Linköping Studies in Science and Technology

Thesis No. 1021

Integrated Optimal Code Generation for Digital Signal Processors

by

Andrzej BEDNARSKI

Akademisk avhandling

som för avläggande av teknologie doktorsexamen vid Linköpings universitet kommer att offentligt försvaras i Planck, Fysikhuset, Linköpings universitet, onsdagen den 7 juni 2006, kl. 13.15

Abstract

In this thesis we address the problem of optimal code generation for irregular architectures such as Digital Signal Processors (DSPs).

Code generation consists mainly of three interrelated optimization tasks: instruction selection (with resource allocation), instruction scheduling and register allocation. These tasks have been discovered to be $\mathbb{NP}$-hard for most architectures and most situations. A common approach to code generation consists in solving each task separately, *i.e.* in a decoupled manner, which is easier from a software engineering point of view. Phase-decoupled compilers produce good code quality for regular architectures, but if applied to DSPs the resulting code is of significantly lower performance due to strong interdependences between the different tasks.

We developed a novel method for fully integrated code generation at the basic block level, based on dynamic programming. It handles the most important tasks of code generation in a single optimization step and produces an optimal code sequence. Our dynamic programming algorithm is applicable to small, yet not trivial problem instances with up to 50 instructions per basic block if data locality is not an issue, and up to 20 instructions if we take data locality with optimal scheduling of data transfers on irregular processor architectures into account. For larger problem instances we have developed heuristic relaxations.

In order to obtain a retargetable framework we developed a structured architecture specification language, xADML, which is based on XML. We implemented such a framework, called OPTIMIST that is parameterized by an xADML architecture specification.

The thesis further provides an Integer Linear Programming formulation of fully integrated optimal code generation for VLIW architectures with a homogeneous register file. Where it terminates successfully, the ILP-based optimizer mostly works faster than the dynamic programming approach; on the other hand, it fails for several larger examples where dynamic programming still provides a solution. Hence, the two approaches complement each other. In particular, we show how the dynamic programming approach can be used to precondition the ILP formulation.

As far as we know from the literature, this is for the first time that the main tasks of code generation are solved optimally in a single and fully integrated optimization step that additionally considers data placement in register sets and optimal scheduling of data transfers between different registers sets.

This work has been supported by the Ceniit (Center for Industrial Information Technology) 01.06 project of Linköpings universitet, ECSEL (Excellence Center for Computer Science and Systems Engineering), and the RISE (Research Instituet for Integrational Software Engineering) project supported by SSF (Stiftelsen för Strategisk Forskning), the Swedish Foundation for Strategic Research.

Department of Computer and Information Science
Linköpings universitet
SE-581 83 Linköping, Sweden

ISBN 91-85523-69-0 ISSN 0345-7524

Linköping Studies in Science and Technology
Dissertation No. 1021

Integrated Optimal Code Generation for Digital Signal Processors

by

Andrzej Bednarski

Department of Computer and Information Science
Linköpings universitet
SE-581 83 Linköping, Sweden

Linköping 2006

Printed by LiU-Tryck, Linköping 2006

To Leonia . . .

Abstract

In this thesis we address the problem of optimal code generation for irregular architectures such as Digital Signal Processors (DSPs).

Code generation consists mainly of three interrelated optimization tasks: instruction selection (with resource allocation), instruction scheduling and register allocation. These tasks have been discovered to be $\mathcal{NP}$-hard for most architectures and most situations. A common approach to code generation consists in solving each task separately, *i.e.* in a decoupled manner, which is easier from a software engineering point of view. Phase-decoupled compilers produce good code quality for regular architectures, but if applied to DSPs the resulting code is of significantly lower performance due to strong interdependences between the different tasks.

We developed a novel method for fully integrated code generation at the basic block level, based on dynamic programming. It handles the most important tasks of code generation in a single optimization step and produces an optimal code sequence. Our dynamic programming algorithm is applicable to small, yet not trivial problem instances with up to 50 instructions per basic block if data locality is not an issue, and up to 20 instructions if we take data locality with optimal scheduling of data transfers on irregular processor architectures into account. For larger problem instances we have developed heuristic relaxations.

In order to obtain a retargetable framework we developed a structured architecture specification language, xADML, which is based on XML. We implemented such a framework, called OPTIMIST that is parameterized by an xADML architecture specification.

The thesis further provides an Integer Linear Programming formulation of fully integrated optimal code generation for VLIW architectures with a homogeneous register file. Where it terminates successfully, the ILP-based optimizer mostly works faster than the dynamic programming approach; on the other hand, it fails for several larger examples where dynamic programming still provides a solution. Hence, the two approaches complement each other. In particular, we show how the dynamic programming approach can be used to precondition the ILP formulation.

As far as we know from the literature, this is for the first time that the main tasks of code generation are solved optimally in a single and fully integrated optimization step that additionally considers data placement in register sets and optimal scheduling of data transfers between different registers sets.

This work has been supported by the Ceniit (Center for Industrial Information Technology) 01.06 project of Linköpings universitet, ECSEL (Excellence Center for Computer Science and Systems Engineering), and the RISE (Research Instituet for Integrational Software Engineering) project supported by SSF (Stiftelsen för Strategisk Forskning), the Swedish Foundation for Strategic Research.

Acknowledgments

The work presented in this thesis could never be accomplished without the support of many people. I would like to take the opportunity here to formally thank them and apologize for those I forgot to mention.

First of all I would like to thank my supervisor Christoph Kessler for his precious help and advice during this whole period of Ph.D. study. He spent a large amount of his time guiding me in research work and writing. For me he was the perfect supervisor that a Ph.D. student may have ever expected. This work could have never been completed without Christoph's support.

Also, I would like to thank Peter Fritzson who offered me the opportunity to join Programming Environments Laboratory (PELAB) in March 1999, and triggered in me the interest for code generation.

Many thanks go to Peter Aronsson for diverse administrative arrangements and his effort for making me feel here in Sweden like at home. Today I am aware that without his help I would have probably never joined Linköpings universitet.

I would like as well to thank all members of PELAB, who create a stimulating environment, and in particular Jens Gustavsson, Andreas Borg, and John Wilander. A particular memory to Emma Larsdotter Nilsson who unfortunately passed away too early. Special thanks go to Levon Saldamli for relevant discussions concerning C++ issues but, also as a friend, for making my spare time enjoyable. Further, I am also thankful to Bodil Mattsson Kihlström, our secretary, who makes not only my stay at the university easier, but also of all other members of PELAB. A great thank to Jon Edvardsson, Tobias Ritzau, and Damien Wyart for tremendous help with LaTeX and motivating discussions. I am grateful to Mikhail Chalabine for helping me with the thesis cover and to Mattias Eriksson for proofreading the thesis. A thank to professor Petru Eles and Alexandru Andrei from Embedded Systems Laboratory (ESLAB) of Linköpings universitet for providing us access to CPLEX installation.

