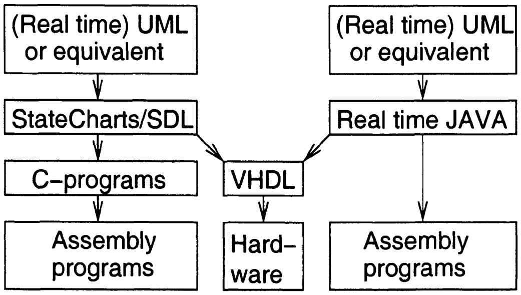


Figure 1.1. Approaches for specifying embedded systems

Due to the increasing amount of embedded software and due to the limited amount of trained embedded software engineers available, high software productivity is required. High software productivity can be obtained by specifying embedded software at a high level of abstraction, using compact specifications. This is reflected in the proposals for a real-time extension of the unified modeling language UML. At a slightly lower level, StateCharts and SDL are being used, as well as real-time extensions of Java. At still lower levels we find programs written in C, VHDL and assembly languages. Note that C is used as an intermediate step for most translations from higher level descriptions into machine languages. Hence, the efficient generation of machine programs from higher level specifications requires that C compilers generating efficient code for embedded processors are available.

3.2. Efficient compilers

These compilers have to meet the general goal of generating efficient code. However, C compilers for embedded processors are notoriously known for their poor code quality. Detailed studies of the code quality of available compilers were made in the context of the *DSPStone project* at the Technical University of Aachen [77]. In this project, manually generated assembly language programs were compared with compiled code. Some of the results are shown in fig. 1.2.

According to fig. 1.2, data memory overhead for compiled code can almost reach a factor of 5. Even worse, cycle overhead can reach a factor

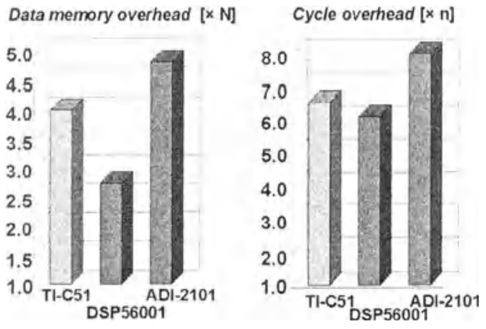


Figure 1.2. Overhead of compiled code for ADPCM algorithm

of 8. That means, up to $\frac{7}{8}$ of the useful processor cycles are needed just to compensate for the negative effect of the compiler. This is a serious problem for battery-operated systems. As a result, many embedded systems are still implemented in assembly languages, partially because the energy consumption of compiled code would be unacceptable.

However, a number of researchers are trying to improve the quality of the code generated by available compilers [196]. In some cases, zero or negative overhead has been reported. Due to the progress made in this context and due to the increasing number of processors on the market, other properties of compilers are gaining interest.

3.3. Retargetable compilers

The property gaining the largest increase is retargetability, which can be defined as the possibility of changing a compiler generating code for target processor A such that it will generate the code for target processor B. Obviously, this definition contains some vagueness. To some extent, every compiler is retargetable since changing almost all the compiler code can still be called a change of the original compiler. We call this a **very low level of retargetability**. At the other extreme, there are compilers that can generate code for another processor after just setting some switch. For currently available systems, the level of retargetability is somewhere in between. Recently, a number of approaches for generating retargetable compilers have been published. This led to the idea of presenting in this book a survey of the approaches that exist at the beginning of this century. Significant progress has been made since the last comprehensive survey [15] was published.

When comparing different approaches, we have to distinguish between various kinds of retargetability.

One form of changing the target processor is by changing some processor parameters such as the number of registers by using command-line arguments and simple configuration files. We will call this a **parameterized compiler**. Apart from a flexible number of registers, flexible cost functions for instructions and possibly some other parameters, the instruction set of parameterized compilers is essentially fixed. These compilers are relatively easy to design.

A more complex situation appears when the instruction set can be changed. Due to the availability of tools such as IBURG [195] and OLIVE [132], changing some instructions is not a major problem anymore as long as the overall structure of the instruction set, the register architecture etc. remain unchanged.

A very serious problem is to provide good optimizations for all possible targets. In general it can be stated that a price has to be paid for retargetability: it is much more difficult to supply good optimization techniques for a wide range of possible architectures than for a small range.

Another distinction is that between **user-** and **developer-retargetability**. For user-retargetability, changing the target processor is easy enough to be done by compiler users. User retargetability is typically restricted to very similar processors.

Why are people interested in retargetable compilers? There is a number of reasons for this interest:

- Due to the large variety of embedded processors, designing a new compiler for each and every processor is *too costly*. Also, it would frequently become available too late to be really used in the critical phases of the design.
- Retargetable compilers help to understand the *mutual dependencies* between computer architectures, instruction sets, compilers and the resulting code.
- A retargetable compiler can always be used as a *first step* towards a fully optimizing, target-specific compiler. Hence, it is useful even if we do not achieve the required code quality with a retargetable compiler.
- A very important application of retargetable compilers is *design space exploration*. Details about this will be presented in chapter 2.

In chapter 3 we will introduce some common terms and techniques of compiler construction. Some of the early work on retargetability will be presented in chapter 4. Possibly the most important chapter of this book is chapter 5, comprising a comprehensive set of descriptions of the major contributions in this area. Finally, chapter 6 provides a summary and outlook. A tabular overview of important tools is given in appendix A.

Chapter 2

COMPILERS IN EMBEDDED SYSTEM DESIGN

1. Design flow and hardware/software codesign

Applications of retargetable compilers have to be seen in the context of the overall design flow of embedded systems. Different design flows are being used. Nevertheless, fig. 2.1 can serve as an example representing a wide range of realistic design flows.

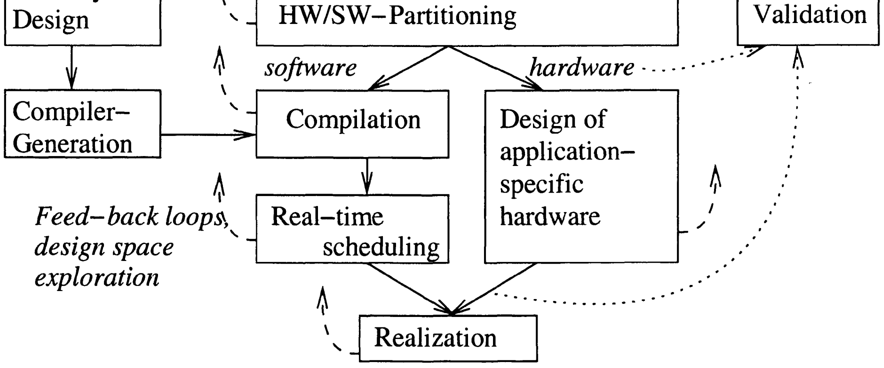

Specification
Early decisions, algorithm selection...
Processor–& Memory– Design
Operation to process mapping, HW/SW–Partitioning
Validation
software
hardware
Compiler– Generation
Compilation
Design of application– specific hardware
Feed–back loops, design space exploration
Real–time scheduling
Realization

Figure 2.1. Possible design flow

First of all, the embedded system is specified using the techniques mentioned in the previous chapter. Next, we have to map operations to threads of control, called tasks or processes. The processes then have to be mapped to hardware components. Possible hardware components include processors, custom *application specific hardware components* (ASICs) and *field programmable gate arrays* (FPGAs). A key advantage of custom application specific hardware is its speed and energy efficiency. In contrast, a key advantage of software is its flexibility, but implementing the same functionality in software does not result in the same level of performance and energy efficiency that can be obtained with ASICs. Generally, designers try to use only the necessary amounts of specialized hardware and to map as much functionality as possible to software. FPGAs also provide flexibility. Up till now, they are not power- and area-efficient and they are therefore not considered in this book, even though the situation may soon change.

Will all embedded systems be totally implemented in software in the future, due to increasing processor speeds? Certainly not, for a variety of reasons:

- some specialized interfaces to I/O devices are typically required for embedded systems,
- in parallel to increasing processor performance, the applications are also becoming more and more demanding, especially for multimedia applications ("*By the time MPEG-n can be implemented in software, MPEG-n+1 will have been proposed*"),
- it may be necessary to use specialized hardware in order to reduce the energy consumption.

Therefore, we have to assume that, in general, embedded system design involves both the design of specialized hardware and the design of a processor- and software-based part of the system. Hence, *partitioning* of operations into those that will be implemented in specialized hardware and those that will be implemented in software is required.

Partitioning is based on estimating the cost and the performance of implementing operations (or sets of these) either in hardware or in software. Mathematical programming or iterative procedures can then be used for partitioning. A closer look at different techniques for computing the estimates reveals that *compiling* processes is a way of finding cost and performance estimates for software [38]. This means that compilers may already be needed for hardware/software partitioning.

After partitioning, the design flows for hardware and for software are to some extent independent (see fig. 2.1), apart from design validations involving hardware and software.

In general, it is very difficult to predict the effect of early design decisions on consequences for the resulting design models. Therefore, it may be necessary to have design iterations. These design iterations correspond to control flow and are indicated by dashed lines in fig. 2.1.

According to the design flow, compilers are needed in different phases during the design of embedded systems:

- possibly already during hardware/software partitioning for computing *cost/performance estimates*,
- for *code generation* after hardware/software partitioning,
- as *verification* tools which check if certain operations can be implemented on certain hardware configurations.

In general, compilers are included in control flow loops for design space exploration and therefore they also play an important role for this design phase.

Standard, non-retargetable compilers can be used for all applications just mentioned. However, traversing a large design space requires either hundreds of compilers for all possible processor architectures or a retargetable compiler which can be configured to generate code for the currently considered target architecture.

Due to the importance of retargetable compilers for design space exploration (DSE), we will cover DSE in more detail in the following.

2. Design space exploration

2.1. Levels in the design space

The design space of embedded systems is typically quite large: numerous options exist for the choice of the algorithms, processors, memory systems, packages etc. There are also less legacy problems caused by code compatability requirements. Since embedded systems have to be efficient, this freedom has to be exploited in order to generate very efficient designs. These can be found by traversing the design space. This space is n-dimensional, where each dimension corresponds to a design choice for which alternatives exist.

Fig. 2.2 shows a possible conceptual view of the design space [25]. At the top level, we start with a specification of the design. This is frequently called *back-of-the-envelope* (even though envelopes can hardly be used to sketch all key ideas of the design, let alone a full specification). Specifications originally only exist in people's minds. A subset of

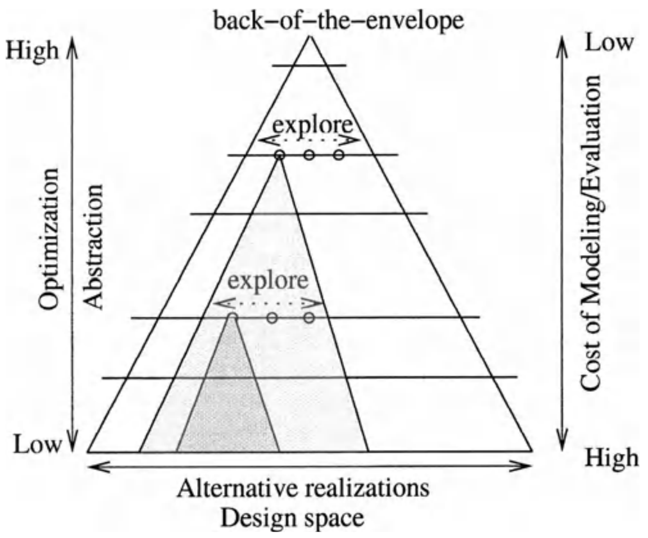


Figure 2.2. Conceptual view of the design space

these can be captured in the initial written specification. Unfortunately, currently available specification techniques do not allow us to express exactly what should be captured. Some properties cannot be specified in formal languages (for example, how do you specify "user-friendliness"?). In many cases, it is necessary to specify algorithms, even though many different algorithms computing the same function exist. Due to these reasons, we consider the specification to be the set of ideas about the product that exist in the head of the person(s) specifying the embedded system.

During the design process, many different design decisions can be taken at all levels of abstraction. It is well-known that it is very expensive to change a decision taken at a high level after discovering at a low level that this decision should be revised.

In the context of this book, we distinguish between a certain number of levels (there may be other levels in other contexts). These will be the levels that we will consider:

1 The highest level, at which we will be considering alternatives, will be the **algorithm level**. At this level, alternatives consist of different options that exist for the essential algorithms of the embedded system to be designed.

Examples include different DCT or FFT algorithms, different algorithms for character recognition, or different algorithms for data compression.

2 At the next lower level, we consider different **options for implementing algorithms**. These are options that are beyond the scope of currently available compiler optimizations. For example, we may find that only small regions of an array are needed at any given time and we may hence decide to fold different regions, thereby possibly saving a significant amount of memory space.

3 For software-based realizations, there are different alternatives for **mapping operations to processes**. For the same type of realizations, the **mapping of processes to processors** has to be optimized. This mapping is expected to become more important in the future. Both mappings may be part of the partitioning between those operations that will be implemented in hardware and those that will be implemented in software (the so-called **hardware/software partitioning**). Both mappings can also be generated in an independent design step.

4 For embedded systems, there may be a large amount of **choices for the peripheral components** (displays, keys, disk drives) as well as for the power supply subsystem. We will not consider these choices in this book.

5 Due to the rapidly increasing processor speeds, the **memory subsystem** is becoming extremely important and more options have to be considered in order to design a high-speed, lower power memory subsystem.

6 For the processors that are used, there are various options as far as the **instruction sets** are concerned.

7 Finally, we mention several options that exist for the **internal structure of processors** (micro-architecture).

In the following, we will examine the usefulness of compilers (and of retargetable compilers in particular) for exploring the design space. We will present examples of applications of retargetable compilers. These examples will show that the use of retargetable compilers for DSE is becoming a standard technique for medium and lower levels of the decision tree.

2.2. Algorithm selection

Automatically selecting the right algorithm has been a dream of many researchers. In the digital signal processing (DSP) domain, there has been the dream of automatically finding the right motion estimation algorithm. In numerical mathematics, there has been the dream of automatically switching between sparse and dense representations of matrices. After many years of research, some results are available. These include the work in the SPIRAL project on the automatic transformation of algorithms, in particular DSP algorithms. The design flow in the SPIRAL project is shown in fig. 2.3.

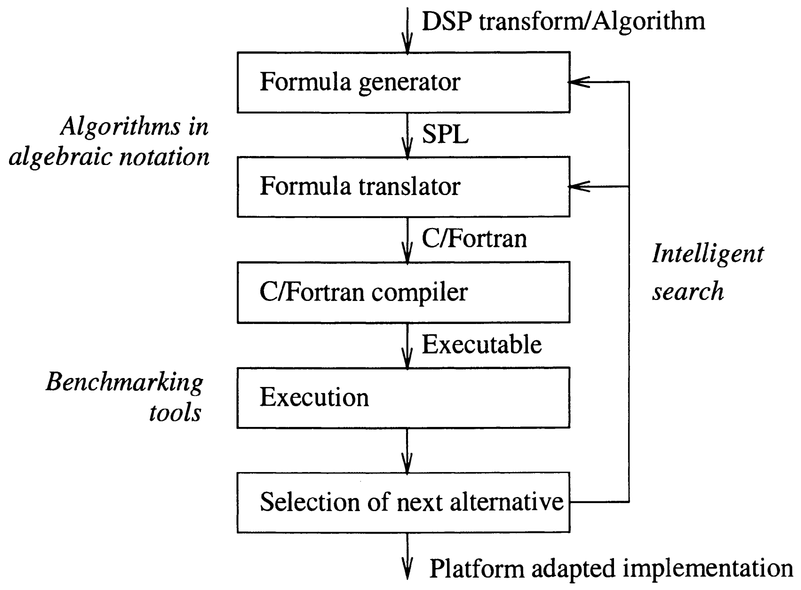


Figure 2.3. Selection of algorithms in the SPIRAL project

Different algorithms are generated from the same algebraic description of a certain DSP transform. Each algorithm is then compiled and evaluated. The result of an evaluation controls the generation of the next algorithm. Using clever search strategies [42], it has been possible to try only certain design points and to focus on the promising parts of the design space.

In the cited work, standard compilers are used within the feedback loop. If retargetable compilers were used, it would be possible to analyze dependencies between algorithms and processor architectures even for architectures for which a standard compiler does not exist.

2.3. Options for implementing algorithms

Design space exploration techniques at this level have be studied at IMEC (Leuven, Belgium) in the context of their ATOMIUM and ADOPT projects [12, 5, 11].

These projects focus mainly on multimedia applications. Such applications require very large amounts of data to be stored, transfered, and processed. Consequently, very large data storage and transfer capacities are needed, and the cost for these is one of the largest contributions to the overall system cost. In order to reduce this cost, design transformations designed in the ATOMIUM project aim at reducing these capacities. For example, not all of the regions of an array may be needed at the same time. Hence, these regions may be folded such that at any time, only a small fraction of the array needs to be stored. Folding arrays reduces the memory size requirements and consequently also the power consumption and improves performance. A detailed analysis of the tradeoffs between different techniques for implementing storage and transfer mechanisms is the key feature of the ATOMIUM project.

Transformations such as array folding reduce the memory size, but may increase the amount of necessary computations. For example, various if-statements may be required in order to distinguish between the different regions of an array and complex modulo operations may be required for mapping index values to the small set of locations implementing that array. Transformations developed in the ADdress OPTimization project (ADOPT) project try to reduce the amount of necessary computations and in particular try to remove the additional computations inserted during the ATOMIUM transformations. The techniques are based on the use of optimizing source code level transformations which are organized in two stages: a processor target independent stage and a processor (family) specific one.

The overall result is a modified algorithm which typically requires smaller memories, less energy, less computations, and which consequently shows better performance.

In the ADOPT and ATOMIUM projects, standard compilers are used for evaluating the benefits of applied transformations. More advanced compilers could integrate ATOMIUM and ADOPT techniques directly. Retargetable compilers could be used for analyzing a larger design space.

2.4. Process mapping and HW/SW partitioning

For most of the approaches for specifying embedded systems, specifications include *concurrent processes.* Partitioning of the specification into processes is, first of all, a matter of convenience. Consider, for example, an answering machine. It it very convenient to describe the monitoring of the incoming line and the keys as more or less independent processes. It may be inefficient to implement a system using exactly those processes that are described in the specification. Instead, Instead, it may be more efficient to merge or to split processes used in the specifications:

- For example, if the execution of process A included in the specification will always be followed by the execution of process B, it is very wise to merge A and B in order to reduce context switching overhead.
- On the other hand, splitting a process into two may be very useful if that process includes a wait-for-input operation. In that case, all operations following that wait operation should be turned into a separate process and that process should be triggered if the input is available. This way, a blocking wait-for-input operation consuming processor time can be turned into an event-controlled processor scheduling requiring no active waiting.

Techniques for analyzing the design space in this context have been described by Thoen [7].

Due to the increasing use of multiprocessor systems, finding a mapping of processes to processors is also required. This mapping is very much correlated to the process splitting and merging discussed above and can hardly be done independently. Finding such a mapping is also very much affected by decisions to map certain operations to hardware or to software.

Taking such decisions is called *hardware/software partitioning.* In hardware/software partitioning, we try to find a mix of hardware and software such that costs are minimized while meeting all design constraints. A huge amount of techniques for hardware/software partitioning have been proposed. For example, we can map all operations to hardware and then gradually move operations to software as long as performance constraints are met. In another approach [38], a comprehensive mathematical model of the optimization problem is generated and the cost function is minimized using mathematical programming techniques. However, at the current state of the art, it is not feasible to generate sufficiently good partitionings without using any feedback loops. Rather, feedback loops are necessary to let results of one partitioning step have an effect on the next partitioning (see dashed line in fig. 2.1).

The effect of the options at this level can also be evaluated by means of a compiler. Again, retargetable compilers can offer a wider choice of processor architectures.

2.5. Memory system design

Traditionally, the memory system was transparent to compilers, i.e. the memory system was modeled as a single homogeneous array of storage locations. Due to the larger access times and the increased energy consumption of smaller memories, it makes sense to design non-homogeneous memory systems comprising a variety of memories, including combinations of caches and memories mapped to parts of the address space. In order to fully exploit the benefits of such memory systems, compilers should be memory-aware.

Grun et al. designed a methodology for generating such compilers from descriptions of the architecture [19]. Grun, Mishra et al. have described how this methodology can be used for DSE [36]. In particular, they analyzed the performance of six different memory configurations attached to a TI 6211 processor. Table 2.1 contains these configurations.

Configuration	L1 cache	L2 cache	SRAM	stream buffer	DRAM
1	128B; l=1; LRU	-	-	256; l=4	l=20
2	128B; l=1; LRU	-	2k; l=1	-	l=20
3	128B; l=1; FIFO	2k; l=4; FIFO	-	-	l=20
4	128B; l=1; LRU	2k; l=4; LRU	-	-	l=20
5	128B; l=1; FIFO	2k; l=4; FIFO	1k; l=1	-	l=20
6	-	-	8k; l=1	-	l=20

Table 2.1. Size, latency and replacement policies for the six memory configurations

Options include two levels of caches, an SRAM mapped into the address space (a so-called *scratch-pad memory*), and a stream buffer. In addition, a DRAM background memory is assumed. Fig. 2.4 shows results of the DSE for the six configurations. Obviously, it is not a very good idea not to use any cache (see configuration 6) for these applications.

Other key parameters of the memory system include the type and number of memory ports, especially for VLIW (very long instruction word) processors. These have a major influence on the overall system structure. Jacome et al. explore these options of the design space. For speed reasons, estimation techniques rather than retargetable compilers are used [23].

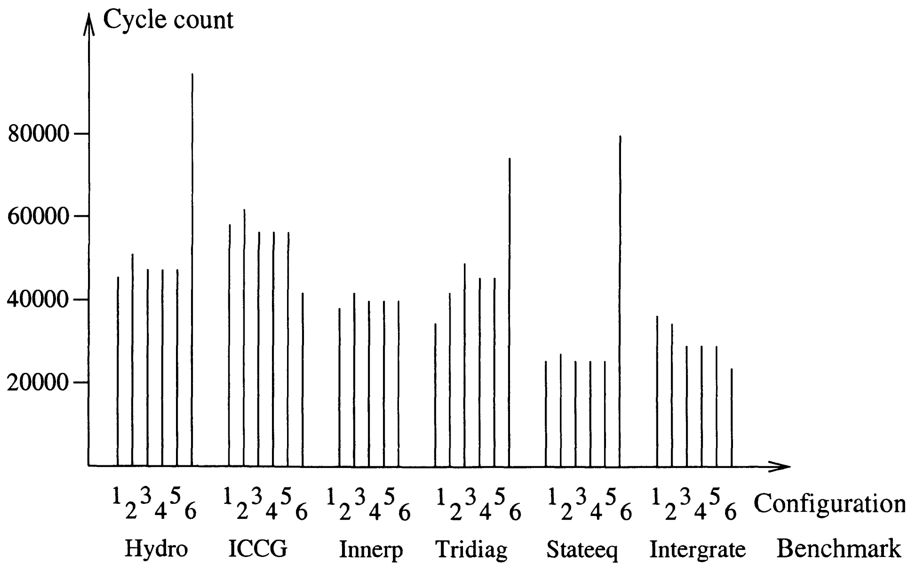


Figure 2.4. Cycle counts for the memory configurations of table 2.1.

2.6. Instruction set options

2.6.1 Design of VLIW machines

Due to the much reduced instruction set legacy problem in embedded systems, instruction sets can be customized for certain applications.

One key opportunity resulting from this is the opportunity to use VLIW instruction sets. With such instruction sets, several instructions can be started at the same clock cycle. The corresponding increased performance can be used for scaling down the supply voltage, resulting in significant energy savings.

An example of a search for power-optimized instruction sets is described by Kin et al. [26]. Kin analyzes the effect of architectural parameters such as the number of functional units, the issue width, cache sizes etc. on the energy consumption during the execution of multimedia benchmark programs. A framework for the selection of power-efficient media processors is used. The IMPACT tool suite [152] (see section 5.3.2) is used to generate code for different architectures. Fig. 2.5 shows one of the results. The figure shows that processors requiring a larger chip area can be more energy efficient than smaller processors. The main reason for this is that larger chip areas permit more parallelism and hence the same deadline can be met with lower supply voltages.

The IMPACT tool suite can generate code for a wide range of VLIW machines. For other machines it is less appropriate. Due to the difficulty

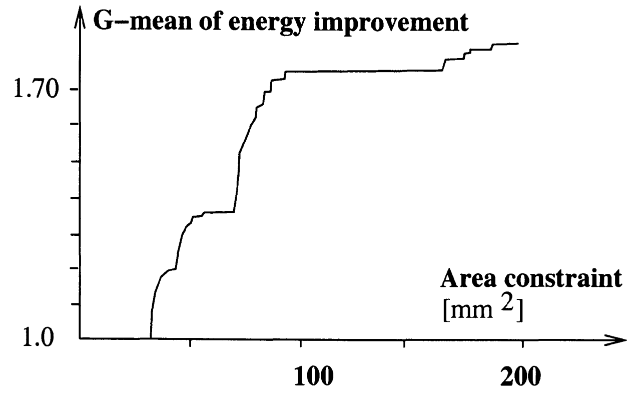


Figure 2.5. DSE with a compiler, considering energy

of designing good retargetable compilers, researchers have tried to avoid retargetable compilers and to find other ways of estimating the behavior of hardware with respect to different evaluation metrics. Various researchers have recently published papers in this direction [16, 20]. Good retargetable compilers, however, can make separate estimation tools for computing these metrics obsolete.

Some early high-level synthesis tools were also designed for *instruction set synthesis*. The MIMOLA synthesis system (MSS) is an example of this. Figure 2.6 shows a result from one the tools of the MSS, MSSH [30].

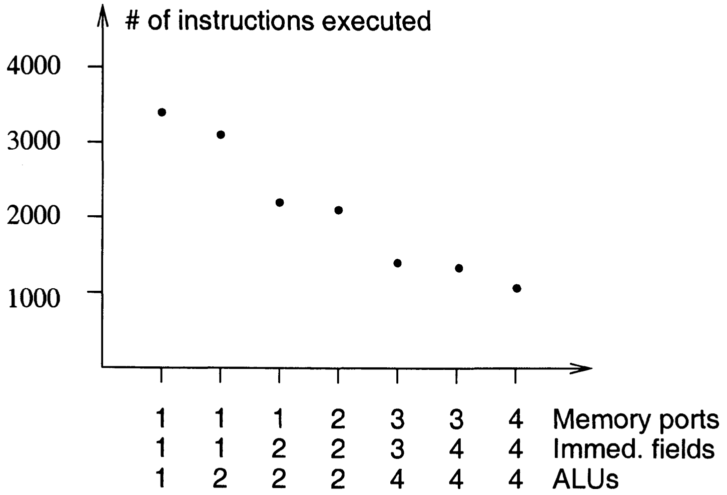


Figure 2.6. DSE with synthesis tool MSSH

MSSH considers constraints for the main resources: the number of memory ports, the number and types of functional units (or ALUs) and the number of immediate instruction fields. MSSH can therefore be used for DSE. The VLIW instruction set is an implicit result of the synthesis process focusing on the main resources.

Comprehensive DSE for VLIW machines containing *clusters of functional units* (adders, multipliers etc.) and associated memory ports has been implemented by Lapinskii, Jacome et al. Fig. 2.7 shows the overall approach [27].

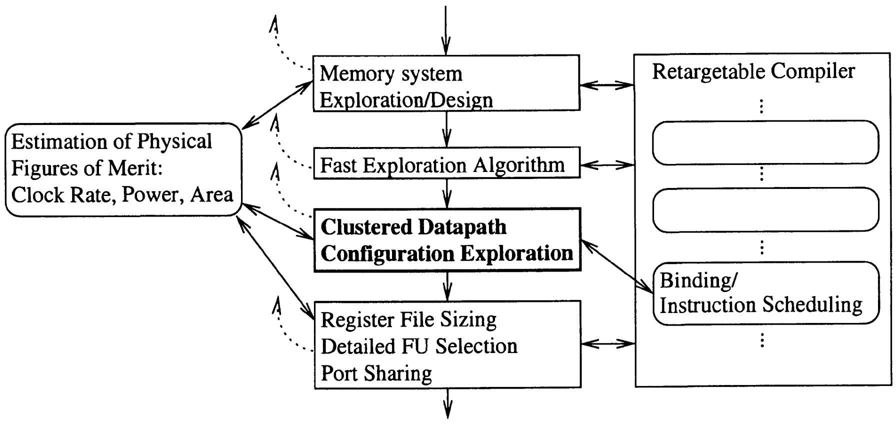

Memory system Exploration/Design
Fast Exploration Algorithm
Clustered Datapath Configuration Exploration
Register File Sizing
Detailed FU Selection
Port Sharing
Estimation of Physical Figures of Merit: Clock Rate, Power, Area
Retargetable Compiler
Binding/ Instruction Scheduling

Figure 2.7. DSE in Lapinskii's approach

First of all, the memory subsystem is designed and a broad schedule of memory operations is performed. Next, the fast DSE algorithm [23] just mentioned is used to identify reasonable ranges for key design parameters such as the maximum number of functional units per data path cluster (N_F), the number of clusters (N_C) and the number of busses (N_B).

The kernel of the DSE (shown in bold face in fig. 2.7) consists of an exploration of values of the triple (N_F, N_C, N_B) providing good cost/performance tradeoffs. The types of functional units are considered to be of secondary importance and decisions about these are postponed. Promising sections of the three-dimensional space are traversed. For each point in the space, the instruction scheduling component of a retargetable compiler schedules operations and computes the resulting latency. Table 2.2 shows the type of results that can be generated this way.

An interesting observation is that some partitioned architectures achieve the same performance as very expensive centralized architectures. For example, the architecture with N_C=3 clusters containing N_F=3 functional units each and the architecture with N_C=1 cluster and 8 func-

	N_F										
	2	3	4	5	6	7	8	9	10	11	12
N_C											
1	37	23	19	14	12	11	10	9	8	8	7
2	19	12	10	9	8	8	8	8	7		
3	13	10	9	8							
4	11										

Table 2.2. Latency for algorithm DCT-DIT for N_B=2 and bus latency = 1

tional units both achieve a latency of 10. The large number of ports of the register file required for the cluster makes the latter very expensive.

This kernel of Lapinskii's DSE algorithm is followed by a detailed design of functional units and sizing of register files. The generation of instruction sets is implicit in Lapinskii's approach.

2.6.2 Design of non-VLIW machines

Focusing more on explictly modeling instruction sets, Huang et al. try to synthesize instruction sets from algorithms using the ASIA framework [21]. In ASIA, both the algorithm(s) and a generic micro-architecture are given. ASIA includes a "retargetable mapper" which maps operations to data path units. From this mapping, instruction sets are extracted. The retargetable mapper is not a real retargetable compiler, but it is quite similar.

Imai et al. have designed the PEAS design system, the currently available version being PEAS-III (see section 5.4.2). This system has been used for designing ASIPs (Application Specific Instruction set Processors). Examples in experiments include a MIPS R3000 compatible processor, DLX, a simple RISC controller, and the PEAS-I core. In the experiments, the easiness of design and modification procedure with the goal of improving design quality in terms of performance and hardware cost has been proven: *It has been confirmed that the design method used in PEAS-III is effective to design space exploration for simple pipelined processors* [1].

2.6.3 Word length optimization

In embedded system design, standard data types of the C language are insufficient to describe the word length that is actually required for the variables. Accordingly, a number of extensions to C have been proposed which allow the user to exactly specify the word length for the different variables.

One particular instance of this is the Valen-C language, designed by Yasuura et al. [50]. Examples of variable definitions in Valen-C are shown in fig. 2.8, left.

Yasuura et al. have designed a processor model called *Soft-core*. This model includes generic parameters such as the word length for data. This model is synthesizable for all reasonable values of these parameters, resulting in the ability to automatically generate layout data. The tool suite comes with a retargetable compiler which can map different data types to the word length defined by the generic parameter. Two examples can been seen in fig. 2.8, center and right.

```
int12 x;
int20 y;
int24 z;
```

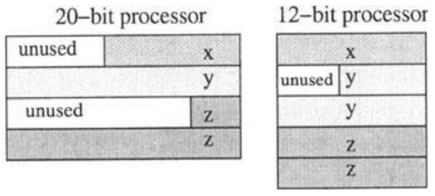


Figure 2.8. Mapping of Valen-C data types to hardware-supported word lengths

Setting the generic word length parameter to different values, the effect of this design parameter on the resulting speed, size and energy consumption can be analyzed. Fig. 2.9 shows this effect.

length of variables	# of variables
4	257
8	257
14	3
39	258

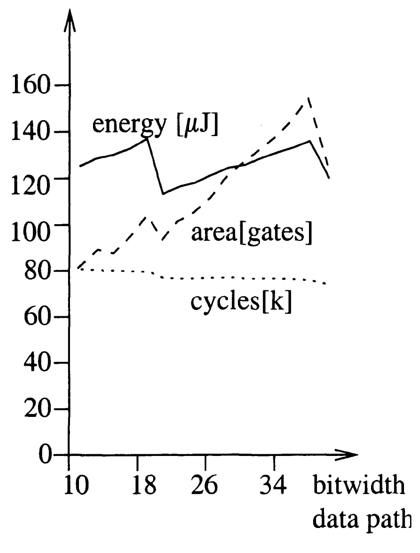


Figure 2.9. Effect of machine word length on some figures of merit

It is obvious that this analysis requires a retargetable compiler. The compiler designed by Yasuura et al. is discussed in more detail in section 5.4.3.

2.6.4 Register file sizing

The size of register files is another parameter that can be fixed during the design. Traditionally, decisions concerning the size of the register files were based on an empirical analysis. Most designers had no options anyway, because they had to use existing processors. Embedded system design opens new opportunities in this context, since application- or domain-specific processors can be used. Also, it is important to precisely analyze figures of merit such as the delay time and the energy consumption of register files of different sizes.

A detailed study concerning the code size, the number of clock cycles and the energy consumption was done by Wehmeyer, Jain et al. [48]. All results are for an ARM-based processor executing the THUMB instruction set, for which the number of registers is considered a generic variable. Fig. 2.10 shows an analysis of the effect of the register file size on the power and energy consumption.

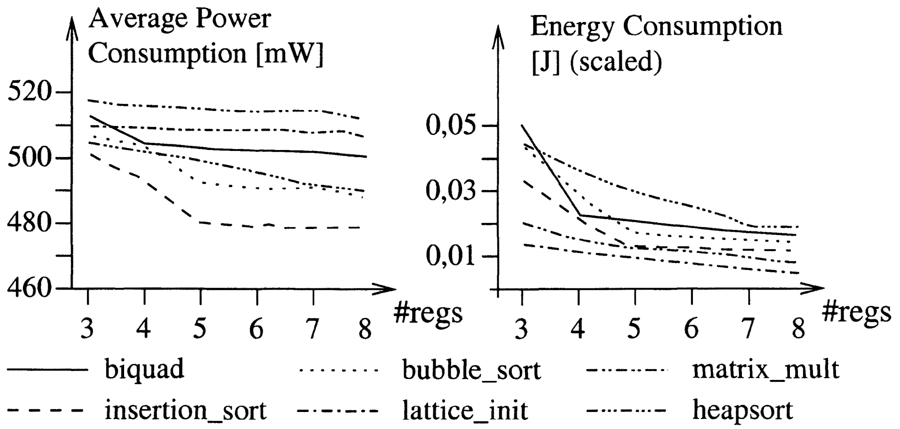


Figure 2.10. DSE of the register file size

An interesting observation is the fact that increasing the register file size from 3 to 8 can reduce the energy consumption by a factor of about 5.

Again, it is obvious that this analysis requires a retargetable compiler. In this particular case, the retargetability of the used compiler is limited and will not be discussed in more detail; it was just employed for demonstrating possible applications of a retargetable compiler.

The ARM processor issues at most one instruction per clock cycle. For multiple-issue machines, such as VLIW processors, a larger set of registers may be useful. An analysis of the effect of the number of registers of different figures of merit for VLIW processors was done by Valero et al. [8]. Fig. 2.11 shows the corresponding results. The large number of registers that can be used at the same time is due to the parallelism of the VLIW machines and due to algorithms that can be parallelized.

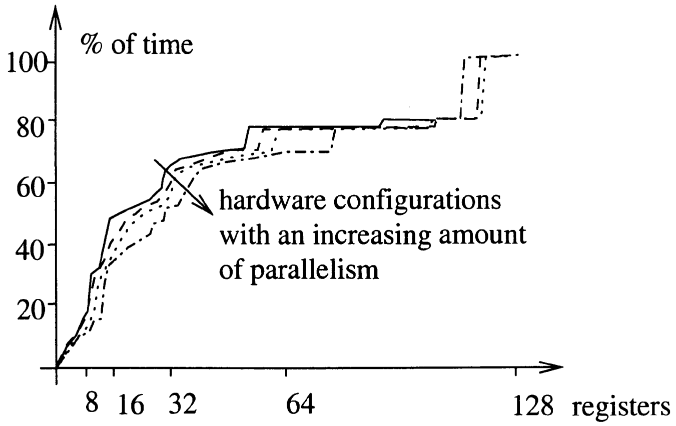


Figure 2.11. Number of concurrently used registers for VLIW machines

2.7. Micro-architectural options

Various options also exist at the micro-architectural level. DSE at this level does not require the availability of a retargetable compiler, since these options, by definition, have no effect on the instruction set. However, the cost functions used by the compiler could be affected by decisions taken at this level. Hence, design space experiments at this level can mostly be performed with a compiler for a fixed architecture provided that this compiler can be configured using cost function definitions.

Examples of DSE at the micro-architectural level that do not require retargetable compilers include

- an analysis of the tradeoff between in-order completion and out-of-order completion in super-scalar pipelines [35],
- an analysis of the tradeoff between using a multiply unit and a multiply coprocessor [35].

For these cases, cycle-true simulators have to reflect the details at the micro-architectural level.

There is one exception, though, in which flexibility of the compiler is required even for changes at the micro-architectural level: if features are added at the micro-architectural level which require new compiler optimizations to be exploited.

An example of this is the use of a scratch pad memory mapped into the address space, which is not visible at the instruction set level. Typically, specific optimizations have to be added to the compiler to exploit such a feature [45]. General optimization techniques for general models of memory hierarchies are normally not available in any retargetable compiler, so far. Hence, they have to be hand-coded.

Up to a certain extent, DSE can be done with automatic synthesis tools. Synthesis tools can typically accept certain design constraints, like type and number of arithmetic/logic units (ALUs) and memory ports. These constraints may be sufficient to analyze designs characterized by these metrics.

For designers wanting to specify further levels of detail, just specifying the type and the number of ports and ALUs is not enough. All the paths may have to be specified in order to really study the effect of different design options that exist. These options may include different ways of implementing pipelines, like this was done by De Gloria and Faraboschi [18]. Fig. 2.12 shows one of their results. Unfortunately, the authors do not provide many details about the retargetable compiler which they designed.

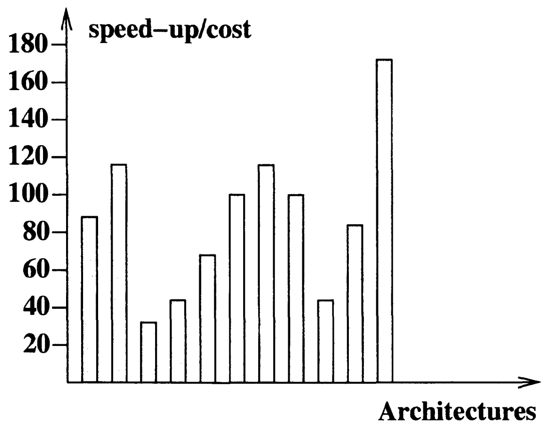


Figure 2.12. DSE with a compiler, considering interconnect

3. Design verification

For architectures with heterogenous register files, it is not easy to always know the effect of deleting hardware resources during some optimization step. It can easily happen that the designer deletes too many resources. Compilation can be used to check if a certain hardware can actually perform a certain operation like a transfer between two specialized registers. Retargetable compilers driven by some hardware description which allows machine registers to be referred to can check this.

Takagi [46] described a method for checking if given register transfer structures are able to execute certain register transfers. During this checking, the ability to provide the required control code for all functional units has to be guaranteed. This means that the functionality of generating control code has to be implemented, even though Takagi's tool does not really output control code.

In the context of the MIMOLA system, using a retargetable compiler for design verification was described by Nowak [32]. Also, special retargetable compilers can be employed for generating test programs from a description of test patterns to be applied to internal processor nodes [216] (see also section 5.5.7).

We have now seen what retargetable compilers can be used for. In the next chapter, we will discuss some fundamental techniques for implementing compilers in general and retargetable compilers in particular.

Chapter 3

SOME COMPILER TECHNOLOGY BACKGROUND

Since many approaches to retargetable compilation share some concepts, e.g. concerning source language frontends, intermediate representations, and basic code generation techniques, in this chapter we give some essential background information from a practical viewpoint. In particular, we will not discuss detailed algorithms, but mainly focus on the terminology. For more comprehensive "classical" compiler technology background, several recent textbooks are available, such as [53, 54].