I would like to give credits to my master students, without whom this work would probably never progress so far. Special thanks go to Anders Edqvist who provided me with relevant feed back and contributed significantly in the

implementation and improvement of the project. The credits for writing ARM specifications and improving xADML specifications go to David Landén. A thank to Yuan Yongyi for providing me with the Motorola MC56K specifications. Further, I would like to thank Andreas Rehnströmer who implemented the first version of the graphical tool for writing xADML specifications.

Also a thank goes to administrative staff at IDA, in particular Lillemor Wallgren and Britt-Inger Karlsson.

Finally I would like to thank my friends and my family who encouraged me during these long years of expatriation. I also would like to thank Mingmin whom I love tenderly.

Andrzej Bednarski
Linköping, May 2006

Contents

Chapter 1.

Introduction

This chapter is a general introduction to our research area, that is optimal code generation for irregular architectures. We define what an irregular architecture is and give a short introduction to compilation and code generation. Further, we motivate the interest of optimal code generation and summarize the contributions of our work.

1.1. Introduction to Compilation and Code Generation for DSP

A Digital Signal Processor (DSP) is an integrated circuit whose main task is to process a digital input signal and produce an output signal. Usually the input signal is a sampled analog signal, for instance speech or temperature. An Analog-Digital Converter (ADC) transforms analog signals into a digital form, *i.e.* the input signal for a DSP. The output signal from the processor can be converted back to an analog signal with a Digital-Analog converter (DAC), or remain in digital form. Figure 1.1 depicts such a DSP system.

Today DSP systems represent a high volume of the electronic and embedded applications market that is still growing. Mobile phones, Portable Digital Assistants (PDAs), MP3 players, microwave-ovens, cars, air planes, *etc.* are

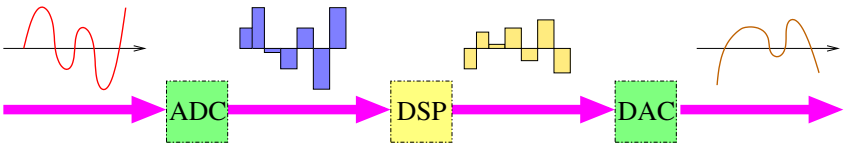

Figure 1.1.: Model of a DSP system.

only some examples equipped with a DSP system. Most embedded systems implement signal filters that are based on signal theory. Many filter computations are based on convolution:

$$x(n) = \sum_{i=0}^{n} x(i)\delta(n-i)$$

that involves the fundamental operation of Multiplication and Accumulation, for short MAC. Thus a MAC instruction is implemented in hardware on almost all classes of DSPs for efficiency purposes. The MAC instruction is an example of various hardware solutions to cope with high market requirements.

Manufacturers produce DSP processors with different technical solutions that meet computational requirements imposed by the market. As a consequence, processors exhibit more and more irregularities: specialized register sets, multiple memory banks, intricate data paths, *etc*. Thus, high requirements are achievable only if the code exploits DSP features and irregularities efficiently.

For regular architectures, currently available compiler techniques produce high quality code, but if applied to DSPs hand made code is hundreds of percent better [Leu00a]. This is because compiler techniques for standard processors are unable to efficiently exploit all available resources. Thus, for hot parts of the applications, DSP experts often write code directly in target assembly language and integrate it into automatically generated assembly code. Hand-coded parts of an application are generally of high quality but come with several drawbacks:

- High cost: it is due to the need of highly qualified staff and longer production time.
- Maintainability: it is difficult to update an application that is written partially in high level language, such as C, and in assembly language.
- Hardware update: if the platform is replaced all parts of the assembly code are likely to be totally rewritten. This is time consuming in terms of implementation and testing.

Moreover, high competition in electronics market increases the time to market demand drastically. Programmers are eager to write embedded applications in a high level language such as C instead of target depend assembly code and leave the optimization task to the compiler. There are numerous reasons for using the C language, mainly its popularity in industry for embedded

systems and its flexibility. However, some basic operators implemented in a DSP are of higher level than basic operators in C. For example, on most DSPs the following C statement `a=b*c+a;` requires a single target instruction, called multiply and accumulate, noted MAC. To map such a statement to the MAC instruction the compiler needs to perform pattern matching on the intermediate code.

Additionally, retargetability of a tool is an issue for fast adaptation to new emerging hardware.

1.2. Compilation Process

In general, a compiler is a software tool that translates programs written in a high level language into equivalent programs in object code or machine language that can be executed on a particular target architecture (could be a virtual machine as well). The compilation process inside a compiler consists of at least five steps [ASU86]:

- *Lexical analysis* takes as input character strings and divides them into tokens. Tokens are symbolic constants representing strings that constitute the vocabulary of the input language (*e.g.* "=", words, delimiters, separators, *etc.*). Lexical analysis produces error messages if the input character strings are incorrectly formed. The output of lexical analysis consists of a stream of tokens.

- *Syntactic analysis*, or parsing, processes the stream of tokens and forms a high level Intermediate Representation (IR) such as parse trees or abstract syntax trees. A tree is a data structure accessed by the top node, called *root*. Each node of a tree is either a leaf or an internal node that has one or more child nodes. A syntax tree is a compressed representation of a parse tree. An operator appears as an internal node of a tree, where its children represent operands.

- *Semantic analysis* takes as input a high level IR and verifies that the program satisfy semantic properties of the input language. Usually, after semantic analysis the IR is transformed to lower level, also known as *intermediate code generation* phase.

- *Optimizations* that are target independent such as dead code elimination, local and global subexpression elimination, loop unrolling, *etc.* are performed on a high and/or low level IR. The choice, order and level of

optimization depend on the overall optimization goal. Machine independent optimizations are considered as add-ons intended to improve code quality in various aspects.

- *Code generation* transforms the low level IR, or the intermediate code into equivalent machine code. Besides the tasks of instruction selection, instruction scheduling, and register allocation, numerous machine dependent optimizations are performed inside the code generator.

A compiler includes additionally a *symbol table*, a data structure which records each identifier used in the source code and its various attributes (type, scope, *etc.*). The five tasks with the symbol table are summarized in Figure 1.2.

1.3. Motivations

The first criterion of a compiler is to produce correct code, but often correct code is not sufficient. Users expect programs to use available hardware resources efficiently. The current state-of-the-art in writing highly optimized applications for irregular architectures offers two alternatives:

- Writing the applications directly in the assembly code for the specific hardware. Often, it is possible to automatically generate assembly code for those parts of the program that are not critical for the application, and only concentrate on the computationally expensive parts and write code for them by hand.

- Obtain highly optimized libraries from the hardware provider for a given target architecture. Then, the work consists in identifying parts of the application that may use a given library and call it. However, there are few companies that can spend sufficient effort in implementing highly optimized general purpose libraries, that are as well handwritten directly in the assembly language by experts. Designing a good library may be a difficult task, and it may work only for specific application areas.

With the solutions above it is possible to generate highly optimized code but at high cost in terms of man months. Further, the code needs to be rewritten almost completely if the underlying hardware changes, since the methods are rarely portable. In terms of software engineering, maintainability is difficult as long as there is no unified framework where third-party components can be updated more frequently than the application itself.

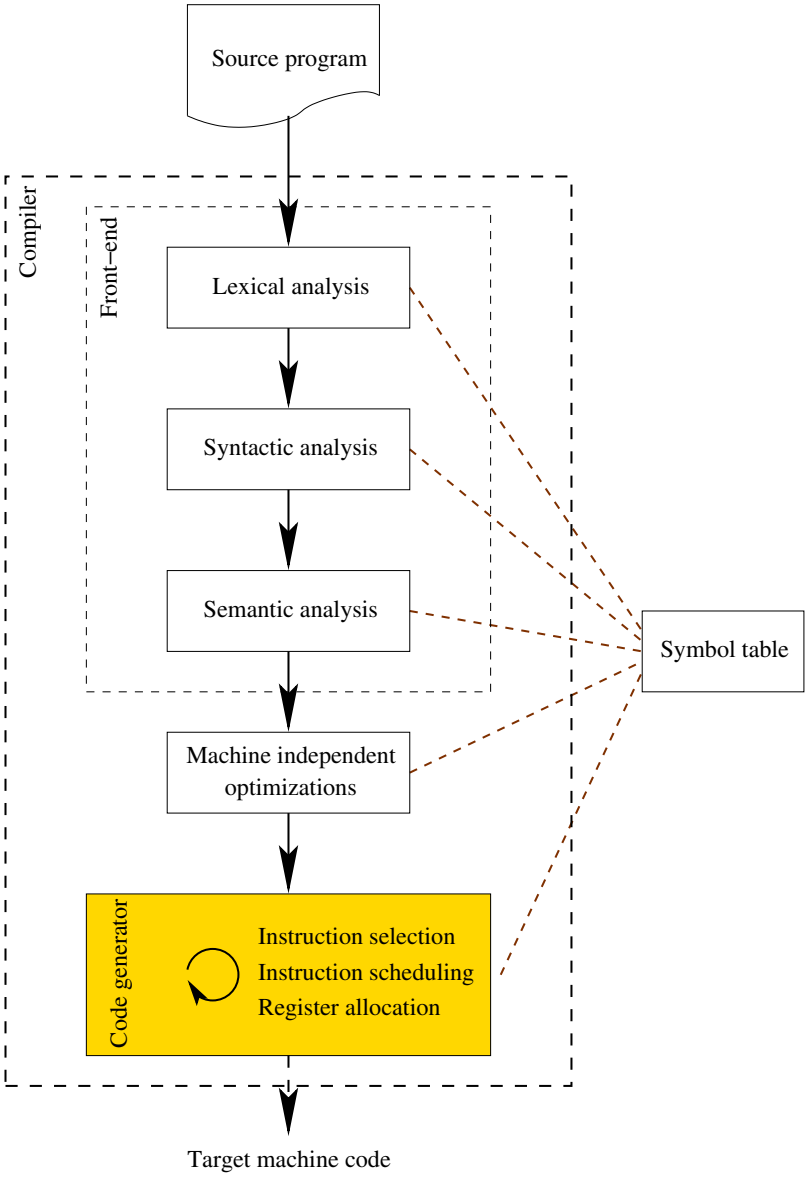

Figure 1.2.: Global view of a compilation process for a specific target machine.

Within this work we aim at improving the current state-of-the-art in compiler generation for irregular architectures, where the user keeps writing applications in a high level language and the compiler produces highly optimized code that exploits hardware resources efficiently. We focus particularly on code generation and aim at producing an optimal code sequence for a given IR of an input program. Optimal code sequence is of particular interest for fixed applications where the requirements are difficult to meet, and often a dedicated hardware is required. Furthermore, from the hardware designer viewpoint, information about optimal code sequence may influence further design decisions.

Moreover, we consider the retargetability issue. The code generation framework should not encapsulate target hardware specific, but take the hardware description as input information together with the application specified in a high level language. From the hardware description it is either possible to generate a compiler (code generator generator), or parameterize an existing framework. A code generator generator is usually more difficult to build, thus in this thesis we implement a prototype of a parameterizable retargetable framework for irregular architectures.

1.4. Research Interest

In the rest of this thesis by IR we mean a low level IR where data dependences are represented as a Directed Acyclic Graph (DAG). A DAG $G = (V, E)$ is a graph whose edges $e \in E$ are ordered pairs of nodes $v \in V$ and there is no path that starts and ends at the same node. IR nodes correspond to operations (except for the leave nodes) and the children of a node are the operands. The graphical representation of the IR can be described in textual form by *three-address code*. Three-address code consists of an instruction with at most three operands. Such a representation is close to the assembly language. For instance for a binary operation a three-address code is shown below.

```
destination = operand1 operation operand2
```

In general, the compilation process does not work on the whole program or the whole function at once but on a set of smaller units called *basic blocks*. A basic block is a set of three-address codes (or IR statements) in which the flow of control enters only at the beginning and leaves only at the end. Each time the basic block is entered all the statements are executed exactly once.

Code generation research started at the time when the first computers appeared. Today it is known that for most hardware architectures and for IR

dependence graphs that form DAGs, most important problems of optimal code generation are $\mathcal{NP}$-hard.

Often, the quality of a compiler is evaluated using a set of benchmarks and compared either to figures obtained by another compiler on the same set of benchmarks or to hand optimized code. However, in both cases it is impossible to indicate how far from the optimal code the generated code sequence is because the optimum is not known. We do research aiming for a novel method for fully integrated code generation, that actually allows to produce *optimal* code for problem instances that could not be solved optimally with the state-of-the-art technology.