A general overview of compiler phases, as we discuss it here, is given in fig. 3.1. However, it is important to mention that there is no unique "one size fits all" compiler organization. Naturally, there are a lot of trivial dependency constraints (e.g. IR optimization must follow source code analysis, and register allocation must follow code selection). However, particularly the organization of optimizations at the intermediate representation level as well as the detailed backend organization may show significant variations in different compilers. This is due to the fact that some target processors do not require certain optimizations, the available compilation time may limit the amount of potential optimizations, or certain passes are simply not available in a given compiler infrastructure.

With respect to code optimization, which is particularly important for embedded systems, the compiler's capabilities are limited anyway. One can theoretically show that it is impossible to design a compiler which generates optimal code for all input programs. This means that we can just try to perform "as good as we can" within a reasonable amount of compilation time, but there will always be some improvement opportunities left (Appel [53] calls this the *full employment theorem for compiler writers*).

One key approach to obtain good code quality (since we generally do not know the optimum, "good" mostly denotes code quality similar to what we can achieve by hand-coding in assembly) is to study the *mutual dependence* of compiler passes. Due to complexity[1] and software engineering reasons, a practical compiler may be subdivided into dozens of separate phases, each of which may impose unnecessary restrictions on subsequent phases. The *phase coupling* approach aims at eliminating this problem by intermingling traditionally separate compiler phases, so as to generate better code. We will address this issue later in this chapter.

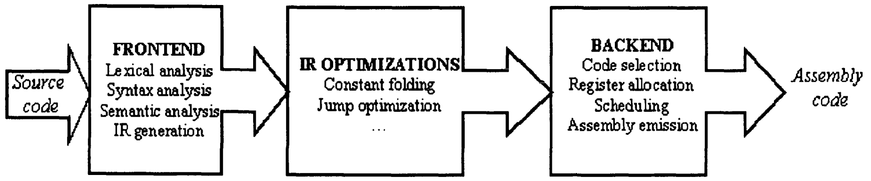


Figure 3.1. Coarse compiler phase organization

1. Frontend

The task of the source language frontend is to analyze a given source program, check for errors, and (in case of a correct input) generate an intermediate representation (IR) for the subsequent compiler passes. A frontend typically comprises the following three phases:

Lexical analysis. Initially, the input program is nothing but a string of ASCII characters. Lexical analysis recognizes substrings from the input stream and combines then into *tokens*. Each token denotes a primitive element of the source language, e.g. a keyword, a number, an

[1]Many subtasks in code generation, such as global register allocation, scheduling under resource constraints, and address code optimization, are known to be *NP-hard*. The set NP includes all *decision* problems that can be solved in polynomial time by a nondeterministic Turing machine, while P is the corresponding set for deterministic Turing machines. A decision problem Π is *NP-complete*, if $\Pi \in NP$, and Π is "at least as difficult" as all other problems in NP, which can be formally proven by constructing a polynomial-time transformation between problems. Here, we usually have to deal with *optimization* problems. Optimization problems with cost functions, that can be computed in polynomial time, can be solved in polynomial time, if their decision counterpart can be solved in polynomial time. Optimization problems with an NP-complete decision counterpart are called *NP-hard*. For the detailed theory cf. [192]. Exactly solving NP-hard and NP-complete problems most likely requires exponential-time algorithms (unless $P = NP$, which is a major open problem in complexity theory).

identifier, or an operator symbol. While tokens for keywords or operators are simply represented by integer numbers, identifier or number tokens are attributed, e.g. with a string or a numerical value.

Most primitive language elements can be represented by *regular expressions*, which can be parsed by means of *finite state machines* (FSMs). The FSM reads input characters and stores them in its internal states, until some language element (e.g. a "while" keyword or a floating point number) has been recognized. Then it emits the corresponding token, and returns to its initial state to parse the next input word.

As a by-product, lexical analysis also suppresses white spaces, newlines, and tabs from the input. A special handling is usually required for comments that show a "balanced parentheses" structure (like "/* ... */" in C or "(* ... *)" in PASCAL). These cannot be directly parsed with FSMs due to their finite number of states, but can easily be handled with special support functions.

Tools performing lexical analysis are called *scanners*. Scanners can conveniently be built with the widespread UNIX tool LEX. LEX reads a lexical specification in the form of a list of regular expressions and generates C code for the corresponding FSM that recognizes these expressions. Fig. 3.2 shows a part of a LEX specification for the ANSI C language. The interface to the FSM generated by LEX is the "yylex()" function. The syntax analyzer calls it each time it requires a new token.

Syntax analysis. While lexical analysis views the input program as a string of regular expressions, the syntax analyzer, or *parser*, considers it as a token string produced by the scanner. The parser analyzes the syntactic structure of the token string w.r.t. an underlying *context free grammar*

$$G = (T, N, R, S)$$

where T is a finite set of *terminals* (the set of possible tokens), N is a finite set of *nonterminals*, and $S \in N$ is the *start symbol*. The *rule set* R contains rewrite rules of the form

$$X \rightarrow (T \cup N)^*$$

i.e., X can be replaced by a string of symbols from T and N. The term *context free* denotes that X has to be a member of set N. As a result, the parser produces a *parse tree*, that represents a derivation of the input program from G's start symbol.

Context free grammars cannot be parsed by FSMs but require *stack automata*. One way to implement a stack automaton for grammar G is to read one input token after another, in left-to-right order, from the scanner, and in each step to perform one out of two possible actions:

```
// declaration of some special regular expressions
D                       [0-9]       // decimal digits
IS                      (u|U|l|L)*  // integer suffixes
...

// C declaration part
%{
// token definitions
#define AUTO      290
#define REGISTER  291
#define CASE      292
#define CHAR      294
#define BREAK     319
...
%}

// regular expression list
%%
// C keywords
"auto"          { return(AUTO); }
"break"         { return(BREAK); }
"case"          { return(CASE); }
"char"          { return(CHAR); }
"const"         { return(CONST); }
"continue"      { return(CONTINUE); }
...
// integer constant is a string of decimal digits,
// followed by optional suffix
{D}+{IS}?       { return(INT_CONSTANT); }
// C operators
...
"&="            { return(AND_ASSIGN); }
"^="            { return(XOR_ASSIGN); }
"|="            { return(OR_ASSIGN); }
">>"            { return(RIGHT_OP); }
"<<"            { return(LEFT_OP); }
...
```

Figure 3.2. Partial LEX specification for ANSI C

shift or *reduce*. "Shift" means the current input token is pushed onto the stack for later use, while "reduce" denotes the application of a grammar rule from R. Which action is to be taken can be determined from the current stack contents, the current input token, and some lookahead on the remaining token stream. For instance, consider the case that the input character sequence is a C assignment like

```
    x = x + 1;
```

for which the scanner would generate the token sequence
<id> <eql> <id> <plus> <const> <semicolon>

The parser reads the tokens and performs shift actions until the expression "`x + 1`" has been found, which can be reduced to an "<expr>" nonterminal:

<id> <plus> <const> $\rightarrow$ <expr>

The "expr" nonterminal replaces the sequence "<id> <plus> <const>" on the top of stack. Now, when reading the next token "<semicolon>" the parser knows that a statement has been finished and reduces the whole token sequence to an assignment nonterminal:

<id> <eql> <expr> <semicolon> $\rightarrow$ <asgn>

Writing parsers manually is a tedious job, but fortunately it can be automated by LEX's sibling tool YACC [55] (or some more recent extensions like GNU's FLEX and BISON tools). YACC reads a context free grammar specification and generates C code for a corresponding shift-reduce parser. A example for an ANSI C parser is given in fig. 3.3.

The interface to the YACC-generated parser is the "yyparse()" function. After setting up the input file, a call to this function from the compiler driver program generates the complete parse tree. In case a syntax error is encountered during parsing, a user-defined "yyerror()" function is called, which can be used to emit a detailed error message. YACC also shows limited support for *error recovery*, which allows to continue parsing even after an error has occurred, so as to accelerate the usual edit-and-compile cycles during program development.

Semantic analysis. Programming languages like C are not completely context free, but are actually *context sensitive*. However, parsing context sensitive grammars is much more complicated, and the above approach with context free grammars works well in practice if we perform additional analyses with special support functions. In an ANSI C compiler, analysis tasks which are typically not part of the parser but are performed afterwards within a dedicated *semantic analysis* phase include the following:

- Book-keeping for symbol declarations, including data type, storage model, and scope.
- Checking for correct operand type combinations in expressions.
- Checking whether an assignment has an "lvalue" on its left hand side, e.g. assignments to constants, complete arrays, and functions are not allowed in C.
- Checking whether target labels of "goto" statements are defined in the current function.

```
// token declarations, these are automatically converted
// into #defines for the LEX spec
%token IDENTIFIER CONSTANT STRING_LITERAL SIZEOF
%token PTR_OP INC_OP DEC_OP LEFT_OP RIGHT_OP LE_OP GE_OP EQ_OP NE_OP
%token AND_OP OR_OP MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN ADD_ASSIGN
...

// grammar rules, nonterminals are implicitly declared
%%
// primary expression
primary_expr
        : identifier
        | CONSTANT
        | STRING_LITERAL
        | '(' expr ')'
        ;

// expression with postfix operator
postfix_expr
        : primary_expr
        | postfix_expr '[' expr ']'                // array access
        | postfix_expr '(' ')'                     // function call
        | postfix_expr '(' argument_expr_list ')'  // function call
        | postfix_expr '.' identifier              // struct access
        | postfix_expr PTR_OP identifier           // indir struct access
        | postfix_expr INC_OP                      // post-increment
        | postfix_expr DEC_OP                      // post-decrement
        ;

// unary operator with expression
unary_expr
        : postfix_expr
        | INC_OP unary_expr          // pre-increment
        | DEC_OP unary_expr          // pre-decrement
        | unary_operator cast_expr   // unary operation
        ;
...
```

Figure 3.3. Partial YACC specification for ANSI C

It is convenient to partially integrate semantic analysis into syntax analysis by means of *attribute grammars.* An attribute grammar is constructed "on top" of a context-free grammar $G = (T, N, R, S)$. In an attribute grammar, all symbols $s \in T \cup N$ are annotated with an attribute set $A(s)$. An attribute $a \in A(s)$ can be considered as a container for semantical information about a symbol, such as its type or scope.

The value of an attribute $a \in A(s)$ is determined by an *attribute definition* $D(a)$, an equation attached to a grammar rule in which a occurs. Attributes fall into two classes: *synthesized* and *inherited* attributes. An attribute $a \in A(s)$ is called synthesized, if s is a nonterminal on the left hand side of the rule, and its definition only depends on attributes of grammar symbols on the right hand side of the same rule. Conversely, inherited attributes belong to symbols on the right hand side of a rule r, and their value only depends on attribute values of r's left hand side. As an example, consider the ANSI C grammar rule

```
jump_statement: BREAK ';'
```

where "`jump_statement`" is a nonterminal and "`BREAK`" and "`;`" are terminals, or tokens. One attribute we would typically like to add to "`jump_statement`" is a flag "correct" that signals whether or not a given jump statement is semantically correct, so that we can emit an error message if required. In our little example, the jump statement is obviously correct, if it is a correct "break" statement (this statement is used in C for "unstructured" termination of loops). So, the attribute "`jump_statement.correct`" has to be *synthesized* from `BREAK`'s correctness attribute, which leads to the attribute definition

```
jump_statement.correct = BREAK.correct
```

When is the "break" statement correct, though ? In case there is no syntax error (which would cause an error message already in the parser), it has to be ensured that the "break" is located inside of a loop. The "break" statement itself does not know this, so we need to pass this information by means of an *inherited* Boolean attribute, say "inloop", which signals whether the jump statement is part of a loop (i.e. it is contained in a FOR, WHILE, or DO compound statement):

```
BREAK.inloop = jump_statement.inloop
```

Now, we can easily give the attribute definition for the correctness of "BREAK"

```
BREAK.correct = BREAK.inloop
```

which makes our attribute definitions complete.

Any useful attribute grammar for semantic analysis needs both synthesized and inherited attributes. While YACC only provides some support for synthesized attributes, there are several *attribute grammar processing tools* available, that permit the automatic generation of integrated parsers and attribute evaluators. An example is OX [56], an extension of LEX and YACC. From an attribute grammar specification, OX generates normal LEX and YACC specifications while including C code for attribute evaluation and hence semantic analysis. Amongst others, OX has been used to develop the C frontend for the LANCE compiler system (see section 5.1.7). More details can be found in [93].

As the frontend is almost completely machine independent, it is obviously perfectly retargetable to different processors. In order to generate an intermediate representation for a concrete machine, it merely requires some parameters, like endianess, type bit widths, and memory alignment.

2. Intermediate representation

After successful analysis, the output of the frontend is an *intermediate representation* (IR) of the input source code. The IR can be generated from the parse tree in a post-frontend pass or by means of synthesized attributes when using the above attribute grammar approach.

The main purpose of the IR is to provide a clean and simple data structure that supports IR optimization passes as well as code generation in the backend. Some important IR formats are described in the following.

Three address code. While source programs written in languages like C may show a very complex structure, *three address code* provides a much simpler view of the program, where all statements (except for function calls with multiple arguments) have at most three operands: up to two arguments and one result. Any source program can be lowered down to three address code by inserting *auxiliary variables* or *temporaries*, that store intermediate results of complex computations. As an example, consider the C assignment

```
x = a + b - c * d
```

which translates to three address code as follows:

```
t1 = a + b
t2 = c * d
x  = t1 - t2
```

Three address code is convenient for implementing data flow analysis and IR optimization tools. However, for certain purposes it is advantageous to have a more high-level IR that retains control structures like loops and conditionals, because otherwise these need to be reconstructed from the three address code. A high-level IR facilitates loop transformations and control code optimizations. Muchnik [54] even proposes three IR formats at different abstraction levels. The description of the SUIF and LANCE compiler systems in chapter 5 provides examples for several IR formats.

Control/data flow graphs. While three address code is essentially just another textual representation of the source code, *flow graphs* provide more explicit information about the program semantics and hence open more optimization opportunities. We can certainly generate machine code directly from three address code with a statement by statement translation scheme, but the result would be very poor in terms of code quality.

First of all, the control flow needs to be analyzed. Normally, this is done on a function-by-function basis. The IR of a function is split into *basic blocks.* A basic block

$$B = (s_1, \dots, s_n)$$

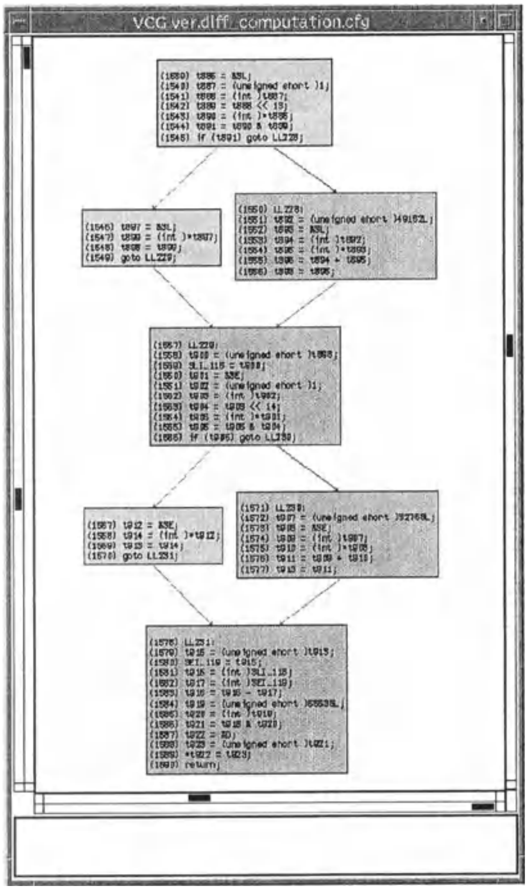
is a sequence of IR statements of maximum length that meets two conditions:

- B can only be entered at statement s_1, and
- B can only be left at statement s_n.

Thus, if statement s_1 is executed, then it is guaranteed that all other statements in B are executed as well. This property is important, since it allows to partially rearrange (e.g. during code selection or scheduling) the computations in B without causing undesired side effects. Identifying the basic blocks in a three address code IR of a function can be easily done by looking for statements that have an impact on control flow, i.e. labels, as well as goto and return statements.

The *control flow graph* (CFG) is a data structure that visualizes all possible flows of control between the basic blocks of a function. For some function F, the CFG is a directed graph $G_F = (V, E)$, where each node $b_i \in V$ represents one of F's basic blocks, and each edge $e = (b_i, b_j) \in E \subseteq V \times V$ represents the fact that block b_j might be executed directly after block b_i. The CFG can be constructed immediately once the basic blocks have been identified. Fig. 3.4 shows a CFG for a sample function visualized by means of the VCG graph display tool [57]. It contains 7 basic blocks, each of which is represented in the form of three address code. Nodes with two outgoing edges represent blocks that end with a conditional jump, while blocks with a fanout of one end with a "goto" or just "fall through" into the next block. Finally, blocks without outgoing edges have a "return" statement at the end. In case the CFG is not fully connected, there is *unreachable code*, which can be eliminated without changing the program behavior.

Within the scope of a single basic block, it is not useful to introduce control flow dependencies, since this would restrict optimization opportunities. Instead, here we are more interested in the *data dependencies*



```
(1539) t886 = &SL;
(1540) t887 = (unsigned short )1;
(1541) t888 = (int )t887;
(1542) t889 = t888 << 13;
(1543) t890 = (int )*t886;
(1544) t891 = t890 & t889;
(1545) if (t891) goto LL228;
```

```
(1546) t897 = &SL;
(1547) t899 = (int )*t897;
(1548) t898 = t899;
(1549) goto LL229;
```

```
(1550) LL228:
(1551) t892 = (unsigned short )49152L;
(1552) t893 = &SL;
(1553) t894 = (int )t892;
(1554) t895 = (int )*t893;
(1555) t896 = t894 + t895;
(1556) t898 = t896;
```

```
(1557) LL229:
(1558) t900 = (unsigned short )t898;
(1559) SL1_118 = t900;
(1560) t901 = &SE;
(1561) t902 = (unsigned short )1;
(1562) t903 = (int )t902;
(1563) t904 = t903 << 14;
(1564) t905 = (int )*t901;
(1565) t906 = t905 & t904;
(1566) if (t906) goto LL230;
```

```
(1567) t912 = &SE;
(1568) t914 = (int )*t912;
(1569) t913 = t914;
(1570) goto LL231;
```

```
(1571) LL230:
(1572) t907 = (unsigned short )32768L;
(1573) t908 = &SE;
(1574) t909 = (int )t907;
(1575) t910 = (int )*t908;
(1576) t911 = t909 + t910;
(1577) t913 = t911;
```

```
(1578) LL231:
(1579) t915 = (unsigned short )t913;
(1580) SE1_119 = t915;
(1581) t916 = (int )SL1_118;
(1582) t917 = (int )SE1_119;
(1583) t918 = t916 - t917;
(1584) t919 = (unsigned short )65535L;
(1585) t920 = (int )t919;
(1586) t921 = t918 & t920;
(1587) t922 = &D;
(1588) t923 = (unsigned short )t921;
(1589) *t922 = t923;
(1590) return;
```

Figure 3.4. Sample control flow graph

between the statements, since these have a strong impact on code generation. For some block $B = (s_1, \ldots, s_n)$ we say that statement s_j is data dependent on statement s_i, with $i < j$, if s_i defines a value used in s_j, so that s_i needs to be executed before s_j in the machine code.

Data flow analysis (DFA) is the process of computing the data dependencies. Local DFA (within the scope of basic blocks) is relatively easy if we neglect load/store dependencies and function calls with potential side effects and just focus on the local variables. Then the relationship between value definitions and uses can be determined on the basis of symbol names by tracing back each use to its last definition in the current block. However, we must conservatively assume that each store and each function call potentially manipulate all local variables whose

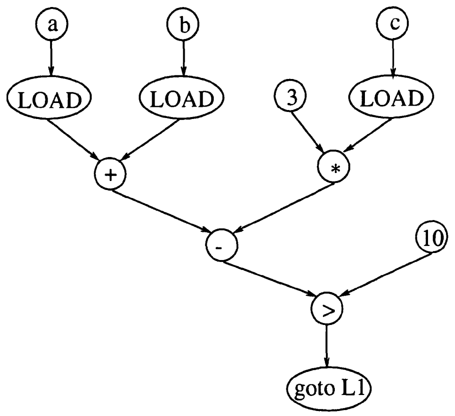


Figure 3.5. DFG representation of a basic block

address is being computed (e.g. by the C operator "&"). Otherwise, a more complex *alias* and *interprocedural* analysis is required.

The result of local DFA is a *data flow graph* (DFG). A DFG for a basic block B is a directed acyclic graph $G_B = (V, E)$, where each node $v \in V$ represents

1 a primary input (variable or constant), or

2 an operation, or

3 a primary output (variable).

An edge $e = (v_i, v_j) \in E \subset V \times V$ represents the fact that the value defined by v_i is used by v_j. Note that in the DFG model all temporary variables from the three address code are transformed into graph edges. Fig. 3.5 shows an example DFG for the following basic block (a, b, and c are assumed to be global variables, read via LOAD operations).

```
t1 = a + b;
t2 = 3 * c;
t3 = t1 - t2;
t4 = t3 > 10;
if (t4) goto L1;
```

In the special case that the DFG is connected and no node has a fanout larger than one, we call it a *data flow tree* (DFT). DFTs are free of *common subexpressions*, and they are the basic data structures for many code selection techniques.

If the CFG and DFG data structures are mixed, so that each CFG node represents a DFG instead of a basic block, the graph is called a *control/data flow graph* (CDFG). Note that the use of the CDFG data structure is not necessarily limited to the IR level. It is frequently used also for optimization at the assembly level in the backend. In this case, the DFG nodes represent concrete machine instructions instead of abstract machine independent operations.

Global data flow analysis. The local data flow analysis (DFA) mentioned above is insufficient for many purposes, particularly for several important IR optimizations. Global DFA determines the data dependencies within the scope of an entire function. Since within the CFG control flow may join at certain blocks, in general there can be multiple definitions reaching a certain use of a variable. In order to check, for instance, whether a variable is guaranteed to be constant at a certain program point, all its definitions, possibly contained in other blocks, have to be investigated. Likewise, all potential uses of some definition have to be known in order to identify redundant code.

One way to perform global DFA is to embed the local DFA into an iterative *work list* algorithm that considers all basic blocks. At the beginning, all blocks are inserted into the work list. For each block B, DFA is performed for each control path into B found in the CFG. In case of a cyclic CFG (containing loops), in general multiple DFA iterations over the same block are required. Whenever a change in the previous data flow information for some block has been detected, its CFG successors are inserted into the work list again in order to propagate the new information. The process continues until the work list is empty and a fix point has been reached. For more details and alternative methods for global DFA, see e.g. [54, 58]

IR transformations and optimizations. There are many possible compiler passes that transform the IR in a machine independent way. Some transformations (which we call optimizations) result in shorter or faster IR code, while others just rewrite the IR in order to generate optimization opportunities for other passes or the backend. In the following we mention some frequently used IR transformations.

Function inlining: Replacement of function calls by copies of the bodies of the called functions. This reduces the calling overhead and permits better optimization opportunities for further passes. However, code size may increase significantly.

Static single assignment: In case the IR is in a form such that each variable statically has a unique definition, it is said to be in static single assignment (SSA) form. SSA form simplifies data flow analysis and may also be beneficial for reducing spill code during register allocation.

Loop unrolling: Loops can present an obstacle to instruction schedulers for instruction-level parallel processors. Partially duplicating, or unrolling, the loop body can help, since it increases the basic block size of loop bodies and exhibits more parallelism.

Loop invariant code motion: Moving loop invariant computations outside of the loop. This can result in a significant performance gain for loop intensive applications.

Induction variable elimination: Given some loop counter variable i running from i_1 to i_n during loop execution, any variable j that shows a linear dependence on i of the form $j = k \cdot i + c$, for constant k and c, is called an induction variable. These frequently arise in the IR from scaled indices in array accesses. The potentially expensive multiplication can be eliminated by initializing j with $k \cdot i_1 + c$ and incrementing j by k in each iteration (assuming a step width of one for variable i).

Constant folding: Compile-time constant expressions can be replaced by the computed values. Care has to be taken to avoid undesired side effects due to finite word length effects or arithmetic exceptions.

Constant propagation: Variables known to have a constant value at a certain program point can be replaced by that constant. This might give additional opportunities for constant folding.

Copy propagation: A copy operation is an assignment of the form "$a = b$" for two variables a and b. Subsequent uses of a can thus be replaced by uses of b, provided that a cannot get redefined in between. Dead code elimination may later remove the copy operation.

Common subexpression elimination: Identical computations at different program points can be replaced by a single computation whose result is stored in a temporary variable, so that it can be reused. Care has to be taken for machines with few registers, since aggressive common subexpression elimination can lead to a huge amount of spill code.

Dead code elimination: Computations whose results are never needed (and which do not show side effects) can be safely eliminated from the program.

Reassociation: Reordering of arithmetic expressions by application of algebraic transformations, so as to enable better opportunities for constant folding.

Jump optimization: Removal of jump chains, redundant jumps, minimization of unconditional jumps, and unreachable code elimination.

Clearly, many of the above transformations have an impact on each other, and usually multiple iterations are required to achieve the best result. With respect to retargetability, there is usually some degree of mutual dependence with the backend: Even though the IR itself is mostly machine independent, different machines may benefit from different IR transformation procedures.

3. Backend

3.1. Code selection

Once the compiler enters the backend, it starts dealing with machine specific aspects. Code selection is typically the first backend phase and maps machine independent IR statements and operations into machine specific processor instructions. Several implementation choices are described in the following.

Statement based code selection. The simplest way to perform code selection is to start with three address IR code and to translate each IR statement into equivalent assembly instructions step by step. This is particularly easy to implement due to the simple structure of three address code. However, this approach is not always likely to produce optimal results, since it might be possible to cover multiple IR statements with a single instruction. A typical example is the *multiply-accumulate* (MAC) instruction in DSPs, which performs a single-cycle multiply and add operation in a chained fashion. On CISC processors, complex memory addressing modes, such as "base plus offset" can be exploited to implement multiple IR statements with a single instruction.

Another problem arises if the target machine has special purpose registers. Since the machine instructions communicate via registers, it has to be ensured that there is not too much overhead due to register-to-register data move operations, but the statement based approach cannot take this into account during code selection. For RISC targets with homogeneous register files, however, statement based code selection can give satisfactory results, since there are hardly complex instructions, and late improvement of the selected code is still possible by means of *peephole optimization.*

Tree based code selection. In case of target machines with complex instructions and/or special purpose registers, a tree based approach to code selection is more favorable, which works on data flow trees (DFTs) as introduced in section 3.2. As exemplified in fig. 3.5, a DFT generally represents a complex computation, that covers multiple IR statements at a time. DFTs can be constructed either directly from three address code, or by first constructing a data flow graph (DFG), followed by splitting the DFG at its common subexpressions, which results in a forest of DFTs.

Today's most common approach to code selection is *tree parsing*, which can be efficiently implemented by tree pattern matching and dynamic programming [59, 60, 61]. The target instruction set is modeled as a *tree grammar*

$$G = (T, N, R, S, w)$$

where T is a set of *terminals*, N is a set of *nonterminals*, R is a set of *rules*, $S \in N$ is the *start symbol*, and $w : R \rightarrow \mathbb{R}$ is a *cost metric* for rules, which may reflect optimization goals like code size, performance, or power consumption.

Intuitively speaking, the nonterminals in N are mostly used to model hardware resources that can store data (registers, memories), while the terminal set T is used to represent operators and constants in a DFT. The grammar rules in R can be used to *derive* DFTs from the start symbol S. Like for usual string grammars, a derivation step in G means to replace the occurrence of a nonterminal $n \in N$ in a tree by another tree T, which is possible if the rule $n \rightarrow T$ is in R. The *tree language* $L(G)$ generated by grammar G is equal to the set of all possible DFTs.

The rules $p \in R$ are generally used to model the behavior of an instruction in the form of a small *tree pattern*. For instance, for an ADD instruction that computes the sum of two register contents and assigns the result to another register, the rule

$$reg \rightarrow PLUS(reg, reg)$$

would be used, where $reg \in N$ and $PLUS \in T$. Concerning the language *generated* by grammar G, this rule allows to *derive* a subtree with a root labeled $PLUS$ and two subtrees from reg. Conversely, if we talk about tree *parsing*, the rule can be used to *reduce* a subtree rooted by $PLUS$ and two subtrees (that have already been reduced to reg) to nonterminal reg. In any case, *using* the rule means to instantiate an ADD instruction during code selection.

Since grammar rules essentially model instructions, the task of code selection for a DFT T is equivalent to finding a derivation in G for T

from the start symbol S. Since the rules in R are weighted by function w, there are *optimal* derivations, i.e., derivations such that the sum over the weights $w(p)$ over all instances of rules p used in the derivation is minimal.

Using tree parsing, an optimal derivation for a DFT T can be found as follows: In a *bottom-up* traversal, all nodes x in T are labeled with a set of triples (n, p, c), where $n \in N$, $p \in R$, and $c \in \mathbb{R}$. These triples represent the fact that node x can be reduced to nonterminal n at a total cost of c with rule p. The rule p implicitly determines the nonterminals, which the subtrees of node x (if any) must be reduced to in order to make p applicable for x. In general, multiple triples are annotated to x, which represent alternative derivations.

When the root of T has been reached, all alternative derivations potentially leading to an optimum are known. One optimal derivation is now explicitly constructed in a *top-down* traversal of T. For the root node, the triple (S, p, c) is selected (S is the start symbol), for which c is minimal over all alternative triples at the root. In turn, rule p now implies the optimal derivations for the subtrees at the next lower level in T, since the nonterminals which they must be reduced to are identical to the nonterminals on the right hand side of p. This traversal is recursively continued until the leaves of T have been reached and the derivation has been completely emitted. An example for this process is given in fig. 3.6.

Though tree parsing generally does not produce globally optimal solutions, it shows a number of important advantages:

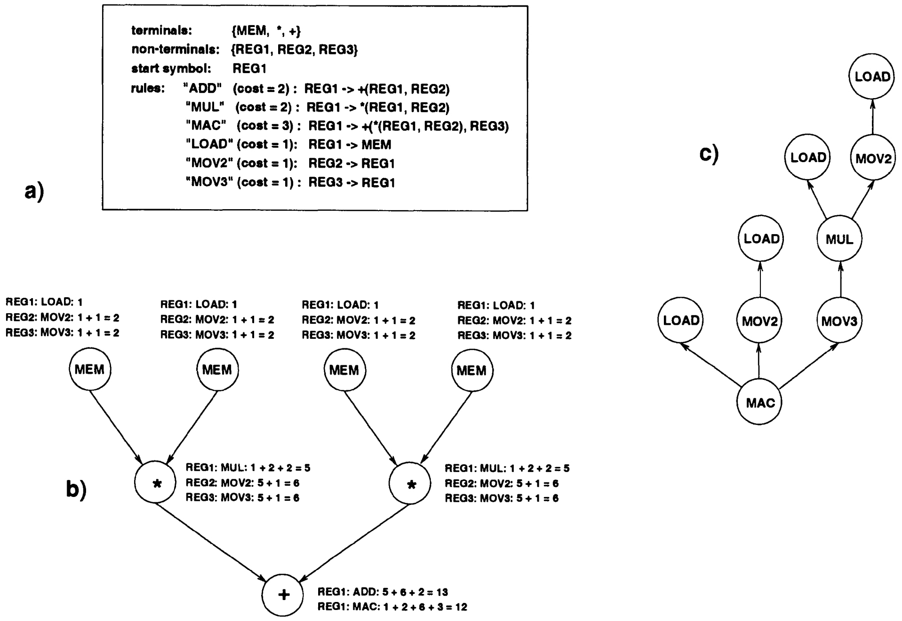
1 It requires only *linear time* in the DFT size.

2 It selects an *optimal set* of instruction pattern instances for each single DFT.

3 It allows for modeling *complex instructions* and special purpose registers.

These characteristics make tree parsing a very popular technique for code selection. In chapter 5 we will see that different variants have actually been used in many retargetable compilers. Moreover, we will present some tools for automatic generation of tree parsing code selectors from grammar specifications, which make tree parsing easily retargetable.

Graph based code selection. Code selection can result in better solutions if it is generalized towards data flow graphs (DFGs) with common subexpressions (CSEs). The reason is that splitting DFGs at the CSEs, so as to obtain DFTs, can impose unfavorable restrictions. An



a)

```
terminals:      {MEM, *, +}
non-terminals:  {REG1, REG2, REG3}
start symbol:   REG1
rules:   "ADD"  (cost = 2) :  REG1 -> +(REG1, REG2)
         "MUL"  (cost = 2) :  REG1 -> *(REG1, REG2)
         "MAC"  (cost = 3) :  REG1 -> +(*(REG1, REG2), REG3)
         "LOAD" (cost = 1):   REG1 -> MEM
         "MOV2" (cost = 1):   REG2 -> REG1
         "MOV3" (cost = 1) :  REG3 -> REG1
```

Figure 3.6. Example for tree parsing: a) Tree grammar specification, b) DFT with annotated nonterminal/rule/cost triples. There are two alternatives, ADD and MAC, for the root. MAC is selected, because it covers two operations at a time and therefore results in a cheaper derivation with a cost of 12 instead of 13 for ADD. c) Optimum derivation tree

example is given in fig. 3.7. Part a) shows a sample DFG with the multiply node being a CSE with two uses. In a tree based approach, the DFG would be split into two DFTs, where the multiply DFT writes its result into some storage resource (shaded box in part b), and the second DFT with the two add operations reads that value twice. Suppose, the target machine offers add, multiply, and MAC instructions with equal costs. Then, three instructions (one multiply, two adds) are required to implement the DFTs from fig. 3.7 b). However, as shown in fig. 3.7 c), there is a cheaper solution that results from duplicating the CSE and using two MACs for covering.

Another aspect of DFG based code selection is that it can lead to a better exploitation of special purpose registers with fewer data move instructions between the DFTs. While graph based code selection can result in significantly better code for irregular target architectures like DSPs, the computational complexity is unfortunately much higher than in tree parsing. Generally, optimal DFG based code selection is an NP-hard problem [58] and hence (most likely) requires exponential worst-case runtime. However, sometimes heuristic search strategies can be

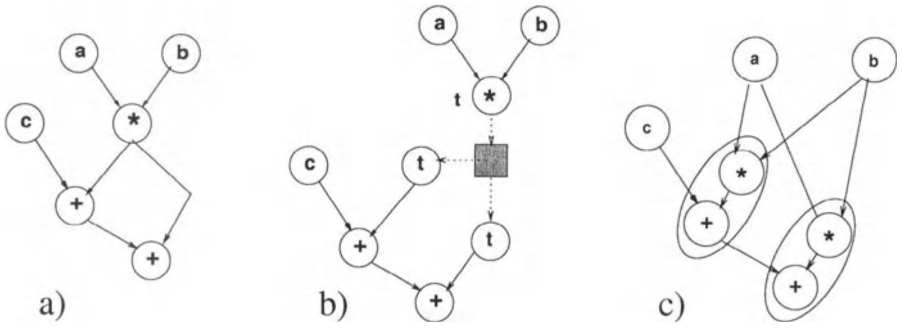


Figure 3.7. Code selection for DFGs: a) sample DFG, b) conventional splitting into two DFTs, c) duplicating the CSE to exploit two MAC instructions

applied to guarantee a reasonable amount of runtime while still achieving good results. Examples are given in [189, 62]. For a class of regular target processors, even exact solutions can be computed efficiently by a generalization of the tree parsing approach [63].

Global code selection. Naturally, the best results can be achieved when performing code selection in a global context, i.e. within the scope of an entire procedure or function. As illustrated in fig. 3.7, for instance, it can be favorable to duplicate CSEs. However, some CSE used in a DFG could also be defined in a different basic block, which is not visible when performing local code selection. Hence, there is a tight dependency between code selection and global CSE elimination at the IR level.

Special care has to be taken in case of code selection across basic block boundaries. As has been exemplified in the previous sections, machine instructions may be selected that span multiple IR statements. Such a selection is generally only valid within the scope of basic blocks. In case of an instruction covering statements from different blocks, undesired side effects may occur, since the dynamic control flow between blocks is generally not known at compile time.

3.2. Scheduling

Code selection maps a program into assembly instructions but it does not assign concrete execution times to instructions. This is the task of *instruction scheduling*. However, the term scheduling is used in different meanings, and many compilers in fact employ multiple scheduling passes in the backend. We can coarsely distinguish scheduling techniques that work on *sequential* code, while others deal with exploitation of instruction level *parallelism*. Likewise, there is a distinction between *local* schedulers (working at the basic block level) and *global* schedulers operating on a loop or even an entire function.

The most common data structure for local scheduling is a *dependency graph* (DG), i.e. an edge-weighted directed acyclic graph $G = (V, E, w)$. Each member of the node set V represents an instance of some machine instruction. The edges $e = (v_i, v_j) \in E \subseteq V \times V$ denote scheduling dependencies, which exist in three forms:

Data dependence: v_i defines a value used by v_j. Therefore, v_i has to be scheduled before v_j. Data dependence edges correspond to the DFG edges described in section 3.2.

Anti-dependence: v_i writes to some storage resource R, and v_j reads from R but uses a value defined by a different instruction. Therefore, v_j must not be scheduled after v_i. In case the target processor permits writing and reading a register within a single cycle, v_i and v_j may be scheduled in the same cycle.

Output dependence: v_i and v_j write to the same storage resource R. Then, the schedule must preserve the original ordering of v_i and v_j imposed by the IR code.

A scheduler assigns a start time $t(v)$ to each node $v \in V$. Generally, the goal is to construct a schedule with minimum total execution time. Any DG edge $e = (v_i, v_j)$ has a weight $w(e)$ that denotes the minimum start time difference between $t(v_j) - t(v_i)$ in a valid schedule. Thus the DG edges, together with the available amount of processor resources, impose the constraints for the scheduler. The instruction I represented by v_i may take multiple cycles to generate its result, so $t(v_j) - t(v_i)$ may be an arbitrary integer number.

It might be the case that new instructions can be issued before I has been completed. In this case, we say that I has *delay slots*, which should be exploited during scheduling by filling them with useful instructions instead of no-operations (NOPs). For instance, RISCs frequently have jump instructions with delay slots, which arise from instruction pipelining. Jump delay slots can be represented in the DG by edges with a negative weight.

Sequential scheduling:. In case of a target processor without instruction level parallelism, the task of the scheduler is to define a linear ordering of instructions, based on a topological sort of the DG. As the resource constrained scheduling problem is generally NP-hard [192], a number of heuristics are in use [114].

Perhaps the most popular one is *list scheduling*, an effective class of scheduling algorithms with a worst case execution time quadratic in the number of DG nodes. It is based on the notion of the *ready set*. This set denotes all DG nodes ready to be scheduled at a certain point of time, since all its DG predecessors have already finished execution. At the beginning, the ready set consists of all primary DG inputs, and it changes dynamically when removing DG nodes that have already been scheduled. In each step, the list scheduler heuristically picks one of the ready set members and appends it to the partial (initially empty) schedule constructed so far. The ready set is updated accordingly, and the process is iterated until all DG nodes have been scheduled. The schedule quality critically depends on the heuristic for selecting one of the ready nodes.

Another aspect of sequential scheduling is that the register allocation phase described in section 3.3.3 requires to know the *live range* of values in a program, which depends on the linear order of instructions. Even though scheduling is not necessarily to be fully performed before register allocation, at least some form of sequential scheduling has to be done before register allocation can take place. From a register allocation viewpoint, this scheduling should be performed in such a way that value life times are minimized.

Code compaction:. Local scheduling is frequently referred to as *code compaction*, if the target processor is a VLIW-like machine showing instruction level parallelism. Furthermore, code compaction frequently assumes that *all* required instructions, including spill code (section 3.3.3) and address code (section 3.3.4), have already been generated, making code compaction typically a late compiler pass.

Essentially the same techniques as for sequential scheduling can be used, but the scheduler is also responsible for exploiting the parallel functional units (FUs), so as to achieve the highest performance. In case of alternative FUs, capable of implementing the same set of instructions, FU assignment is a non-trivial task. A simple, yet effective, heuristic to optimize FU utilization in this case is *version shuffling* [114].

Software pipelining:. A major compiler problem with highly parallel VLIW-like processors is to keep the large number of functional units (FUs) busy, even though typical application programs do not show a high degree of potential parallelism. Software pipelining [64] is a very effective assembly-level loop scheduling technique that maximizes FU utilization for loop bodies. As an example, we consider the TI C6x, a VLIW DSP with 8 parallel FUs, whose coarse data path is shown in fig. 3.8.