From a code generation point of view, a program is a directed graph, also called *Control Flow Graph* (CFG) whose nodes represent basic blocks. In this thesis, and as a first step, we focus on code generation on basic block level, *i.e. local code generation*. Generalization for global code generation is planned for future work.

Integration of several $\mathcal{NP}$-hard problems into a single optimization pass increases considerably the complexity. We are aware that generating optimal code is not feasible for large problem instances, such as basic blocks with hundreds of instructions.

1.5. Contributions

In this thesis we provide both a dynamic programming and an integer linear programming approach to retargetable, fully integrated optimal code generation that makes it possible to precisely evaluate a code generation technique. Further, we see other domains that can benefit from optimal code generation:

- The method is suitable for optimizing critical parts of a fixed (or DSP) application. Since the proposed optimization technique requires a large amount of time and space, it is intended only for the final code generation before shipping the code on the hardware.

- For hardware designers it is possible to evaluate an architecture during the design phase, and take decisions upon the resulting feedback. Thus, it is possible to modify a hardware, reevaluate an application and analyze the influence on code quality of such a modification.

As part of the discussion of future work in Chapter 10 we provide further possible extensions and contributions.

1.6. Origin of the Chapters

Kessler started considering the problem of optimal code generation in 1993 with different scheduling algorithms [Kes98, Keß00]. In year 2000, he joined Linköpings universitet and under his guidance we continued research on optimal code generation. A large part of the material in this thesis originates from the following publications:

- A. Bednarski and C. Kessler. Exploiting Symmetries for Optimal Integrated Code Generation. In *Proc. International Conference on Embedded Systems and Applications*, pages 83–89, June 2004.
- C. Kessler and A. Bednarski. Energy-Optimal Integrated VLIW Code Generation. In Michael Gerndt and Edmund Kereku, editors, *Proc. 11th Workshop on Compilers for Parallel Computers*, pages 227–238, July 2004.
- C. Kessler and A. Bednarski. Optimal integrated code generation for VLIW architectures. To appear in *Concurrency and Computation: Practice and Experience*, 2006.
- A. Bednarski and C. Kessler. Optimal Integrated VLIW Code Generation with Integer Linear Programming. Accepted for publication at Euro-Par 2006 Conference in Dresden, 2006.

1.7. Organization of the Thesis

The rest of the thesis is organized as follows:

- **Chapter 2** introduces the code generation problem and focuses on digital signal processor issues. In this chapter we motivate why, if searching for an optimal code sequence, we require an integrated approach.
- **Chapter 3** provides the necessary notations and formalisms used in the rest of the thesis. Additionally, we provide a formal model of our target architecture.
- **Chapter 4** provides the dynamic programming algorithm for determining a time-optimal schedule for regular and irregular architectures. In this chapter we also formally prove that the dynamic programming algorithm to find an optimal solution.

- **Chapter 5** is an extension of our work in the area of energy aware code generation. Our method is generic and can be easily adapted to other domains where it is possible to define a monotonic cost function of the schedule length. The chapter presents a method for energy optimal integrated code generation (for generic VLIW processor architectures) based on an energy model from the literature.
- **Chapter 6** presents an optimization technique that reduces time and space usage of the dynamic programming algorithm. We exploit a so-called partial-symmetry property of data dependence graphs that results in higher compression of the solution space.
- **Chapter 7** provides an integer linear programming (ILP) formulation that fully integrates all phases of code generation as a single integer linear problem. The formulation is evaluated against the dynamic programming approach and we show the first results.
- **Chapter 8** provides the specification of our structured hardware description language called *Extended Architecture Description Mark-up Language* (xADML), based on Extensible Mark-up Language (XML).
- **Chapter 9** classifies related work in the area of decoupled and integrated code generation solutions.
- **Chapter 10** gives different possible extensions of the thesis work.
- **Chapter 11** concludes the thesis.

Chapter 2.

Introduction to Code Generation for Digital Signal Processors

In this chapter we introduce the code generation problem and concentrate more on digital signal processor (DSP) architectural issues. Additionally we motivate why we need an integrated approach when we are searching for an optimal code sequence. Further we provide related work in the area of integrated code generation.

2.1. Motivations

The advances in the processing speed of current microprocessors are caused not only by progress in higher integration of silicon components, but also by exploiting an increasing degree of instruction-level parallelism in programs, technically realized in the form of deeper pipelines, more functional units, and a higher instruction dispatch rate. Generating efficient code for such processors is largely the job of the programmer or the compiler back-end. Even though most superscalar processors can, within a very narrow window of a few subsequent instructions in the code, analyze data dependences at runtime, reorder instructions, or rename registers internally, efficiency still depends on a suitable code sequence.

The high volume of the embedded processor market asks for high performance at low cost. Digital Signal Processors (DSPs) with a VLIW or clustered VLIW architecture became, in the last decade, the predominant platform for high-performance signal processing applications. In order to achieve high code quality, developers still write critical parts of DSP applications in assembly language. This is time consuming, and maintenance and updating are difficult. Traditional compiler optimizations developed for standard processors

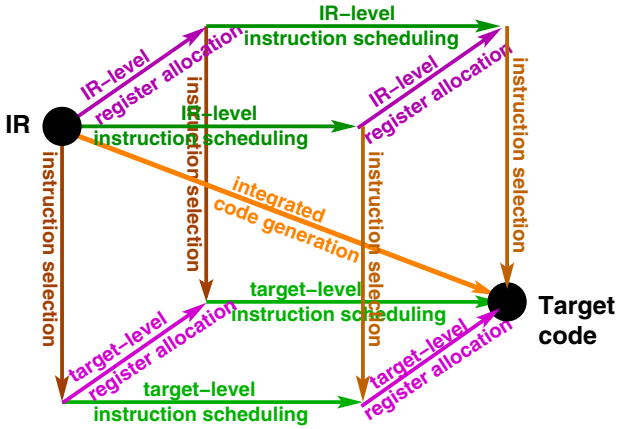

Figure 2.1.: The tasks of code generation as phase-decoupled or integrated problem. For clustered VLIW architectures, a fourth problem dimension (partitioning) could be added.

still produce poor code for DSPs [Leu00a]. and thus do not meet the requirements One reason for this is that the main tasks in code generation, such as instruction selection, instruction scheduling and register allocation, are usually solved in separate, subsequent phases in the compiler back-end, such that the interdependences between these phases, which are particularly strong for DSP architectures, are partially ignored. Hence, we consider the integration of these tasks into a single optimization problem.

2.2. Main Tasks of Code Generation

The task of generating target code from an intermediate program representation can be mainly decomposed into the interdependent subproblems of instruction selection, instruction scheduling, and register allocation. These subproblems span a three-dimensional problem space (see Figure 2.1). Phase-decoupled code generators proceed along the edges of the cube, while an integrated solution directly follows the diagonal, considering all subproblems simultaneously.

2.2.1. Instruction Selection

Instruction selection maps the abstract operations given in an intermediate representation (IR) of the input program to machine-specific instructions of the

target processor, where our notion of an instruction also includes a specific addressing mode and a (type of) functional unit where it could be executed, if there are several options. For each instruction, one can specify its expected cost (in CPU cycles) and its semantics in terms of equivalent IR operations, where the latter is usually denoted in the form of a (tree) pattern. Hence, instruction selection amounts to a pattern matching problem with the goal of determining a minimum cost cover of the IR with machine instructions where cost usually denotes latency but also may involve register requirements, instruction size, or power consumption. For tree-structured IR formats and most target instruction sets, this problem can be solved efficiently. The dynamic programming algorithm proposed by AHO and JOHNSON [AJ76] gives a minimal-cost covering for trees in polynomial time for most target instruction sets.

But, for DAGs minimal-cost instruction selection is $\mathcal{NP}$-complete [Pro98], see also Appendix A. There are several heuristic approaches that split the DAG into disjoint subtrees and apply tree pattern matching [EK91, PW96] on these.

2.2.2. Instruction Scheduling

Instruction scheduling is the task of mapping each instruction of a program to a point (or set of points) of time when it is to be executed, such that constraints implied by data dependences and limited resource availability are preserved. For RISC and superscalar processors with dynamic instruction dispatch, it is sufficient if the schedule is given as a linear sequence of instructions, such that the information about the issue time and the functional unit can be inferred by simulating the dispatcher's behavior. The goal is usually to minimize the execution time while avoiding severe constraints on register allocation. Alternative goals for scheduling can be minimizing the number of registers (or temporary memory locations) used, or the energy consumed.

2.2.3. Register Allocation

Register allocation maps each value in a program that should reside in a register, thus also called a *virtual register*, to a physical register in the target processor, such that no value is overwritten before its last use. If there are not enough registers available from the compiler's point of view, the live ranges of the virtual registers must be modified, either by:

- *coalescing*, that is, forcing multiple values to use the same register. Register coalescing identifies copy operations of values that do not interfere

with each other and are not overwritten after the copy. Then, the unnecessary copy is eliminated, and occurrences of the copied value are replaced with the original value.

- *spilling* the register, that is, storing the value of that register back to the memory and reloading it later, thus splitting the live range of value such that a different value can reside in that register in the meantime.

Even for superscalar processors, which usually expose quite many general-purpose registers to the programmer and internally may offer even more by hardware register renaming, spilling caused by careless use of registers should be avoided if possible, as generated spill code cannot be recognized as such and removed by the instruction dispatcher at runtime, even if there are internally enough free registers available. Also, spill code should be avoided especially for embedded processors because more memory accesses generally imply higher energy consumption. Finally, if less virtual registers are live across a function call, less registers must be saved to the stack, which results in less memory accesses, too. If the schedule is fixed and spilling cannot be avoided completely, the goal is to find a fastest instruction sequence that includes necessary spill code. Another strategy is to find a new schedule that uses less registers, possibly by accepting an increase in execution time.

2.2.4. Partitioning

Partitioning of data and instructions across clusters becomes an issue for clustered VLIW architectures, where not every instruction can use every register as operand or destination register, due to restrictions implied by data paths or instruction set encoding. We group registers that exhibit equal constraints on addressability as operands in all instructions of a given instruction set into a *register class* (a formal definition will be given later). While we could even treat partitioning as a fourth problem dimension, the partitioning of data across register classes can be considered a part of the register allocation task, and the mapping of instructions to a unit in a specific cluster for execution can be included in the instruction selection problem.

2.3. Optimization Problems in Code Generation

We refer to solving the problem of generating *time-optimal* code for a given program fragment, that is, code that needs a minimum number of clock cycles to execute, as *time optimization*. Likewise, by *space optimization*, we denote solving the problem of determining *space-optimal* code, that is, code that

needs a minimum number of registers (or temporary memory locations) for storing intermediate results without spilling. For non-pipelined single-issue architectures, this problem is also known as *minimum register instruction sequencing (MRIS)* problem. By adding an execution time deadline or a limit in the number of registers, we obtain constrained optimization problems such as *time-constrained space optimization* or *space-constrained time optimization*, respectively.

2.3.1. Code Generation Techniques

During the last two decades there has been substantial progress in the development of new methods in code generation for scalar and instruction-level parallel processor architectures. New retargetable tools for instruction selection have appeared, such as IBURG [FHP92, FH95]. New methods for fine-grain parallel loop scheduling have been developed, such as software pipelining [AN88, Lam88]. Global scheduling methods like trace scheduling [Fis81, Ell85], percolation scheduling [Nic84, EN89], or region scheduling [GS90] allow to move instructions across basic block boundaries. Also, techniques for speculative or predicated execution of conditional branches have been developed [HHG+95]. Finally, high-level global code optimization techniques based on data flow frameworks, such as code motion, have been described in [KRS98].

Most of the important optimization problems in code generation have been found to be $\mathcal{NP}$-complete. Hence, these problems are generally solved by heuristics. Global register allocation is $\mathcal{NP}$-complete, as it is isomorphic to coloring a live-range interference graph [CAC+81, Ers71] with a minimum number of colors. Time-optimal instruction scheduling for basic blocks is $\mathcal{NP}$-complete for almost any nontrivial target architecture [AJU77, BRG89, HG83, MPSR95, PS93] except for certain combinations of very simple target processor architectures and tree-shaped dependency structures [BGS93, BG89, BJPR85, EGS95, Hu61, KPF95, MD94, PF91]. Space-optimal instruction scheduling for DAGs is $\mathcal{NP}$-complete [BS76, Set75], except for tree-shaped [BGS93, SU70] or series-parallel [Güt81] dependency structures. Instruction selection for basic blocks with a DAG-shaped data dependency structure is assumed to be $\mathcal{NP}$-complete, too, and the dynamic programming algorithm designed for (IR) trees can no longer guarantee optimality for DAGs, especially in the presence of non-homogeneous register sets [Ert99].

Optimal selection of spill candidates and optimal a-posteriori insertion of spill code for a given fixed instruction sequence and a given number of available registers is $\mathcal{NP}$-complete even for basic blocks and has been solved by dynamic programming or integer linear programming for various special cases of

processor architecture and dependency structure [AG01, HKMW66, HFG89, MD99].