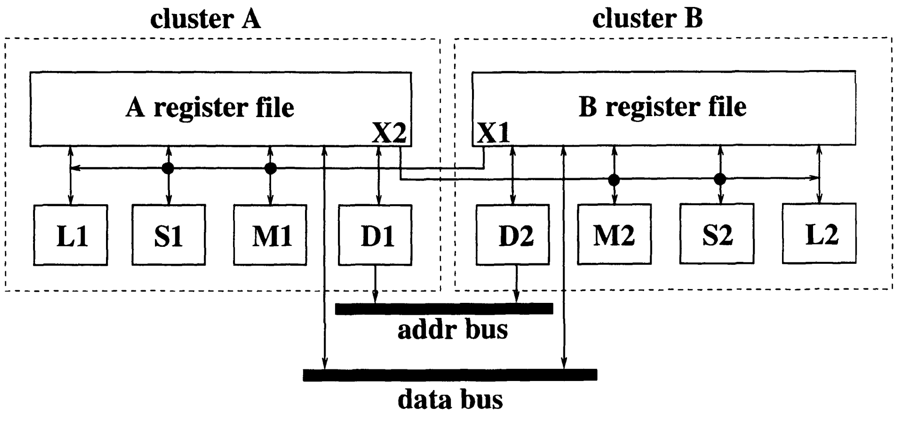


Figure 3.8. Data path of a TI C6x VLIW processor

Consider the following C function for dot product computation. Here, we have unrolled the loop body once to exhibit some intra-loop parallelism.

```
int dotp(short a[], short b[])
{ int sum0, sum1, i;

  sum0 = sum1 = 0;
  for (i = 0; i < 100; i += 2)
  { sum0 += a[i] * b[i];
    sum1 += a[i+1] * b[i+1]; }

  return sum0 + sum1;
}
```

A C6x assembly code for the dot product loop is shown below. Despite the unrolling the FU utilization is still low, as only few operations are executed in parallel (denoted by "||"). However, the code is in fact performance-optimal for the given loop structure. It requires 9 cycles per iteration, thus (for 50 iterations) a total of 450 cycles.

```
         mnemonic   FU     operands

           LDH      .D2    *++B4(4),B6 // load 16 bit
||         LDH      .D1    *++A4(4),A5 // load 16 bit
           LDH      .D2    *++B4(2),B5  // load 16 bit
||         LDH      .D1    *++A4(2),A6  // load 16 bit
           SUB      .L2    B0,1,B0      // decrement loop counter
```

```
[ B0]   B       .S1     L2          // branch if zero
        NOP             1           // empty delay slot
        MPY     .M1X    B6,A5,A5    // multiply
        MPY     .M1X    B5,A6,A6    // multiply
        NOP             1           // empty delay slot
        ADD     .L1     A6,A0,A0    // compute sum0
||      ADD     .S1     A5,A3,A3    // compute sum1
```

The weak FU utilization can be clearly seen when illustrating the schedule by means of a simple *reservation table* that shows the FU occupation by the different instructions over the time T (fig. 3.9).

FU/T	0	1	2	3	4	5	6	7	8
D1	LDH	LDH							
D2	LDH	LDH							
M1						MPY	MPY		
M2									
L1									ADD
L2			SUB						
S1				B					ADD
S2									

Figure 3.9. Reservation table for dot product loop body

Software pipelining optimizes FU utilization by initiating new loop iterations before previous ones have been completed. In a software pipelined loop, multiple iterations from the original source code are simultaneously active, each in a different stage of completion.

On the TI C6x, for instance, a LOAD instruction has a delay of 5 instruction cycles, a branch (B) takes 6 cycles, and a MPY needs 2 cycles before its result is valid. However, new instructions may be issued earlier, since the FU executing an instruction is (virtually) occupied only for a single cycle. Exploiting these delay slots, we can issue two LOADs in each cycle, and a MPY each time a LOAD pair has been finished 5 cycles later. Then, we get a reservation table as shown in fig. 3.10.

As can be seen, the FU utilization gets higher, and from cycles 7 and 8 onwards, once the software pipeline is in a "steady state", the schedule repeats periodically every two cycles. The code from cycles 0 to 6 needs to be executed once and is called the *prologue.* Likewise there is an *epilogue* (not shown) for finally cleaning up the pipeline. The code from cycles 7 and 8, however, becomes the new loop kernel, which thus needs only two cycles per iteration. Including prologue and epilogue, the dot product computation time is reduced to 106 cycles only, indeed a significant improvement over the above version.

FU/T	0	1	2	3	4	5	6	7	8
D1	LDH	LDH	LDH	LDH	LDH	LDH	LDH	LDH	LDH
D2	LDH	LDH	LDH	LDH	LDH	LDH	LDH	LDH	LDH
M1							MPY		MPY
M2						MPY		MPY	
L1									ADD
L2			SUB		SUB		SUB	ADD	SUB
S1				B		B		B	
S2									

FU/T	9	10	11	12	13	14	15	16	17
D1	LDH	LDH	LDH	LDH	LDH	LDH	LDH	LDH	LDH
D2	LDH	LDH	LDH	LDH	LDH	LDH	LDH	LDH	LDH
M1		MPY		MPY		MPY		MPY	
M2	MPY		MPY		MPY		MPY		MPY
L1		ADD		ADD		ADD		ADD	
L2	ADD	SUB	ADD	SUB	ADD	SUB	ADD	SUB	ADD
S1	B		B		B		B		B
S2									

Figure 3.10. Reservation table for software pipelined dot product loop body

The disadvantage of software pipelining is a potential increase in code size, due to the prologue and epilogue code. Additionally, there is a possible need to perform a run-time check whether there are enough loop iterations to make the software pipeline applicable, or whether a non-pipelined code version has to be used.

Global scheduling:. Local and loop scheduling techniques like software pipelining do not help much in case of control-dominated code, where the control flow graph (CFG) consists of a large number of small basic blocks. Small blocks are very unlikely to exhibit enough parallelism to ensure good resource utilization for a VLIW processor. Global scheduling techniques, such as *Trace Scheduling* [65], aim at eliminating this bottleneck by allowing instructions to be moved over basic block boundaries.

This works roughly as follows: By means of profiling, one can identify critical paths in the CFG. The sequence of basic blocks along such a path is called a *trace.* Neglecting the block boundaries, Trace Scheduling compacts a trace by means of a local scheduling algorithm, as if it were a single large basic block. Generally, this will give a much denser schedule for the trace than what would result from locally scheduling the blocks one by one, so that the critical path is shortened. However, a lot of undesired side effects may be incurred by moving instructions out of their original blocks. Therefore, *compensation code* has to be inserted (e.g. by duplicating instructions) that "repairs" the side effects. This is also the major disadvantage of Trace Scheduling, which tends to significantly increase code size. An alternative global scheduling technique, *Percolation Scheduling* [66], reduces the code size overhead by avoiding code duplication whenever possible.

3.3. Register allocation

For sake of simplicity, code selection and early scheduling frequently abstract from the physical register resources of the target machine. Instead, it is assumed that the target has an infinite number of *virtual* or *symbolic* registers. Each time the code selector needs a new storage resource, it generously allocates a new unique virtual register. The task of the register allocator is to finally assign virtual to physical registers in such a way that the limits of the target machine are met.

The number of virtual registers used in early compiler phases may well exceed the available number of physical registers. In case there are insufficient physical registers, the register allocator must generate *spill code.* Spilling a register means temporarily storing its content to memory, and reloading it when it is required again. Naturally, the optimization goal of the register allocator is to minimize the amount of spill code.

In most cases, register allocation is based on the notion of *live ranges.* The live range of a virtual register v starts right after the instruction I_D that defines it and extends over all instructions lying on a control flow path leading to some instruction I_U that uses v. Thus, *liveness* at a certain program point means that the register contents are still required and should not be overwritten before their last use, after which the virtual register eventually "dies".

The following two approaches are generally used for register allocation. They differ in complexity and efficacy, and the best choice also strongly depends on the concrete target machine.

Local register allocation. Local register allocation works within the scope of a single basic block. This approach is computationally efficient, but shows a limited optimization effect, since many loads and stores are generally required between the basic blocks. However, if the target machine has only very few registers, then local register allocation is a reasonable approach, since it is unlikely that values can be kept in registers for a long time without spilling.

The central data structure in most register allocation techniques is the *interference graph.* This is an undirected graph $G = (V, E)$, where each node $v \in V$ represents a virtual register. The set E contains an edge $e = \{v_i, v_j\}$, whenever the live ranges of v_i and v_j intersect. Hence, edge e indicates that v_i and v_j should be mapped to different physical registers. Otherwise, spill code has to be inserted.

As defined in section 3.2, a basic block is a straight line sequence of statements. At the time of register allocation, these statements are normally machine instructions instead of IR statements. In any case, the basic block property implies that all live ranges of virtual registers are *intervals* of program points (at least after having performed SSA transformation). For instance, consider the following statement sequence annotated with liveness information.

```
(1)  b = 1;            // live in: -
(2)  c = 2;            // live in: b
(3)  a = b + c;        // live in: b, c
(4)  d = a * 2;        // live in: b, a
(5)  e = b / 3;        // live in: b, d
(6)  return e - d;     // live in: d, e
```

Computing the live ranges leads to the following intervals:

```
       1 2 3 4 5 6
  a:  [      x     ]
  b:  [  x x x x   ]
  c:  [    x       ]
  d:  [        x x ]
  e:  [          x ]
```

If the interference graph is such that all live ranges are intervals, then the so-called *left edge algorithm* [67] can be used for optimal register allocation in approximately linear time. Given a set $\{R_1, \ldots, R_k\}$ of physical registers, the algorithm moves a scan line from left to right over the intervals and assigns a free register R_i with minimum index i to each new live range. Likewise, it frees the register allocated for each live range that ends at the current scan line position. For the above example, two registers are sufficient.

If there is a solution with at most k registers, the algorithm will find it. Otherwise, if the number of live registers exceeds k, one register needs to be selected for spilling, so as to break its live range into sub-intervals. In this case it is favorable to select that register with the maximum forward distance to its next use, since this will minimize the number of further conflicts.

Global register allocation. If the target machine has a large number of registers, such as a RISC processor, then register allocation should be generalized towards entire functions. Like in the case of local allocation, we can build the interference graph to represent the live range conflicts. However, in contrast to the local approach, the live ranges are generally no intervals, and the left edge algorithm cannot be applied.

```
x = f(0); // live in: -
if (x<3)  // live in: x
{
  a1 = f(x);          // live in: x
  b1 = f(a1);         // live in: a1
  c1 = f(a1*b1);      // live in: a1, b1
  x  = a1 + b1 * c1;  // live in: a1, b1, c1
}
else c1 = 0;          // live in: x

if (x<2)              // live in: c1, x
{
  c2 = f(c1);         // live in: c1
  a2 = f(c1);         // live in: c1, c2
  b2 = f(a2*a2);      // live in: a2, c2
  x  = c2 + f(a2*b2); // live in: a2, b2, c2
}
return x; // live in: x
```

Figure 3.11. Multi-block code and variable liveness

Instead, global register allocation amounts to a general *graph coloring* problem. Given the interference graph $G = (V, E)$ and k physical registers $\{R_1, \ldots, R_k\}$, a "color" (i.e. a number in $\{1, \ldots, k\}$) needs to be assigned to each $v \in V$, such that different colors are assigned to all node pairs (v_i, v_j) for which $\{v_i, v_j\} \in E$. An example is given in figs. 3.11 and 3.12. In case that G is not k-colorable, no register allocation without spill code exists.

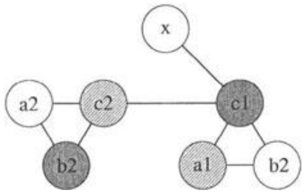


Figure 3.12. Interference graph and its 3-coloring for the code from fig. 3.11

Like many other code optimization problems, general graph coloring is NP-hard [192]. However, there are a number of effective heuristics. One of the most popular ones is Brigg's algorithm [68]. Starting with an interference graph $G = (V, E)$, it is based on the observation that any node $v \in V$ with less than k neighbors is non-critical w.r.t. coloring:

If the graph G' that results from G by removing v and all its incident edges is k-colorable, then also G is k-colorable, since a valid color for v can always be found once the coloring of G' is known. Brigg's algorithm iteratively removes such nodes, pushes them onto a stack, and tries to color the remaining graph G'. Each time G' contains only nodes with degree greater or equal to k, one node is selected for spilling and is removed and pushed onto the stack as well.

When G' is empty, the original graph is reconstructed in reverse order by iteratively popping nodes from the stack. In case a spill candidate is popped, colorability is not guaranteed, and spill and reload instructions are generated on demand. Since the addition of spill code modifies the live ranges and their interference, the algorithm has to be repeated until a fix point without further spill requirements has been obtained.

There are a number of refinements to this basic algorithm, including *coalescing* to eliminate redundant move instructions and handling of *precolored* nodes that a priori represent physical registers. Brigg's algorithm *optimistically* assumes that all virtual registers might be mapped to physical registers, and generates spill code only in case this assumption is violated. Conversely, there are also *pessimistic* approaches [69], which start from the assumption that all virtual registers must be stored in memory, and afterwards aim at keeping as many as possible in registers.

The presented techniques are obviously retargetable w.r.t. the number k of physical registers in a machine, since k is a parameter of the register allocator. However, particularly for irregular architectures, standard techniques like graph coloring should be extended by more sophisticated algorithms, that explicitly take special purpose registers into account.

3.4. Address code optimization

Address code optimization is mainly useful for DSPs with a dedicated *address generation unit* (AGU), as depicted in fig. 3.13. The AGU generally comprises *address registers* (ARs) and *modify registers* (MRs). All indirect addressing in such a DSP takes place via the ARs, and the available memory addressing modes are very limited.

ARs can be updated in parallel to any load or store instruction by adding or subtracting some constant c, provided that c either resides in an MR, or c is contained in a machine specific (and typically small) *auto-increment range* [-r,r]. Due to the instruction level parallelism, such auto-increment operations can be considered to be of zero cost. On the other hand, non-parallel address computations always require extra code. Therefore, the compiler should aim at organizing address computations

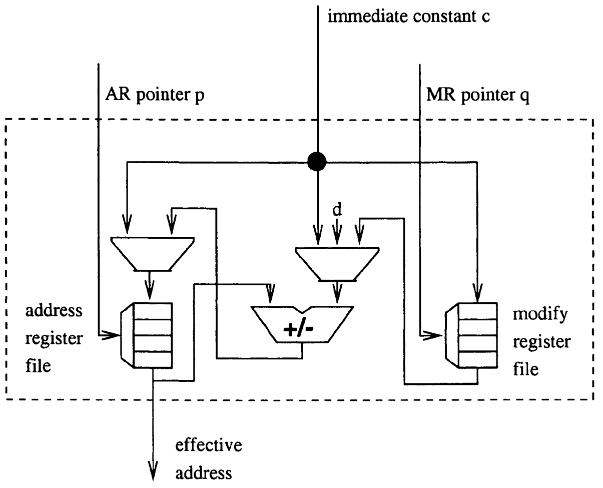


Figure 3.13. Address generation unit (AGU) in DSPs

in such a way, that auto-increment addressing is applicable as often as possible.

Offset assignment. First of all, this optimization can be done for the *scalar variables* (and also spill locations) that reside in the machine's runtime stack. The amount of potentially parallel address computations mainly depends on how well the layout of variables in memory is adapted to the given variable access sequence, which is exemplified in fig. 3.14. Since scalar variables are located at some offset relative to the frame pointer, the optimization problem of determining the best variable layout is commonly known as the *offset assignment* problem.

Offset assignment is an effective optimization, and therefore it has been implemented in several existing compilers. Due to NP-completeness of the problem, exact solutions usually cannot be computed. Triggered by work from Bartley and Liao [70, 115], today there are a large number of good heuristics (see e.g. [93] for a detailed overview), many of which are also capable of handling a wide variety of AGU configurations w.r.t. the number of AR, MRs, and auto-increment ranges.

Array address code optimization. Offset assignment is based on the fact that the compiler can freely arrange the layout of scalar variables. In contrast, array elements have to have a fixed layout w.r.t. each other. Still, the auto-increment capabilities of AGUs can be exploited for address code optimization. The main idea is to minimize the amount of spill code for ARs, that are used for array accesses in loop bodies. As

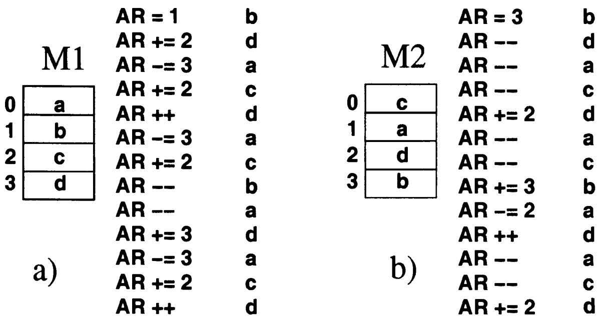


Figure 3.14. Example for scalar address code optimization for an AGU with a single AR and auto-increment range [-1,1]. The variable access sequence is $S = (b,d,a,c,d,a,c,b,a,d,a,c,d)$. a) Memory layout $M1$, where variables are assigned to memory locations 0 ... 3 in alphabetical order. The sequence of address computations needed to generate the memory addresses corresponding to S is given in C-like notation. Only 4 operations are auto-increment/decrement. b) Improved layout $M2$ with 8 auto-increment/decrement address computations.

an example, consider the following loop with 7 accesses to some array A:

```
for (i = 2; i <= N; i++)
{
  A[i+1]  // a1
  A[i]    // a2
  A[i+2]  // a3
  A[i-1]  // a4
  A[i+1]  // a5
  A[i]    // a6
  A[i-2]  // a7
}
```

For the corresponding address computations, any number of ARs from 1 to 7 can potentially be allocated. Assuming an AGU with an auto-increment range of [-1,1], using only a single AR results in poor code, since then most address computations between neighboring array accesses in the loop (e.g. "-3" for $a3 = A[i+2]$ and $a4 = A[i-1]$) cannot be implemented by auto-increment. Conversely, if we used 7 ARs (one for each access), only auto-increment would be required, but potentially too many ARs would be in use to keep them in the physically available resources.

The key idea in array address code optimization is to let array accesses *share* ARs as much as possible (similar to general register allocation, where virtual registers share physical registers), without introducing extra address code inside a loop body. Sharing is possible, whenever the address distance between neighboring array accesses falls into the auto-increment range. For the above example, for instance, only 3 ARs are required, while still using only auto-increment addressing:

```
AR1 = &A[3];
AR2 = &A[2];
AR3 = &A[0];
for (i = 2; i <= N; i++)
{
  *AR1--     // a1
  *AR2--     // a2
  *AR1--     // a3
  *AR2++     // a4
  *AR1++     // a5
  *AR2++     // a6
  *AR3++     // a7
}
```

Array address code optimization is a non-trivial task, especially when inter-iteration constraints must be obeyed. Several heuristics and exact techniques are available, including [122, 127, 71, 145, 72]. Like in the case of offset assignment, different AGU configurations can frequently be handled, making the techniques applicable for retargetable compilation for DSPs.

Address code optimization is typically one of the last compiler passes, since it requires detailed knowledge about all memory accesses. However, it has to be followed by a post-scheduling or compaction pass, that parallelizes address computations newly inserted into the assembly program.

3.5. Phase coupling issues

Due to the huge overall complexity of the compilation problem, a compiler is subdivided into numerous different phases. This is a simple divide-and-conquer methodology, but it implies that all phases have to be executed in a certain order. The non-trivial problem of determining the best sequence of phases is known as the *phase ordering problem.*

Many phases may have an impact on the optimization opportunities of subsequent phases. The IR optimization "constant propagation", for instance, is usually intended to improve the efficacy of "constant folding", which in turn can generate further constants for propagation. Therefore, it is clear that these two optimizations should be applied iteratively until no more optimization is possible.

Particularly in the backend, however, phase ordering is not as simple, because some phases unnecessarily *restrict* the search space for subsequent phases. Moreover, this type of phase interdependence is frequently *cyclic.* We illustrate this problem by some examples:

Code selection and register allocation: Code selection maps the IR into assembly instructions. Typically, this phase also decides which values will reside in virtual registers. Moreover, in case of multiple register files, code selection also decides which destination register file will be chosen for a certain operation, since there is usually an optimum w.r.t. the instruction cost metric. However, only during register allocation it turns out whether the choice was optimal if the required spill code is taken into account, too.

Register allocation and scheduling: Register allocation folds virtual registers into physical registers. Typically, it will reuse a certain physical register for multiple virtual registers. This can introduce unnecessary *false dependencies* (or anti-dependencies) that obstruct the scheduler. On the other hand, sequential scheduling determines the detailed live ranges of virtual registers, and thus clearly has an impact on the results of register allocation.

Scheduling and address code optimization: Address code optimization relies on the detailed memory access sequence produced by a scheduler. The variable layout in memory and the amount of auto-increment operations depend on the access sequence, and it might be that a different, yet valid, alternative schedule would lead to lower addressing costs. Since address code optimization inserts new instructions, it also has an impact on the results of code compaction.

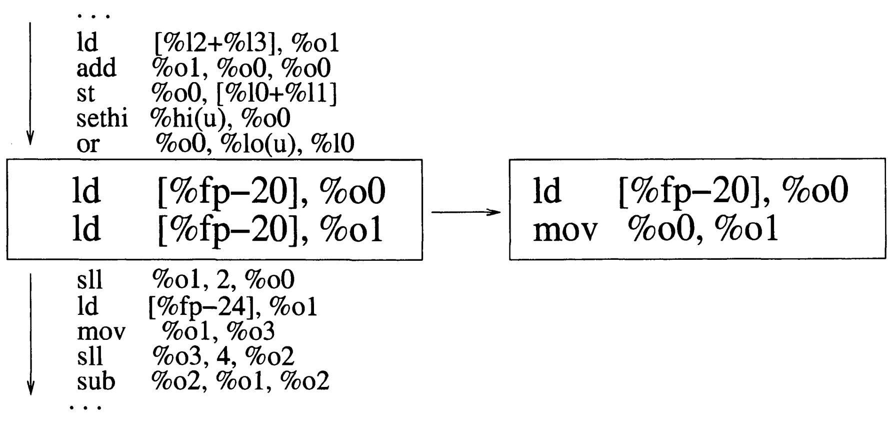
The result of these mutual dependencies is that generally *any particular phase ordering* will produce some overhead in code quality for certain input programs. The phase ordering problem is even more significant for embedded processors with an irregular architecture like in DSPs. Therefore, *phase coupling* approaches are used that lead to a tight phase interaction in order to achieve better code quality.

The simplest way of phase coupling is the *iterative* execution of multiple phases, so as to revise potentially poor decisions based on back annotation from subsequent phases. Other approaches try to *estimate* the impact of some phase on subsequent phases at an early point of time. The highest degree of phase coupling is achieved when multiple phases are actually *combined* into a single one. However, the corresponding algorithms are difficult to design, and this "ideal" phase coupling frequently results in relatively high compilation times.

The case studies in chapter 5 give a number of examples how phase coupling has been implemented in retargetable compilers.

3.6. Peephole optimization

After all compilation phases have finished, it is frequently useful to make one last quick run over the generated code in order to perform some late local improvements. Even though each of the previous optimization passes will have done a good job on its own, it might be that their combination and/or ordering led to some "unbeautiful" instruction sequences, which still could be easily improved.



```
 . . .
ld     [%l2+%l3], %o1
add    %o1, %o0, %o0
st     %o0, [%l0+%l1]
sethi  %hi(u), %o0
or     %o0, %lo(u), %l0

ld     [%fp-20], %o0            ld    [%fp-20], %o0
ld     [%fp-20], %o1    --->    mov   %o0, %o1

sll    %o1, 2, %o0
ld     [%fp-24], %o1
mov    %o1, %o3
sll    %o3, 4, %o2
sub    %o2, %o1, %o2
 . . .
```

Figure 3.15. Example for peephole optimization (SPARC assembly code)

For this purpose, a small window is moved over the assembly code step by step, and only that piece of code currently visible through this "peephole" is considered for optimization (fig. 3.15). Typical peephole optimizations include elimination of redundant loads, and substitution of partial instruction sequences by cheaper ones w.r.t. a given cost metric.

Peephole optimization is frequently implemented in a retargetable fashion by providing the compiler with a set of match/replace rules that describe the candidate instruction sequences and their replacement. An example for this methodology is the retargetable peephole optimizer PO [73]. The more powerful the peephole optimizer, the less effort has to be paid in early code optimization phases. In an extreme case, one can simply generate some poor (yet valid) initial code and leave the whole optimization job to the peephole optimizer. For instance, such an approach is used in Zephyr/VPO (section 5.1.6).

Before we describe existing compiler systems in more detail (chapter 5), the next chapter will summarize the historical development of retargetable compiler research.

Chapter 4

HISTORICAL OVERVIEW

1. Contributions from the compiler community

Contributions from the compiler domain that can be used for designing retargetable compilers, include the following:

- methods for code generation,
- register allocation,
- front end generation, and
- intermediate language design.

A subset of these will be discussed in the next sections of this chapter.

1.1. UNCOL

Generating compilers for new target processors from existing compilers easily has been a goal since many decades.

One of the first proposals consisted of a clear separation between compiler frontends and backends in the UNCOL project [6]. The key idea was the introduction of a common intermediate language for m different source languages and n different target processors. Using such an intermediate language, m frontends and n backends have to be written, instead of the $m \times n$ compilers that were required if compilers translated all source languages directly into all machine languages (see fig. 4.1).

Mixing different source languages, using a single library for all source languages and a unified trace and debugging environment across all source languages are welcome byproducts of this approach. It has been used used by some companies in order to provide compilers for different

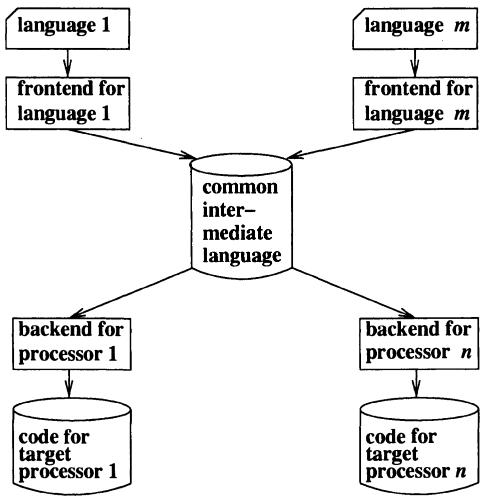


Figure 4.1. UNCOL approach for reducing effort of writing compilers

source languages and target processors (including, for example, the first workstation vendor Apollo Computers Inc.).

The UNCOL approach still requires manually written frontends and backends. Generating frontends was very much simplified with the availability of tools like LEX and YACC and standard techniques such as recursive descent [49]. Follow-up research also aimed at avoiding the work of writing backends. We will present major contributions in this area, covering contributions from microprogramming and standard compilers separately.

1.2. Code generation for expressions

As mentioned in the previous chapter, one of the standard actions performed within a compiler is *code generation.* Code generation is responsible for finding – for each operation of the source program – a corresponding set of machine operations. The fact that this can be done using term rewriting has been common knowledge for many years. It is also common knowledge, that rewriting is *ambiguous*: there are many sequences of machine instructions that implement a source program. A key problem is to find an efficient sequence of machine operations. Different approaches have been used for this.

The approach of Granville and Graham [17] to code generation is based on LR(1) parsing. This means that *string parsing* is used instead of the tree parsing introduced in the previous chapter. For each target architecture, an LR(1) parser is generated. Like other LR(1) parsers, Glanville's parser performs shift, reduce, accept and error actions. The shift action reads the next symbol from the intermediate representation. The reduce action applies a rule, corresponding to the generation of an instruction, and shortens the stack. The accept action is performed when a program has been compiled and the error action corresponds to cases in which the source program cannot be compiled. The parser is designed such that it tries to exploit special purpose instructions. However, no explicit cost model is used and hence, optimality cannot be guaranteed.

In order to generate low-cost instruction sequences, Cattell proposed the so-called *maximum munching method* [4]. In this approach, the instruction pattern covering the largest segment of the IR representation is always replaced first. Again, optimality cannot be guaranteed. A survey on the start of the art in retargetable compilation in the early eighties was published by Ganapathi [15].

BURS theory is a more recent special instance of a term-rewriting approach [41]. BURS stands for *bottom up rewrite systems.* BURS theory can be used to generate optimal code sequences, similar to tree parsing techniques based on tools like IBURG and OLIVE. The relation between tree parsing and tree automata is described by Wilhelm et al. [14].

2. Contributions from microprogramming

Contributions from the microprogramming domain that can be used for designing retargetable compilers, include the following:

- scheduling techniques,
- explicit target machine models, and
- explicit consideration of machine resources.

Work on microprogramming led to some important results on scheduling and resource allocation which are beneficial for current compilers which also have to consider hardware details and not just the instruction set. Programming contemporary VLIW processors requires the use of scheduling techniques which initially were developed for applications in microprogramming. The important effect of microprogramming on future design technologies was phrased very nicely on a cover of ACM's SIGMICRO Newsletter: *Microprogramming is dead – long live microprogramming!* Hence, it is interesting to look at some of the origins of these techniques in microprogramming. Scheduling was discussed in detail in chapter 3. Therefore, we focus on the other contributions from microprogramming in this section.

2.1. Motivation

Following Maurice Wilkes, many of the early computers were implemented using microprogramming. With this approach, machine instructions were interpreted by microinstructions. Microinstructions typically reside in a small and fast microprogram memory and have full access to all hardware blocks. Microprograms were almost always written at the micro-assembly level, using specialized, machine-dependent micro-assemblers.

Writing microprograms at the micro-assembly level was very time-consuming, since many hardware details had to be taken care of. Therefore, researchers started to look for ways of writing micrograms at a higher level of abstraction and interest in microcode compilers started to rise. It was immediately obvious that microcode compilers should provide some level of target-independence. Reasons for this include the following:

- There was a *huge variety* of microinstruction sets.
- Only *few microprograms* were written for each instruction set. This and the huge variety of instruction sets made traditional compiler development too costly.
- Microcode-compilers were needed in the *very early phase* of a processor design project and waiting for the completion of some slow compiler development project was impossible. Hence, traditional compiler development was too slow.

It was obvious that retargetable microcode compilers would solve the problem.

2.2. Early work

Surveys of very early work on retargetable microcode generation were published by Bushell [3] and Sint [44]. It seems like this work was not very successful.

However, taking this work into account, Dasgupta concluded that the design of a retargetable microcode compiler compiling from a single high-level language initially was too difficult. Therefore, Dasgupta proposed the use of a microprogramming language schema S* [10]. S* represents a family of languages sharing the same high-level language elements (including the elements describing control) but also including some target-machine specific statements. S* was used to automatically generate simulators in a "retargetable firmware development system" [9] at the University of Aachen. Interestingly enough, researchers from the same university have more recently designed a technique for the retargetable generation of very fast simulators [40].

2.3. First retargetable microcode compilers

The first microprogram compiler that received major attention was the "machine-independent efficient microprogram generator" MPG by Takanobou Baba and his co-workers [2]. The MPG system used a machine description section (MDS) and an algorithm description section (ADS) as its input. The MDS consisted of a *control section* (describing the controller) and a *controlled section* (describing the data path). ADS is a relatively low-level programming language. In the ADS, assignments can refer to all machine registers. Assignments are translated into sets of operations available for a target machine, called micro-operations. During the translation, MPG tries to find efficient sets of micro-operations. Emphasis of MPG, however is on allocating microinstructions – which are composed of micro-operations – to the microprogram memory. Addressing microprogram memories typically is very complex. For example, many machines used pairs of microinstructions and branches were only allowed if the two possible branch destinations were from the same pair of instructions. The authors of the MPG system focused on handling such situations. Decomposition of large expressions, arrays, pointers, procedures etc. were not considered. The MPG system was used for generating microprograms for HITAC8350 and HP2100A processors, for which a surprisingly low overhead of just 12 % respectively 6 % was reported.

Vegdahl [47] –like Cattell being from CMU– tried to extend Cattell's work to microprogramming. Cover generation was based on Cattell's maximum munching technique. Other components of the backend, like constant generation, had to be written manually.

One of the first commercial systems was the IDAS system from JRS Research Labs. The input languages selected for IDAS suit the needs of the US Department of Defense: ADA for algorithm description and VHDL for hardware description. JRS specifically focused on using retargetable compilation for design space exploration: *One can make a change in the hardware (e.g., delete an ALU, add a Multiplier, reduce the amount of power available) and directly measure the impact of the change on the performance of the program* [24]. Accordingly, JRS is one of the tool providers in the RASSP (Rapid-Prototyping of Application Specific Signal Processors) initiative. In addition to the support of ADA, C was added as another language for describing algorithms. Focus was very much on the commercial exploitation of the underlying technology and hardly any detailed description of it is publicly available.

Other contributions were made by Robert Mueller and his group. Their Horizon compiler [37] also allowed compilation of low-level programs. A C-derivative called Micro-C was used as the language for describing algorithms. In Micro-C, variables correspond to machine resources. Language operators correspond to operators available in hardware. Code-generation was based on finding paths in the micro-architecture from sources to sinks. Target specific information about the paths and functional units in the micro-architecture was encoded in PROLOG and PROLOG was then used to find paths along which required data moves could be implemented. An attempt was made to commercialize Horizon through QTC Corp., Beaverton. Design space exploration was clearly among the goals of QTC [51]. Later, QTC took the tools off the market.

2.4. The MIMOLA project

Synthesizing application-specific micro-architectures from algorithmic descriptions of the applications was the initial goal of the MIMOLA project [52]. MIMOLA stands for *machine-independent microprogramming language.* The MIMOLA project and a corresponding independent project [39] at the Carnegie-Mellon-University were the first two projects in high-level synthesis (even though that name was not used at that time). The MIMOLA Software System (MSS) incorporated tools for high-level synthesis, code generation, test generation, synthesis and schematic generation in a consistent way [33]. For the MIMOLA project, it was possible to start from an algorithm and a partial micro-architecture, the latter describing some of the ideas of the designer about the target hardware. A completely specified micro-architecture was included as a special case of a partially specified architecture. The MIMOLA project also assumed the availability of a program memory and was capable of translating algorithms into binary machine instructions to be stored in the program memory. Performing this translation for a fully specified micro-architecture is equivalent to retargetable compilation. Accordingly, work on retargetable compilers began in the late seventies. Focus was on machines with very long instruction words, now called VLIW machines, but then called horizontally microprogrammed machines. The term microprogramming was used despite the fact that no machine-level programming on top of the microprogramming level was assumed.

The MIMOLA language provides mechanisms for describing hardware structures and algorithms. It is clearly oriented towards synthesis [34] and – except for early versions – is based on PASCAL. The first version of the retargetable compiler was called MSSV [28, 29] (the more recent MSSQ is described in section 5.4.1). MSSV – just like Horizon – performed code selection by trying to find paths form source to sink resources in the micro-architecture. Merging of several micro-operations (so-called bundling) was performed for inputs of n-ary hardware operators such as ALUs. In contrast to Horizon, several possible paths were considered and forwarded to the scheduler MSSC. Just like in Horizon, matching of algorithm and hardware was performed at a low level, without trying to extract the instruction set from the description of the micro-architecture. Increased interest in retargetable compilation led to the publication of unpublished material in [31].

The brief description of the early work in the MIMOLA project concludes our presentation of the roots of retargetability. In the next chapter, we will present more recent approaches to retargetability.

Chapter 5

RETARGETABLE COMPILER CASE STUDIES

This chapter describes a selection of retargetable compilers and code generation techniques. We focus on a representative list of specific tools and techniques. Due to the limited space, we naturally cannot cover many interesting details, but these are mostly available in several publications anyway. Instead, our goal is to highlight their advantages, limitations, and novel concepts, as well as to put the different approaches into context. Additionally, we mention practical issues like availability and licensing terms of software. Clearly, also many other compiler techniques besides the ones mentioned here are retargetable in the sense that they show a certain degree of machine independence. However, we focus on tools and techniques that explicitly use some kind of *machine model* in order to adapt the compiler to different targets. In addition, we restrict our review to approaches that have at least some relation to compilers for *embedded systems.*

The tool overview is mainly categorized by target processor classes. First, we present retargetable compilers and compiler infrastructures for general-purpose processors (GPPs). Then, we switch to domain or application-specific machine classes like DSPs, VLIWs, and ASIPs. Important point solutions and techniques that do not well fit into this list of categories are summarized separately in section 5. Finally, we describe a set of commercial compiler systems that emphasize retargetability. Appendix A provides a chronological tabular overview of important tools together with WWW links to their home pages and/or available software.

1. Retargetable compilers for GPPs

1.1. GCC

Among the most well-known and widespread retargetable compilers is GCC, which is part of the GNU free software project. GCC mostly is used as a C/C++ compiler, but it also comprises frontends for Fortran, Java, and some exotic languages.

The GCC compiler for most platforms can be downloaded from [75]. The web site also provides documentation and background information. Red Hat [76] provides an MS Windows port called "cygwin", which emulates a Unix environment under MS Windows. The compiler falls under the *GNU public license*, which mainly means that the software can be freely used, modified, and distributed, provided that the corresponding source code is still made freely available.

There are GCC backends for numerous OS platforms and target processors, in most cases CISCs and RISCs (including Sparc, MIPS, Alpha, Intel x86, and M68000). However, GCC originally was not designed as a "clean" retargetable compiler, but the target machine description capabilities have been extended on demand over the time. This is what the manual says about GCC's portability:

"*The main goal of GCC was to make a good, fast compiler for machines in the class that the GNU system aims to run on: 32-bit machines that address 8-bit bytes and have several general registers. Elegance, theoretical power and simplicity are only secondary.*

GCC gets most of the information about the target machine from a machine description which gives an algebraic formula for each of the machine's instructions. This is a very clean way to describe the target. But when the compiler needs information that is difficult to express in this fashion, I have not hesitated to define an ad-hoc parameter to the machine description. The purpose of portability is to reduce the total work needed on the compiler; it was not of interest for its own sake."

The code generation process in GCC consists of about 20 passes, that revolve around an intermediate representation (IR) called RTL (register transfer language). A nice feature is that the RTL code after each pass can be dumped into a readable file to monitor the IR modifications. The first pass is the frontend, which generates an initial RTL for a given source program. In contrast to other IR formats, RTL already consists of machine-specific instruction patterns. RTL generation takes place on a simple statement-by-statement basis, modern techniques like tree parsing are not applied.

Next, there are a number of standard optimization passes, e.g. jump optimization, common subexpression elimination, and loop optimization. Register allocation is split into a local and a global pass. Likewise, instruction scheduling is distributed over several passes. The first scheduling pass aims at instruction reordering in order to avoid pipeline stalls. The second scheduling pass essentially does the same, but also takes the spill code resulting from register allocation into account. Yet another scheduling pass is responsible for filling possible delay slots, and finally assembly code is emitted for the optimized RTL code. Thus, GCC's code generation process follows a quite conventional approach. In particular, there is hardly any phase coupling, and there are no built-in optimizations dedicated to embedded processors.

A target machine description for GCC typically comprises three files: a machine description (MD) file, a C header file for macro definitions, and a C source file with processor-specific support routines. GCC's retargeting mechanism uses these files to reconfigure the compiler source code, which afterwards can be compiled and linked to produce code for the given machine.

The main purpose of the MD file is to inform the compiler about the available instruction set and some specific optimizations. Mostly, "define_insn" constructs are used to define available instructions. Such a construct generally specifies a name, an RTL template, an assembly output template, and matching constraints. This is shown in the following example:

```
(define_insn "subsf3"
  [(set (match_operand:SF 0 "register_operand" "=f")
        (minus:SF (match_operand:SF 1 "register_operand" "f")
                  (match_operand:SF 2 "register_operand" "f")))]
  ""
  "subf\\t%0,%1,%2")
```

The name "subsf3" informs the compiler that the pattern describes the subtraction of single precision floating point numbers, which can be exploited in RTL generation. "match_operand" serves as a placeholder for actual operands to be inserted during code generation. All three operands in this example have a unique number (0,1,2) and are of type "single float" (SF). The string "register_operand" is a predicate ensuring that only registers can be used as operands of the instruction. The letter "f" further restricts the operands to be floating point registers, where "=f" denotes that operand 0 (the destination register) is write-only in this context. Finally, an assembly output template is provided, which emits a "subf" mnemonic, followed by the operand registers, in case the "subsf3" pattern has been used for matching a C expression.

The MD file may also contain so-called *expander definitions* that inform the compiler about operations too complex to be handled with a single machine instruction. A "define_expand" construct therefore allows to specify how operations can be implemented by sequences of RTL instructions. Finally, one can specify machine-specific peephole optimization. A "define_peephole" construct in the MD file tells GCC how a certain sequence of instructions can be replaced by a more cost-effective sequence, possibly dependent on some matching conditions. The following shows an example from a GCC port to the TI TMS320C25 DSP, developed at the University of Toronto. There, a load from memory and an addition are combined into a single instruction (LTA – load and accumulate).

```
(define_peephole
  [(set (reg:QI 2)  (match_operand:QI 0 "memory_operand" "m"))
   (set (reg:QI 0)  (plus:QI (reg:QI 0) (reg:QI 1)))]
  ""
  "LTA  %0")
```

The C header file with target-specific macros partially consist of purely numerical parameters. An example is given in the following, where endianess, the minimum addressable storage unit, and some type bit widths are defined.

```
#define BYTES_BIG_ENDIAN 1
#define BITS_PER_UNIT 8
#define INT_TYPE_SIZE 32
#define SHORT_TYPE_SIZE 16
#define LONG_TYPE_SIZE 32
```

Many others of the huge amount of GCC's machine macros are more complex and, for instance, specify the register names, register classes, the syntax for assembler directives, and argument passing conventions, just to name a few. Finally, the C file with support routines amongst others typically defines highly machine-specific things like the assembly code sequences to be emitted for *function prologues* and *epilogues.*

In total, one can say that the machine description required by GCC is quite complex, and it certainly takes some time to be able to retarget the compiler to a completely new processor. On the other hand, GCC's description complexity can also be considered an advantage, since it allows for a high degree of flexibility for handling special hardware details. This explains why GCC is available for a significant number of different GPPs, and is actually in intensive practical use.

When it comes to embedded systems, however, the applicability of GCC strongly depends on the target processor. In the DSP area, for instance, GCC has been ported to the Analog Devices 2101 and the Motorola 56001. The DSPStone benchmark [77] showed that a performance overhead of 400 % or more of compiled code versus hand-written assembly is not unusual. This indicates limitations of GCC w.r.t. irregular processor architectures. The compiler is hardly capable of exploiting small heterogeneous register files, instruction-level parallelism, or multiple memory banks. The report [78] describes a project where GCC has been ported to Thor, a stack-oriented embedded RISC processor. The authors show in detail the problems encountered when porting GCC to an "unusual" architecture, and they describe some workarounds. Among the main problems mentioned is GCC's need for a byte-addressable memory and the sparse documentation.

In summary, GCC is generally a good choice when the target processor is a GPP for which a compiler port already exists, or which is at least compatible to GCC's intended target machine class. In this case a big plus for GCC is that it comes with a comprehensive set of support software, such as assembler, linker, debugger, and standard C/C++ headers and libraries. Additionally, GCC is a very stable compiler. These are probably the reasons why processor core vendors like ARC [79] and Tensilica [80] have chosen GCC as a primary development platform.

In case the target machine does not really fit into the GCC concept, porting gets very difficult, since the GCC source code is somewhat hard to patch, and the resulting code quality may be expected to be poor.

1.2. LCC

The "little C compiler" LCC has been developed at Princeton University. The current version V4.1 can be downloaded together with its source code from [81]. The compiler is easy to install for Unix and Linux machines, and there is a special development branch called LCC-Win32 for PC/Windows platforms [82].

In contrast to GCC, LCC is actually a lightweight C compiler. The total source code has a size of only about 13,000 lines. Nevertheless, it is a complete and retargetable ANSI C compiler. Due to its small size, it is possible to study LCC's anatomy in detail. A big advantage in this context is that the source code is well documented in the form of a book [83]. The standard LCC distribution comes with built-in backends for MIPS, Alpha, Sparc, and Intel x86 target processors. Also the design of the backends is documented in the LCC book, which provides a good insight into LCC's retargeting capabilities.

Similar to GCC, retargetability has not been a design goal for LCC right from the beginning. Instead, it evolved into its current state over the years. This is what the documentation says:

"*There was no separate design phase for LCC. It began as a compiler for a subset of C, so its initial design goals were modest and focused on its use in teaching about compiler implementation in general and about* *code generation in particular. Even as LCC evolved into a compiler for ANSI C that suits production use, the design goals changed little.* "

Now LCC can be considered a stable C compiler, even though it is not as widespread as GCC in the Unix world. For PC/Windows platforms, however, LCC is frequently preferred over GCC, since the LCC-Win32 version comes with a relatively comfortable graphical development environment.

The LCC copyright permits the free use of the software for research and education, as well as redistribution. In contrast to the GNU public license, there is no need to publish the source code as well. Building commercial products on top of LCC, however, is more problematic, since the copyright is with a publisher. Again, this situation is different for the LCC-Win32 version, for which professional support is also available.

Being a lightweight compiler, LCC's code optimization capabilities are certainly limited. There are no global optimizations, even no global register allocation, and local optimizations are limited to simple passes like constant folding and common subexpression elimination. As a consequence, the code quality may be expected to be somewhat lower than GCC's in general. On the other hand, LCC is extremely fast, but this is usually not too important in the context of embedded system design.

The code generation procedure works roughly as follows. The ANSI C frontend translates a given C source into data flow graphs (DFGs). The DFG operators essentially correspond to the C language operators, but they also carry type and size information. Therefore, the DFG operators are machine-dependent. Fig. 5.1 shows an example with a piece of C code and the DFG format dumped by LCC.

```
int a;
int f(int* p)
{ return a + *p + 1; }

5. ADDRGP2 a      // address of global var "a"
4. INDIRI2 #5     // load a
8. ADDRFP2 p      // address of parameter "p"
7. INDIRP2 #8     // load p
6. INDIRI2 #7     // load *p
3. ADDI2 #4 #6    // integer add
9. CNSTI2 1       // constant "1"
2. ADDI2 #3 #9    // integer add
1. RETI2 #2       // integer return
```

Figure 5.1. C code and LCC's intermediate representation

Code selection is performed by tree parsing, based on a tree grammar specification of the target instruction set. The code selector itself is generated by means of the LBURG tool, a variant of IBURG [74]. The resulting symbolic machine code is linearized and passed to the register allocator. The register allocator performs local allocation of registers, i.e. in a basic block oriented fashion. Finally, assembly code is emitted.

Similar to GCC, the target processor for LCC is modeled in the form of a machine description file. In contrast to GCC, however, there is usually only a single, quite compact, MD file. The MD file contains two main parts: a target instruction set description and a set of C support functions.

The instruction set is described as a tree grammar, very similar to the notation used for IBURG/OLIVE (sections 5.5.3.1 and 5.5.3.2). First, the grammar terminals are defined. As mentioned above, LCC's terminal symbols are machine specific to a certain extent. This is exemplified in the following, where some terminals used in LCC's x86 backend are shown:

```
%term ADDF4=4401
%term ADDF8=8497
%term ADDI4=4405
%term ADDI8=8501
%term ADDP4=4407
%term ADDP8=8503
%term ADDU4=4406
%term ADDU8=8502
```

All symbols refer to an ADD operation, where the fourth letter denotes the type (F = float, I = integer, P = pointer, U = unsigned). Finally, a number is used to inform the compiler about the type size in bytes (e.g. 4 for a float, 8 for a double).

Next, the actual instruction patterns are described in the form of tree grammar rules, together with some corresponding output assembly template and an optional pattern cost value. Again we illustrate this by an example. The rule

```
    reg: ADDI4(reg,mrc)  "?mov %c,%0\nadd %c,%1\n"  1
```

describes a 4-byte integer ADD operation, where `reg` is a nonterminal denoting a general-purpose register, and `mrc1` is another nonterminal that may match a memory, register, or constant operand. Following the rule, a string specifies the assembly instruction (or instruction sequence) to be emitted in case the rule has been selected. In this case, there is a "mov" followed by an "add" instruction. The symbols with a "%" prefix serve as *placeholders* for the actual values of the nonterminals to be inserted later. Here, "%c" denotes the symbol on the left hand side of the rule (which is going to be a physical register afterwards), while "%0" and "%1" refer to the argument nonterminals. The special character "?" at the beginning of the assembly template tells LCC's code generator to suppress emission of the first assembly instruction ("mov") in case it turns out to be redundant. Finally, a cost value can be specified. In the above example, this is a constant "1", but also C functions may optionally be called for more complex cost computations.

In case the required assembly output for some rule cannot be specified with a simple assembly template as above, LCC offers an escape mechanism. If the template starts with a "#" character, the code generator calls a special C support function in order to emit code for the respective rule.

The remainder of the MD file specifies a fixed set of about 20 target-specific C support functions. There is a dedicated initialization function used to inform the compiler about available registers and register classes. Other functions are used, for instance, to emit function prologues and epilogues, and to guide the emission of assembler directives and segmentation information.

In order to handle special register constraints, there are also functions that permit to bind certain operations to specific registers. The following code fragment is taken from LCC's x86 backend.

```
static void target(Node p) {
        switch (specific(p->op)) {
        case MUL+U:
                setreg(p, quo);
                rtarget(p, 0, intreg[EAX]);
                break;
...
}
```

The support function **target** generally has the form of a switch statement that selects over the different DFG operators. The above specification defines that for unsigned multiplication the destination has to be a register class called "quo", while the left argument has to be allocated in register EAX.

Like GCC, the LCC compiler certainly has a preference for more or less regular RISC/CISC processor architectures with a byte-addressable memory. Modeling irregular DSP data paths is more difficult, since the built-in register allocator might need to be bypassed. Nevertheless, the existing backends indicate that LCC is surprisingly flexible. Due to the rather concise code generation interface, retargeting LCC will generally be simpler than in the case of GCC. On the other hand, code quality may be expected to be lower due to the missing standard optimizations and the local register allocator. In addition, LCC does not comprise an instruction scheduler, which has to be implemented as a postpass optimization if required.

1.3. Marion

Marion [84, 85] is a retargetable compiler designed for RISC architectures. Target processors handled by Marion include Motorola 88k, Intel i860, and MIPS R2000. Special emphasis is put on effective retargetable instruction scheduling and the coupling of the traditionally separated scheduling and register allocation phases in order to maximize code quality. It is assumed that the target machine has a load-store architecture with a general purpose register file, and that the functional unit usage by each instruction is known at compile time.

Marion uses LCC's C frontend for generating an intermediate program representation. The target machine is described in a language called Maril. A target model in Maril consists of three sections: a resource declaration, a runtime model, as well as an instruction set description.

The resource section declares processor entities like registers, memories, functional units, and pipeline stages. The runtime model mainly defines compiler properties like calling conventions, stack frame layout, and the use of certain registers for special purposes (e.g. registers available for general allocation and the frame and stack pointer registers).

Finally, the instruction section models the available machine instructions in detail. This includes the assembly syntax, as well as the instruction behavior in terms of a C expression. Instructions with side effects (such as auto-increment) cannot be modeled, though. Additionally, an instruction specification describes its resource usage on a cycle-by-cycle basis (similar to a reservation table), a cost value, the instruction latency, and the number of delay slots. The following example taken from [85] illustrates this concept with the example of a load instruction:

```
%instr ld r,r,#const16    // assembly syntax
   { $1 = m[$2+$3]; }     // behavior
   [ IF;ID;IE;IA;IW; ]    // used pipe stages
   ( 1,3,0 )              // cost, latency, delay slots
```

The code generator first maps the IR into the described assembly instruction set by a tree-based greedy heuristic. In case that IR constructs have no direct correspondence to assembly instructions, the mapping has to be described manually, either by rewrite rules or by special C support functions. Next, global register allocation and instruction scheduling are performed. Both phases rely on standard techniques each (graph coloring and list scheduling, respectively), but a heuristic phase coupling is provided in order to partially eliminate the well-known phase ordering problem: doing register allocation first may restrict the scheduler's instruction reordering opportunities, while performing scheduling first may lead to superfluous spill code.

Experimental results for the targets mentioned above indicate that the retargetable phase coupling approach between register allocation and scheduling works and produces good code. The Marion system is actually a good example for the fact that retargetability and high code quality are not necessarily contrary goals, provided that only a certain processor class is targeted. Another strength is its capability of code generator generation from quite concise machine models. Limitations of Marion concern the expressiveness of the Maril modeling language, low compilation speed and robustness, and the lack of global IR optimizations and a more powerful code selector.

1.4. PAGODE

PAGODE is a backend generator for RISC targets [90, 91] that has been developed within the European research project COMPARE [92]. It reads a target machine specification in the SCALA language and generates code selector, register allocator, scheduler, and assembly code emitter for the given target. A SCALA description captures all target machine characteristics required for generating these tools: instruction templates with semantics, assembly format, cost metrics, and pipelining restrictions, as well as available storages and registers.

Compilers generated with PAGODE require some source language frontend that generates a machine-independent intermediate representation (IR). For this purpose, a coupling to the CoSy (section 5.6.1) frontends has been implemented. The code selector uses tree parsing to map the IR into a common machine-dependent low-level IR (LIR), which the remaining code generation phases operate on. After code selection, register allocation via graph coloring is performed. Finally, a list scheduler is applied to each basic block, so as to minimize pipeline hazards, and assembly code is emitted.

PAGODE has been used to generate a Sparc backend, but results on code quality have not been published. The system emphasizes modularity and extensibility of generated backends. On the other hand, there is no phase coupling between code selection, register allocation, and scheduling, but only standard techniques are used for code generation. Hence, the code quality may be expected to be lower in general than e.g. in the Marion system, and retargeting to embedded processors with irregular architectures is obviously not supported.

1.5. SUIF/Machine SUIF

Stanford University's SUIF system is actually more an optimizing frontend rather than a retargetable compiler. The software can be downloaded in two versions from [86].

The original version SUIF1 comprises an ANSI C frontend built on LCC's frontend and also provides Fortran entry capabilities via a Fortran to C translator. On the backend side, a MIPS code generator is provided. According to the licensing information, the software may be freely used, modified, and redistributed for any commercial or non-commercial purpose. SUIF has been designed primarily as a compiler research framework, and for this purpose it is widely used.

The optimization focus of SUIF is on *parallelizing transformations*, but also classical scalar optimizations like constant folding and propagation are included. The compilation and optimization process is intended to be very transparent and extensible by using a file exchange format between all compiler phases. In this way, new optimizations can be inserted at any time in a "plug-and-play" fashion. Naturally, this makes SUIF somewhat slower than usual compilers.

SUIF1 has no particular support for retargeting to different processors, but it is possible to dump the intermediate representation in C syntax at any time, so that an existing C compiler may serve as a "backend".

SUIF's machine-independent IR can retain high-level C constructs like loops, conditionals, and array accesses, a feature that facilitates the intended complex program transformations. Alternatively, a "low-SUIF" IR may be used, where all high-level constructs are lowered down to assembly-like, yet machine-independent, code. Optimization passes will generally require either the high or the low IR format, however. IR access and manipulation are possible via a C++ class library. We illustrate the different IR-levels with a small C example:

```
int A[10];

void main(int a,int b,int i)
{
  A[i] = a < b ? 10 : 20;
}
```

The high-SUIF IR for this example (exported in C syntax) looks as follows. As can be seen, the conditional expression is still visible in its original form.

```
void main(int a, int b, int i)
  {
    int suif_tmp0;

    if (a < b)
      {
        suif_tmp0 = 10;
      }
    else
      {
        suif_tmp0 = 20;
      }
    A[i] = suif_tmp0;
    return;
  }
```

In contrast, in the low-SUIF format the conditional is replaced by jumps and labels, and also the array access is converted into a pointer access:

```
void main(int a, int b, int i)
  {
    int suif_tmp0;

    if (a >= b)
        goto L1;
    suif_tmp0 = 10;
    goto __done2;
  L1:
    suif_tmp0 = 20;
  __done2:
    *(int *)((char *)A + i * 4) = suif_tmp0;
    return;
  }
```

The new version SUIF2 represents a complete revision of SUIF, and is currently in a beta release stage. There are, however, conversion utilities for backward compatibility to SUIF1. The IR has been redesigned to be more modular and extensible. The terms of use are similar to SUIF1, except for the C frontend. SUIF2 is based on EDG's commercial C++ frontend [87], therefore only the binaries are available, and free use is restricted to research purposes.

Just like SUIF1, also the new version does not directly support retargeting to different processors. However, the Machine SUIF project at Harvard University [88] aims at filling this gap. It complements SUIF with a low-level but still partially machine-independent IR, that shows a one-to-one correspondence to assembly instructions. The low-level IR is generated from the normal SUIF IR by a dedicated lowering pass.

The goal is to keep most of the source code required for low-level optimizations and code generation (e.g. register allocation and instruction scheduling) machine independent and thus reusable, while encapsulating the machine-specific details in a target library. In this way, backends for Alpha and x86 targets have been constructed with Machine SUIF. However, it still has to be investigated whether the low-level IR is expressive enough to handle more typical cases of embedded processors.

In summary, the SUIF/Machine SUIF packages are certainly a good starting point for research on retargetable code generation, since they provide a large part of the required compiler infrastructure for free. On the other hand, SUIF has so far hardly been applied to code generation for embedded application-specific processors. An exception is the SPAM compiler described in section 5.2.4.

1.6. Zephyr/VPO

The Zephyr compiler infrastructure has been designed by the University of Virginia in cooperation with Princeton University. Together with the SUIF system, it forms a main component of the U.S. National Compiler Infrastructure project. While SUIF focuses on high-level code transformations, Zephyr emphasizes the machine-dependent code optimizations. It supports different language frontends, such as EDG's C++ frontend and LCC's C frontend. Alternatively, the required intermediate representation (IR) can be generated via SUIF (which gives access to further frontends) and be converted into Zephyr's internal format. The Zephyr software, including source code, can be downloaded from [89]. It is copyrighted by the University of Virginia.

Zephyr supports both different IRs and different target machines. The IR is described in the machine-independent Abstract Syntax Desciption Language (ASDL), a language that supports file exchange of tree-like IR data structures between compiler components, possibly written in different programming languages. Information about the target machine is captured in a language family called CSDL (Computer Systems Description Languages). Different CSDL dialects are responsible for describing different aspects of the target machine, e.g. there are CSDL sub-languages for describing assembly and binary formats of machine instructions, instruction semantics, calling conventions, and pipeline structure. Even though this approach results in a rather heterogeneous machine model, all CSDL models for a given target share the same data structures for instructions and their respective impact on the processor state.

A main idea in Zephyr is that the IR in a first step is mapped to a naive assembly code implementation without particular optimization effort. The unoptimized assembly is afterwards optimized by the VPO (Very Portable Optimizer) tool. VPO has its origins in the retargetable peephole optimizer PO [73], which also influenced GCC.

The mapping from the IR to unoptimized assembly takes place via a *code expander*. There has to be one dedicated code expander for each combination of IRs and target models. The code expander describes the mapping of each IR construct into an equivalent sequence of RTL assignments. In order to save development time, it is recommended to keep new code expanders extremely simple. Generation of correct machine code is sufficient in this step, since VPO will later take care of the optimization process (fig. 5.2).

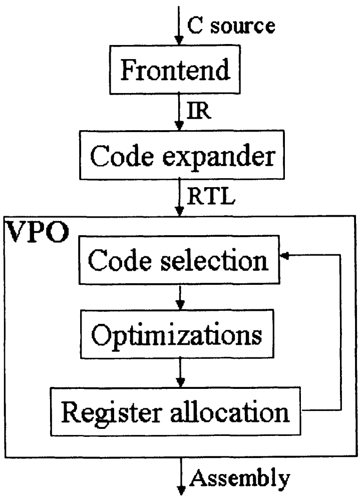


Figure 5.2. Compilation flow in Zephyr/VPO

On the unoptimized code VPO iteratively applies optimizations until some fix point is reached, at which no further optimization seems possible. Available optimizations include common subexpression elimination, loop unrolling, code motion, function inlining, and strength reduction. A key idea is that all these optimizations are machine-independent, and hence reusable, even though they are performed on machine-dependent code. Machine-independence is achieved by performing only such transformations that do not alter the predefined RTL semantics of the assembly code. On the other hand, machine-dependence is preserved by a special VPO module that checks whether the modified assembly still obeys a *machine invariant*, in the sense that all RTL assignments still correspond to exactly one instruction on the target machine. Transformations not satisfying this invariant are rejected.

After the code optimization process has terminated, register allocation by graph coloring is performed, since all transformations take place on RTL code with virtual registers. Afterwards, the assembly syntax description part of the CSDL machine model can be used to emit valid assembly code. A retargetable postpass instruction scheduler is obviously not included in VPO but has to be developed separately if required.

With respect to retargetable compilation the major advantages of the Zephyr/VPO approach are twofold: Retargeting is comparatively easy, since it is not required to specify target-dependent optimizations, as these are left to VPO. Moreover, all code optimizations once implemented in VPO in principle can be reused for all target machines, even though not all optimizations might be effective for each target.

The main limitations of Zephyr are the strict decoupling of code selection, optimization, and register allocation, as well as the requirement of a one-to-one mapping between RTL assignments and assembly instructions, which restricts its use mostly to clean RISC and CISC machines. Irregular architectures frequently found in embedded processors, including special-purpose registers, complex instructions, and limited parallelism are not directly supported, but require machine-specific extensions of VPO.

1.7. LANCE

The LANCE C compiler system, developed at the University of Dortmund [93], comprises an ANSI C frontend, a C++ API for accessing the intermediate representation, a set of standard IR optimizations, as well as a backend interface. The C frontend can be downloaded from [94], while the complete V2.0 system can be licensed for research purposes on request. Supported platforms are Unix, Linux, and MS Windows.

The LANCE system is not as easily retargetable as GCC or LCC, since it does not include a complete backend generation module. Hoever, design of target-specific code generators is supported by the backend interface. LANCE is almost completely machine-independent and has no implicit preference for a certain target processor class. Thus, backends can be designed actually for almost any target. The system covers several compiler passes, from C source analysis over IR optimizations, down to generation and visualization of control and data flow graphs.

LANCE combines ideas from SUIF and LCC. Similar to SUIF, the different compiler and optimization passes operate on a common IR and communicate via file exchange. This makes LANCE comparatively slow, but it enables the same "plug-and-play" extensibility concept as in SUIF:

IR optimizations can be inserted (or omitted) at any point of time. The IR format, however, is kept in the form of pure assembly-like three address code. While a high-level IR as in SUIF better supports certain optimizations, the LANCE IR has intentionally been chosen as a very simple format, so as to make it easily understandable when writing new optimization passes.

Similar to LCC, LANCE can generate data flow graphs that can be directly fed into code selectors generated by tools like IBURG or OLIVE. In contrast to LCC, however, the DFG operators (or the tree grammar terminals, respectively) are machine-independent. This reduces the number of operators (the total number happens to be 42, as opposed to more than 200 in a typical LCC model) and thereby the number of required tree grammar rules, at the expense of more complicated code generator action functions.

We exemplify the LANCE compilation procedure with the same small piece of C code as in our above SUIF example:

```
void main(int a,int b,int i)
{
  A[i] = a < b ? 10 : 20;
}
```

The C frontend translates this into the following IR file:

```
void main(int a_3,int b_4,int i_5)
{
 int t1;
 int t2;
 char *t3;
 int t4;
 char *t5;
 int *t6;

        t1 = a_3 < b_4;
        if (t1) goto LL1;
        t2 = 20;
        goto LL2;
 LL1:
        t2 = 10;
 LL2:
        t5 = (char *)A;
        t4 = i_5 * 4;
        t3 = t5 + t4;
        t6 = (int *)t3;
        *t6 = t2;
        return;
}
```

The frontend assigns a unique numerical suffix to all local identifiers, so as to flatten the possibly nested local scopes in a C function. Additionally, it inserts auxiliary variables ("t" prefix) in order to break complex expressions. All high-level language constructs are tranformed into conditional jump/label constructs, and all implicit type casts as well as pointer and array index scaling are made explicit in the IR. The latter naturally requires machine-dependent type size information. This is passed to the frontend through a small configuration file, which also contains the type alignment information required e.g. for computing structure component offsets.

Similar to SUIF's C export facility for the low-SUIF IR, the IR generated by LANCE is still a valid (but even lower-level) C program. In fact it is pure, flattened three address code in C syntax. This means that the IR can be compiled and executed on a host just like the original C source. This feature is exploited for *validation* of the C frontend, the IR optimization tools, as well as the backend interface. Based on this methodology and a large suite of heterogeneous C test programs, a reasonably good stability of the LANCE system has been achieved, and any newly designed IR optimization pass can be easily validated without the need for a backend or simulator in the same way.

The next step in the compilation procedure is normally the iterative application of IR optimizations like constant folding, dead code elimination, or loop-invariant code motion. However, in our above simple example there is not much to optimize. The final step, therefore, is the translation of the IR into the data flow tree format required for code selection. For our example this looks as follows:

```
* Function 'main'
* Basic block 1:
* Tree 1:
 (cs_CJUMP [IR stm 2: 'if (t1) goto LL1;']
  (cs_LESS [IR exp 8: 'a_3 < b_4' C type: int  ]
   (cs_READARG [IR exp 6: 'a_3' C type: int  ] arg no 1)
   (cs_READARG [IR exp 7: 'b_4' C type: int  ] arg no 2)))
* Basic block 2:
* Tree 1:
 (cs_WRITE [IR stm 3: 't2 = 20;']
  (cs_INTCONST [IR exp 2: '20' C type: int  ]))
* Tree 2:
 (cs_JUMP [IR stm 4: 'goto LL2;'])
* Basic block 3:
* Tree 1:
 (cs_LABEL [IR stm 5: 'LL1:'])
* Tree 2:
 (cs_WRITE [IR stm 6: 't2 = 10;']
  (cs_INTCONST [IR exp 3: '10' C type: int  ]))
* Basic block 4:
* Tree 1:
 (cs_LABEL [IR stm 7: 'LL2:'])
* Tree 2:
 (cs_STORE [IR stm 12: '*t6 = t2;']
  (cs_CAST [IR exp 24: '(int *)t3' C type: int * ]
   (cs_PLUS [IR exp 21: 't5 + t4' C type: char * ]
    (cs_CAST [IR exp 14: '(char *)A' C type: char * ]
     (cs_GLOBALSYM [IR exp 13: 'A' C type: int * ]))
    (cs_MULT [IR exp 17: 'i_5 * 4' C type: int  ]
     (cs_READARG [IR exp 16: 'i_5' C type: int  ] arg no 3)
     (cs_INTCONST [IR exp 4: '4' C type: int  ]))))
  (cs_READ [IR exp 27: 't2' C type: int  ]))
* Tree 3:
 (cs_VOIDRETURN [IR stm 13: 'return;'])
```

Function main is subdivided into four basic blocks, each of which comprises one or more trees. The trees are shown in a textual format, where in each line the operator ("cs_" prefix, followed by an identifier in capital letters) is followed by some debug and type information. For validation purposes, also C syntax export is possible. The indentation indicates the parent/child relationships between the tree nodes. For instance, tree 1 in block 1 denotes a conditional jump dependent on a "less" comparison of function arguments a and b. The LANCE C++ library provides macros and functions that make the generated data flow trees directly accessible to code selectors generated with tools like IBURG and OLIVE.

Retargeting LANCE requires more compiler know-how and backend design effort than systems like GCC or LCC. However, the system is fairly easy to use or to extend due to its clean software architecture and simple IR. It has been applied in a number of prototype compilers for embedded processors. Among others, there is a power-optimizing backend for the ARM7 Thumb RISC instruction set [95]. LANCE also serves as the primary compiler infrastructure at the technology transfer company ICD [98], where commercial C compilers for Infineon's Network Processor architecture [96, 97] and Systemonic's [99] HiperSonic DSP have been developed.

2. Retargetable compilers for DSPs

2.1. CBC

The CBC compiler designed at the TU Berlin [100, 101, 102, 104] is part of a larger DSP tool design effort that also includes retargetable instruction set simulation. CBC is a retargetable compiler for DSPs. It comes without a C frontend, but generates its intermediate program representation directly from a flow graph description of the application.

The target processor is described in nML ("not a Machine Language") [105], which provides a behavioral, instruction-oriented model to the compiler. Similar to other languages, e.g. Maril, an nML description starts with a resource specification in terms of available registers and memories. Functional units are not explicitly captured. A typical resource declaration in nML is shown in the following, which declares a 1024 × 16-bit memory.

```
mem the_memory[1024,int(16)]
```

Available types in resource declarations also include unsigned, floating point, fixed point, and Boolean numbers. However, it is not clear why the resources are typed, and not the operations as one might expect.

The instruction set itself is described in a hierarchical fashion, in the form of an attribute grammar. In this way, nML exploits the tree-like structure of many instruction sets, which in turn permits a concise factored description, as opposed to a lengthy flat list. The basic modeling entity in nML is an *operation*, that includes a certain behavior, assembly syntax, and a binary encoding. An example is:

```
opn add()
action = { reg1 = reg2 + reg3; }
syntax = format("ADD")
image = format("010")
```

Each operation typically comes with three *attributes*: The *action* part describes the instruction behavior in the form of C-like assignments, where the operands are declared resources or constants. The *syntax* attribute accounts for the assembly syntax to be emitted by the compiler, where the "format" construct is used for formatted output similar to the "printf" function in C. Finally, the *image* attribute specifies a partial binary encoding that belongs to the operation.

A special type of operation refers to sub-instructions that merely serve for operand computation. In nML, such an operation is called a *mode*. A mode has syntax and image attributes like a normal operation, but no action attribute. Typical applications are specification of addressing modes or indexed access to a register file.

nML allows to factor operations by means of so-called *OR-rules*. Such a rule defines a name for a set of alternative operations. For instance, if we had defined another operation "sub" analogous to the above "add", then the OR-rule

```
opn alu_op = add | sub
```

allows to refer to either "add" or "sub" by a single identifier "alu_op". More complex operations employing "add" or "sub" as subroutines can now be described concisely by referring to the attributes of "alu_op" instead of explicitly enumerating the suboperations. For instance, if we want to describe a predicated instruction dependent on some flag, we could use the following construct:

```
opn conditional(ins: alu_op)
action = { if FLAG == 1 then ins.action;
                        else "NOP"; endif; }
image = format("11 %s",ins.image)
syntax = format("FLAG ? %s",ins.syntax)
```

Here, a sub-instruction of type "alu_op" is passed as a parameter to operation "conditional", which uses all attributes of "alu_op" to describe a full (predicated) instruction.

The CBC compiler reads the nML target model and generates either assembly or binary machine code for the input flow graph. First, a *lowering pass* maps the abstract flow graph operations into equivalent sequences of machine instructions. The actual code selection is performed by a tree parsing technique [106]. The required tree grammar rules are automatically generated by flattening the nML model, but according to [101] still some manual work is required to make the code selector fully operational.

A special feature of CBC's code generator is a combined register allocation and scheduling technique called *data routing* [107]. This aims at efficient handling of irregular data paths as frequently encountered for DSPs, where tight coupling of code generation phases is a must. A similar approach has also been developed by Rimey and Hilfinger [108]. The CBC data router is driven by a list scheduler and tries to avoid expensive spilling of special-purpose registers on the fly. A problem with this approach is that *scheduling deadlocks* may result, which have to be avoided by a dedicated and potentially very time-consuming algorithm.

Together with the corresponding instruction set simulator, the design of CBC showed that generation of different development tools from a single target model is possible. However, both tools are obviously no longer in use, and results on code quality for realistic targets have never been reported according to our knowledge.

Still, the introduction of the nML modeling language had quite some impact on further projects. Examples include the CHESS compiler (section 5.6.2), which is based on nML, and the LISA processor modeling language (section 5.5.5), which was strongly inspired by nML. A project at IIT Kanpur [109] dealt with automatically generating LCC machine descriptions from nML models. A demonstrator model has been developed for the PowerPC. However, fully automatic retargetability has not been achieved, since important MD sections like register classification and C support routines were not extractable from nML models.

2.2. REDACO

Similar to CBC, the REDACO compiler designed at TU Vienna [110, 111, 112] does not include a programming language frontend, but it starts from a data flow graph (DFG) description of the application program. It targets fixed-point DSPs with irregular data paths.

REDACO takes its machine-dependent information from a target architecture description file (TADF). The TADF uses a relatively straightforward syntax to specify available machine registers as well as instruction patterns together with possible argument and result registers. Additionally, the TADF captures combinations of instructions that qualify for parallel execution under certain constraints.

The code generation procedure revolves around a *Trellis Diagram* data structure (fig. 5.3). This data structure is used to cope with the special-purpose register architecture of typical target processors. There is one Trellis Diagram for each of the available machine instructions (e.g. an ADD or a register-to-register transfer). Each diagram is a graph representation of a machine instruction and its permissible combination of operand and destination registers. Graph nodes represent registers or register sets, while cost-weighted edges represent instructions. Thus, any path in a Trellis Diagram denotes an instance of the underlying machine instruction for a particular combination of operand and destination registers. This information is used by the code generator to ensure that only valid instruction/register combinations are generated.

REDACO comprises a Trellis Diagram generator that extracts all required diagrams from a given TADF. Naturally, this task needs to be performed just once per target machine. Also the program intermediate representation (IR) is converted into a Trellis Diagram format. Some expansion step may be required to ensure that there is a one-to-one correspondence between IR operations and machine instructions. The code generator picks one data flow tree after another from the IR and replaces its internal operators by the corresponding Trellis Diagram. The resulting diagrams are connected to each other by insertion of dedicated data transfer Trellis Diagrams that represent possibly required loads, stores,

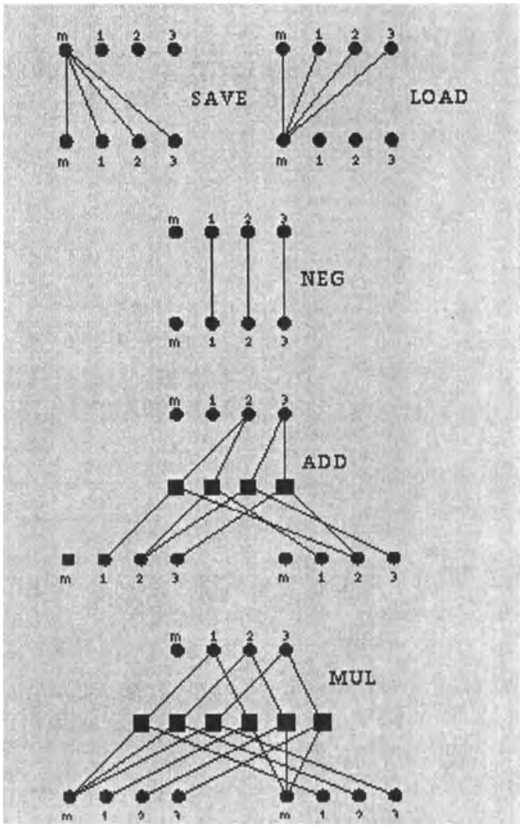


Figure 5.3. Trellis Diagrams for different machine instructions (taken from [113])

or register-to-register moves. In this way, the data flow tree is eventually converted into a *Trellis Tree.*

This Trellis Tree implicitly represents all its possible implementations by machine instructions and also the respective register usage. A *dynamic programming* algorithm is used to detect the minimum cost paths from the tree leaves to the root, so that finally a minimum cost code selection is found. Roughly, this corresponds to the tree parsing approach used in other compilers. However, REDACO uses a slightly generalized notion of *constrained* data flow trees. These may also incorporate common subexpressions (which are normally represented by separate trees) and thus have to be attributed with a partial evaluation order. In addition, the compiler keeps alternative optimal solutions to achieve higher optimization freedom in the subsequent phases.

The selected code tree is linearized and passed to the register allocator and compactor. The register allocator, which is based on an interference graph and is coupled to the compactor, exploits the register assignment choices remaining after the Trellis Tree traversal, e.g. for efficiently communicating values between data flow trees and minimizing spill code. The compactor is based on the *critical path heuristic* [114] and aims at optimum local parallelization of generated instructions in accordance with the constraints in the TADF description. One of the last optimization phases in REDACO is offset assignment (section 3.3.4), aiming at a high utilization of auto-increment capabilities of the address generation unit for efficient access to local variables.

REDACO has been retargeted to different fixed-point DSPs, including TI C25, ADSP-210x, and Motorola 56k. For some small input DFGs it has been capable of generating code whose quality comes close to handwritten assembly. These results show that good code quality can be achieved even for irregular target machines such as DSPs when using advanced code generation techniques. The main limitations of REDACO are its focus on a narrow class of targets and the missing frontend for a programming language like C. Since the tool only compiles DFGs, function calls, aggregate data structures, and control flow are obviously not supported.

2.3. CodeSyn/FlexWare

The FlexWare system from STMicroelectronics performs semi-automatic generation of different software development tools from machine descriptions, including compiler, simulator, debugger, and profiler [118, 119, 120, 121, 122].

The first approach to retargetable compilation within the FlexWare project was the model-based CodeSyn compiler. The target model in CodeSyn is described in three parts. First, there is a set of available *instruction templates*, each specified as a small tree-shaped pattern that represents a piece of computation (e.g. an ADD or a conditional jump) performed by a certain target instruction. This format strongly resembles the one used in tree parsing based approaches to code generation. Additionally, assembly syntax and opcode information is annotated to the templates.

The second part is a structural *connectivity graph* that reflects the interconnections between processor resources like registers, ALUs, and memory. The graph model mainly serves to inform the compiler about possible ways of data transportation within the data path. Finally, there is a *resource classification* section in the target model that accounts for available physical registers, register classes, and functional units. The register classification refers to the inputs/outputs of the instruction templates. Additionally, a mapping of instruction templates to the functional units has to be specified.

The compiler first translates a given input C program into an internal control/data flow graph (CDFG) model. Similar to CBC, abstract CDFG operations are lowered down to machine operations in a rewriting phase. On the lowered CDFG, pattern matching is applied w.r.t. to the specified instruction template set, so as to find an optimum covering of subgraphs by machine instructions. As opposed to many other compilers, CodeSyn uses a custom code selection technique based on a so-called *prune tree* data structure that implicitly enumerates all possible covers. Eventually, however, the covers are selected by means of dynamic programming, so that the results should be very similar to the tree parsing technique in general.

Following code selection, greedy register allocation is performed. Special focus is on handling of irregular register architectures. Each virtual register is characterized by a set of candidate physical registers, and register allocation heuristically starts with assigning those virtual registers with the lowest number of candidates. Potentially required register-to-register moves and spills are generated on-the-fly. Remaining freedom in register allocation is finally exploited via a *left edge algorithm* that locally aims at spill code minimization.

The last step in CodeSyn is an instruction scheduling pass. This is implemented as a code compaction algorithm that, based on the specified resource occupation of instructions, aims at parallelizing generated instructions under the given constraints by means of a list scheduling approach.

CodeSyn has been retargeted to a DSP-like Nortel ASIP, which also has been the driver for the development of the compiler. The compiler has been applied to some very small C programs, for which an average code size overhead of 19 % over hand-written assembly was found.

Further evaluation of CodeSyn showed that the approach is somewhat restricted concerning the class of possible target processors. As a consequence, a new generation of retargetable compilers called FlexCC has been developed. In contrast to CodeSyn, FlexCC is *rule-driven*. This means that the entire code generation process is steered by manually specified, machine-dependent translation rules. The required rules mainly fall into two classes. The first set of rules describes the mapping of C code constructs into instructions of a *virtual machine*. This machine is functionally similar to the intended target, but provides no instruction-level parallelism.

This "virtual" code selection step also comprises register allocation, based on local graph coloring like in CodeSyn. The second rule set refers to assembly-level peephole optimizations. These are used in a compilation phase called *target machine mapping*, where (similar to the approach in GCC) partial instruction sequences can be refined to exploit highly target-specific instructions and to accommodate restrictions. Also, function calls and control statements are handled during this phase. As in CodeSyn, the final phase in FlexCC is code compaction for exploitation of parallism, followed by assembly code emission.

In summary, FlexCC is a very pragmatic approach to retargetable compilation, not necessarily restricted to DSPs. The main advantage is high flexibility, since the rule concept allows to capture almost arbitrary idiosyncrasies in the target architecture. Unfortunately, the expressiveness of the rules used in FlexCC is not fully clear from the publications. Generally, it may be expected that the retargeting effort is comparatively high, and that the code quality heavily depends on the suitable specification of machine-dependent code generation rules, since there is not much optimization performed automatically by the compiler kernel.

FlexCC has been retargeted to a microcontroller used in a video telephone chip at SGS Thomson. For some medium size C programs, the compiler generated code of 1 % less size on average compared to the hand-written reference code. Another target for FlexCC was an SGS Thomson fixed-point DSP. For two medium size C inputs, the observed compiler overhead ranged between 0 and 26 %, dependent on the C programming style. The total retargeting effort, including validation and integration, was about 8 person months.

An interesting feature of the FlexWare system is that is also comprises further software development tools beyond the compiler. With this extensive tool support, retargetable compiler technology well supports design space exploration. The system is in in-house use for different processors at STMicroelectronics [123] and is not publicly available. An ongoing project deals with retargeting FlexWare to a new class of network processors. Currently, there is also a shift towards a new FlexCC version, based on ACE's CoSy system described in section 5.6.1.

2.4. SPAM

SPAM is a joint project ("Synopsys, Princeton, Aachen, MIT") focused on retargetable compilers for fixed-point DSPs [124, 129, 131, 127, 130, 125, 128, 126]. It builds on the SUIF compiler (section 5.1.5). The source code can be downloaded for research and development purposes from [132]. The distribution comprises demonstrator backends for the TI C25 and Motorola 56k DSPs.