For the general case of DAG-structured dependences, various algorithms for time-optimal local instruction scheduling have been proposed, based on integer linear programming *e.g.* [GE92, Käs00a, WLH00, Zha96], branch-and-bound [CC95, HD98, YWL89], and constraint logic programming [BL99]. Dynamic programming has been used for time-optimal [Veg92] and space-optimal [Kes98] local instruction scheduling.

2.4. Phase Ordering Problem

In most compilers, the subproblems of code generation are treated separately in subsequent *phases* of the compiler back-end. This is easier from a software engineering point of view, but often leads to suboptimal results because the strong interdependences between the subproblems are ignored. For instance, early instruction scheduling determines the live ranges for a subsequent register allocator; where the number of physical registers is not sufficient, spill code must be inserted a-posteriori into the existing schedule, which may compromise the schedule's quality. Also, coalescing of virtual registers is not an option in that case. Conversely, early register allocation introduces additional ("false") data dependences and thus constrains the subsequent instruction scheduling phase.

Example Let us illustrate the phase ordering problem by a simple example. We consider a pipelined single-issue architecture, *i.e.* only one instruction can be issued per clock cycle. Figure 2.2 shows a situation where performing register allocation prior to target instruction scheduling decreases the number of possible schedules by adding extra constraints, and thus may miss the optimal solution. Let us consider that the register allocator allocates values computed by instructions `a`, `b` and `c` to registers r_1, r_2, and r_1 respectively. We observe that instruction `b` needs to be computed before instruction `c`, since `c` overwrites the value of register r_1 that is required for computing `b`. Therefore, this register assignment adds an extra dependence edge, the dashed edge in the figure, sometimes called "false" dependence that ensures that the content of r_1 is not overwritten before its last use.

There is only one possible target schedule after register allocation that is compatible with such a register allocation, *i.e.* `a, b, c`. Now, if we suppose that the instruction `c` has a latency of two clock cycles, and instructions `a` and `b` take only one clock cycle, we obtain a schedule with a total execution time of four clock cycles. Consequently, if we are optimizing for the shortest

Figure 2.2.: Performing register allocation before instruction scheduling adds additional constraints to the partial order of instructions.

execution time, we miss the optimal schedule, *i.e.* `a, c, b` that computes the same result in three clock cycles, where `b` and `c` are overlapping in time. ■

Moreover, there are interdependences between instruction scheduling and instruction selection: In order to formulate instruction selection as a separate minimum-cost covering problem, phase-decoupled code generation assigns a fixed, context-independent cost to each instruction, such as its expected or worst-case execution time, while the actual cost also depends on interference with resource occupation and latency constraints of other instructions, which depends on the schedule. For instance, a potentially concurrent execution of two independent IR operations may be prohibited if instructions are selected that require the same resource.

Example An example, adapted from [NN95], illustrates the interdependence between the phases of instruction selection and instruction scheduling. Let us consider an architecture with three functional units: an adder, a multiplier, and a shifter. We assume a single-issue architecture. For the following integer computation,

$$b \leftarrow a \times 2$$

it is possible to choose three different target instructions: multiply, addition and left shift. Table 2.1 lists the three possibilities with their respective unit occupation time. We assume that, for this target architecture and this example each possible instruction runs on a different functional unit.

The code selection phase, if performed first, would choose the operation with the least occupation time. In our example, the left shift instruction would be selected, associating the shifter functional unit with the multiplication operation. However, in the subsequent scheduling phase it might be the case that the left shift instruction should be scheduled at time t_{i+1}, but the shift unit is already allocated for computing another operation at that time t_i, such as $y = x << 3$ (see Figure 2.3). Figure 2.3 illustrates that the choice of using a left shift operation instead of addition produces final code that takes one clock

Table 2.1.: Possible operations to perform the integer computation $b \leftarrow a \times 2$, with their occupation time in clock cycles.

Operation	Func. unit	Unit occupation
add(b, a, a)	adder	4 cycles
mul(b, a, 2)	multiplier	5 cycles
lshift(b, a, 1)	shifter	3 cycles

cycle longer, although the adder requires one additional occupation clock cycle. Figure 2.3 (a) shows the resulting unit occupation schema if the shifter unit performs the integer multiplication; Figure 2.3 (b) illustrates the schedule if using the adder unit instead, which is shorter by one clock cycle.

t_i :	lshift(y, x, 3)	t_i :	lshift(y, x, 3)
t_{i+1} :	NOP	t_{i+1} :	add(b, a, a)
t_{i+2} :	NOP	t_{i+2} :	NOP
t_{i+3} :	lshift(b, a, 1)	t_{i+3} :	NOP
t_{i+4} :	NOP	t_{i+4} :	NOP
t_{i+5} :	NOP		

(a) Using the shifter (b) Using the adder

Figure 2.3.: Here, using an adder results in a shorter execution time than using a shifter.

■

Furthermore, on clustered VLIW processors, concurrent execution may be possible only if the operands reside in the right register sets, which is an issue of register allocation. Especially for DSP processors where data paths and instruction word encodings are optimized for a certain class of applications, such interdependences can be quite involved. Hence, the integration of these subproblems and solving them as a single optimization problem is a highly desirable goal, but unfortunately this increases the overall complexity of code generation considerably.

2.5. Integrated Approaches

There exist several heuristic approaches that aim at a better integration of instruction scheduling and register allocation [BSBC95, FR92, GH88, KG92].

For the case of clustered VLIW processors, the heuristic algorithm proposed by Kailas *et al.* [KAE01] integrates cluster assignment, register allocation, and instruction scheduling; heuristic methods that integrate instruction scheduling and cluster assignment were proposed by Özer *et al.* [OBC98], Leupers [Leu00b] and by Nagpal and Srikant [NS04].

Nevertheless, the user is, in some cases, willing to afford spending a significant amount of time in optimizing the code, such as in the final compilation of time-critical parts in application programs for DSPs. However, there are only a few approaches that have the potential—given sufficient time and space resources—to compute an optimal solution to an integrated problem formulation, mostly combining local scheduling and register allocation [BL99,Käs00a,Leu97]. Some of these approaches are also able to partially integrate instruction selection problems, even though for rather restricted machine models. For instance, Wilson *et al.* [WGHB94] consider architectures with a single, non-pipelined ALU, two non-pipelined parallel load/store/move units, and a homogeneous set of general-purpose registers. Araujo and Malik [AM95] consider integrated code generation for expression trees with a machine model where the capacity of each sort of memory resource (register classes or memory blocks) is either one or infinity, a class that includes, for instance, the TI C25. The integrated method adopted in the retargetable framework Aviv [HD98] for clustered VLIW architectures builds an extended data flow graph representation of the basic block that explicitly represents all alternatives for implementation; then, a branch-and-bound heuristic selects an alternative among all representations that is optimized for code size.