While SUIF performs C source code analysis and machine-independent IR optimizations, SPAM's retargetable backend library called TWIF is responsible for assembly code generation. TWIF consists of a set of C++ data structures and algorithms that are customizable or parameterizable and hence are capable of generating code for different targets. Among others, TWIF contains the main backend data structures like call graphs, control flow graphs, and data flow graphs for basic blocks, as well as an intermediate format for assembly code.

The SPAM compiler has been designed as a *developer retargetable* compiler. Thus, in contrast to tools like CBC (section 5.2.1) or RECORD (section 5.2.5) there is no homogeneous processor model in some description language, from which the compiler derives the required information. Instead, TWIF offers a suite of retargetable code generation and optimization modules, from which the compiler developer can select the required ones and adapt them to the new target. Hence, focus is on the reuse of source code instead of automatic retargeting support. TWIF contains a number of innovative DSP-specific code optimization techniques, which include:

Code selection for irregular architectures: Like many other compilers, SPAM makes uses of tree parsing for code selection. The OLIVE tool (section 5.5.3.2) is used to generate the code selector source code from a tree grammar specification. While tree parsing originally was mainly intended for regular CISC-like target architectures, it has been pointed out in [129] that an adaptation to irregular machines is relatively easy, if special purpose registers are represented by *dedicated nonterminal symbols* in the tree grammar. The TI C25 target, for instance, has three special purpose registers TR, PR, and ACCU, and using one nonterminal for each register ensures that the required register-to-register moves are automatically generated already during tree parsing. In this way, register allocation is partially coupled with code selection. For a certain, yet narrow class of target machines satisfying the so-called *register transfer graph (RTG) criterion* is has been shown that even optimal, spill-free schedules can be generated with this method. This is illustrated in fig. 5.4, which shows a data flow tree with two subtrees T_1 and T_2 rooted at nodes v_1 and v_2. Suppose that code selection is such that registers r_1 and r_2 are assigned to v_1 and v_2, but also to two further nodes w_1 and w_2 within T_1 and T_2. In this case spill code cannot be avoided, since neither T_1 nor T_2 can be scheduled first without overwriting the opposite subtrees's destination register. The RTG criterion, together with an appropriate scheduling algorithm, ensures that such *deadlock situations* cannot occur. However, optimality here is restricted to the sequential assembly code, while exploiting instruction level parallelism is postponed to a later code compaction step.

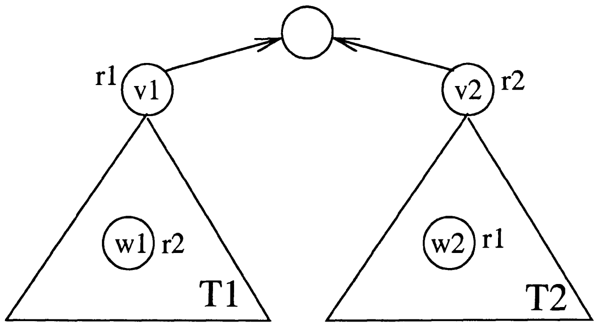


Figure 5.4. Potential register allocation deadlock

Exploitation of dual memory banks: A number of DSP architectures like the Motorola 56k or the Analog Devices ADSP-210x show dual (X/Y) memory banks for sake of an increased memory access bandwidth (fig. 5.5). In order to utilize such an architecture within a compiler, it is necessary to *partition* the program variables between the X and Y memory banks. This is a difficult task, since the partitioning problem in itself is complex, and numerous constraints w.r.t. the register usage and the instruction encoding have to be met. As a consequence, most existing compilers for dual memory bank DSPs (such as the GCC port for the Motorola 56k) simply neglect one of the two banks or leave the partitioning to the programmer by means of *compiler intrinsics*. In contrast, SPAM comprises a variable partitioning module for X/Y memory banks. After a pre-compaction step of the input program, given as symbolic assembly code, memory bank allocation and register allocation take place in a single phase. These problems are mapped to a *constraint graph* labeling problem. The constraint graph nodes represent variables to be mapped to X/Y memory banks or registers. The graph edges are used to reflect both the costs associated with a certain labeling and the code generation constraints imposed by the target DSP. The labeling is performed by a *simulated annealing* optimization algorithm. A problem with this approach is the sometimes huge runtime requirement. Alternative variable partitioning techniques are described in [134, 135, 136].

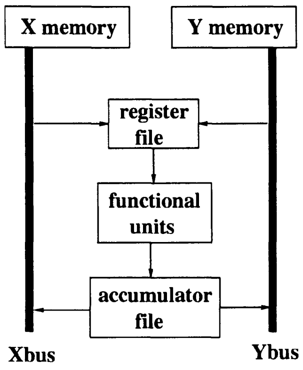


Figure 5.5. Coarse architecture of a dual memory bank DSP

Exploitation of address generation units: TWIF comprises modules for graph-based offset assignment for local variables (section 3.3.4), that can be fully parameterized by the number of available address registers and the auto-increment range [115]. These techniques can also be applied for partial design space exploration [130]. Additionally, the TWIF library offers algorithms for address register allocation within loop bodies [127]. This work has recently been extended towards handling of arbitrary control flow [133].

Besides the above techniques, TWIF offers reusable implementations of several other analysis and optimization passes such as data flow analysis on IR or assembly code, graph coloring register allocation, local code compaction, and exploitation of *zero-overhead loops.*

The SPAM compiler has been retargeted to two standard DSPs (TI C25 and Motorola 56k) and a custom DSP from Fujitsu. As mentioned above, retargeting SPAM does not mean changing some target processor model, but adapting the SPAM backend to the new target, while aiming at the highest possible reuse rate of the TWIF library functions. Software reuse is facilitated by a clean C++ class implementation of the optimization modules that make use of the *virtual function* concept in C++. The developer can derive a new machine-specific class from one of TWIF's optimization base classes, and provide the machine-specific details to the base class methods via its virtual functions. In this way, a relatively high degree of code sharing of about 60 % has been achieved between the three different backends. However, the main limitation of this library reuse approach to retargetability is that the reuse rate will be much lower in case of a new target machine, whose architecture differs significantly from SPAM's previous targets. This is due to the fact that the development of the existing optimization modules has been largely driven by architectural features of specific machines.

The quality of code generated by SPAM has been measured for several small C programs, such as the DSPStone benchmarks [77]. The code size overhead as compared to hand-written assembly typically ranges between 0 and 70 %, which is quite good for a retargetable compiler. The best results have been achieved for the Fujitsu target.

2.5. RECORD

RECORD ("Retargetable Compiler for DSPs") was developed at the University of Dortmund [137, 138, 116, 139, 140] as a successor of the MSSQ compiler described in section 5.4.1. Like CBC, REDACO, and SPAM, it is a retargetable compiler for a class of fixed-point DSPs. In contrast to other approaches, RECORD derives the required target machine information solely from a *hardware description language* (HDL) model.

Like MSSQ, RECORD uses the MIMOLA HDL (see also sections 2.2 and 5.4.1) [141] for this purpose, a language that resembles structural VHDL. A MIMOLA processor model essentially describes the target machine by a hierarchical RT-level (RTL) netlist of components, where leaf components are described behaviorally. Examples are given in figs. 5.6 and 5.7.

```
MODULE ALU (IN i1, i2: (15:0); OUT outp: (15:0); IN ctr: (1:0));
BEHAVIOR IS
 BEGIN
  outp <- CASE ctr OF
            0: i1 + i2;
            1: i1 - i2;
            2: i1 AND i2;
            3: i1;
          END;
 END;
```

Figure 5.6. MIMOLA model of an ALU

In contrast to its predecessor MSSQ, RECORD is not restricted to pure RTL HDL models, but also accepts *behavioral* models and *mixed-style* models. In a purely behavioral processor model, only the target instruction set is described in the form of a single complex MIMOLA component, while hiding the internal RT-structure [142].

The advantage of using an HDL for describing the target processor model is twofold: The user has a high degree of freedom in modeling

```
MODULE Reg16bit (IN inp:(15:0); OUT outp:(15:0); IN enable:Bit);
BEHAVIOR IS
 VAR S: (15:0);
 BEGIN
  IF enable THEN S := inp;
  outp <- S;
 END;
```

Figure 5.7. MIMOLA model of a 16-bit register

the target, due to the expressiveness of the HDL. Dependent on the available documentation (instruction set, RT schematic, or something in between), the most convenient modeling style can be chosen for each target. Additionally, using real HDL models eliminates the need for developing a number of different processor models, and the resulting problems of model equivalence checking: The same HDL model may be used for synthesis, simulation, and code generation.

However, HDL models are sometimes too low-level for retargetable compilation, in particular when specified at the RTL. Therefore, the RECORD compiler comprises an *instruction set extractor*, that converts an arbitrary-style MIMOLA model into a flat list of RT-level assignments by enumerating all possible data transfer paths through the netlist. Simultaneously, the corresponding partial binary opcodes are extracted for each RTL assignment. Internally, these opcodes are represented as Boolean functions by means of Binary Decision Diagrams (BDDs) [143]. In this way, the instruction set extractor can efficiently determine the subset of RTL assignments that are valid w.r.t. the instruction encoding scheme, as well as the groups of RTL assignments that can be scheduled in parallel without encoding or resource conflicts.

An example RT structure is given in fig. 5.8. For instance, computing the sum of the contents of registers R1 and R3 and storing the result in R5 requires a certain setting of specific instruction word bits I.(n) and *mode registers* MR1 and MR2. The instruction set extractor determines the required control bit values and stores the corresponding opcodes or mode register settings. In case *alternative opcodes* are found for the same RTL assignment, all alternatives are kept, so as to obtain higher freedom for the code compaction phase.

The input source language for RECORD is DFL, a data flow language designed for DSPs [144]. DFL shows some resemblence to C, but it has a data flow semantics and provides better support for describing DSP-specific algorithms like filters. The DFL source code is first transformed into an internal control/data flow graph model and afterwards decom-

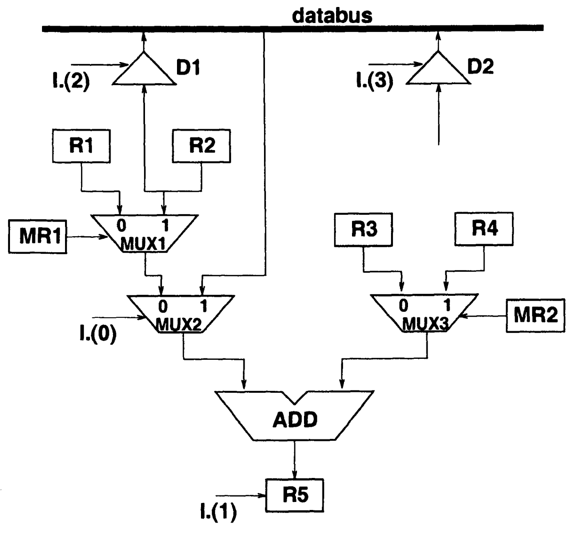


Figure 5.8. Partial RT-level hardware structure

posed into data flow trees. The RTL assignment list delivered by the instruction set extractor is converted into a tree grammar specification for the IBURG tool (section 5.5.3.1), which automatically generates a target-specific code selector. This step has to be performed only once per target architecture.

Like in the SPAM compiler [126], register-specific tree patterns are used to partially integrate allocation of special-purpose registers into the code selection phase. Once all program values have been assigned to specific registers or register files, a local register allocator based on the *left edge algorithm* [67] aims at spill code minimization in the generated sequential assembly code.

RECORD comprises two further DSP-specific code optimization techniques:

Address code optimization: It is assumed that the target processor comprises a parallel address generation unit (AGU) with auto-increment support as explained in section 3.3.4. The AGU needs to be part of the HDL model, and the detailed AGU configuration (number of address registers, presence of modify registers, etc.) is automatically extracted. Similar to SPAM, RECORD uses the available AGU operations for optimized address code generation for local variables and array accesses. For this purpose, several improved optimization techniques have been developed [116, 145, 146]. Address code optimization is applied to the previously generated sequential assembly code with symbolic memory accesses. Afterwards, the generated AGU operations are inserted into the sequential code at the required positions.

Code compaction: RECORD uses a local code compaction technique based on *Integer Linear Programming* [140]. This technique permits optimal compaction of basic blocks, and it is also capable of meeting timing constraints w.r.t. the schedule length. However, due to the high runtime requirements the block size has to be restricted to about 50 assembly instructions. Excessively large blocks have to be split into subblocks before compaction.

The RECORD compiler has been applied to the TI C25 DSP and to several custom DSPs. The behavioral MIMOLA model for the C25 has a size of approximately 900 lines of code. For the DSPStone benchmarks, the code size overhead of compiled code versus hand-written assembly is comparable to that of the SPAM compiler. This is presumably due to the fact that the overall organization of the backend is very similar in both compilers.

The main innovation of RECORD is the instruction set extraction approach which allows to handle mixed-style HDL processor models. Since the compiler can be retargeted very quickly, it allows to perform a limited design space exploration for custom DSPs given as an RTL netlist or an instruction set model, as has been demonstrated in [137]. The main limitations of RECORD are the missing C frontend (there is hardly any DSP software written in DFL), the missing support for multi-cycle instructions, and the lack of machine-independent standard optimizations, which makes the tool less suitable for large programs.

3. Retargetable compilers for VLIWs

3.1. ROCKET

ROCKET is a retargetable compiler for microprogrammed, pipelined architectures. It has been developed at Colorado State University [147, 148] and has its roots in the HORIZON compiler [149] (see also chapter 4). ROCKET has been targeted towards Alpha, i860, and RS/6000. It reads C input programs based on the LCC frontend and first performs several global standard optimizations, including common subexpression elimination, algebraic simplification, as well as constant folding and propagation on an intermediate representation. Alternatively, also a Fortran frontend is available.

After IR optimization, code selection takes place, and data flow graph (DFG) representations for the basic blocks are built. Dependence analysis between the DFG nodes also includes *memory disambiguation*, in order to exhibit more parallelism for the scheduling phase.

The code generator is driven by a machine description that consists of four sections:

1 Resources annotated with timing information

2 Separation of VLIW instructions into distinct fields

3 Machine operations together with their activation patterns from the instruction fields and output assembly syntax

4 A data path description for driving the code selector

As in the Marion compiler, special emphasis in ROCKET is on the coupling of scheduling and register allocation in order to reduce the unfavorable effects of the phase ordering problem ("early" vs. "late" register allocation). In particular, detailed register assignment and spill code insertion are executed only after a code compaction pass, which performs local parallelization of the sequential code. The local scheduler is embedded into a global scheduler that also supports software pipelining.

The core of the register allocator is implemented by a traditional graph coloring approach. Since spill code insertion within compacted code is quite difficult, a feedback loop between register allocation and compaction is used. This avoids false dependencies that usually affect compaction, at the expense of sometimes very high computation time requirements. An alternative approach to phase coupling has been described in [150], which toggles between early and later register assignment dependent on the generated schedule lengths.

More recently, code generation techniques in ROCKET have been extended into different directions, e.g. handling of clustered VLIW architectures with multiple register files [151]. Reported experimental results, however, mainly deal with hypothetical target machines. ROCKET is more a research compiler than a robust tool. Software is available on special request.

3.2. IMPACT

The IMPACT system [152] is mainly an optimizing C frontend, with emphasis on instruction-level parallel (ILP) architectures. The C frontend itself is built upon the EDG [87] frontend. IMPACT includes classical "Dragon Book" [58] optimizations, as well as advanced ILP-improving program tranformations. These include code layout for functions, as well as *superblock* and *hyperblock* formation which increase the scheduling and optimization scope for backends via *predicated execution* support.

There are two levels of program intermediate representation: Hcode (which preserves high-level language constructs) and Lcode (a machine-independent RISC assembly like format). Additionally, there is a more machine-oriented low-level IR called Mcode, which allows for annotating machine-specific information like delay slots of instructions.

IMPACT provides additional support modules like control flow profiling, C source file restructuring, function inlining, and emission of C code as a "backend". There are also backends for real processors like MIPS R2000/3000, Sparc, x86, and i860, that reuse IMPACT's built-in optimizations and whose code quality competes well with native compilers. However, there is no dedicated retargeting formalism like in GCC or LCC.

The IMPACT software can be downloaded from [153]. Supported platforms are HP-UX, SunOS/Solaris, and Linux. The software is generally free for academic, research, and "internal business" purposes. Licenses for full commercial use can be negotiated. There are, however, special terms for different sub-packages used in IMPACT (such as the EDG C frontend source code, the GNU C preprocessor, and a BDD package), which have their own license conditions.

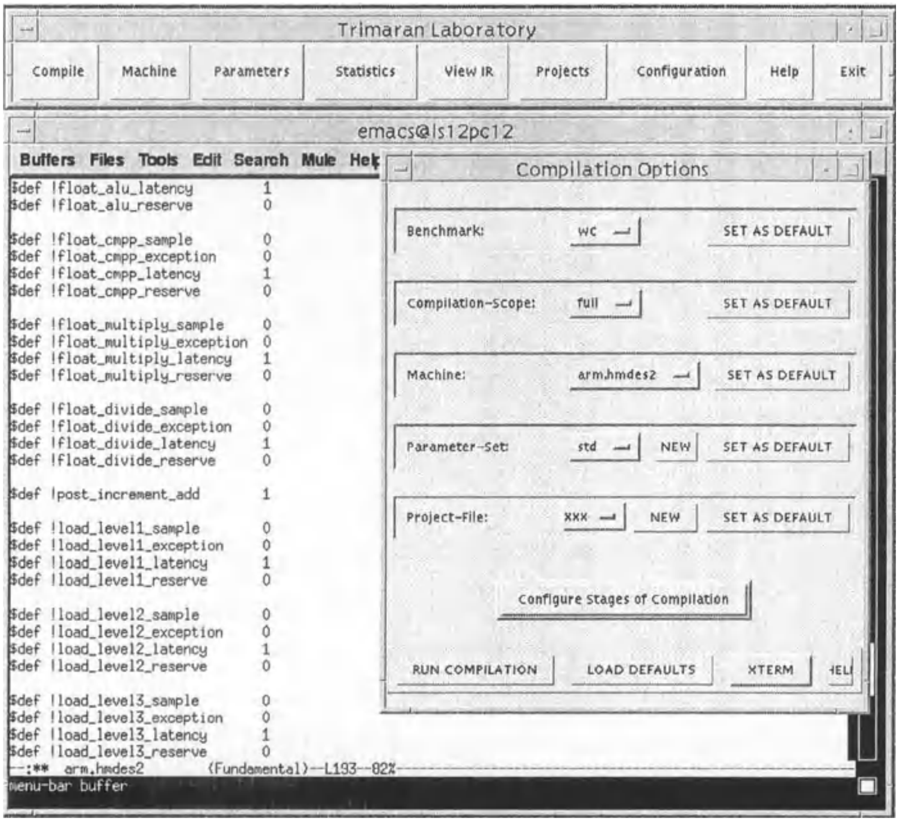
3.3. Trimaran

The Trimaran system, mainly developed at HP Research Labs, is an extensible compiler framework for research on code optimization techniques, with a philosophy similar to SUIF. However, focus is on instruction-level parallel (ILP), VLIW-like processors and backend optimizations. The software, together with comprehensive documentation, can be downloaded from [154] for HP-UX, Linux, and Solaris platforms. The license terms are very similar to those of IMPACT.

A major part of Trimaran is a retargetable C compiler that makes use of the IMPACT system as an optimizing frontend. Additionally, there are simulation and profiling capabilities. The primary target processor class is called HPL-PD, a parameterizable *explicitly parallel instruction computing (EPIC)* meta-architecture (even though the Trimaran distribution also comprises an ARM RISC backend called Triceps). EPIC represents a generalization of VLIW, as for instance also the memory hierarchy is made visible to the compiler.

One main purpose of Trimaran is architecture exploration within the HPL-PD processor class. Therefore, the Trimaran backend ELCOR can be configured in many ways via a machine description file. This includes the specification of register files, number and types of functional units, instruction word length, and instruction latencies. As Trimaran's focus is on ILP processors, dedicated architectural features like *speculative* and *predicated execution*, software pipelining support, and the detailed memory system can be varied by the user. Many other features, like the available instruction set and the controller architecture are largely predefined, though.

The target configuration takes place via a textual description of the architecture parameters in MDES, a relatively complex database oriented language. However, there is a clean procedural interface between MDES and the compiler, and the Trimaran distribution already comes with several example processor models. Simple parameters like the number of registers or instruction latencies can be easily reconfigured via a comfortable GUI (fig. 5.9). Generally, it is recommended to start with modifying an existing machine model, instead of writing new ones from scratch.



```
$def !float_alu_latency         1
$def !float_alu_reserve         0

$def !float_cmpp_sample         0
$def !float_cmpp_exception      0
$def !float_cmpp_latency        1
$def !float_cmpp_reserve        0

$def !float_multiply_sample     0
$def !float_multiply_exception  0
$def !float_multiply_latency    1
$def !float_multiply_reserve    0

$def !float_divide_sample       0
$def !float_divide_exception    0
$def !float_divide_latency      1
$def !float_divide_reserve      0

$def !post_increment_add        1

$def !load_level1_sample        0
$def !load_level1_exception     0
$def !load_level1_latency       1
$def !load_level1_reserve       0

$def !load_level2_sample        0
$def !load_level2_exception     0
$def !load_level2_latency       1
$def !load_level2_reserve       0

$def !load_level3_sample        0
$def !load_level3_exception     0
$def !load_level3_latency       1
$def !load_level3_reserve       0
--:**  arm.hmdes2        (Fundamental)--L193--82%--
menu-bar buffer
```

Benchmark: wc
Compilation-Scope: full
Machine: arm.hmdes2
Parameter-Set: std
Project-File: xxx

Figure 5.9. Session with the Trimaran GUI

The C source program is first compiled into a graph-based intermediate representation, with comprehensive visualization capabilities. ELCOR's code generator is not completely predefined, but is actually structured as a toolbox of modules for control and data flow analysis, ILP-improving transformations (e.g. *if-conversion*), acyclic and loop scheduling, and finally register allocation. Partially, these transformations are already performed by the IMPACT frontend. The desired organization of compiler passes can be configured by means of a script. Similar to SUIF and LANCE, this gives the user the opportunity to add custom optimizations at any time.

Once the compiler has been retargeted, Trimaran's cycle-true simulation and performance monitoring tools allow to estimate the quality of the processor configuration for given application programs. The generated statistics include the cycle count, memory trace, profile information, and resource utilization. The system includes a GUI with extensive visualization capabilities for the result data. In this way, the user gets valuable feedback about the architectural decisions.

Trimaran has been validated with a large set of test programs, many of which belong to the SPEC92/95 benchmark suite [155]. It is based on more than 100 person years of R&D, and there are many installations worldwide. In summary, Trimaran is a comprehensive software package that can be considered an ideal platform for research on retargetable ILP compilers. It benefits from the fact that it focuses on a relatively narrow and parameterizable target architecture class, within which it is easily retargetable and generates highly optimized code. On the other hand, this is also Trimaran's main limitation. It will be very difficult to retarget the tools to some irregular target architecture like a fixed point DSP or a network processor.

3.4. Trimedia

The Trimedia is a fixed/floating point VLIW processor family, originally developed by Philips and since recently by the new company Trimedia Technologies [157]. It is mainly intended for multimedia applications such as video conferencing.

Being a complex VLIW machine with 128 registers and 5 issue slots, the Trimedia is explicitly intended to be programmed in a high-level language. Therefore, the software development kit comes with a C/C++ compiler based on the EDG frontend [87]. The compiler comprises a set of local and global optimizations like constant folding, common subexpression elimination, and *tree height reduction* (especially important for

VLIW). Also if-conversion is performed, as the instruction set is fully predicated.

Since the different Trimedia CPU generations show few differences in the overall instruction set architecture, the C/C++ has been designed to be retargetable through a textual machine description file [158]. A typical machine description file starts with a classification of instruction names into functional groups like load, store, or binary and unary arithmetic. The behavior of the instructions is obviously predefined, as it is not explicitly specified in the MD file. Next, there is a mapping of instructions to the functional units, ordered by latency, which is required for the scheduler. The MD is completed by the number of registers and issue slots, a unit-to-slot mapping, and an enumeration of instruction opcodes.

Being an industrial product, the Trimedia compiler is naturally accompanied by assembler, simulator, and profiler tools. It is a good example of the fact that retargetable compilers are a reasonable approach when different processor generations within a common class of architectures need to be supported, and that retargetability finds its way into industrial practice. Naturally, the architectural scope of the Trimedia compiler is quite limited as compared to research compilers like Trimaran.

3.5. AVIV

AVIV from MIT is a retargetable code generator focused on VLIW architectures [160]. It builds on the SUIF and SPAM compiler infrastructures (sections 5.1.5 and 5.2.4). These frontends transform a given C/C++ input program into a sequence of basic block data flow graphs (DFGs) which form the main input for AVIV.

The target processor is described in the *instruction set description language* ISDL [159], a language designed for modeling processors with instruction level parallelism. An ISDL model comprises six sections:

Instruction format: Similar to the ROCKET compiler, it is assumed that the instruction word is subdivided into several fields, each of which controls a part of the VLIW data path.

Global definitions: In this section, symbolic names (*tokens*) related to the target assembly syntax are defined, e.g. register names and constants. For sake of more convenient use of tokens in the instruction specification, *factoring* by means of nonterminals is also supported, e.g. a set of alternative operands can be factored into a single nonterminal. The nonterminals may also be attributed with *action code*, similar to the UNIX tool YACC.

Storage resources: This section caputures the available registers and memories, together with their size and bit width. Distinct declarations are used for memories, register files, single registers, control registers, the stack, and the program counter.

Instruction set: Based on the declared instruction format and the storage resources, the behavior of instructions is described field by field by means of an RTL assignment notation. Additionally, operation names and cost and timing information for each instruction are captured.

Constraints: It is assumed that all VLIW instruction fields by default can be activated in parallel. However, certain combinations of fields may be excluded due to resource or encoding conflicts. These conflicts are explicitly modeled as Boolean constraints, so as to prevent generation of invalid machine code.

Optional architectural details: This section can be used to capture optional information for the compiler, e.g. concerning instructions with delay slots.

Based on the ISDL target model, AVIV first converts each DFG into a data structure called *split-node DAG*. This is an extension of the original DFG that represents all possible alternatives of implementing the DFG on the given target machine. This is achieved by incorporating two additional node types (besides the given operator nodes) into the DFG: *split nodes* and *data transfer nodes*. Split nodes arise from duplicating all original DFG operator nodes for each functional unit that can implement them. Data transfer nodes represent the possible need for inserting move instructions to shuffle data between the units.

Naturally, the split-node DAG size can be huge for non-trivial basic blocks. Therefore, AVIV uses a number of heuristics to prune the search space. First of all, only a subset of possible DFG coverings is considered in detail. This subset is determined heuristically by using the estimated amount of instruction level parallelism and the required number of data transfers. Then a *clique covering* of the split-node DAG is computed, where each clique corresponds to a set of operations that can be covered by a single VLIW instruction. During this step, illegal instructions that violate the instruction format constraints are also eliminated. For the selected covering, detailed register allocation is performed by graph coloring. The covering algorithm already ensures that no more spill code will be inserted during this phase. On the other hand, superfluous spill and reload instruction may still be present. Therefore, a final peephole optimization is performed that eliminates redundant spills and partially performs a re-compaction of the code.

The main innovation of AVIV is the concept of phase-coupled code generation based on the split-node DAG data structure that implicitly enumerates all possible mappings to the target processor. This helps to reduce the negative effects on code quality usually observed when using strictly separated code generation phases. On the other hand, the large number of heuristics used to prune the huge search space may still compromise optimality. Experimental results have been reported for a hypothetical VLIW architecture, for which good code quality has been achieved within reasonable amounts of compilation time. However, it is not clear how AVIV performs for real-life target processors, which typically show more code generation constraints. Since ISDL requires an explicit enumeration of such constraints, the processor models will also tend to grow quite complex for such machines.

3.6. Mescal

Mescal is a new research project at the Gigascale Silicon Research Center (GSRC) that deals with the development of a software and hardware design environment for programmable processors. The system is intended to support architecture exploration for a set of applications, including *network processing.*

Emphasis in Mescal is also on representation and exploitation of concurrency at different levels of abstraction. As a consequence, a VLIW processor architecture has been chosen for the primitive processing elements. A retargetable compiler is being implemented that maps C programs onto configurable multi-processor VLIW architectures. Similar to Trimaran, the IMPACT system has been selected as the infrastructure for the C compiler.

The Mescal system is currently under development, and further publications or software are not yet available. A project overview and some advance presentations can be found at [161].

4. Retargetable compilers for ASIPs

4.1. MSSQ

MSSQ [162, 163, 164] has been developed at the University of Kiel as the successor of the MSSV compiler (see section 4.2). It is a retargetable compiler for ASIPs modeled in MIMOLA, a hardware description language that has already been exemplified in section 5.2.5. The HDL model has to be given as an RT-level netlist comprising all controller and data path components of the target machine. As an example, figs.

5.10 and 5.11 show the schematic of a very simple target processor and its complete MIMOLA model.

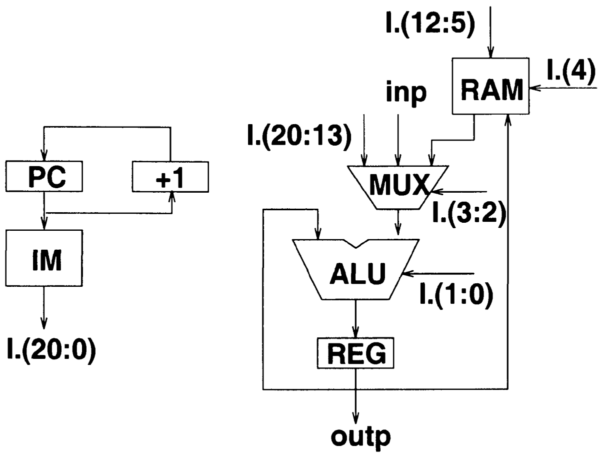


Figure 5.10. Schematic of a simple 8-bit processor. "I" denotes the instruction word (21 bits), whose fields are connected as control and data inputs to data path components.

The user-editable MIMOLA model is internally represented as a *connection operation graph* (COG). The COG nodes represent operators available in the processor hardware, while its edges denote possible data transfer paths via the connections (wires or buses). This COG structure, which partially abstracts from the concrete external MIMOLA model, is used by the MSSQ code generator to find a mapping of a given application program to target machine code.

The source program to be compiled is specified in the MIMOLA *programming langage*, a hardware-oriented superset of PASCAL. Extensions to PASCAL include

- **Predefined variable locations:** The statement
  `VAR x : (15:0) AT Regl;`
  declares a 16 bit variable x located at register Regl.

- **References to physical storages:** Instead of using abstract variables, physical registers and memories can be directly referenced, e.g. in an assignment to some accumulator register:
  `ACCU := ACCU + M[1];`

- **Bit-level addressing:** Subranges of operands may be referenced by appending a bit vector index range. The following assignment loads variable x with the least significant 16 bits of register ACCU:
  `x := ACCU.(15:0);`

```
MODULE SimpleProcessor (IN inp:(7:0); OUT outp:(7:0));
STRUCTURE IS
TYPE InstrFormat = FIELDS      -- 21-bit horizontal instruction word
                    imm:      (20:13);
                    RAMadr:    (12:5);
                    RAMctr:       (4);
                    mux:        (3:2);
                    alu:        (1:0);
                  END;
      Byte = (7:0); Bit = (0);  -- scalar types

PARTS                          -- instantiate behavioral modules
 IM: MODULE InstrROM (IN adr: Byte; OUT ins: InstrFormat);
      VAR storage: ARRAY[0..255] OF InstrFormat;
      BEGIN ins <- storage[adr]; END;
 PC, REG: MODULE Reg8bit (IN data: Byte; OUT outp: Byte);
           VAR R: Byte;
           BEGIN R := data; outp <- R; END;
 PCIncr: MODULE IncrementByte (IN data: Byte; OUT inc: Byte);
          BEGIN outp <- INCR data; END;
 RAM: MODULE Memory (IN data, adr: Byte; OUT outp: Byte; FCT c: Bit);
       VAR storage: ARRAY[0..255] OF Byte;
       BEGIN
        CASE c OF: 0: NOLOAD storage; 1: storage[adr] := data; END;
        outp <- storage[adr];
       END;
 ALU: MODULE AddSub (IN d0, d1: Byte; OUT outp: Byte; FCT c: (1:0));
       BEGIN                  -- "%" denotes binary numbers
        outp <- CASE c OF %00: d0 + d1; %01: d0 - d1; %1x: d0; END;
       END;
 MUX: MODULE Mux3x8 (IN d0,d1,d2: Byte; OUT outp: Byte; FCT c: (1:0));
       BEGIN outp <- CASE c OF 0: d0;  1: d1; ELSE: d2; END; END;

CONNECTIONS
 -- controller:                    -- data path:
 PC.outp       -> IM.adr;          IM.ins.imm    -> MUX.d0;
 PC.outp       -> PCIncr.data;     inp        -> MUX.d1;    -- primary input
 PCIncr.outp   -> PC.data;         RAM.outp   -> MUX.d2;
 IM.ins.RAMadr -> RAM.adr;         MUX.outp   -> ALU.d1;
 IM.ins.RAMctr -> RAM.c;           ALU.outp   -> REG.data;
 IM.ins.alu    -> ALU.c;           REG.outp   -> ALU.d0;
 IM.ins.mux    -> MUX.c;           REG.outp   -> outp;      -- primary output
END; -- STRUCTURE
LOCATION_FOR_PROGRAMCOUNTER PC;
LOCATION_FOR_INSTRUCTIONS IM;
END; -- STRUCTURE
```

Figure 5.11. Complete MIMOLA description of the processor from fig. 5.10

- **Module calls:** Hardware components can be called like procedures with parameters, so as to enforce execution of certain operations. For instance, if the processor description contains a component named `AdderComp`, this component can be "called" in an assignment:

```
x := AdderComp(y,z);
```

- **Operator binding:** Operations can be bound to certain hardware components, e.g. in the assignment

  ```
  x := y +_AdderComp z;
  ```

  the addition is bound to component `AdderComp`.

Since all these extensions are optional, the user can select from a variety of "programming styles", either more abstract or more hardware-specific. Pure PASCAL programs are possible as well as programs close to the machine code level.

Before code generation takes place, the high-level source program is *lowered* down to an RT-level program. All declared user variables are bound to storage modules, and variable references are substituted by references to the corresponding storages. The process of variable binding may be steered through reservations provided by the user. Otherwise, variables are bound to arbitrary storage modules of sufficient capacity. Furthermore, all high-level control structures like FOR, WHILE, and REPEAT loops are replaced by IF-constructs, with explicit reference to the program counter register (PC). The following example shows a piece of source code and the corresponding RTL program:

```
source code:

VAR x, y, z: integer;
REPEAT
  y := y + z;
  x := x - 4;
UNTIL x < 0;

RTL code: (let x, y, z be bound to Mem[0], Mem[1], Mem[2])

lab:
     Mem[1] := Mem[1] + Mem[2];
     Mem[0] := Mem[0] - 4;
     PC := (IF Mem[0] >= 0 THEN lab ELSE INCR PC);
```

The REPEAT/UNTIL loop is replaced by a conditional assignment to the program counter `PC`. If `Mem[0] >= 0` is true at the end of the loop body, then the branch to label `lab` is taken. Otherwise, `PC` is incremented so as to point to the next instruction after the loop. Replacement rules for high-level control structures are contained in an external library, which can be edited by the user. In this way, the most appropriate replacements can be defined for each particular target processor. On a DSP for instance, it might be favorable to replace FOR-loops by hardware loops.

During code generation, the RTL program is mapped to the COG model of the target machine. This involves three main steps:

Code selection and temporary allocation: Each RTL assignment is represented as a data flow tree T. If a subgraph within the COG that matches T can be determined, then the assignment can be implemented by a single instruction. In case the assignment is too complex, it is split into a sequence of simpler assignments, while allocating temporary registers on the fly. MSSQ uses no optimized code selection technique such as tree parsing. However, it ensures by an exhaustive search in the COG that additional cycles due to temporary allocation are only inserted into the generated code in case they are actually required.

Checking for conflicts: It has to be ensured that the selected partial instructions have no conflict w.r.t. resource usage and instruction encoding. For this purpose, MSSQ maintains a data structure called *I-trees* (instruction trees). An I-tree is a representation of alternative partial instruction word settings, where each node represents one partial instruction. All nodes in an I-tree that lie on the same path are combined by AND, i.e. they have to be simultaneously set, while nodes on different paths represent alternatives. Such alternatives frequently arise from the fact that there may be multiple ways of routing data through the data path (e.g. via multiplexers or buses).

The use of I-trees is exemplified in fig. 5.12. Part a) shows an RTL assignment represented as a data flow tree. Fig. 5.12 b) shows two alternative matching subgraphs in the COG, and part c) depicts the corresponding I-tree. The left matching subgraph requires that the two constants "00000001" and "10000000" are provided simultaneously at instruction word bits 13 down to 6, which naturally leads to a conflict. On the other hand, the right subgraph causes no conflict since one of the constants is provided by decoder DEC.

Encoding selection and compaction: The generated partial instructions are passed to a heuristic code compactor aiming at generation of dense schedules in case the target machine shows instruction level parallelism. Possible alternative instruction encodings found in the previous phases are exploited here to select the ones that result in the tightest schedule.

The main strength of MSSQ is that it derives all required target information solely from a homogeneous HDL model written in MIMOLA. This has allowed to retarget the compiler to a large number of different machines, including several ASIPs as well as some standard processors. The large degrees of freedom both in processor modeling and source program specification make MSSQ a powerful tool for design space ex-

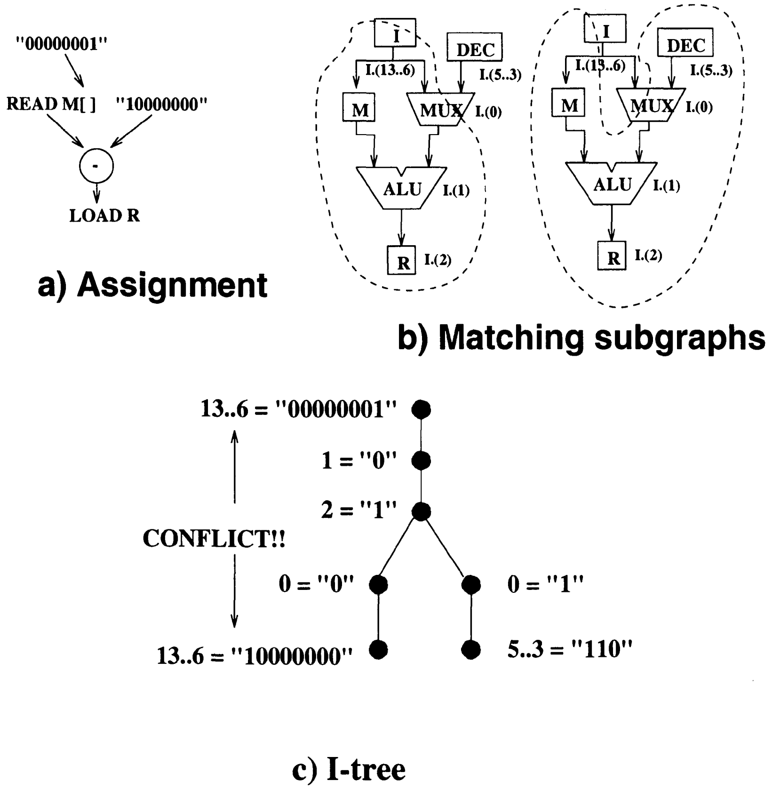


Figure 5.12. Alternative partial instructions and their I-tree representation

ploration during ASIP design, with some emphasis on instruction-level parallelism.

Limitations of MSSQ mainly lie in the range of possible target processors (only single-cycle instructions), sometimes poor code quality due to a simple mapping approach and a missing global register allocator, as well as high compilation times. However, many of these restriction have been removed in MSSQ's successor, the RECORD compiler (section 5.2.5), while retaining the advantages of the HDL model approach to retargetability.

4.2. PEAS

PEAS is a hardware/software codesign project at Osaka University [165]. The main idea is to automatically synthesize ASIPs together with the required software development tools, based on knowledge about applications and design constraints. An overview is given in fig. 5.13.

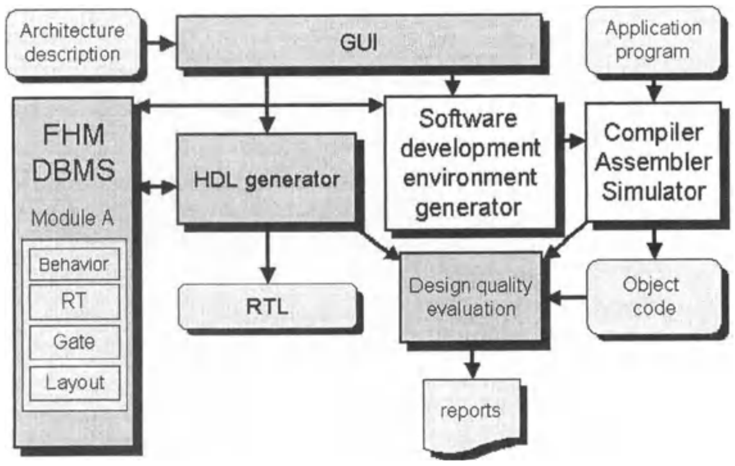


Figure 5.13. PEAS system overview (PEAS-III version), ©PEAS project, Osaka University

Retargetable C compiler generation within the architectural scope of PEAS target processors is also part of this project.

PEAS has evolved over three generations so far. For PEAS-I [166], a RISC based processor kernel with one ALU and a general-purpose register file has been defined, which can be tuned towards certain applications by adding custom functional units and modifying the number of registers. Based on an *application analyzer* module, the processor kernel's instruction set is customized so as to best fit the performance requirements of the intended applications under given area and power consumption constraints. For compiler generation, the GCC compiler has been ported to the RISC kernel. The fine-tuning of the compiler towards a certain ASIP configuration takes place via local modifications in GCC's machine description files, so as to accommodate special instructions and register file sizes.

For the PEAS-II version [167], modifications have been made to the processor kernel, which in this case shows a VLIW architectural style. Still, the instruction set and the pipeline are largely fixed. The application analyzer module uses a branch-and-bound technique to derive an optimum processor configuration w.r.t. the number and types of functional units in the VLIW data path. Like in PEAS-I, the C compiler is based on GCC, which has been enhanced by a dedicated instruction scheduler in order to exploit the available parallelism. However, from the publications it is not fully clear how the compiler supports custom functional units and what the resulting code quality is.

In the most recent version, PEAS-III [168], two important changes have been made. The target processor is no longer based on a kernel with a fixed predefined instruction set, but it can be entirely specified via a GUI in the form of a machine description. This gives more flexibility as compared to earlier versions. The machine description contains the following sections:

1 An *architecture parameter specification* that describes the general architecture type (e.g. VLIW) and the detailed pipeline stages as well as instruction delay slots.

2 A *resource declaration* for registers, ALUs, and their simulation and timing models. Available resources are retrieved from a module library and are configured w.r.t. bit width and functionality.

3 An *instruction set definition* that captures instruction types, fields, and opcodes in a hierarchical fashion.

4 A *micro-operation description* which defines the detailed behavior of instructions, separated into the different pipeline stages.

In contrast to the earlier system versions, the compiler generator in PEAS-III is based on ACE's CoSy system (section 5.6.1). This is possibly due to the fact that dropping the concept of a fixed processor kernel in favor of a more flexible approach complicates retargeting of GCC, which inherently has a preference for RISC architectures. The PEAS-III compiler generator transforms the above machine description format into the backend specification required by CoSy. First, a set of mapping rules for the code selection pass is generated. This is performed by converting the micro-operation description into an intermediate structural format. Resembling the instruction set extraction procedure in the RECORD compiler, this structural model is then used to extract the final instruction patterns. The information required for CoSy's register allocation and instruction scheduling modules are more or less explicit in the target processor model.

Similar to the Trimaran system, the idea of describing the target processor "graphically" via a GUI instead of using a dedicated machine description language certainly provides a comfortable retargeting mechanism. However, so far no detailed results about the compiler generator's target architecture range and the code quality are available. Experience from earlier projects (e.g. MSSQ and RECORD) shows that a very large range of possible targets usually compromises code quality, and, vice versa, high code quality can only be achieved by focusing the target domain. As the project is still under development, it is not obvious which direction will finally be taken in PEAS-III.

4.3. Valen-C

Valen-C ("variable length C") is an extension to the C programming language, dedicated to optimization of ASIP architectures. The corresponding C compiler VCC has been developed mainly at Kyushu University [50]. The software can be downloaded from [169]. It is copyrighted by a Japanese research agency, but it can be virtually freely used under certain preconditions.

The target processor range for VCC is a relatively narrow class of RISC machines. The detailed target instruction set, its register configuration, stack usage, as well as type bit widths and alignment are described in a configuration file.

The main motivation for Valen-C is the fact that in many embedded system designs the processor word length might not be well adapted to the word length required by some given application, which leads to a waste of on-chip memory area. The normal C language does not address this problem, since there are only a small number of type bit widths available, typically 8, 16, or 32 bits. Therefore, the language extensions made in Valen-C allow to specify the *exact bit width* of each program variable, e.g. the type "int23" specifies a 23-bit integer variable. This feature allows the programmer to perform a more detailed design space exploration for ASIPs than possible with a regular C compiler. As the target machine word length may differ from the application program word lengths, the mapping of operations on variables specified with an arbitrary exact bit width to the given target (e.g. 23-bit arithmetic on an 11-bit machine) is a time-consuming task, and in fact it is the main purpose of the VCC compiler to perform this translation automatically. By repeatedly recompiling and simulating some application for different target word lengths, an ASIP can be tuned towards the application (fig. 5.14).

A given Valen-C program is first transformed into an equivalent ANSI C program. Then the SUIF C frontend (section 5.1.5) is used for compilation into the SUIF intermediate representation, and several machine-independent optimizations are performed by means of the SUIF library.

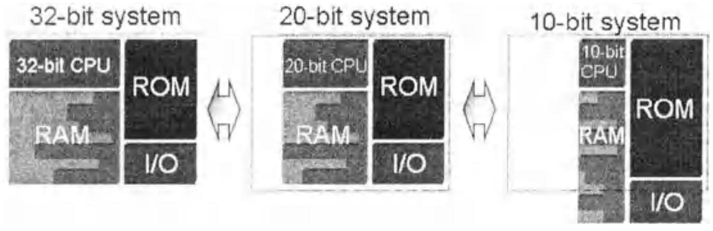


Figure 5.14. Customizing the bit width of ASIP chip components with Valen-C, ©Kyushu University

Next, lowering phases take place during which operations on variables whose width exceeds the target word length are expanded into equivalent instruction sequences that can be executed on the target machine. These phases are followed by code selection, register allocation, and assembly code emission, which are rather straightforward due to the largely predefined target architecture. An instruction scheduler is not included so far.

Since there are already many retargetable RISC compilers such as GCC or LCC, the Valen-C compiler VCC does not bring much innovation from a pure retargetability viewpoint. Moreover, it has a quite limited architectural scope. However, VCC is one of the few compilers that can automatically handle variables with arbitrary bit widths. Due to the automatic translation into assembly code via ANSI C and SUIF, a given Valen-C program can be easily remapped to different memory width configurations of a RISC-like ASIP, and in this way the memory width can be customized so as to minimize on-chip memory area consumption. While reducing the RAM width generally reduces RAM area and the CPU core size, it tends to increase program ROM area, due to the fact that the expansion of instructions on "wide" variables requires more instructions, which in turn implies lower code density. Due to this tradeoff, there is usually some global optimum for total chip size, which VCC can help to determine. Results have been reported for an ADPCM decoder application, where the reduction of the original word length of 32 bits to 18 bits led to a chip area reduction of about 50 %, together with a reduction in energy consumption of 35 % (for more results, see also section 2.2.5).

4.4. EXPRESS

EXPRESS is an ongoing retargetable compiler project at UC Irvine. It is part of a larger project on design space exploration for embedded systems, which also includes design capture, simulation, and *memory hierarchy optimization.* The tools revolve around the architecture description language EXPRESSION [170]. EXPRESSION models can be written manually, or can be generated from a schematic entry GUI called V-SAT.

EXPRESSION processor models are given in a LISP-like notation and show a mixed behavioral/structural style with the following overall structure:

Operations specification: Opcodes, operands, and RTL behavior of each available processor instruction. Concise descriptions are facilitated by factoring constructions for alternative operands.

Instruction description: This section describes the permissible grouping of operations to parallel VLIW instructions. Each instruction is described by a set of issue slots, each of which in turn corresponds to a functional unit.

Operation mappings: Rules in this section specify the mapping of machine-independent operations into machine-specific assembly instructions. Also simple algebraic transformations can be captured.

Components specification: A list of RT-level components of the target processor, annotated with a list of executable operations and timing information.

Pipeline and transfer paths: A specification of pipeline stages, the binding of operations to stages, and an enumeration of valid transfer paths between the RTL components.

Memory subsystem: Storage units like register files, memories, and caches are described separately in this section.

While some concepts are adopted from other languages (e.g. nML and MIMOLA), EXPRESSION innovates in the detailed pipeline and memory modeling. *Reservation tables* needed for instruction scheduling can even be automatically extracted from the processor model [171], which strongly simplifies frequent retargeting during design space exploration.

The EXPRESS compiler [172] itself, which is retargeted based on the information in the EXPRESSION model, appears to be in an early stage yet. An overview is given in fig. 5.15. EXPRESS uses the GCC C frontend and performs different global optimizations like loop unrolling. The backend is built on top of the *Mutation Scheduling* technique [173], which aims at coupling the code selection and register allocation phases with a global instruction scheduler by comparing several alternative ways to map data flow graphs into machine code.

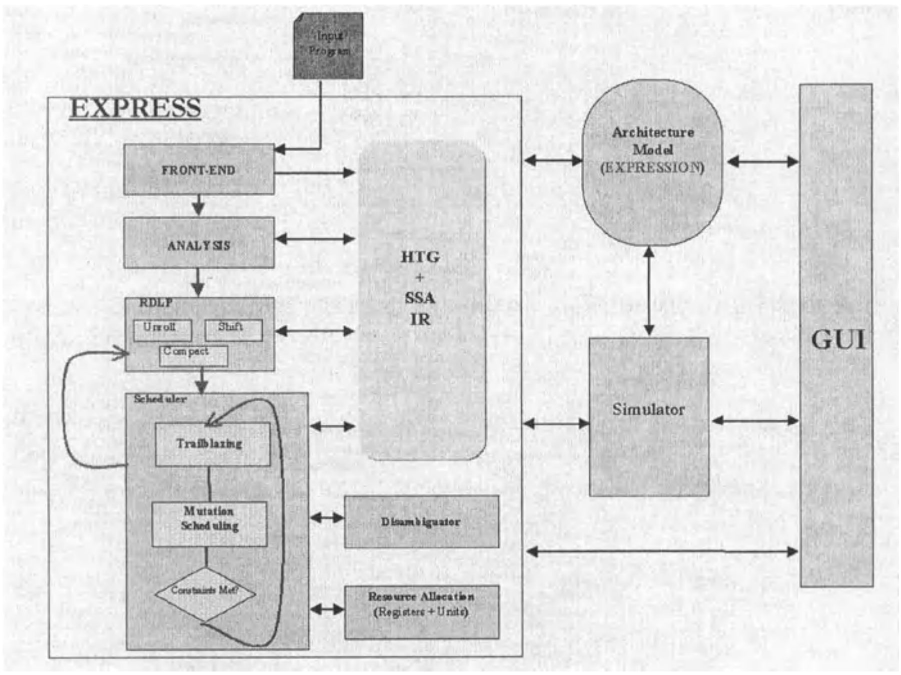


Figure 5.15. EXPRESS compiler overview, ©Center for Embedded Computer Systems, UC Irvine

EXPRESS also emphasizes the phase ordering problem for different code optimizations. It contains provisions for adaptive *dynamic phase ordering*, as opposed to the static phase ordering in traditional compilers. Naturally, this has to be paid with higher compilation times. EXPRESSION models for compilation have been developed for the TI C6x and Motorola 56k DSPs , but experimental results on code quality for these targets have not yet been reported.

4.5. BUILDABONG

BUILDABONG [174, 175] is an ongoing project at the University of Paderborn, aiming at ASIP optimization by architecture/compiler codesign. It is conceptually somewhat similar to EXPRESS and uses an Abstract State Machine (ASM) model of the target processor. BUILDABONG comprises the description language XASM for ASM specification, but XASM models can also be generated from a schematic entry tool. The ASM model has been inspired by the internal processor model used in the RECORD compiler (section 5.2.5). The target processor is considered as a machine which in each instruction cycle executes the same set of parallel *guarded RT operations* of the form

```
if (condition) then <do register transfer>
```

The guard conditions are formed by binary opcodes, modify register states or dynamic conditions (e.g. comparison results). While RECORD extracts this notation from an HDL processor model, BUILDABONG generates it from a schematic and uses an explicit notation of guarded RT operations in the XASM language. The project comprises four main phases:

Architecture description: The target processor architecture is captured as an RT-level structure via a graphical schematic entry tool called ArchitectureComposer. This tool can be downloaded from [176]. The ArchitectureComposer provides a library of parameterizable RTL components like registers, memory, ALUs, and multiplexers which can be instantiated and connected by the user to form an RTL data path netlist (fig. 5.16). An equivalent XASM model can be exported.

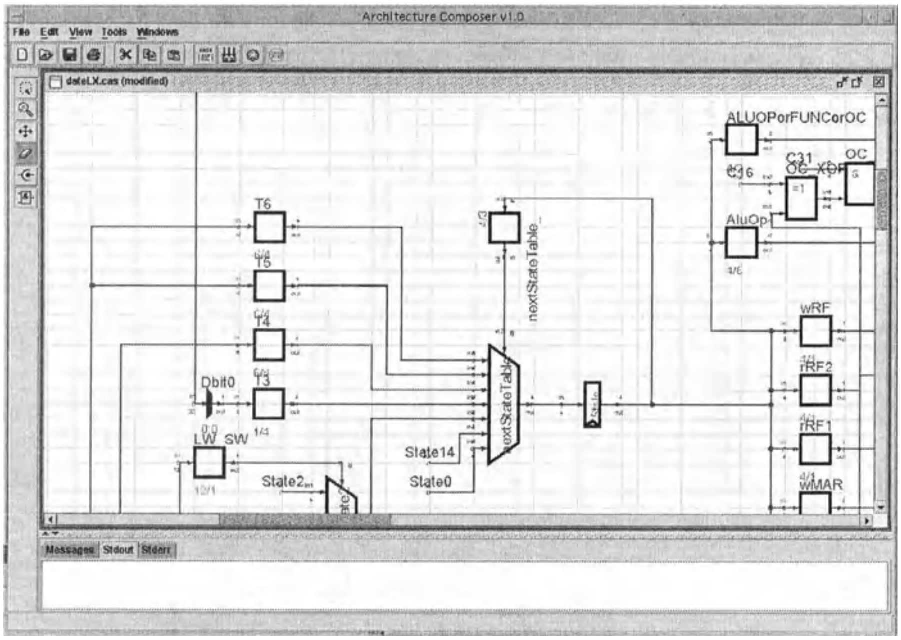


Figure 5.16. ArchitectureComposer tool in BUILDABONG

Simulator generation: The XASM model also forms the basis for generating a target-specific instruction set simulator. For this purpose, the XASM model is translated into C code, which is compiled and linked together with an arbitrary word length arithmetic package and a graphical debugger interface. Also semi-automatic parser generation for assembly program inputs is supported, based on an instruction grammar specification. The debugger can be used to monitor the simulation of guarded RT operations and it displays the current machine state (e.g. register contents or pipeline states). It has been applied to an XASM model of the TI C6x VLIW DSP. However, it is not yet clear how the simulator performs as compared to other recent compiled simulation approaches like [177, 178, 206].

Compiler generation: Generating target-specific compilers in BUILDABONG is still in a research state. The RECORD project showed that the guarded RT operation or ASM notation is suitable for retargetable code generation, and the BUILDABONG compiler is intended to handle a larger scope of architectures, with emphasis on VLIW. It is based on the LCC frontend and an intermediate language called TIL. The target machine is described in MAML (Machine Markup Language). There is also a special script language for specifying high-level optimization strategies. Results on the exact architural scope and code quality have not yet been reported.

Architecture exploration: Just like in related ASIP design and compiler systems like MSSQ, EXPRESS, or PEAS, the ultimate purpose of the BUILDABONG project is to support architecture exploration under certain constraints and optimization goals by means of the generated compiler and simulator tools.

5. Special retargetability techniques

5.1. Code generation methods

5.1.1 Balakrishnan's microcode compiler

Balakrishnan and Bhatt [179] describe a simple retargetable microcode generator. It reads source programs written in a register transfer level (RTL) language and generates machine code for microprogrammed target architectures. Similar to the philosophy of the MSSQ compiler, target machines are supposed to be predefined or synthesized from a behavioral specification. The microoperation set of the target is described in a special language, which permits the specification of resources, instruction fields, and the behavior of resources dependent on instruction field opcodes. From this target model, the code generator first extracts the set of feasible microoperations, which needs to be performed only once per target (this inspired the instruction set extraction tool in the RECORD compiler). As the source program is already given as RTL code, the actual code generation procedure is rather straightforward: The code generator simply scans the extracted feasible operation set for a *match* against each RTL statement. This generates sequential microcode, which is later compacted with standard heuristics to exploit the available parallelism. This approach clearly emphasizes the quick translation of low-level source programs to a class of microprogrammed targets, without particular efforts in code optimization.

5.1.2 Mavaddat's formal language approach

The formal language approach by Mavaddat et al. [181, 182] is based on parsing of data flow graphs (DFGs) w.r.t. to a parallel rewrite system (*Lindenmayer system*) that represents a target data path. This approach can be considered as a generalization of the tree parsing approach described in section 3.3.1. Register transfer operations are modeled as grammar rules, and code generation means constructing a derivation w.r.t. the given instruction set grammar. While tree parsing only works for data flow trees, the main advantage of Mavaddat's approach is that a complete DFG is compiled within a single phase, including register allocation for common subexpressions and local exploitation of instruction-level parallelism. Thus, it achieves a *perfect phase coupling* at the basic block level and generates optimal code.

The major problem with this approach is that parsing for parallel rewrite systems is much more complex than string or tree parsing. It requires an exhaustive search that typically involves backtracking. Mavaddat presents a number of heuristics and pruning rules in order to reduce the runtime requirements. With these techniques, small code generation problems for simple target data paths can be solved within a few CPU seconds. However, the techniques have mainly been designed for microcode generation for synthesized data paths. As a consequence, many features found in realistic targets, such as multicycle instructions or complex addressing modes, cannot be directly handled. Therefore, the formal language approach to code generation is so far mainly of theoretical interest, and results for realistic machines have not been reported so far.

5.1.3 Langevin's automata theoretic approach

The code generation approach by Langevin at el. [183, 184] is conceptually close to Mavaddat's. The target data path is considered as a finite state machine (FSM). Its *initial state* represents the situation that all input registers have been loaded with arguments. Any RT operation (or valid combination of parallel RTs) executed on the data path is equivalent to a *state transition* in the FSM. Finally, the FSM has to be in a *goal state*, where all results of some DFG computation are available in the output registers.

Langevin's approach starts from the goal state and performs a backward state traversal, so as to determine the smallest number of state transitions in the FSM model that leads back to the start state. Effectively, this means to compute the shortest schedule for executing the input DFG on the given data path.

Like in Mavaddat's technique, an ideal phase-coupled code generation is achieved with this method that allows for optimal code quality. Again, the main problem is computation time. Since for each state potentially all predecessor states have to be investigated (which may be hundreds or thousands on non-trivial data paths), the search space gets extremely large. In order to avoid exhaustive runtime requirements, Langevin presents alternative state space exploration strategies, based on *mixed depth first/breadth first* traversal. However, even though high compilation speed is usually given low priority in embedded code generation, it may still be arguable whether compilation times in the order of one CPU hour for a basic block is still acceptable. Similar to Mavaddat's work, the automata theoretic approach provides good insight into phase-coupled code generation for DFGs, but applicability to realistic code generation problems is still very limited.

5.1.4 Römer's automata theoretic approach

Römer proposes an alternative automata theoretic approach in [185, 186]. Like the above approaches, it deals with optimized phase-coupled mapping of DFGs to irregular target data paths. The data path is considered as an FSM that performs state transitions by (potentially parallel) RT operations, and the code generator aims at finding the shortest state sequence that leads from an input state to a goal state where all results are available.

Römer uses an efficient representation of FSM states by a *bit matrix*: For a DFG with n nodes and a target machine with m registers, matrix A contains $n \times (m + 2)$ bits. Any bit a_{ij} denotes that the result of DFG node i is available in register j. Two extra columns are used to represent the fact that some DFG node may be available as an immediate constant or in memory instead of one of the m registers. This representation shows the advantage that state transitions can be simulated by simple Boolean operations, which can be efficiently executed by word-level logical instructions on the compiler host.

Obviously, a shortest schedule is achieved by computing a shortest path from the initial state to the goal state in the state diagram of the underlying FSM. However, the complexity is exponential in the number of DFG nodes, with large constant factors determined by the instruction set size and the number of registers. Therefore, Römer proposes a heuristic solution, where in each step only the M best successor states of a given state are investigated. For the special case $M = 1$, this heuristic is virtually equivalent to the fast standard list scheduling approach. In contrast, larger values of M lead to a more thorough search space exploration at the expense of higher runtimes. In this way, the user can easily study the trade-off between compilation time and code quality.

Like related approaches on formal methods for code generation, the approach is currently restricted to basic blocks. It has been implemented in a C compiler based on the LANCE system (section 5.1.7) and has been applied for mapping DSP routines into assembly code for the M3 DSP platform [187]. The results indicate that the technique works in practice, but detailed results on the code quality/time trade-off (choice of parameter M) still have to be reported.

5.1.5 Wilson's ILP based code generator

The Integer Linear Programming (ILP) approach by Wilson et al. [190, 191] aims at phase-coupled code generation for irregular architectures. The main idea is to transform the code generation problem as a whole into some well-known optimization problem, ILP, for which efficient solvers already exist. This approach first of all avoids the need to design specific optimization algorithms, and there is hope that by exploiting efficient *ILP solvers*, small to medium size problems can still be solved optimally.

ILP can be stated as the problem of maximizing a linear objective function of the form

$$f(x_1, \ldots, x_n) = c_1 \cdot x_1 + \ldots + c_n \cdot x_n$$

under the following system of constraints:

$$\begin{array}{rcl} a_{11} \cdot x_1 + \ldots + a_{1n} \cdot x_n & \leq & b_1 \\ \vdots & & \vdots \\ a_{m1} \cdot x_1 + \ldots + a_{mn} \cdot x_n & \leq & b_m \end{array}$$

All values a_{ij}, c_i, and b_i are real constants, while the *solution variables* $x_1, \ldots, x_n$ have to be integers. Like many important optimization problems, ILP is NP-hard [192], so that optimal solutions most likely require exponential worst case runtime in the input size.

An integrated ILP formulation must comprise solution variables, whose values finally exactly determine the generated code, as well as all constraints, in the form of linear equations and inequations, imposed by the target machine. In Wilson's code generator, the target specific information is stored in a data base, and for each data flow graph (generated by a C frontend) the corresponding ILP model is automatically generated. In more detail, the solution variables account for the control step binding of DFG operations, the instruction pattern selected for each operation, and the registers used for implementing DFG edges, i.e., data transports. The constraints ensure that operations are covered by exactly one pattern, the selection of valid operand and results registers, the correct scheduling order of operations, and other validity conditions. Finally, the objective function minimizes the total execution time of the DFG implementation. The actual machine code can be determined by solving the ILP with some external tool, and deriving the generated instructions.

The ILP approach provides a theoretically elegant way to solve the phase-coupling problem particularly for irregular target architectures. In addition, it is relatively easy to retarget, since the target machine features are captured only in the ILP constraints. However, the main problem is the sometimes extremely high compilation time due to the use of ILP solvers. Therefore, pure ILP approaches to code generation cannot be expected to work for large programs within reasonable compilation time. Consequently, Wilson's technique can only be applied to small pieces of code. Obviously, it has not been integrated into a complete compiler, and experimental results have not been published. Nevertheless, ILP might be a good point solution for heavily optimizing some small "hot spots" of an application program.

5.1.6 FACTS

Researchers at Philips and TU Eindhoven have focused on code generation for in-house DSPs and ASIPs under performance constraints [210, 211]. As the target architecture and the timing of the application are supposed to be given, the code generation problem is considered as a pure *constraint satisfaction problem.* Recent efforts concentrate on the FACTS system [212, 213, 214, 215], a retargetable constraint based compiler project for a class of parameterizable VLIW DSPs. The target machines show a clustered register file architecture with *multi-casting support* for intermediate results, i.e., results can be written into multiple register files in order to partially suppress costly data move operations.

Given some instance of the target processor class with certain resource constraints and an application with a performance constraint, a phase-coupled code generation procedure is performed that consists of the following steps:

Operation assignment: Operations of the input program (given as a set of data flow graphs) are bound to sets of equivalent functional units.

Lifetime serialization: Operations are partially ordered in such a way that the register file capacity constraints are not violated.

Scheduling: Details assignment of operations to control steps.

Binding: Detailed selection of functional units and registers for intermediate values.

These phases cooperate with a *constraint analyzer*: Each time an assignment decision is made, the analyzer checks whether valid code can still be generated under the given timing and resource constraints. In case of a constraint violation, a backtracking step is initiated, and the previous decision is revised. Internally, the constraint analyzer works with graph models and coloring techniques.

The FACTS approach is particularly suitable for DSP applications and architecture exploration. By exploiting knowledge about constraints during code generation, reasonable runtimes (in the order of CPU minutes for large DFGs) are achieved, even though a phase-coupled approach is used. Also the capability of handling multi-casting instructions is very important for clustered VLIW processors. Reported experimental results, however, are still few. It would be interesting to see how the approach performs for standard processors such as the TI C6x VLIW DSP.

5.1.7 Bashford's CLP based code generator

Bashford also considers the code generation problem for irregular target architectures as a constraint satisfaction problem. The approach is somewhat related to the ILP techniques mentioned above, but Bashford uses a PROLOG-like *constraint logic programming* (CLP) language for problem modeling. CLP can be considered more powerful than ILP, since solution variables may have arbitrary domains, and there is support by a real programming language instead of just linear equations.

CLP is particularly suitable for describing code generation problems for irregular targets, such as DSPs, where the instruction set frequently shows special constraints that are not easily captured in some imperative language algorithm. An example is an instruction template of the form

```
X1 := X2 + X3
```

where X1, X2, and X3 represent certain register sets. However, it is usually not possible to use the complete cross product of these sets as argument and result registers in some instance of this instruction. Instead, different constraints have to be obeyed. For instance, if X1 = $\{a_1, a_2\}$, X2 = $\{b_1, b_2\}$, and X3 = $\{c_1, c_2\}$, then the target typically imposes constraints like "if X2 = b_1 and X3 = c_1 then X1 must be a_2" or "X1 = a_1 is only valid if X3 = c_2" and the like. To make things even more complicated, valid register combinations may also depend on other instructions issued in parallel. In Bashford's code generator such instruction templates together with their constraints are concisely represented as *factored register transfers* (FRTs).

The code generation process itself is considered as a *labeling process*, during which solution variables are assigned members of their respective domains, such that some objective function reflecting code quality is optimized. The solution variables account for the FRT instances selected for DFG nodes, the detailed register allocation, as well as scheduling. A key point in this approach is that *alternative solutions* are kept as long as possible, without unnecessarily restricting subsequent code generation phases as in traditional approaches. Decisions are only made in case they are really required, and the built-in constraint solving mechanism ensures that all restrictions imposed by the target machine and the DFG dependencies are met. In this way a high degree of phase coupling is achieved, and heuristics are required only at a few places.

As a consequence, the CLP approach generates quite good code quality, and it can be applied to realistic machines. A CLP based C compiler based on LANCE (section 5.1.7) has been implemented. An experimental evaluation for an ADSP-2100 DSP and some of the DSPStone benchmarks [77] showed that the average performance overhead of compiled code (vs. hand-written reference assembly code) is only 21 %, where the GCC based native compiler produced an overhead of 245 %. Frequently, the quality of hand-written code has been achieved and sometimes even exceeded.

The main restrictions of the approach are its limitation towards basic block level optimization, the somewhat unusual implementation method CLP (which does not easily integrate with other compiler modules), and the high runtimes requirements (up to several CPU minutes) for computing optimal solutions. However, Bashford also proposed a *partitioning heuristic* for large input DFGs. In his experiments, this led to an average reduction in runtime to less than one CPU second at the expense of only a one percentage point loss in code quality.

5.1.8 Yamaguchi's code generator

Yamaguchi et al. [180] present an approach to handle a specific problem in retargetable code generation for irregular architectures: checking for *existence of data transfer paths.* In irregular architectures like DSPs, it is frequently the case that there is no direct route in the data path to transfer a value from one location to another, and even if a route exists, it may still be invalid due to instruction encoding constraints. Yamaguchi's algorithm aims at finding valid mappings of data flow graphs (DFGs) to the target architecture under such constraints. For this purpose, the data path is modeled structurally as a netlist of RTL components and their interconnect. Additionally, the model captures constraints on parallelism. Boolean functions are defined that indicate, e.g., whether two operations cannot be executed in parallel since they share a resource. The code generation procedure comprises three phases. First, alternative bindings of DFG operations to data path resources are computed. All alternatives are implicitly enumerated by constructing a binary decision diagram (BDD [143]) representation of the binding constraints. The second phase maps DFG edges onto possible data paths. Backtracking is employed in case of constraint violations. Finally, the completely bound DFG is scheduled via a list scheduling algorithm, and spill code is inserted in case of register capacity violations, which again might involve some backtracking. In spite of the backtracking, the compilation times are quite low. However, results have been published only for small examples, and it is not clear whether a full compiler has been implemented.

5.2. Retargetable compilers for microcontrollers

5.2.1 Krohm's compiler

A retargetable C compiler for a class of simple application specific microcontrollers is presented in [193]. Its backend has been implemented based on the Graham/Granville *string parsing* approach (see chapter 4). The underlying instruction set grammar is automatically generated from a machine description file, which captures registers, operators, addressing modes, and instruction patterns. Several transformations are performed on the instruction set grammar, such as compaction and ambiguity elimination.

The string parsing based code selection approach does not offer many opportunities for code optimization. Instead, due to the orientation towards control-dominated applications, Krohm's compiler primarily uses a number of efficient *control flow transformations* to achieve good code quality. These include standard techniques like unreachable code and jump chain elimination, as well as optimized linear ordering of basic blocks and exploitation of short jump instructions. The efficacy of these transformations has been demonstrated for several large C application programs. Additionally, the compiler performs pattern-based peephole optimizations and graph-coloring global register allocation.

Krohm's techniques have been implemented in a working compiler system. It is interesting, since in contrast to most other systems it represents a practical retargetable compiler approach explicitly tuned towards microcontrollers. However, the underlying string parsing based code selector is somewhat outdated due to the invention of tree parsing, and code quality results for real machines have obviously not been published.

5.2.2 SDCC

The small device C compiler is a partially retargetable compiler for microcontrollers, where backends are currently available for Intel 8051 and Zilog Z80. The software, including sources, can be downloaded from [194]. It runs under Linux and MS Windows and falls under the GNU public license. The package also contains assembler, linker, simulator and debugger software.

The SDCC compiler performs a number of standard machine independent optimizations, including global common subexpression elimination, loop optimizations, constant folding and propagation, copy propagation, and dead code elimination. The backend comprises a global register allocator and supports *inline assembly* as well as *compiler intrinsics.*

There is no external machine description file in SDCC, but most target-specific information is hard-coded in the compiler. Since the source code is freely available, adaptions towards other microcontrollers are certainly possible. However, the code selection and register allocation modules are rather machine-specific. Only the peephole optimizer is fully retargetable, since it is based on an external description file with simple replacement rules for instruction sequences.

SDCC is a quite comprehensive C compiler infrastructure for microcontrollers, which provides a good starting point for constructing compilers for new targets within its processor class. However, frequent retargeting or architecture exploration are not supported.

5.3. Code generator generators

5.3.1 IBURG

IBURG [74] is an implementation of the tree parsing technique for code selection presented in section 3.3.1. It reads a Backus-Naur specification (fig. 5.17) of a tree grammar for some target instruction set and generates C source code for a target specific code selector.

```
<tree grammar>     ::= { <declaration> } %% { <rule> }

<declaration>       ::=  %start <nonterminal>
                       | %term { <terminal> = <terminal no> }

<rule>              ::=  <nonterminal> : <tree> = <rule no> (<cost>);

<tree>              ::=  <terminal> ( <tree> , <tree> )
                       | <terminal> ( <tree> )
                       | <terminal>
                       | <nonterminal>

<terminal>,
<nonterminal>       ::= <character string>

<terminal no>,
<rule no>, <cost>   ::= <integer>
```

Figure 5.17. Meta-grammar for specification of tree grammars in IBURG

The tree grammar specification starts with a declaration of numbered terminals and the grammar start symbol. The remaining nonterminals need not to be declared explicitly, as they are implicit in the grammar rules. The next section is an enumeration of all grammar rules, given in a simple tree pattern form. Each rule has to have a unique number, which can later be used to identify rules instantiated in a derivation of a data flow tree (DFT) for assembly code emission. Optionally, each rule can also be assigned an integer cost value. Like in LEX and YACC, also regular C code can be included in the IBURG input file, but it must not be interleaved with the actual grammar specification.

Fig. 5.18 shows a simple tree grammar for IBURG. There are 6 terminals, 3 nonterminals ("reg" is the start symbol), and 7 rules. This grammar allows code generation for simple DFTs consisting solely of loads, stores, and some arithmetic operations and constants.

The C output generated by IBURG for this grammar comprises a number of tables and functions. The tables store information like grammar symbol numbers, target nonterminals of rules, operator arities, as well as some optional debug information. The main interface to the compiler driver program is the generated "burm_label" function, which takes some DFT as an input parameter. The DFT data structure can be defined nearly arbitrarily, as IBURG accesses all relevant information via editable C macros.

```
%start reg
%term Assign=1 Constant=2 Fetch=3 Four=4 Mul=5 Plus=6
%%
con:  Constant                = 1 (0);
con:  Four                    = 2 (0);
addr: con                     = 3 (0);
addr: Plus(con,reg)           = 4 (0);
addr: Plus(con,Mul(Four,reg)) = 5 (0);
reg:  Fetch(addr)             = 6 (1);
reg:  Assign(addr,reg)        = 7 (1);
```

Figure 5.18. Sample IBURG tree grammar

```
switch (op) {
...
case 6: /* Plus */
        assert(l && r);
        if (    /* addr: Plus(con,Mul(Four,reg)) */
                r->op == 5 && /* Mul */
                r->left->op == 4 /* Four */
        ) {
            c = l->cost[burm_con_NT] + r->right->cost[burm_reg_NT] + 0;
            if (c + 0 < p->cost[burm_addr_NT]) {
                    p->cost[burm_addr_NT] = c + 0;
                    p->rule.burm_addr = 3;
            }
        }
        {       /* addr: Plus(con,reg) */
            c = l->cost[burm_con_NT] + r->cost[burm_reg_NT] + 0;
            if (c + 0 < p->cost[burm_addr_NT]) {
                    p->cost[burm_addr_NT] = c + 0;
                    p->rule.burm_addr = 2;
            }
        }
        break;
...
}
```

Figure 5.19. C output fragment for sample grammar

Fig. 5.19 shows a fragment of the generated C code which is responsible for matching any DFT T with a "Plus" terminal at its root. First it is tested whether the "multiply-accumulate" rule (number 5 in fig. 5.18) can be applied by checking T's right kid. If the rule matches, then the costs of rule 5 are summed up with the accumulated costs of T's kids when reducing these to nonterminals "con" and "reg", respectively, to form a new cost value *c*. If *c* turns out to be lower than the current cost value for reducing *T* to the target nonterminal "addr" (stored in "`p->cost[burm_addr_NT]`"), then *c* is recorded as the new minimum cost. The remainder of the C code performs a similar processing for rule number 4.

Normally, the user does not have to care about such details of the generated code, except that debugging becomes necessary in case of unexpected behavior of the code selector during the development phase. The result of the code selection process is a derivation tree for the input DFT, which can be traversed in a subsequent pass to emit assembly code. However, this is not directly supported by IBURG.

IBURG is a very compact, efficient, and stable tool. It eliminates the need to implement tree parsing based code selection from scratch and it supports quick retargeting to different instruction sets. However, its functionality is restricted to code selection only. Among others, IBURG has been used in the RECORD compiler, where tree grammar specifications are automatically generated from hardware description language models. The software including source code is available at [195]. Its use is free for research purposes. The generated C output code may also be used in products, provided that the code is delivered at no charge.

5.3.2 OLIVE

OLIVE is an improved version of IBURG. It has essentially the same functionality but includes several important enhancements that make code selector development more comfortable.

First, the cost attributes of grammar rules are no longer restricted to integer constants, but *arbitrary cost functions* may be used instead. While in many cases constant costs still suffice, there are special applications where cost functions are more powerful. One example is to dynamically enable or disable certain rules during multi-pass code generation [196]. Rules can be conditionally disabled by assigning an "infinite" cost value, which inhibits the selection of a rule even though it matches some DFT.

Another important improvement over IBURG is the introduction of *action functions* in OLIVE. An action function is a piece of C code that is executed each time some rule has been selected during tree parsing. This works as follows: Normally, the code selector makes two passes over a DFT. After computing the optimum DFT cover in the first pass by means of the "burm_label" function (as in IBURG), the second pass is explicitly invoked by calling the action function for the start symbol. Typically, this function will in turn contain calls to the action functions for the kids, and this process recursively continues until all DFT nodes have been visited again.

The most important purpose of action functions is the emission of assembly code, even though also other phases like local register allocation can be integrated here. In contrast to IBURG, which completely decouples code emission from the parsing phase, this permits a cleaner, syntax-driven code generation approach.

```
%term AND
%declare <char*> reg;

reg: AND(reg,reg)
{
  $cost[0] = 1 + $cost[2] + $cost[3];
}
=
{
  char* vreg1, *vreg2, *vreg3;
  vreg1 = $action[2]();
  vreg2 = $action[3]();
  vreg3 = NewVirtualRegister();
  printf("AND %s,%s,%s",vreg1,vreg2,vreg3);
  return vreg3;
};
```

Figure 5.20. Use of actions functions in OLIVE

Fig. 5.20 shows a tree grammar fragment that outlines the use of action functions. Like in IBURG, all terminals have to be declared. OLIVE additionally requires a declaration of nonterminals, where this also implicitly declares the corresponding action function interface. In the example, nonterminal "reg" is declared in such a way that its action function is a parameterless C function returning a string. This string is used to identify a virtual register name, so that different instances of using nonterminal "reg" can be distinguished.

The sample grammar rule is used to match "AND" operations in a DFT. The arguments must reside in registers, and also the result will be written to a register. Similar to IBURG, the cost part sums up the inherent rule cost (here 1) and the cost of the subtrees, so as to induce a quality metric for the code selector. The action function (following the "=" character) looks like a regular C function. It calls the action functions of the subtrees to get the associated virtual register names. Next, a new unique virtual register is allocated for the result, and an assembly instruction is emitted. The name of the result register is returned for subsequent use upwards in the DFT.

Like IBURG, OLIVE is a stable and efficient tool. Its extensions over IBURG make the specification of code selectors quite comfortable. The OLIVE tool including the source code is included in the distribution of the SPAM compiler [132], of which it forms a central component. There it has been used to develop several DSP backends. Also the LANCE system provides an interface to OLIVE based code selectors. The license conditions for OLIVE are the same as for IBURG.

5.3.3 BEG

BEG is a backend generator developed at the University of Karlsruhe [197]. It generates code selectors and register allocators for a target machine modeled by a code generator description (CGD) file. Instruction scheduling is obviously not included. The backend source code is emitted in Modula-2 or C.

Like IBURG and OLIVE, BEG generates a tree parser for the code selector. A CGD therefore contains rules for tree pattern matching, cost attributes, as well as assembly code templates to be emitted in case of a match. Additionally, matching conditions can be specified, e.g. the permissible range of integer constants as immediate operands of instructions.

For the register allocation part, BEG first requires an enumeration of all available registers. The set of admissible registers for instruction operands and destinations can be annotated at each rule. In addition, special rule attributes can be used to inform the register allocator that some register besides the destination is changed as a side effect, in which case that register gets spilled. BEG generates two alternative register allocator variants: "general" and "on-the-fly". The general allocator is slower but handles a wider range of target machines, while the faster allocator may fail in case of too complex expressions due to restricted spilling capabilities. Both register allocators are very local, as they process only one data flow tree at a time.

The functionality of BEG somewhat resembles that of LCC, even though many implementation details are different and BEG has its roots in Modula-2 rather than C. Like LCC, BEG requires a number of handwritten support routines for the backend. Unfortunately, BEG itself has no source language frontend. On the other hand, BEG permits the specification of the intermediate representation as a part of the CGD, and several things like constraining the valid register classes for instruction patterns are certainly easier than in LCC.

BEG has been used in backend generation for MOCKA, a popular compiler for Modula-2. Additionally, BEG has been the basis for the backend generator of the CoSy system (section 5.6.1). The original version of BEG is copyrighted by GMD Karlsruhe, and binaries for Linux are available at [198]. The package also contains example CGDs for IBM 370 and Motorola 68020 targets. However, it is not clear whether BEG is still supported as a stand-alone tool.

5.4. Assembly-level optimization

5.4.1 SALTO

In contrast to most other systems presented here, the SALTO system [199] developed at INRIA/France is focused on retargetable *assembly-level optimizations.* It does not include a source language frontend nor a built-in library of optimizations, but it offers the user a framework for implementing his own machine-dependent code transformation and optimization tools. The architectural scope of SALTO is mainly in the area of VLIW processors.