Most of these approaches are based on integer linear programming, which is again a $\mathcal{NP}$-complete problem and can be solved optimally only for rather small problem instances. Otherwise, integration must be abandoned and/or approximations and simplifications must be performed to obtain feasible optimization times, but then the method gives no guarantee how far away the reported solution is from the optimum. Admittedly, ILP is a very general tool for solving scheduling problems that allows to model certain architectural constraints in a flexible way, which enhances the scope of retargetability of the system. ILP has also been used for several other integrated approaches, by Wilson [MG95], Leupers [Leu97], and Govindarajan *et al.* [GYZ+99].

2.6. DSP Challenges

High performance requirements push DSP manufactures to build more and more irregular processors. A typical irregular feature consists in dedicated register sets, also called special purpose registers. Thus an instruction can only

be executed if the operands reside in special registers and/or it writes the result in another specific register or register set. In some DSPs dedicated register sets are context dependent.

Additionally, to increase data bandwidth, manufacturers of DSPs provide separate data memory banks and support for word-parallel execution.

Today there exists different techniques that improve the overall code quality without significant changes in the compiler. Here, we enumerate some of currently applied ad-hoc methods:

- Provide user directives to help the compiler to identify specific opportunities for code improvement.
- Allow direct access to the target specific instructions, *i.e.* allow the user to manually write a part of the code directly in the assembly language of the architecture to perform certain critical operations of the application. Wagner and Leupers [WL01] provide access to the target processor with so-called compiler-known functions, or compiler intrinsics. Compiler known-functions bring higher level abstraction provided by the hardware into the programming language.
- Build ahead generic highly optimized libraries for most of the digital signal processing arithmetic operations. Thus, hardware providers do not only manufacture a processor, but additionally related libraries. Usually assembly and hardware experts write such libraries directly in assembly language.

On the one hand, the methods enumerated above considerably improve the final object code. But they are not portable and considerable time and effort must be spent in rewriting applications and libraries for a different hardware.

Further, contrary to general purpose processors, DSPs offer a reduced number of addressing modes. Often, an addressing mode offers the possibility of auto-decrementing or auto-incrementing an address. A restricted number of addressing modes may be practical if programmers write applications directly in assembly code, but it imposes on the compiler to carefully place data in memory to efficiently access them, since address arithmetic instructions are limited. Leupers [Leu00a] addresses the issue of offset assignment that consists in rearranging local variables in memory such that the address generation unit can access them efficiently with auto-increment and auto-decrement operations.

Summarizing, DSPs exhibit numerous irregularities and thus increase considerably the complexity of high quality code generation. Hence, producing highly optimized code for DSPs is a challenging task.

2.7. Need for Integrated Code Generation

In Section 2.4 we mentioned the existence of dependences between different phases in a decoupled code generation. These interdependences are "stronger" for irregular architectures, that present intricate structural constraints. In order to produce optimal code it is necessary to combine the three main tasks of code generation into a single optimization phase. In phase decoupled code generators the code generation is illustrated as a path along the edges of the code generation cube (see Figure 2.1), while integrated solutions follow directly the diagonal of the cube from the IR to target code.

For IRs in form of directed acyclic graphs (DAGs) and most architectures, most subproblems of code generation are $\mathcal{NP}$-hard (see Section 2.3.1). Considering them as a single problem leads to a complex and challenging optimization problem.

For instance, if we consider an architecture with general purpose registers (regular architecture) and an IR-level DAG with n nodes the number of all possible (sequences) IR-level schedules is less than $n!$, the number of permutations of the operations. Due to the data dependence constraints the number of schedules depends on the DAG structure and is usually less than the upper bound. Brute-force enumerating all possible schedules is feasible for small problem instances, with up to 15 instructions per basic block even if only space-optimization is considered [Kes98].

For irregular architectures, where accessible registers for a given instruction are context dependent, registers need to be carefully allocated. The location of operands may influence the possibility of concurrent execution of given instructions.

Example Let us consider two independent simple operations, an addition and a multiplication. For the Hitachi SH7729 SH3-DSP [Hit99], if the operands for addition are located in registers `Y0` and `Y1` and for the multiplication in `X0` and `A1` then both cannot be scheduled simultaneously. The registers of the destination also influence the possibility of parallelism, but we do not consider it in this example. Then, possible target schedules are:

```
t1: ADD Y0, Y1, Y0          t1: MUL X0, A1, X0
t2: MUL X0, A1, X0   and    t2: ADD Y0, Y1, Y0
```

However, if the operands and result destinations are carefully chosen, then both operations can be executed simultaneously:

```
t1: ADD X0, X1, X0 || MUL Y0, Y1, Y0
```

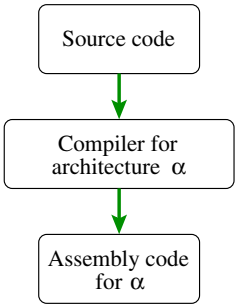

Figure 2.4.: In a dedicated compilation system, the compiler embeds the information of the target architecture.

■

This shows that the number of possible target schedules may be larger than the number of permutations of instructions. In the case of a fully integrated solution of the three problems, the complexity is even higher.

2.8. Retargetable Code Generation

Writing compilers is generally time consuming, and consequently expensive. In the worst case, once the compiler is available it might turn out that the target hardware is already obsolete. Therefore, it is important, for a code generation system to be easily reconfigurable for different architectures, *i.e.* to be a *retargetable code generation* system.

Generally, most compilation systems that come with a processor are dedicated compilers for that given architecture. However, in a design and development phase it is desirable to have a retargetable system. In Section 1.2 we described a classical view of a compiler that is a dedicated compiler for a specific hardware. Figure 2.4 illustrates a dedicated compiler for an architecture α. In such designs the back-end is specific for a given architecture, but also it may not be clearly separated from the rest of the compiler, and often the hardware information is spread within the whole compiler. In order to produce code for a different hardware, say β, it is necessary to spend considerable time and effort in porting the existing α compiler for β architecture.