Retargeting SALTO works via a LISP-like machine description language, which has been inspired by GCC's machine description format. It specifies the available resources (i.e. anything that has an impact on scheduling), reservation tables for modeling pipeline behavior, and assembly language syntax. As a demonstrator, a model for the Trimedia TM1000 VLIW DSP (section 5.3.4) has been developed.

From the machine specification, an *assembly language parser* can be automatically generated. The parser generates an internal program representation, which can be accessed via the SALTO API. The API gives application programs access to data structures for control flow graphs, basic blocks, and single instructions. The data structures can be attributed with application-specific information.

SALTO has been used to implement some simple optimizations like list scheduling. It also serves as a basis for research projects on the trade-off between software pipelining and loop unrolling as well as general phase coupling. Additionally, a *compiled simulation* generator based on SALTO has been developed [200], which has been verified for the TM1000 and generally outperformed the native interpretive simulator.

The SALTO approach is interesting, since it allows to develop and evaluate machine-specific optimizations and phase orderings at a reduced effort. However, for architecture exploration a compiler would be required that at least generates valid, unoptimized assembly code from a programming language like C.

5.4.2 PROPAN

Like SALTO, the PROPAN system from Saarland University [201, 202] is intended for retargetable assembly-level code optimization. However, PROPAN is more oriented towards DSP targets, and similar to Wilson's approach mentioned in section 5.5.1.5 it comprises Integer Linear Programming (ILP) based optimization routines. There is also some conceptual similarity to the Zephyr/VPO approach (section 1.6), but PROPAN incorporates highly machine specific optimizations. An overview is given in fig. 5.21.

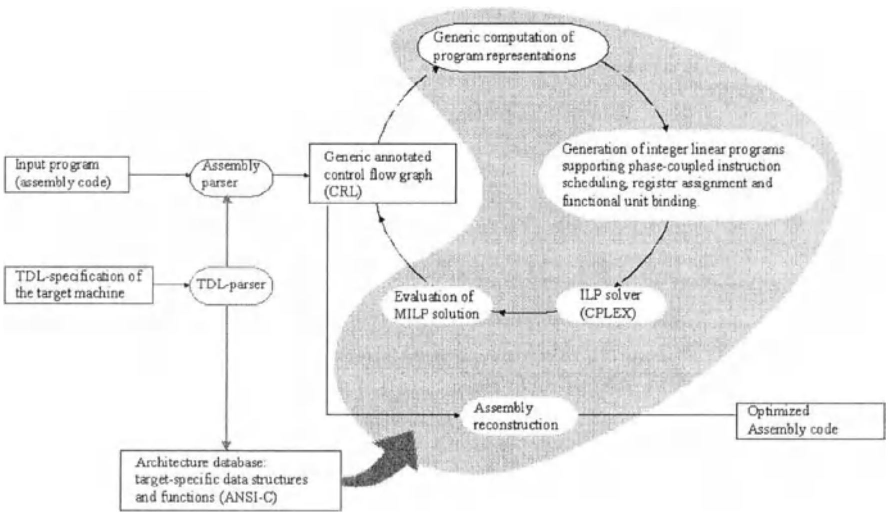

Generic computation of program representations
Input program (assembly code)
Assembly parser
Generic annotated control flow graph (CRL)
Generation of integer linear programs supporting phase-coupled instruction scheduling, register assignment and functional unit binding
TDL-specification of the target machine
TDL-parser
Evaluation of MILP solution
ILP solver (CPLEX)
Assembly reconstruction
Optimized Assembly code
Architecture database: target-specific data structures and functions (ANSI-C)

Figure 5.21. PROPAN overview (taken from [203])

One important motivation for moving retargetability from the compiler down to the assembly level is that existing (but possibly poor) compilers can be reused, while focusing only on the machine-dependent optimization problems. Moreover, several optimizations that normally take place at the intermediate representation level should be repeated at the assembly level anyway. A typical example is loop-invariant code motion, which cannot be fully performed in a machine-independent fashion.

In PROPAN, target machines are modeled in the *target description language* TDL. This language permits modeling of resources like functional units, registers, and memories, as well as a large range of possible assembly instruction formats, including assembler directives and complex algebraic assembly notations. Additionally, TDL captures instruction behavior by means of a special RTL language as well as explicit constraints on instruction-level parallelism. From a TDL model, PROPAN automatically generates an assembly language parser, as well as an architecture data base that stores the target-dependent information.

The generated parser translates a given input program into an internal control/data flow graph structure, which forms the basis for optimization routines. First of all, PROPAN is capable of *register renaming* to remove false dependencies in the initial assembly code. The core optimizations are based on two different ILP approaches for phase-coupled code generation called SILP and OASIC. While other ILP approaches to code optimization are mostly restricted to basic blocks, PROPAN performs optimizations within *superblocks* than span multiple basic blocks. This opens up higher optimization opportunities.

The system has been evaluated for two real-life targets: the Analog Devices ADSP-2106x SHARC and the Philips Trimedia TM1000. Additionally, PROPAN has served as a platform for an industrial postpass optimizer for the Infineon C16x microcontroller. The results for the ADSP-2106x and the TM1000 show that the ILP approaches result in lower code size than standard heuristics such as list scheduling. However, the average improvement is moderate (less than 10 %), and the required computation times are generally quite high, due to NP-completeness of the ILP problem. Nevertheless, the additional computation time may sometimes be justified for the critical loops of embedded applications. Moreover, some reduction is possible by replacing the exact ILP solving with approximation methods, which still yield optimal solutions in many cases.

5.5. LISA

The LISA language [204, 205] developed at ISS, TU Aachen is a processor description language mainly used for retargetable generation of software development tools. LISA has been influenced by nML (section 5.2.1) but it provides some important improvements that enhance its practical applicability. First of all, it allows for cycle-accurate modeling, since the detailed pipeline behavior of the target processor is captured. Secondly, it includes C elements, which facilitates the language use as well as generation of fast compiled simulators [177, 206].

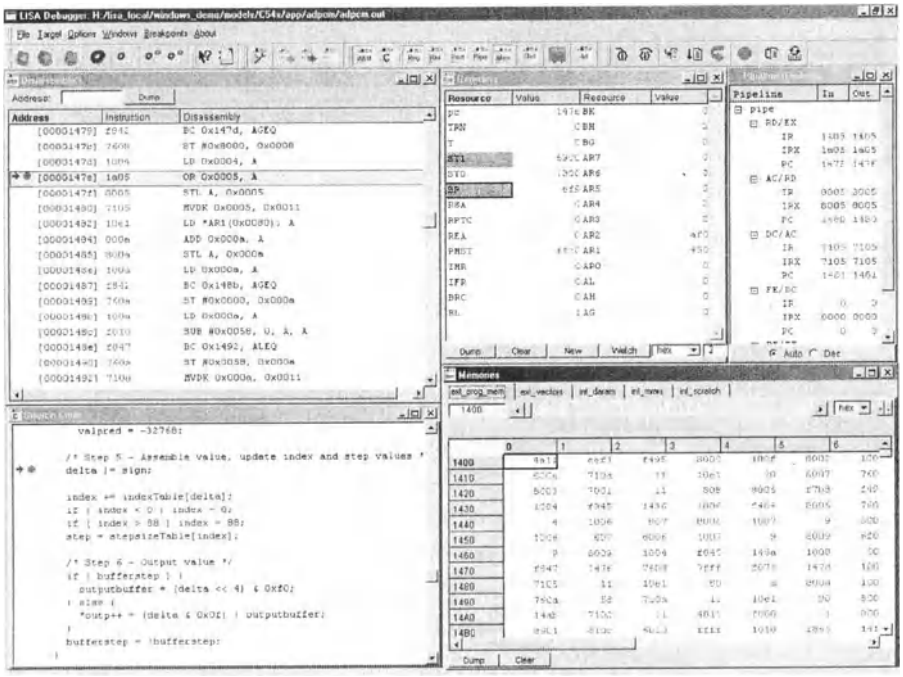
Similar to nML, a LISA model consists of *resource descriptions* and a list of *operations*. The resource section is used to declare available memories, registers, and auxiliary global variables used for sake of simpler descriptions. Additionally, a detailed model of (possibly multiple) pipelines is provided, including *pipeline stages* and registers. Any LISA operation describes a primitive processor operation, e.g. a machine instruction at a certain pipeline stage. This includes instantiation of sub-operations (with factoring capabilities like in nML), a description of the assembly syntax, and the binary encoding. An operation is mostly also attributed with a piece of *behavior*, which is modeled in regular C. Fig. 5.22 shows an example operation from a LISA model of the DLX RISC processor.

```
// logical shift left bound to execute stage
// of the instruction pipeline

OPERATION SLL IN pipe.EX
{
  // instantiation of sub-operations

  DECLARE
    {
      GROUP rs1, rs2, rd = { reg32 };
      INSTANCE r_ralu;
    }

  // binary encoding

  CODING { r_ralu rs1 rs2 rd 0bx[5] 0b110010 }

  // assembly syntax

  SYNTAX { "SLL" rd "," rs1 "," rs2 }

  // effect on machine state

  BEHAVIOR
    {
            temp=(unsigned int)rs1;
            rd = temp << ( rs2 & 0x0000001f);
            rs1=(int) temp;
    }
}
```

Figure 5.22. Sample LISA operation

Operations may be specified at the level of machine instructions. However, for pipelined processor models, typically a more fine-grained structure is used that e.g. subdivides instruction execution into fetch, decode, execute, and write-back operations, each of which is explicitly bound to one of the pipeline stages declared in the resource section. In order to support such models in a simulator, LISA provides explicit pipeline control commands, e.g. for flushing, shifting, or stalling some pipeline.

The LISA language is supported by a number of tools that take a given processor model as an input. First of all, there is a LISA model debugger, which helps to develop correct models for a given initial specification.


LISA Debugger: H:/lisa_local/windows_demo/models/C54x/app/adpcm/adpcm.out
File Target Options Windows Breakpoints About
Address:
Dump
Resource
Value
Pipeline
In
Out
Memories
Dump
Clear
New
Watch
hex
Auto
Dec

```
Address         Instruction   Disassembly
[00001479]      f842          BC 0x147d, AGEQ
[0000147b]      7608          ST #0x8000, 0x0008
[0000147d]      1004          LD 0x0004, A
[0000147e]      1a05          OR 0x0005, A
[0000147f]      8005          STL A, 0x0005
[00001480]      7105          MVDK 0x0005, 0x0011
[00001482]      10e1          LD *AR1(0x0080), A
[00001484]      000a          ADD 0x000a, A
[00001485]      800a          STL A, 0x000a
[00001486]      100a          LD 0x000a, A
[00001487]      f842          BC 0x148b, AGEQ
[00001489]      760a          ST #0x0000, 0x000a
[0000148b]      100a          LD 0x000a, A
[0000148c]      f010          SUB #0x0058, 0, A, A
[0000148e]      f047          BC 0x1492, ALEQ
[00001490]      760a          ST #0x0058, 0x000a
[00001492]      710a          MVDK 0x000a, 0x0011
```

```
        valpred = -32768;

      /* Step 5 - Assemble value, update index and step values *
      delta |= sign;

      index += indexTable[delta];
      if ( index < 0 ) index = 0;
      if ( index > 88 ) index = 88;
      step = stepsizeTable[index];

      /* Step 6 - Output value */
      if ( bufferstep ) {
        outputbuffer = (delta << 4) & 0xf0;
      } else {
        *outp++ = (delta & 0x0f) | outputbuffer;
      }
      bufferstep = !bufferstep;
    }
```

Figure 5.23. LISA debugger GUI, ©ISS, TU Aachen

Once the model has been debugged, assembler, linker, and simulator tools are automatically generated. Generated simulators can be linked to a debugger GUI (fig. 5.23), which provides functionality like source-level stepping, breakpoints, internal state monitoring, and profiling.

Current R&D activities at TU Aachen aim at extending the LISA language applicability into further directions (fig. 5.24), including compiler generation, HDL model synthesis, and co-simulation. In this way, a complete high-level processor design and architecture exploration tool suite will be provided. Prototypes are already in industrial use, and tools will be productized by LISATek Inc. [208]. A different LISA dialect is also being used in automatic simulator and assembler generation tools at Axys Design Automation [207].

Even though compiler generation from LISA is so far not supported and certainly requires some language extensions, we find the language worth mentioning here due to its large practical impact. Besides its use in the Axys tools, LISA has been used to model a variety of real-life processors, including RISCs (e.g. ARM 7 TDMI, MIPS 4k), DSPs (e.g. TI C6x, C54x, Motorola 56k), and microcontrollers (e.g. Intel 8051) at different levels of accuracy, and software development tools, including

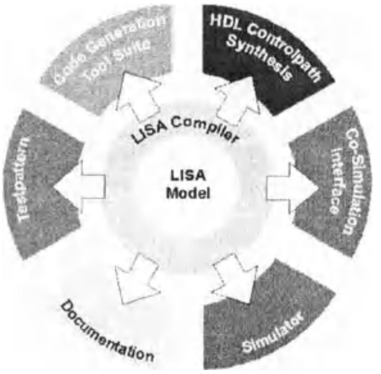


Figure 5.24. LISA development tools, ©ISS, TU Aachen

high-speed cycle-accurate simulators, have been automatically generated. This indicates that LISA is among the most flexible and mature processor description languages currently available.

5.6. Compilers for industrial reconfigurable cores

Retargetability can be put into practice quite easily in case of a very narrow target processor domain. More and more *IP vendors* selling reconfigurable processor cores realize that software development tools like compilers have to be flexible (or retargetable) enough to cope with different instances of a core, so as to give the highest benefit to the customer.

While processor cores that are reconfigurable by some numerical parameters (register file size, bus width, etc.) have been available for some years, only recently IP vendors have started offering processor cores with a customizable instruction set. This concept permits the user to extend the standard core architecture, as it is shipped by the vendor, by new application specific instructions. Sometimes, such instructions are specified through a dedicated instruction modeling language, while in other cases templates are provided that at least facilitate the integration of new instructions into the compiler, simulator, and the underlying HDL synthesis model.

Examples for cores with extensible instruction sets are Tensilica's Xtensa RISC core [80] 3DSP's SP-5 Flex DSP core [209], and ARC Cores' Tangent RISC core [79], all of which come with a retargetable C compiler (Tensilica's compiler is GCC based). A still open problem is the automatic exploitation of new custom instructions in the compiler. Ideally, the compiler should recognize the semantics of new instructions and optimally exploit them during code generation. However, currently available systems are not capable of this and rather use a pragmatic approach, where new instructions are made known to the compiler via *compiler intrinsics.*

5.7. Retargetable test program generation

The RESTART system by Bieker [216] represents a very special type of a retargetable compiler: a retargetable *self-test program generator.* The system exploits the fact that programmable processors may perform self-tests for their RTL components when using appropriate test programs. RESTART generates binary test code for test programs specified in a high-level language.

Like in the MSSQ and RECORD compilers (sections 5.4.1 and 5.2.5), the target processor is described in the MIMOLA hardware description language. The range of possible target processors is very similar to MSSQ. For the specification of test programs, RESTART uses a custom language called TCL (*test code language*). TCL comprises constructs for testing sequential components (e.g. to test whether a certain constant that has been written into a register has been correctly stored) and combinational components. For instance, the TCL statement

```
TEST alu(%00,%0001,%0011);
```

translates into machine code that supplies the binary inputs "%00", "%0001", and "%0011" to an RT component "alu" (specified in the MIMOLA model) and checks whether the expected output is computed. The expected output of "alu" is automatically determined by a built-in RTL simulator. In case of a test failure, a jump to a specified error label is performed, otherwise the program continues with the next test statement.

For sake of more concise test program specifications, TCL also comprises FOR loops. For instance, the loop

```
FOR i := 1 TO 1000 DO TEST ram[i] := #ffff;
```

will be compiled into test code that writes the hex value "#ffff" into the cells 1 to 1000 of memory component "ram" and tests (by means of a comparison instruction) whether the value has actually been stored.

Although TCL is a quite simple programming language, the code generator is relatively complex. Since the TCL statements typically do not allow for a one to one mapping into machine instructions, code selection, register allocation, and scheduling have to be performed as in any other compiler. In addition to binary test programs, RESTART also generates external stimuli files for values that cannot be provided by the processor hardware itself.

The RESTART system has been implemented with *constraint logic programming* (CLP, see section 5.5.1.7). This resulted in a very compact implementation (only about 25 % source code as compared to a traditional implementation in an imperative language). It has been integrated with logic synthesis, test pattern generation, and fault simulation tools [217]. For several simple target processors, a fault coverage between 95 and 99 % has been observed.

The main achievement of RESTART is that test program generation from a high-level language is largely automated. Due to the special application domain of test generation, there are only few relations to other retargetable compilers. However, self-test program generation is certainly an interesting niche application of retargetable compiler technology. The experimental results indicate that the technique works in practice and, in combination with complementary design and test tools, may lead to a good fault coverage for programmable processors.

5.8. Retargetable estimation

As outlined in chapter 2, one main application of retargetable compilers is architecture exploration, so as to determine an optimum target processor architecture for a given set of applications. If the target processor class is DSP, the design of efficient and retargetable compilers is a difficult problem, though. Ghazal et al. [218, 16] propose to circumvent this problem with a *retargetable estimation* methodology. They developed a tool suite, based on the SUIF frontend (section 5.1.5), for predicting the optimum performance of an application on a given target processor. The underlying processor description only comprises parameter tables instead of a detailed architectural model. These parameters include:

- Functional units and restrictions on parallelism
- Instruction set, incuding complex instruction like MAC
- Memory data packing/unpacking support
- Memory addressing support, like auto-increment
- Control flow support, such as zero-overhead loops
- Loop optimizations, like software pipelining

The input source code is processed by SUIF, which also performs standard and loop-level optimization on the intermediate representation. The processor parameters are then used to compute the estimated performance for a close-to-optimal (i.e. hand-written) mapping of the source code into machine code, based on static code analysis and profiling.

The retargetable estimation methodology has been applied to two DSPs (ZSP 16401 and TI C6201) and has been shown to achieve high accuracy (5 % error for DSP kernel routines and 16 % for a full application). The results also show that the native C compilers for the two targets frequently produce a large overhead as compared to hand-written code, which confirms the findings of the DSPStone project [77]. Hence, retargetable estimation might be a reasonable and fast alternative to retargetable compilation during architecture exploration. However, more experimental results would be required to confirm the estimation accuracy and its retargetability, and an optimizing compiler will still be required once the target architecture has been fixed.

5.9. Miscellaneous

For sake of completeness, we briefly mention the following list of tools and WWW resources. Even though there is no direct relation to embedded systems, these provide point solutions that may (or may not) be valuable resources in new retargetable compiler projects.

TenDRA: A portable C/C++ compiler that generates intermediate code in the TDF/ANDF format [219].

Eli: A compiler generation package for integration of compiler components with custom I/O formats [220].

VCODE: A portable dynamic code generator with extremely high compilation efficiency by avoiding the step of intermediate code generation [221].

New Jersey Machine Code Toolkit: Tools for encoding and decoding binary machine code, supporting the implementation of tools like assemblers, disassemblers, linkers, and debuggers [222].

Cocktail: Tools for scanner and parser generation, attribute grammar processing, as well as tree pattern matching [223].

Gentle: A compiler construction toolbox, including parsing, source-to-source translation, and code selector generation [224].

SGI Pro64: C/C++ compiler development tools for Linux/IA-64 platforms [225], based on the GCC frontend.

PAG: Generator for static program analyzers, to be used in program transformation and optimization tools [226].

MLRISC: A retargetable optimizing backend for RISC processors, written in ML [227]

Comprehensive overviews of further compiler construction tools can be found at [228, 229, 230].

6. Commercial retargetable compilers

6.1. CoSy

CoSy is an extensible and retargetable compiler system with frontends for C/C++ (optionally with special language extensions for DSPs), Fortran, and Java. It originated from the European research project COMPARE [92] and has later been commercialized by Associated Compiler Experts (ACE) [231]. The software is available for Unix, Linux, and MS Windows platforms. CoSy licenses typically include a royalty payment scheme for generated compilers. Research licenses are available for universities at a reduced fee.

A basic concept in CoSy is the use of a common *high-level intermediate representation* (IR) that is generated by the different frontends. Like in SUIF, LANCE, or Trimaran this allows to enable or disable certain *IR optimization engines* at any time, which also can dynamically interact with each other (fig. 5.25). CoSy already comes with a library of standard IR optimizations (including constant folding, dead code elimination, strength reduction, and loop unrolling), but it is explicitly designed to be extensible by custom IR optimization passes.

While the IR optimizations are essentially machine-independent, there is also a backend generator based on which machine-specific code generators can be designed. The backend generator is an improved version of the BEG tool (section 5.5.3.3). First, the high-level IR is passed to a lowering engine for replacement of high-level language constructs by low-level statements. Then a tree parsing based code selector maps the lowered IR into sequential assembly code. Compared to tools like IBURG and OLIVE, CoSy uses a relatively comfortable specification mechanism for the underlying tree grammar, which permits the specification of special matching conditions as well as operand and destination registers.

Next, there is a sequential pre-pass scheduling phase, during which instructions are reordered, so as to optimize potential parallelism and minimize virtual register lifetimes. The mapping to physical registers afterwards takes place by a graph-coloring based (either local or global, dependent on runtime requirements) register allocator. In case the target processor shows VLIW-like instruction-level parallelism, a post-pass scheduler, or code compactor, can be called to pack potentially parallel

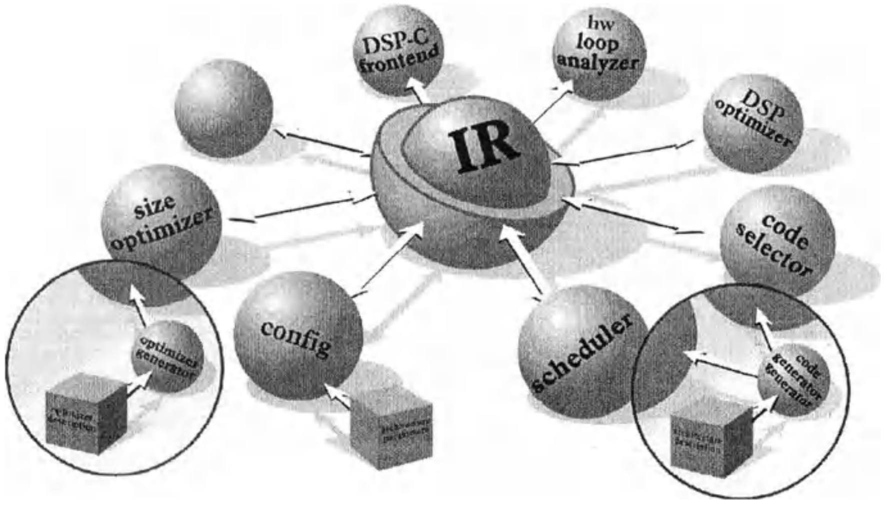


Figure 5.25. CoSy overview, ©ACE - Associated Compiler Experts

instructions into very long instruction words, based on a latency specification of instructions and functional units.

The main advantages of CoSy are its modular extensibility and target-independence, as well as the professional support as a commercial product. The system is used by a number of companies (e.g. Ericsson, Philips, and STMicroelectronics). CoSy users generally find that it is a good platform for quickly designing operational and robust compilers for new targets. On the other hand, the code generation process in CoSy follows a rather classical approach, with few optimizations for irregular architectures and no phase coupling. For a heavily optimizing compiler, new target-specific techniques will generally be required, but integrating such techniques requires significantly more effort than adding new IR optimizations.

6.2. CHESS

Just like CoSy, the CHESS compiler [105, 232, 233] originated from European research projects, whose results have been productized by the startup company Target Compiler Technologies [234]. CHESS is a retargetable C compiler primarily for DSPs. It is supported by further retargetable tools like assembler, linker, and instruction set simulator, as well as VHDL generation from processor models. The tool suite is illustrated in fig. 5.26. Currently supported platforms are Unix and Linux. Licensing terms are subject to negotiation.

CHESS borrows a number of concepts from the CBC and MSSQ compilers (sections 5.2.1 and 5.4.1). Like the CBC compiler, CHESS uses the nML language for target processor modeling. In contrast to the original approach in CBC, however, the action attributes of nML operations are specified by calls to primitive *C library routines* instead of using the built-in nML language elements for describing behavior. The advantage is that limitations of nML can be bypassed, and that a clear simulation semantics is provided by construction. The price for this is that the compiler cannot extract the instruction semantics solely from the nML model anymore, but needs additional mapping information between C library routines and machine instructions.

Like most other compilers, CHESS first performs a set of standard machine-independent optimizations, including common subexpression elimination and induction variable analysis. Fig. 5.27 shows the code generator GUI in CHESS. The code generator uses a control/data flow graph (CDFG) model as an intermediate representation. Also for the target processor a graph model is constructed from the nML specification. This *instruction-set graph* [235] is similar to the connection operation graph used in the MSSQ compiler, in the sense that it also represents hardware resources, interconnections, as well as functional unit operations together with the corresponding partial opcodes. Also the code selection pass resembles the approach taken in MSSQ, since partial opcodes are combined on-the-fly while checking for resource conflicts. The more recent tree parsing approach is not used, but there are provi-

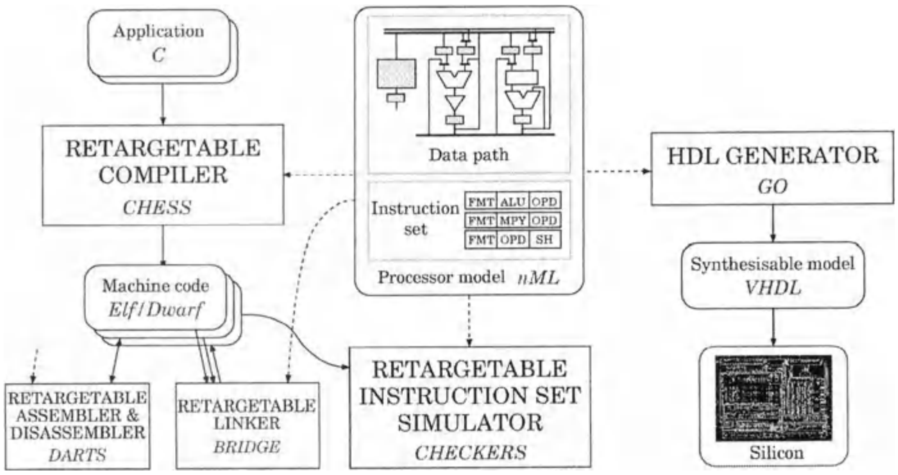


Figure 5.26. Structure of Target Compiler Technologies' retargetable tool suite, ©Target Compiler Technologies

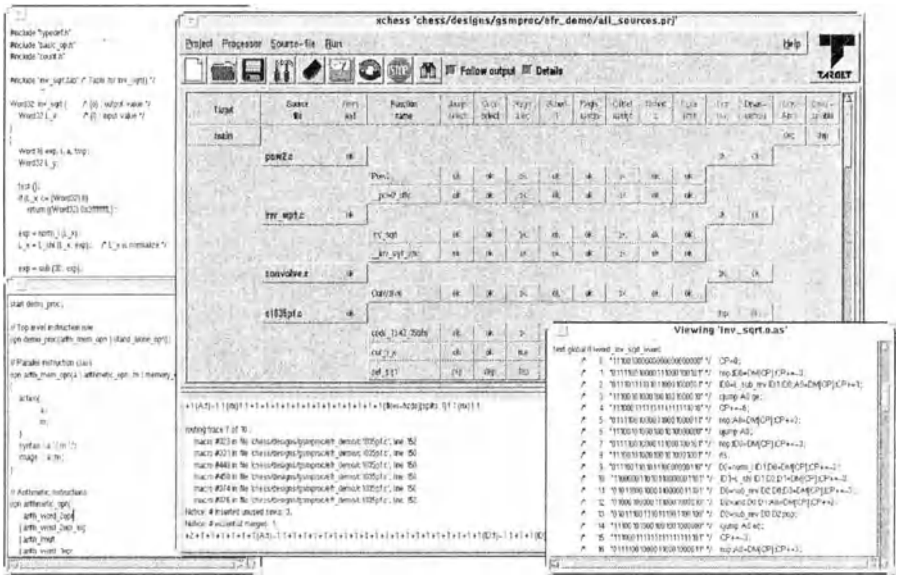

xchess 'chess/designs/gsmproc/efr_demo/all_sources.prj'
Project Processor Source-file Run
Help
Follow output
Details
TARGET
main
pow2.c
inv_sqrt.c
convolve.c
Viewing 'inv_sqrt.o.as'

Figure 5.27. CHESS compiler GUI, ©Target Compiler Technologies

sions for crossing basic block boundaries during code selection in order to better accommodate common subexpressions in the CDFG.

The register allocator in CHESS (similar to CBC) is implemented as a *data router*. This is especially suitable for irregular DSP architectures, since data routing explores different alternative routing paths within the data path, and therefore tends to minimize spill code for special purpose registers. Register allocation is followed by an offset assignment pass, during which local variables are assigned to specific stack frame locations in case the target machine contains an address generation unit as described in section 3.3.4.

The final code generation phase is instruction scheduling, during which generated instructions are compacted, so as to exploit instruction-level parallelism. For this purpose, CHESS relies on a list scheduler, enhanced by global optimizations including software pipelining, delay slot filling, and code hoisting.

The resulting assembly code can be fed into an assembler that is also generated from the nML model. Combined with the use of the retargetable instruction set simulator, the CHESS approach allows to perform architecture exploration for DSPs by iterative compilation and processor fine-tuning. Optionally, a synthesizable *VHDL processor model* can be generated for the final nML model, which establishes a path to hardware synthesis. However, the generated VHDL model does not necessarily represent the most efficient processor hardware implementation.

CHESS is another example of the fact that retargetable compilers are practical tools when focusing on a specific processor class. It also represents a quick flow of research results into an industrial product. Until recently, the main limitation of CHESS has been the lack of support for pipelined architectures, definitely a must for modern processor architectures. The new version CHESS V2, however, has been announced to overcome this limitation by enabling more accurate modeling of pipeline stages in nML. CHESS has been applied to generate code for an Analog Devices ADSP-210x like architecture. Unfortunately, further information about real-life targets and the code quality of CHESS is not publicly available.

6.3. Archelon

The Archelon tool suite [236] comprises a retargetable C compiler, assembler, and linker. The C compiler emits sequential code, which can be passed into further tools within the suite, including a peephole optimizer and a code compactor. The software is available for Unix, Linux, and most MS Windows versions. Pricing is quite low as compared to other commercial retargetable compilers (about US$ 5,000).

The C compiler gets its machine-specific information from a *compiler information file* (CIF). The code compactor, which is separate from the actual compiler, is driven by another description file. The CIF contains all information required for assembly code generation. In contrast to several other compilers (e.g. RECORD) that generate and link compiler components from the target model, the CIF is read every time the compiler is called. This reduces retargeting time at the expense of higher compiler runtimes.

First of all, the CIF file captures essential information like the bit width, signedness, and alignment of the C data types, as well as the machine word length and required assembly output syntax, including directives. Next, there is a description of available register files, their size, and a list of C types they may store. Special registers, like the stack pointer, are explicitly tagged. Further classifications concern scratch and argument registers, as well as registers available for general allocation. Another section in the CIF specifies the stack access and parameter passing conventions. Fig. 5.28 shows an excerpt from a sample CIF.

The compiler performs some basic high-level optimizations like constant folding and common subexpression elimination. Afterwards, assembly code generation takes place in a tree-oriented fashion (from the documentation in [236] it is not fully clear, whether standard tree parsing

```
#REGISTER  SETS
mau := 8; /* byte addressable memory */
regset := R[32] width=32
     optype=int,ptr,ptr2,float,double,longdbl,codeptr
     regtype=char,short,int,ptr,ptr2,codeptr,long,float,double,longdbl;
stkptr  := R[31];      # define the stack pointer register
scratch := R[24-27];  # reserve some scratch (global temporary) registers
argreg  := R[24-25];  # allow arguments to be passed in some registers
color   := R[0-29];    # define which registers will be controlled by
         # the register allocator.

#OPERANDS
operand code_addr codeptr; # pointer to code memory
operand data_addr ptr;     # pointer to data memory
operand const16   sconst -32768 32767; # 16 bit signed constant
...

#FORMATS
format mem_load_ri   src  ri_addr  dest gp_reg;
format mem_store_dir src  gp_reg   dest dir_addr;
format binary_rrr    lsrc gp_reg   rsrc  gp_reg  dest gp_reg;
...

#OPCODES
opcode ldri  mem_load_ri;
opcode ld    mem_load_dir;
opcode st    mem_store_dir;
opcode add   binary_rrr;
...

#CODE  TABLES
code binary(opcode)  # code table for add and subtract
? matches( $right, const16 )  # predicate test: true if rhs is 16 bit constant
{
     opcode|"i" $left,$right,$dest;
}
{
     opcode $left,$right,$dest;
}

oper ADD : sshort  binary( "add" );
oper SUB : sshort  binary( "sub" );
...
```

Figure 5.28. Partial Compiler Information File (CIF) for the Archelon compiler, ©Archelon Inc.

is used, but there are at least strong similarities). The available machine instructions are specified in a special formalism that is subdivided into four sections: operands, formats, opcodes, and code tables. Each entry in the code tables specifies the assembly code to be emitted for a certain intermediate representation operator. Similar to GCC and LCC, there is a predefined set of operators that any machine description must cover so as to generate code for all possible C source programs.

The Archelon compiler also supports some non-standard operators that e.g. allow handling of *zero-overhead loops* and *interrupt functions.* Additionally, the user may define additional operators that are available at the C level via *compiler intrinsics.* Another mechanism of compiler fine-tuning is offered by a tree rewrite formalism that allows to alter data flow trees before code selection takes place.

After code selection a graph coloring register allocator is applied to map virtual to physical registers. The final output of the code generator is sequential assembly code. This code is typically passed to further optimization tools, including a peephole optimizer that performs some simplifications based on user-defined optimization rules. Additionally, there is a code compactor that performs instruction scheduling and parallelization of the sequential code, so as to avoid pipeline stalls and to exploit potential parallelism.

Among others, the Archelon compiler has been applied to several DSP cores, e.g. from Clarkspur and 3DSP. This indicates that the compiler in principle can handle irregular architectures. However, due to a lack of DSP-specific optimizations, some code quality overhead as compared to hand-written code can certainly be expected.

6.4. UCC

Astrosoft [237] offers UCC, a portable C and C++ cross compiler. UCC follows a rather traditional compilation flow: source code analysis, machine-independent optimizations (including constant propagation, common subexpression elimination, loop invariant code motion, induction-variable strength reduction, and dead code elimination), and machine code generation. The target processor is described by means of code generation tables [238], which specify instruction patterns for certain IR constructs, as well as permissible operand and results registers. Dedicated optimizations for embedded processors are obviously not implemented. UCC also has support for *inline assembly* and *C language extensions.* The compiler has been retargeted to the x86 processor family. Unfortunately, further results on code quality and the target processor range have not been published. Whether or not UCC is suitable for embedded systems has yet to be demonstrated.

Chapter 6

SUMMARY AND OUTLOOK

Motivation. Many of today's embedded systems are designed with *programmable processor cores* as their building blocks. As a consequence, software development for embedded processors nowadays plays a significant role in the design flow. Traditionally, most embedded software has been written in assembly languages. However, due to the numerous drawbacks of assembly-level programming and the increasing time-to-market pressure, now there is a shift towards using high-level language compilers.

Embedded systems generally require a different compiler technology than general purpose systems. There is a huge variety of domain or even application specific embedded processors, and we certainly cannot afford to write a new compiler for each new target machine. Therefore, *retargetable compilers* are required, that are capable of generating code at least for a certain class of different processors. Moreover, there is frequently a need for very high code efficiency in terms of performance, size, and/or energy consumption.

While work on retargetable compilers began long before embedded system design became a major issue, they now receive a renewed interest due to the need to perform *design space exploration.* The main purpose of this book is threefold: (1) to provide the essential technical background information on retargetable compilers, (2) to outline the new paradigm of using retargetable compilers for design space exploration, and (3) to provide an up-to-date overview and classification of existing retargetable compiler systems and tools. We hope that this reduces the "entry barrier" to the area of retargetable compilation and stimulates further research projects.

State of the art. Retargetability is normally considered incompatible with the demand for high *code quality.* This is certainly true for a retargetable compiler without any focus on a specific processor class, since each such class requires its own special code optimization techniques. Therefore, a key to successful introduction of retargetable compilers in the system design tool chain is *specialization* towards a single processor class.

For instance, in the domain of standard RISC and CISC processors, freely available compilers like GCC have been quite successfully used, while systems like SPAM, CHESS, and RECORD incorporate dedicated optimization techniques for DSPs. Likewise, there are retargetable compilers tuned towards VLIW machines, e.g. Trimaran and ROCKET. Each of these systems has its limitations w.r.t. the concrete architectural scope that can be handled, but they achieve reasonably good code quality by making a priori assumptions about the target machine.

In case of a new target processor that does not well fit into a standard processor class, systems like SUIF, LANCE, or CoSy can be a good starting point. They incorporate mostly machine independent code optimizations, but they provide a relatively quick path towards generating an operational compiler for an "arbitrary" new target machine. Tools like IBURG and OLIVE may help to further reduce the backend design effort.

Several compilers for ASIPs, e.g. MSSQ, PEAS, and EXPRESS, are explicitly intended for design space exploration. They provide comfortable processor modeling capabilities and also support retargetable simulation, which guarantees short turnaround times.

As described in chapter 5, a huge variety of *processor modeling formalisms* for retargetable compilation are currently in use. These include custom description languages (e.g. CBC and AVIV), sometimes mixed with C source code (e.g. GCC and LCC), as well as parameterized models (e.g. Trimaran), HDL models (e.g. MSSQ and RECORD), or even finite state machine or integer linear programming models. They mainly differ in their modeling capabilities and abstraction level.

Besides the huge amount of optimization techniques known from traditional compiler construction, a lot of research effort has been invested recently in *code optimization for embedded processors.* In particular, this holds for DSPs (e.g. code generation for irregular architectures and address code optimization) and highly parallel machines (e.g. scheduling for clustered VLIW processors). Frequently, such techniques, many of which are beyond the scope of this book, work for a large range of architectures. Hence, they can be integrated into retargetable compilers, in order to further reduce the overhead of compiled code.

Future directions. So far, "compiler-aided" design space exploration is only partially systematic, but frequently follows more a trial-and-error methodology: One can make changes to the target processor model, and compile, synthesize, and simulate in order to somehow converge to a good application-specific architecture, based on the designer's intuition. In contrast, a "clever" retargetable compiler would provide *more detailed feedback* to the processor designer than the plain machine code. For instance, the compiler could report bottlenecks during code selection, scheduling, or register allocation and give suggestions for architectural or even source code changes. In combination with a simulator and profiler, this would shorten the exploration phase and help to reach better design space points.

Also the *code quality* of retargetable compilers needs to be continuously improved, in order to keep pace with the advances in processor architecture. Currently, a trend towards "compiler-friendly" VLIW architectures can be observed, but experience shows that, due to efficiency reasons, realistic machines are rarely clean enough, so that using only off-the-shelf code optimization techniques is sufficient. Therefore, the library of domain specific code optimization techniques should be continuously extended, driven by the newest processor generations.

In this context, also novel approaches (such as assembly-level optimization or exact optimization based on formal methods instead of heuristics) should be further investigated. With few exceptions, so far there are no practical compilers that can switch to time-intensive heavy optimization of the "hot spots" in an application program, e.g. by employing a phase-coupling approach. This feature would largely improve applicability of compilers, since it is still common practice to hand-optimize compiled code, e.g. in time-critical loops.

With respect to *processor modeling formalisms*, each approach has its pros and cons, and we may expect even more modeling formalisms in the future. For narrow processor classes, the parameterized model approach is certainly a good solution. However, for a larger range of target machines, those modeling languages will find their way into practice that represent a good compromise between ease of use, modeling capabilities, and a seamless integration with other modeling languages used in embedded system design, such as SystemC or VHDL.

Appendix A
Tabular overview of compiler tools

system	year of publication or 1st version	preferred processor class	software freely or commercially available	influence from	*WWW home page*
GCC	1987	RISC/CISC	yes		*http://gcc.gnu.org*
MSSQ	1987	ASIP	no	MSSV	
BEG	1989	RISC/CISC	yes		*http://www.first.gmd.de/beg*
ROCKET	1990	VLIW	yes	LCC	*http://www.cs.mtu.edu/~sweany/Rocket.html*
Archelon	1990	–	yes		*http://www.archelon.com*
LCC	1991	RISC/CISC	yes		*http://www.cs.princeton.edu/software/lcc*
Marion	1991	RISC	no	LCC	
IMPACT	1991	VLIW	yes	EDG	*http://www.crhc.uiuc.edu/Impact*
PEAS	1991	ASIP	no	GCC, CoSy	*http://vlsilab.ics.es.osaka-u.ac.jp*
IBURG	1992	–	yes		*http://www.cs.princeton.edu/software/iburg*
CBC	1992	DSP	no		
UCC	1992	–	yes		*http://astrosoft-development.com/english/services/main.html*
PAGODE	1993	RISC	no		

system	year of publication or 1st version	preferred processor class	software freely or commercially available	influence from	*WWW home page*
SUIF	1994	–	yes	LCC, EDG	*http://suif.stanford.edu*
FlexWare	1994	DSP	no	CoSy	
CoSy	1995	–	yes	BEG, EDG	*http://www.ace.nl*
CHESS	1995	DSP	yes	MSSQ, CBC	*http://www.retarget.com*
REDACO	1996	DSP	no		*http://swan.nt.tuwien.ac.at/codegen*
OLIVE	1997	–	yes	IBURG	*http://www.ee.princeton.edu/spam*
Valen-C	1997	ASIP	yes	SUIF	*http://kasuga.csce.kyushu-u.ac.jp/~codesign/Valen-C*
SPAM	1997	DSP	yes	SUIF, OLIVE	*http://www.ee.princeton.edu/spam*
RECORD	1997	DSP	no	MSSQ, SPAM	
LANCE	1997	–	yes	OLIVE	*http://LS12-www.cs.uni-dortmund.de/~leupers*
Trimaran	1998	VLIW	yes	IMPACT	*http://www.trimaran.org*
Trimedia	1998	VLIW	yes	EDG	*http://www.semiconductors.philips.com/trimedia*
AVIV	1998	VLIW	no	SUIF, SPAM	
Zephyr	1998	RISC/CISC	yes	EDG, LCC	*http://www.cs.virginia.edu/zephyr*
EXPRESS	1999	ASIP/VLIW	no	GCC, MSSQ	*http://www.cecs.uci.edu/~aces/projMain.html#expression*
BUILDABONG	2000	ASIP	yes	RECORD	*http://www-date.uni-paderborn.de/RESEARCH/BUILDABONG*

References

[1] A.Kitajima, M. Itoh, J. Sato, A. Shiomi, Y. Takeuchi, M. Imai: *Effectiveness of the ASIP Design System PEAS-III in Design of Pipelined processors*, Asian and South Pacific Design Automation Conference (ASP-DAC), pages 649–654, 2001

[2] T. Baba, H. Hagiwara: *The MPG system: A machine-independent microprogram generator*, IEEE Trans. on Computers, Vol. 30, pages 373–395, 1981

[3] R.G. Bushell: *Higher level language for microprogramming*, Euromicro journal, 4:67–75, 1978

[4] R.G.G. Cattell: *Formalization and automatic derivation of code generators*, Technical report, PhD thesis, Carnegie-Mellon University, Pittsburgh, 1978

[5] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, A. Vandecappelle: *Custom memory management methodology*, Kluwer Academic Publishers, 1998

[6] M.E. Conway: *Proposal for an UNCOL*, Communications of the ACM, Vol. 1, 1958

[7] M. Cornero, F. Thoen, G. Goossens: *Software synthesis for real-time information processing systems*, in [103], 1995

[8] J.-L. Cruz, A. Gonzalez, M. Valero, N. P. Topham: *Multiple-banked register file architectures*, The 27th Annual International Symposium on Computer architecture, pages 316–325, 2000

[9] W. Damm, G. Doehmen, K. Merkel, M. Sichelschmid: *AADL/S* approach to firmware design verification*, IEEE Software, 3:27–37, 1986

[10] S. Dasgupta: *Towards a microprogramming language schema*, 11th Annual Microprogramming Workshop, pages 144–153, 1978

[11] IMEC Desics group: Adopt, http://www.imec.be/desics

[12] IMEC Desics group: Atomium, http://www.imec.be/atomium

[13] *Personal communication*, ELMOS, Dortmund, Germany

[14] C. Ferdinand, H. Seidl, R. Wilhelm: *Tree automata for code selection*, Acta Informatica, pages 741–760, 1994

[15] M. Ganapathi, C.N. Fisher, J.L. Hennessy: *Retargetable compiler code generation*, ACM Computing Surveys, Vol. 14, pages 573–593, 1982

[16] N. Ghazal, R. Newton, J. Rabaey: *Retargetable estimation scheme for DSP architecture selection*, Proceedings of the Asia and South Pacific Design Automation Conference, pages 485–489, 2000

[17] R.S. Glanville: *A machine independent algorithm for code generation and its use in retargetable compilers*, Technical report, PhD thesis, University of California at Berkeley, 1978

[18] A. De Gloria, P. Faraboschi: *An evaluation system for application specific architectures*, Proc. 23rd Ann. Workshop on Microprogramming and Microarchitecture, pages 80–89, 1990

[19] P. Grun, N. Dutt, A. Nicolau: *Memory aware compilation through accurate timing extraction*, Proceedings of the 37th Design Automation Conference, 2000

[20] T.V.K. Gupta, P. Sharma, M. Balakrishnan, M. Malik: *Processor evaluation in an embedded systems design environment*, Proceedings of Thirteenth International Conference on VLSI Design, pages 98–103, 2000

[21] I.-J. Huang: *A case study: Synthesis and exploration of instruction set design for application-specific symbolic computing*, Journal of information Science and Engineering, 14:821–842, 1998

[22] *Personal communication*, Institut für Mikroelektronische Systeme (IMS), Duisburg, Germany

[23] M.F. Jacome, G. de Veciana, V. Lapinksi: *Exploring performance tradeoffs for clustered VLIW ASIPs*, International Conference on Computer-Aided Design (ICCAD), 2000

[24] JRS: *Integrated design automation system (idas)*, Sigmicro Newsletter, 19(1):11–17, 1989

[25] B. Kienhuis: *Design space exploration of stream-based dataflow architectures*, http://www.gigascale.org/systems/forum/5/kienhuis_dse.pdf, 1999

[26] J. Kin, C. Lee, W.H. Mangione-Smith, M. Potkonjak: *Power efficient media processors: design space exploration*, Proceedings of the 36th Design Automation Conference, pages 321–326, 1999

[27] V.S. Lapinskii, M. F. Jacome, G.A. de Veciana: *Application-specific clustered VLIW datapaths: Early exploration on a parameterized design space*, Technical Report UT-CERC-TR-MFJ/GDV-01-1, Computer Engineering Research Center, University of Texas at Austin, 2001

[28] P. Marwedel: *A retargetable microcode generation system for a high-level microprogramming language*, ACM Sigmicro Newsletter, Vol. 12, pages 115–123, 1981

[29] P. Marwedel: *A retargetable compiler for a high-level microprogramming language*, ACM Sigmicro Newsletter, Vol. 15, pages 267–274, 1984

[30] P. Marwedel: *A software-system for the synthesis of computer structures and of microcode (in German)*, Technical report, Habilitation thesis, Kiel, 1985; reprint: report no. 356, Computer Science Dept., Univ. of Dortmund

[31] P. Marwedel: *MSSV: Tree-based mapping of algorithms to predefined structures*, Technical Report 431, Computer Science Dpt., University of Dortmund, 1993

[32] P. Marwedel, L. Nowak: *Verification of hardware descriptions by retargetable code generation*, 26th Design Automation Conference, pages 441–447, 1989

[33] P. Marwedel, W. Schenk: *Cooperation of synthesis, retargetable code generation and testgeneration in the MSS*, EDAC-EUROASIC'93, pages 63–69, 1993

[34] J. Mermet, P. Marwedel, F. J. Ramming, C. Newton, D. Borrione, C. Lefaou: *Three decades of hardware description languages in Europe*, Journal of Electrical Engineering and Information Science, 3, 1998

[35] P. Mishra, N. Dutt, A. Nicolau: *Functional abstraction driven design space exploration of heterogenous programmable architectures*, Int. Symp. on System Synthesis (ISSS), 2001.

[36] P. Mishra, P. Grun, N. Dutt, A. Nicolau: *Memory subsystem description in EXPRESSION*, Technical Report #00-31, Dept. of Information and Computer Science, Univ. California, Irvine, 2000

[37] R.A. Mueller, J. Varghese, V.H. Allan: *Global methods in the flow graph approach to retargetable microcode generation*, 17th Annual Microprogramming Workshop, pages 275–284, 1984

[38] R. Niemann, P. Marwedel: *Hardware/software partitioning using integer programming*, European Design & Test Conference, 1996

[39] A.C. Parker: *Automated synthesis of digital systems*, IEEE Design and Test of Computers, pages 763–776, 1984

[40] S. Pees, A. Hoffmann, H. Meyr: *Retargetable compiled simulation of embedded processors using a machine description language*, IEEE Trans. on Design Automation for Embedded Systems, 2001

[41] E. Pelegri-Lopart, S. Graham: *Optimal code generation for expression trees: An application of BURS theory*, Technical report, Computer Science Division, EECS Department, University of California, Berkeley, 1988

[42] M. Püschel, B. Singer, M. Veloso, J.M.F. Moura: *Fast automatic generation of DSP algorithms*, Proc. ICCS 2001, Lecture Notes of Computer Science 2073, Springer, pages 97–106, 1999

[43] G. Rozenberg, F. Vaandrager (eds.): *Lectures on embedded systems*, Springer Lecture Notes on Computer Science, LNCS 1494, 1998

[44] M. Sint: *A survey of high level microprogramming languages*, 13th Annual Microprogramming Workshop, pages 141–153, 1980

[45] S. Steinke, C. Zobiegala, L. Wehmeyer, P. Marwedel: *Moving program objects to scratch-pad memory for energy reduction*, Technical report, University of Dortmund, Dept. of CS 12, 2001

[46] S. Takagi: *Rule based synthesis, verification and compensation of data paths*, Proc. IEEE Conf.Comp.Design (ICCD'84), pages 133–138, 1984

[47] S.R. Vegdahl: *Local code generation and compaction in optimizing microcode compilers*, PhD thesis and report CMUCS-82-153, Carnegie-Mellon University, Pittsburgh, 1982

[48] L. Wehmeyer, M.K. Jain, S. Steinke, P. Marwedel, M. Balakrishnan: *Analysis of the influence of register file size on the energy consumption, code size and execution time*, IEEE Trans. on CAD, 2001

[49] N. Wirth: *Compilerbau*, Teubner, 2nd edition, 1981

[50] H. Yasuura, H. Tomiyama, A. Anoue, N. Eko Fajar: *Embedded system design using soft-core processor and Valen-C*, Journal of information Science and Engineering, 14:587–603, 1998

[51] J. L. Young: *The software foundry: almost too good to be true*, Electronics, pages 47–51, 1988

[52] G. Zimmermann: *The MIMOLA design system: A computer aided digital processor design method*, Proceedings of the 16th Design Automation Conference, pages 53–58, 1979.

[53] A.W. Appel: *Modern Compiler Implementation in C*, Cambridge University Press, 1998

[54] S.S. Muchnik: *Advanced Compiler Design & Implementation*, Morgan Kaufmann Publishers, 1997

[55] T. Mason, D. Brown: *lex & yacc*, O'Reilly & Associates, 1991

[56] K.M. Bischoff: *Design, Implementation, Use, and Evaluation of Ox: An Attribute-Grammar Compiling System based on Yacc, Lex, and C*, Technical Report 92-31, Dept. of Computer Science, Iowa State University, 1992

[57] G. Sander: *VCG – Visualization of Compiler Graphs*, User Documentation V 1.30, Technical Report, Dept. of Computer Science, University of Saarland, Germany, 1995, software available via ftp://ftp.cs.uni-sb.de/pub/graphics/vcg

[58] A.V. Aho, R. Sethi, J.D. Ullman: *Compilers - Principles, Techniques, and Tools*, Addison-Wesley, 1986

[59] A.V. Aho, M. Ganapathi, S.W.K Tjiang: *Code Generation Using Tree Matching and Dynamic Programming*, ACM Trans. on Programming Languages and Systems 11, no. 4, 1989

[60] A. Balachandran, D.M. Dhamdere, S. Biswas: *Efficient Retargetable Code Generation Using Bottom-Up Tree Pattern Matching*, Comput. Lang. vol. 15, no. 3, 1990

[61] R. Wilhelm, D. Maurer: *Compiler Design*, Addison-Wesley, 1995

[62] B.Wess: *Simulated Evolutionary Code Generation for Heterogeneous Memory-Register DSP Architectures*, European Signal Processing Conference (EUSIPCO), 2000

[63] M.A. Ertl: *Optimal Code Selection in DAGs*, ACM Symp. on Principles of Programming Languages (POPL), 1999

[64] M. Lam: *Software Pipelining: An Effective Scheduling Technique for VLIW machines*, ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 1988

[65] J.A. Fisher: *Trace Scheduling: A Technique for Global Microcode Compaction*, IEEE Trans. on Computers, vol. 30, no. 7, 1981

[66] A. Aiken, A. Nicolau: *A Development Environment for Horizontal Microcode*, IEEE Trans. on Software Engineering, no. 14, 1988

[67] F.J. Kurdahi, A.C. Parker: *REAL: A Program for Register Allocation*, 24th Design Automation Conference (DAC), 1987

[68] P. Briggs: *Register Allocation via Graph Coloring*, Doctoral thesis, Dept. of Computer Science, Rice University, Houston/Texas, 1992

[69] F. Chow, J. Hennessy: *Register Allocation by Priority-Based Coloring*, SIGPLAN Notices, vol. 19, no. 6, 1984

[70] D.H. Bartley: *Optimizing Stack Frame Accesses for Processors with Restricted Addressing Modes*, Software - Practice and Experience, vol. 22(2), 1992

[71] C. Gebotys: *DSP Address Optimization Using a Minimum Cost Circulation Technique*, Int. Conference on Computer-Aided Design (ICCAD), 1997

[72] S. Udayanarayanan, C. Chakrabarti: *Address Code Generation for Digital Signal Processors*, 38th Design Automation Conference (DAC), 2001

[73] J.W. Davidson, C.W. Fraser: *The Design and Application of a Retargetable Peephole Optimizer*, ACM Trans. on Programming Languages and Systems, vol. 2, no. 2, 1980

[74] C.W. Fraser, D.R. Hanson, T.A. Proebsting: *Engineering a Simple, Efficient Code Generator Generator*, ACM Letters on Programming Languages and Systems, vol. 1, no. 3, 1992

[75] Free Software Foundation/EGCS: http://gcc.gnu.org

[76] Red Hat Inc.: http://www.redhat.com

[77] V. Zivojnovic, J.M. Velarde, C. Schläger, H. Meyr: *DSPStone - A DSP-oriented Benchmarking Methodology*, Int. Conf. on Signal Processing Applications and Technology (ICSPAT), 1994

[78] H. Gunnarson, T. Lundqvist: *Porting the GNU C Compiler to the Thor Microprocessor*, Master Thesis, Document No. TOR/TNT/0028/SE, http://www.ce.chalmers.se/~thomasl/publications/thesis95.html, Saab Ericsson Space AB, 1995

[79] ARC Cores: http://www.arccores.com

[80] Tensilica Inc.: http://www.tensilica.com

[81] C.Fraser, D. Hanson: LCC home page, http://www.cs.princeton.edu/software/lcc

[82] J. Navia: *LCC-Win32: a free compiler system for Windows*, http://www.cs.virginia.edu/~lcc-win32

[83] C. Fraser, D. Hanson: *A Retargetable C Compiler: Design And Implementation*, Benjamin/Cummings, 1995

[84] D. Bradlee, R. Henry, S. Eggers: *The Marion system for retargetable instruction scheduling*, ACM SIGPLAN Conference on Programming Language Design and Implementaion (PLDI), 1991

[85] D. Bradlee: *Retargetable Instruction Scheduling for Pipeline Processors*, PhD thesis, University of Washington, Technical report 91-08-07, Department of Computer Science and Engineering, 1991

[86] The Stanford SUIF Compiler Group, http://suif.stanford.edu

[87] Edison Design Group: http://www.edg.com

[88] Machine SUIF: http://www.eecs.harvard.edu/~hube/research/machsuif.html

[89] Zephyr home page: http://www.cs.virginia.edu/zephyr

[90] P. Canalda, L. Cognard, A. Depland, M. Jourdan, M. Mazaud, D. Parigot, F. Thomasset: *PAGODE: a Realistic Back-End Generator*, Technical Report, INRIA Rocquencourt, France, 1995

[91] P. Canalda, L. Cognard, A. Depland, M. Mazaud, F. Thomasset: *IRs and their Specification in the PAGODE Back-End Generator*, Technical Report, INRIA Rocquencourt, France, 1996

[92] ESPRIT project COMPARE home page: http://i44www.info.uni-karlsruhe.de/~vollmer/compare.html

[93] R. Leupers: *Code Optimization Techniques for Embedded Processors*, Kluwer Academic Publishers, 2000

[94] The LANCE V2.0 system: http://LS12-www.cs.uni-dortmund.de/lance

[95] P. Marwedel, S. Steinke, L. Wehmeyer: *Compilation techniques for energy-, code-size-, and run-time-efficient embedded software*, Int. Workshop on Advanced Compiler Techniques for High Performance and Embedded Processors (IWACT), Bucharest, 2001

[96] X. Nie, L. Gazsi, F. Engel, G. Fettweis: *A New Network Processor Architecture for High-Speed Communications*, IEEE Workshop on Signal Processing Systems (SiPS), 1999

[97] J. Wagner, R. Leupers: *C Compiler Design for an Industrial Network Processor*, ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES), 2001

[98] Informatik Centrum Dortmund (ICD): http:/www.icd.de

[99] Systemonic AG, Dresden: http://www.systemonic.com

[100] A. Fauth, A. Knoll: *Translating Signal Flowcharts into Microcode for Custom Digital Signal Processors*, Int. Conf. on Signal Processing (ICSP), 1993

[101] A. Fauth, A. Knoll: *Automated Generation of DSP Program Development Tools Using a Machine Description Formalism*, Int. Conf. on Acoustics, Speech, and Signal Processing (ICASSP), 1993

[102] A. Fauth, G. Hommel, A. Knoll, C. Müller: *Global Code Selection for Directed Acyclic Graphs*, in: P.A. Fritzson (ed.): 5th Int. Conference on Compiler Construction (CC), 1994

[103] P. Marwedel, G. Goossens (eds.): *Code Generation for Embedded Processors*, Kluwer Academic Publishers, 1995

[104] A. Fauth: *Beyond Tool-Specific Machine Descriptions*, chapter 8 in [103], 1995

[105] A. Fauth, J. Van Praet, M. Freericks: *Describing Instruction-Set Processors in nML*, European Design and Test Conference (ED & TC), 1995

[106] C.W. Fraser, R.R. Henry, T.A. Proebsting: *BURG – Fast Optimal Instruction Selection and Tree Parsing*, ACM SIGPLAN Notices 27 (4), 1992, pp. 68-76

[107] R. Hartmann: *Combined Scheduling and Data Routing for Programmable ASIC Systems*, European Conference on Design Automation (EDAC), 1992

[108] K. Rimey, P.N. Hilfinger: *Lazy Data Routing and Greedy Scheduling for Application-Specific Signal Processors*, 21st Annual Workshop on Microprogramming and Microarchitecture (MICRO-21), 1988

[109] S. Mondal: *Compiler Back End Generation from nML Machine Description*, Master Thesis, IIT Kanpur, Dept. of Computer Science & Engineering, 1999

[110] B. Wess: *Code Generation based on Trellis Diagrams*, chapter 11 in [103], 1995

[111] W. Kreuzer, M. Gotschlich, B. Wess: *A Retargetargetable Optimizing Code Generator for Digital Signal Processors*, Int. Symp. on Circuits and Systems (ISCAS), 1996

[112] W. Kreuzer, M. Gotschlich, B. Wess: *REDACO: A Retargetable Data Flow Graph Compiler for Digital Signal Processors*, Int. Conf. on Signal Processing Applications and Technology (ICSPAT), 1996

[113] A. Helm, B. Wess: *Decomposition of Signal Flow Graphs for DSP Compilers Using Trellis Trees in Restricted Environments*, Int. Conf. on Signal Processing Applications and Technology (ICSPAT), 1994

[114] S. Davidson, D. Landskov, B.D. Shriver, P.W. Mallett: *Some Experiments in Local Microcode Compaction for Horizontal Machines*, IEEE Trans. on Computers, vol. 30, no. 7, 1981

[115] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, A. Wang: *Storage Assignment to Decrease Code Size*, ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 1995

[116] R. Leupers, P. Marwedel: *Algorithms for Address Assignment in DSP Code Generation*, Int. Conference on Computer-Aided Design (ICCAD), 1996

[117] B. Wess, M. Gotschlich: *Optimal DSP Memory Layout Generation as a Quadratic Assignment Problem*, Int. Symp. on Circuits and Systems (ISCAS), 1997

[118] C. Liem: *Retargetable Compilers for Embedded Core Processors*, Kluwer Academic Publishers, 1997

[119] C. Liem, T. May, P. Paulin: *Instruction-Set Matching and Selection for DSP and ASIP Code Generation*, European Design and Test Conference (ED & TC), 1994

[120] C. Liem, T. May, P. Paulin: *Register Assignment through Resource Classification for ASIP Microcode Generation*, Int. Conf. on Computer-Aided Design (ICCAD), 1994

[121] C. Liem, P. Paulin, M. Cornero, A. Jerraya: *Industrial Experience Using Rule-driven Retargetable Code Generation for Multimedia Applications*, 8th Int. Symp. on System Synthesis (ISSS), 1995

[122] C. Liem, P.Paulin, A. Jerraya: *Address Calculation for Retargetable Compilation and Exploration of Instruction-Set Architectures*, 33rd Design Automation Conference (DAC), 1996

[123] P. Paulin: *Network Processors: A Perspective on Market Requirements, Processors Architectures, and Embedded S/W Tools*, Design Automation & Test in Europe (DATE), 2001

[124] S. Liao: *Code Generation and Optimization for Embedded Digital Signal Processors*, Ph.D. thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1996

[125] A. Sudarsanam, S. Malik: *Memory Bank and Register Allocation in Software Synthesis for ASIPs*, Int. Conf. on Computer-Aided Design (ICCAD), 1995

[126] G. Araujo, S. Malik: *Optimal Code Generation for Embedded Memory Non-Homogeneous Register Architectures*, 8th Int. Symp. on System Synthesis (ISSS), 1995

[127] G. Araujo, A. Sudarsanam, S. Malik: *Instruction Set Design and Optimizations for Address Computation in DSP Architectures*, 9th Int. Symp. on System Synthesis (ISSS), 1996

[128] G. Araujo, S. Malik, M. Lee: *Using Register Transfer Paths in Code Generation for Heterogeneous Memory-Register Architectures*, 33rd Design Automation Conference (DAC), 1996

[129] G. Araujo: *Code Generation Algorithms for Digital Signal Processors*, Ph.D. thesis, Princeton University, Department of Electrical Engineering, 1997

[130] A. Sudarsanam, S. Liao, S. Devadas: *Analysis and Evaluation of Address Arithmetic Capabilities in Custom DSP Architectures*, Design Automation Conference (DAC), 1997

[131] A. Sudarsanam: *Code Optimization Libraries for Retargetable Compilation for Embedded Digital Signal Processors*, Ph.D. thesis, Princeton University, Department of Electrical Engineering, 1998

[132] SPAM compiler: http://www.ee.princeton.edu/spam

[133] G. Ottoni, S. Rigo, G. Araujo, S. Rajagopalan, S. Malik: *Optimal Live Range Merge for Address Register Allocation in Embedded Programs*, 10th International Conference on Compiler Construction (CC), 2001

[134] D.B. Powell, E.A. Lee, W.C. Newman: *Direct Synthesis of Optimized DSP Assembly Code from Signal Flow Block Diagrams*, International Conference on Acoustics, Speech, and Signal Processing (ICASSP), 1992

[135] M. Saghir, P. Chow, C. Lee: *Exploiting Dual Data-Memory Banks in Digital Signal Processors*, 7th International Conference on Architectural Support for Programming Languages and Operating Systems, 1996

[136] R. Leupers, D. Kotte: *Variable Partitioning for Dual Memory Bank DSPs*, International Conference on Acoustics, Speech, and Signal Processing (ICASSP), 2001

[137] R. Leupers: *Retargetable Code Generation for Digital Signal Processors*, Kluwer Academic Publishers, 1997

[138] R. Leupers, P. Marwedel: *A BDD-based Frontend for Retargetable Compilers*, European Design & Test Conference (ED & TC), 1995

[139] R. Leupers, P. Marwedel: *Retargetable Generation of Code Selectors from HDL Processor Models*, European Design & Test Conference (ED & TC), 1997

[140] R. Leupers, P. Marwedel: *Time-Constrained Code Compaction for DSPs*, 8th Int. System Synthesis Symposium (ISSS), 1995

[141] S. Bashford, U. Bieker, B. Harking, R. Leupers, P. Marwedel, A. Neumann, D. Voggenauer: *The MIMOLA Language V4.1*, Technical Report, University of Dortmund, Dept. of Computer Science, September 1994

[142] R. Leupers: *HDL-based Modeling of Embedded Processor Behavior for Retargetable Compilation*, 11th Int. Symp. on System Synthesis (ISSS), 1998

[143] R.E. Bryant: *Symbolic Manipulation of Boolean Functions Using a Graphical Representation*, 22nd Design Automation Conference (DAC), 1985

[144] Mentor Graphics Corporation: *DSP Architect DFL User's and Reference Manual, V 8.2_6*, 1993

[145] R. Leupers, A. Basu, P. Marwedel: *Optimized Array Index Computation in DSP Programs*, Asia South Pacific Design Automation Conference (ASP-DAC), 1998

[146] R. Leupers, F. David: *A Uniform Optimization Technique for Offset Assignment Problems*, 11th Int. System Synthesis Symposium (ISSS), 1998

[147] P. Sweany, S. Beaty: *Post-Compaction Register Assignment in a Retargetable Compiler*, 22nd Annual Workshop on Microprogramming and Microarchitecture (MICRO-23), 1990

[148] ROCKET compiler: http://www.cs.mtu.edu/~sweany/Rocket.html

[149] R.A. Mueller, M.R. Duda, P.H. Sweany, J.S. Walicki: *Horizon: A Retargetable Compiler for Horizontal Microarchitectures*, IEEE Trans. On Software Engineering, 14 (5), 1988

[150] T. Brasier, P. Sweany, S. Carr, S. Beaty: *CRAIG: A Practical Framework for Combining Instruction Scheduling and Register Assignment*, Int. Conf. on Parallel Architectures and Compilation Techniques (PACT), 1994

[151] J. Hiser, S. Carr, P. Sweany: *Global Register Partitioning*, International Conference on Parallel Architectures and Compilation Techniques, 2000

[152] P. Chang, S. Mahlke, W. Chen, N. Warter, W. Hwu: *Impact: An Architectural Framework for Multiple Instruction Issue Processors*, 18th Int. Symp. on Computer Architecture, 1991

[153] IMPACT home page: http://www.crhc.uiuc.edu/Impact

[154] Trimaran home page: http://www.trimaran.org

[155] SPEC CPU95 Benchmarks: http://open.specbench.org/osg/cpu95

[156] Philips Trimedia: http://www.semiconductors.philips.com/trimedia

[157] Trimedia Technologies: http://www.trimedia.com

[158] K. Vissers, E.J. Pol: *A Retargetable Compiler and Retargetable Simulator for Media Processors*, Handouts 3rd Int. Workshop on Code Generation for Embedded Processors (SCOPES), 1998

[159] G. Hadjiyiannis, S. Hanono, S. Devadas: *ISDL: An Instruction-Set Description Language for Retargetability*, 34th Design Automation Conference (DAC), 1997

[160] S. Hanono, S. Devadas: *Instruction Selection, Resource Allocation, and Scheduling in the AVIV Retargetable Code Generator*, 35th Design Automation Conference (DAC), 1998

[161] Mescal home page: http://www.gigascale.org/mescal

[162] L. Nowak: *Graph based Retargetable Microcode Compilation in the MIMOLA Design System*, 20th Ann. Workshop on Microprogramming (MICRO-20), 1987

[163] P. Marwedel: *Tree-based Mapping of Algorithms to Predefined Structures*, Int. Conf. on Computer-Aided Design (ICCAD), 1993

[164] R. Leupers, P. Marwedel: *Retargetable Code Generation based on Structural Processor Descriptions*, Design Automation for Embedded Systems, Vol. 3, No. 1, Kluwer Academic Publishers, 1998

[165] PEAS project: http://vlsilab.ics.es.osaka-u.ac.jp

[166] A.Y. Alomary, T. Nakata, Y. Honma, J. Sato, N. Hikichi, and M. Imai: *PEAS-I: A Hardware/Software Co-design System for ASIPs*, European Design Automation Conference (EURO-DAC), 1993

[167] N. Ohtsuki, Y. Takeuchi, K. Hamaguchi, M. Imai, et al.: *Compiler Generation in PEAS-II*, 3rd Int. Workshop on Code Generation for Embedded Processors, 1998

[168] S. Kobayashi, Y. Takeuchi, A. Kitajima, M. Imai: *Compiler Generation in PEAS-III: an ASIP Development System*, Int. Workshop on Software and Compilers for Embedded Processors (SCOPES), 2001, http://ls12-www.cs.uni-dortmund.de/scopes-01

[169] Valen-C compiler home page, Kyushu University, Japan: http://kasuga.csce.kyushu-u.ac.jp/~codesign/Valen-C

[170] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt A. Nicolau: *EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability*, Design Automation & Test in Europe (DATE), 1999

[171] P. Grun, A. Halambi, N. Dutt A. Nicolau: *RTGEN: An Algorithm for Automatic Generation of Reservation Tables from Architectural Descriptions*, Int. Symp. on System Synthesis (ISSS), 1999

[172] A. Halambi, A. Shrivastava, N. Dutt A. Nicolau: *A Customizable Compiler Framework for Embedded Systems*, Int. Workshop on Software and Compilers for Embedded Processors (SCOPES), 2001, http://ls12-www.cs.uni-dortmund.de/scopes-01

[173] S. Novack, A. Nicolau, N. Dutt: *A Unified Code Generation Approach using Mutation Scheduling*, chapter 12 in [103], 1995

[174] J. Teich, R. Weper, D. Fischer, S. Trinkert: *BUILDABONG: A Rapid Prototyping Environment for ASIPs*, DSP Deutschland, 2000

[175] D. Fischer, J. Teich, R. Weper: *Modeling and Simulation of Embedded Processors Using Abstract State Machines*, Int. Workshop on Software and Compilers for Embedded Processors (SCOPES), 2001, http://ls12-www.cs.uni-dortmund.de/scopes-01

[176] BUILDABONG project, University of Paderborn: http://www-date.uni-paderborn.de/RESEARCH/BUILDABONG

[177] V. Zivojnovic, S. Tjiang, H. Meyr: *Compiled Simulation of Programmable DSP Architectures*, IEEE Workshop on VLSI Signal Processing, 1995

[178] J. Zhu, D. Gajski: *A Retargetable Ultra-Fast Instruction Set Simulator*, Design, Automation and Test in Europe (DATE), 1999

[179] M. Balakrishnan, P.C.P. Bhatt, B.B. Madan: *An Efficient Retargetable Microcode Generator*, 19th Ann. Workshop on Microprogramming (MICRO-19), 1986

[180] M. Yamaguchi, N. Ishiura, T. Kambe: *Binding and Scheduling Algorithms for Highly Retargetable Compilation*, Asia South Pacific Design Automation Conference (ASPDAC), 1998

[181] M. Mahmood, F. Mavaddat, M.I. Elmasry: *Experiments with an Efficient Heuristic Algorithm for Local Microcode Generation*, Int. Conf. on Computer Design (ICCD), 1990

[182] F. Mavaddat: *On Transforming the Code Generation Problem to a Parsing Problem*, chapter 9 in [103], 1995

[183] M. Langevin, E. Cerny: *An Automata-Theoretic Approach to Local Microcode Generation*, European Conference on Design Automation (EDAC), 1993

[184] M. Langevin, E. Cerny, J. Wilberg, H.-T. Vierhaus: *Local Microcode Generation in System Design*, chapter 10 in [103], 1995

[185] A. Römer, G. Fettweis: *Optimierte Parallele Codeerzeugung*, DSP Deutschland, Munich, 2000

[186] A. Römer, G. Fettweis: *Flow Graph Based Parallel Code Generation*, Int. Workshop on Software and Compilers for Embedded Processors (SCOPES), 1999, http://ls12-www.cs.uni-dortmund.de/scopes-99

[187] G. Fettweis, M. Weiss, W. Drescher et al.: *Breaking New Grounds Over 3000 M MAC/s: A Broadband Mobile Multimedia Modem DSP*, DSP Deutschland, 1998

[188] S. Bashford, R. Leupers: *Phase-Coupled Mapping of Data Flow Graphs to Irregular Data Paths*, Design Automation for Embedded Systems, vol. 4, no. 2/3, Kluwer Academic Publishers, 1999

[189] S. Bashford, R. Leupers: *Constraint Driven Code Selection for Fixed-Point DSPs*, 36th Design Automation Conference (DAC), 1999

[190] T. Wilson, G. Grewal, B. Halley, D. Banerji: *An Integrated Approach to Retargetable Code Generation*, 7th Int. Symp. on High-Level Synthesis (HLSS), 1994

[191] T. Wilson, G. Grewal, S. Henshall, D. Banerji: *An ILP-based Approach to Code Generation*, chapter 6 in [103], 1995

[192] M.R. Gary, D.S. Johnson: *Computers and Intractability – A Guide to the Theory of NP-Completeness*, Freemann, 1979

[193] F. Krohm: *Ein retargierbarer Compiler für anwendungsspezifische Mikrocontoller*, VDI Verlag, ISBN 3-18-146920-3, 1992

[194] Small Device C Compiler: http://sdcc.sourceforge.net

[195] IBURG home page: http://www.cs.princeton.edu/software/iburg

[196] R. Leupers: *Register Allocation for Common Subexpressions in DSP Data Paths*, Asia South Pacific Design Automation Conference (ASPDAC), 2000

[197] H. Emmelmann, F.W. Schröer, R. Landwehr: *BEG – A Generator for Efficient Backends*, ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), SIGPLAN Notices 24, no. 7, 1989

[198] BEG home page: http://www.first.gmd.de/beg

[199] SALTO project, INRIA: http://www.irisa.fr/caps/projects/Salto

[200] R. Amicel, F. Bodin: *A New System for High-Performance Cycle-Accurate Compiled Simulation*, Int. Workshop on Software and Compilers for Embedded Processors (SCOPES), 2001, http://ls12-www.cs.uni-dortmund.de/scopes-01

[201] D. Kästner: *PROPAN: A Retargetable System for Postpass Optimizations and Analyses*, ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES), 2000

[202] D. Kästner: *Retargetable Postpass Optimization by Integer Linear Programming*, Ph.D. Thesis, Saarland University, 2000

[203] D. Kästner: *Retargetable Code Optimization by Integer Linear Programming*, Int. Workshop on Software and Compilers for Embedded Processors (SCOPES), 2001, http://ls12-www.cs.uni-dortmund.de/scopes-01

[204] S. Pees, A. Hoffmann, V. Zivojnovic, and H. Meyr: *LISA - Machine Description Language for Cycle-Accurate Models of Programmable DSP Architectures*, 36th Design Automation Conference (DAC), 1999

[205] A. Hoffmann, A. Nohl, G. Braun, H. Meyr: *A Survey on Modeling Issues Using the Machine Description Language LISA*, International Conference on Acoustics, Speech, and Signal Processing (ICASSP), 2001

[206] A. Hoffman, A. Nohl, S. Pees, G. Braun, H. Meyr: *Generating Production Quality Software Development Tools Using a Machine Description Language*, Design, Automation & Test in Europe (DATE), 2001

[207] Axys Design Automation: http://www.axysdesign.com

[208] LISATek Inc.: http://www.lisatek.com

[209] 3DSP Corporation: http://www.3dsp.com

[210] M. Strik, J. van Meerbergen, A. Timmer, J. Jess, S. Note: *Efficient Code Generation for In-House DSP Cores*, European Design and Test Conference (ED & TC), 1995

[211] A. Timmer, M. Strik, J. van Meerbergen, J. Jess: *Conflict Modelling and Instruction Scheduling in Code Generation for In-House DSP Cores*, 32nd Design Automation Conference (DAC), 1995

[212] B. Mesman, C. Alba Pinto, K. van Eijk: *Efficient Scheduling of DSP Code on Processors with Distributed Register Files*, 12th Int. Symp. on System Synthesis (ISSS), 1999

[213] K. van Eijk, B. Mesman, C.A. Pinto, Q. Zhao, M. Bekooij, J. van Meerbergen, J. Jess: *Constraint Analysis for Code Generation: Basic Techniques and Applications in FACTS*, ACM TODAES, vol. 5, no. 4, Oct 2000

[214] M. Bekooij, B. Mesman, J. van Meerbergen, J. Jess: *Tightly Coupled Operation Assignment and Scehduling for VLIW Processors with FACTS*, Int. Workshop on Software and Compilers for Embedded Processors (SCOPES), 2001, http://ls12-www.cs.uni-dortmund.de/scopes-01

[215] B. Mesman: *Contraint Analysis for DSP Code Generation*, Ph.D. thesis, TU Eindhoven, 2001

[216] U. Bieker, P. Marwedel: *Retargetable Self-Test Program Generation Using Constraint Logic Programming*, 32nd Design Automation Conference (DAC), 1995

[217] U. Bieker, M. Kaibel, P. Marwedel, W. Geisselhardt: *STAR-DUST: Hierarchical Test of Embedded Processors by Self-Test Programs*, European Test Workshop, 1999

[218] N. Ghazal, R. Newton, J. Rabaey: *Predicting Performance Potential of Modern DSPs*, 37th Design Automation Conference (DAC), 2000

[219] TenDRA: http://www.cse.unsw.edu.au/~patrykz/TenDRA

[220] Eli: http://www.cs.colorado.edu/~eliuser

[221] VCODE: http://www.pdos.lcs.mit.edu/~engler

[222] New Jersey Machine Code Toolkit:
http://www.eecs.harvard.edu/~nr/toolkit

[223] Cocktail: http://cocolab.com/html/cocktail.html

[224] Gentle: http://www.first.gmd.de/gentle

[225] SGI Pro64: http://oss.sgi.com/projects/Pro64

[226] PAG, AbsInt GmbH: http://www.absint.com/pag

[227] MLRISC: http://www.cs.nyu.edu/leunga/www/MLRISC/Doc/html

[228] GMD catalog of compiler construction tools: http://www.first.gmd.de/cogent/catalog

[229] Catalog of free compilers and interpreters: http://www.idiom.com/free-compilers

[230] CBEL: http://www.cbel.com/Compilers_Programming

[231] Associated Compiler Experts: http://www.ace.nl

[232] D. Lanneer, M. Cornero, G. Goossens, H. De Man: *Data Routing: A Paradigm for Efficient Data-Path Synthesis and Code Generation*, 7th Int. Symp. on High-Level Synthesis (HLSS), 1994

[233] D. Lanneer, J. Van Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen, G. Goossens: *CHESS: Retargetable Code Generation for Embedded DSP Processors*, chapter 5 in [103], 1995

[234] Target Compiler Technologies: http://www.retarget.com

[235] J. Van Praet, D. Lanneer, G. Goossens, W. Geurts, H. De Man: *A Graph Based Processor Model for Retargetable Code Generation*, European Design and Test Conference (ED & TC), 1996

[236] Archelon Inc.: http://www.archelon.com

[237] Astrosoft: http://astrosoft-development.com/english/services/main.html

[238] V.S. Pavlov, S.A. Mironov: *A Universal C Compiler*, The Journal of C Language Translation, Dec 1992

About the authors

Rainer Leupers is a senior researcher at the Department of Computer Science (Embedded Systems Group) of the University of Dortmund, Germany. His research and teaching activities mainly include software tools and design automation for embedded systems. In addition, he serves as a project manager for industrial tool development at the technology transfer company ICD. Dr. Leupers authored the books "Retargetable Code Generation for Digital Signal Processors" (1997) and "Code Optimization Techniques for Embedded Processors" (2000), published by Kluwer. He obtained the Diploma and Ph.D. degrees in Computer Science with distinction from the University of Dortmund in 1992 and 1997, and he received awards for outstanding theses as well as a Best Paper Award at DATE 2000. Email: leupers@icd.de.

Peter Marwedel received his Ph.D. in Physics from the University of Kiel (Germany) in 1974. He worked at the Computer Science Department of that University from 1974 until 1989. In 1987, he received the Dr. habil. degree (a degree required for becoming a professor) for his work on high-level synthesis and retargetable code generation based on the hardware description language MIMOLA. Since 1989 he is a professor at the Computer Science Department of the University of Dortmund (Germany). He served as the Dean of that Department between 1992 and 1995. His current research areas include hardware/software codesign, high-level test generation, high-level synthesis, and code generation for embedded processors. He is a member of the IEEE Computer society, the ACM, and the Gesellschaft für Informatik (GI). Email: marwedel@acm.org.

Index