Modular compiler toolkits, such as the compilation system CoSy [AAvS94], provide facilities to exchange compiler components, also called engines, and adapt the whole compilation system for a specific goal. Thus, if the target

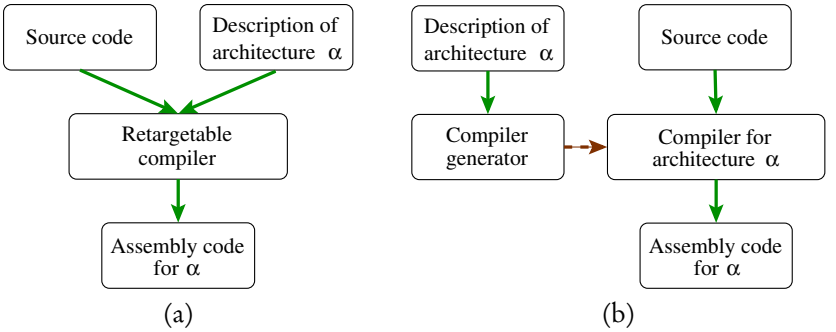

Figure 2.5.: A retargetable compiler takes as input the program and the description of the hardware architecture.

processor changes, it is theoretically sufficient to replace the back-end (engine) of any previously constructed compiler. Modular compiler toolkits facilitate significantly the task of compiler construction, but it is still necessary to write back-ends for each type of target architecture.

In contrast to dedicated code generation systems, retargetable compilers require additionally to the source program a description of the architecture for which to produce code (see Figure 2.5). Thus, to produce code for the α hardware we need to provide the α architecture description and the source program. This looks like an extra overhead compared to a dedicated compiler, but considerably facilitates the migration to some other hardware β, where it is only required to modify the hardware specifications. Figure 2.5 represents two variants of retargetable compilers: (a) parameterized, or dynamically retargetable such as GCC [FSF06] or OPTIMIST [KB05], and (b) generated, or statically retargetable (*e.g.* IBURG).

A retargetable code-generator generator is a framework that takes the hardware description as input and produces a compiler for the specified hardware. OLIVE [BDB90] and IBURG [FHP92] are examples of a retargetable code-generator generator tool. Code-generator generators are generally more complex to write than dynamically retargetable frameworks.

Dynamically retargetable back-ends take the architecture description and the source code simultaneously. A well known example is the GCC C compiler that includes descriptions for several processor architectures.

Chapter 3.

Prerequisites

In this chapter we introduce necessary terminology for integrated optimal code generation that we use in the rest of the thesis. Further, we give the generic model of our target architectures.

3.1. Notations

In the following, we focus on code generation for basic blocks where the data dependences among the IR operations form a directed acyclic graph (DAG) $G = (V, E)$. Let n denote the number of IR nodes in the DAG. An extension to extended basic blocks [Muc97] will be given in Section 4.3.6. For brevity, we often use the IR node names also to denote the values computed by them.

3.2. Modeling the Target Processor

We assume that we are given an in-order issue superscalar or a VLIW processor with f *resources* $U_1, ..., U_f$, which may be functional units or other limited resources such as internal buses.

The *issue width* ω is the maximum number of instructions that may be issued in the same clock cycle. For a single-issue processor, we have $\omega = 1$, while most superscalar processors and all VLIW architectures are multi-issue architectures, that is, $\omega > 1$.

Example The DSP processor TI-C62x (see Figure 3.1) has eight functional units and is able to execute a maximum of eight instructions concurrently in every clock cycle, *i.e.* $\omega = 8$. The full issue width can only be exploited if each instruction uses a different functional unit. ■

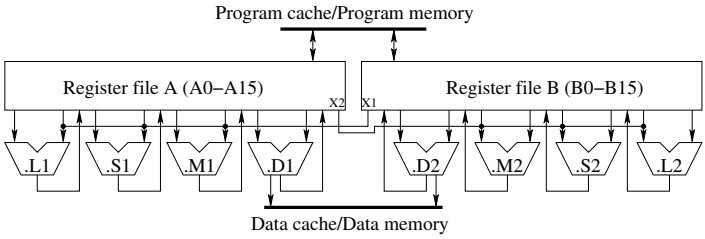

Figure 3.1.: Texas TI-C62x family DSP processor (VLIW clustered architecture).

In order to model instruction issue explicitly, we provide ω separate *instruction issue units* u_i, $1 \leqslant i \leqslant \omega$, which can take up at most one instruction per time unit (clock cycle) each. For a VLIW processor, the contents of all u_i at any time corresponds directly to a (long) instruction word. In the case of a superscalar processor, it corresponds to the instructions issued at that time as resulting from the dynamic instruction dispatcher's interpretation of a given linear instruction stream.

Example The following instruction word for TI-C62x

```
MPY .M1 A1,A1,A4  ||  ADD .L1 A1,A2,A3  ||  SUB .L2 B2,B3,B1
```

consists of three instructions (multiplication, addition and subtraction) issued concurrently, as indicated by `||`. Note that `NOP`s are implicit. The mapping to issue unit slots at time t is

	u_1	u_2	u_3	u_4	u_5	u_6	u_7	u_8
	L1	S1	M1	D1	D2	M2	S2	L2
t	ADD	NOP	MPY	NOP	NOP	NOP	NOP	SUB

where we use mnemonics for each entry. ■

Beyond an issue unit time slot at the issue time, an instruction usually needs one or several resources, at issue time or later, for its execution. For each instruction y issued at time (clock cycle) t, the resources required for its execution at time t, $t+1$, ... can be specified by a *reservation table* [DSTP75], a bit matrix o_y with $o_y(i,j) = 1$ iff resource i is occupied by y at time $t+j$. Let $O_i = \max_y\{j \,:\, o_y(i,j) = 1\}$ denote the latest occupation of resource i for any instruction. For simplicity, we assume that an issue unit is occupied for one time slot per instruction issued.

An instruction y may produce a result, which may, for instance, be written to a register or a memory location. The number of clock cycles that pass from the issue time of y to the earliest possible issue time of an instruction that can use this result is called the *latency* of that instruction, denoted by $\ell(y)$. Hence, the result of instruction y issued at time t is available for instructions issued at time $t + \ell(y)$ or later. We assume here that latency is always nonnegative. A more detailed modeling of latency behavior for VLIW processors is given *e.g.* by Rau *et al.* [RKA99].

3.3. Terminology

Our code generation methods are generally "driven" at the IR level because this gives us the possibility to consider multiple choices for instruction selection.

We call an IR *fine-grained* for a given instruction set $\mathcal{I}$ if the behavior of each instruction in $\mathcal{I}$ can be represented as tree/forest/DAG patterns (see Figure 3.2; a formal definition will be given later) covering one or several operations in the IR DAG completely. In other words, there is no "complex" IR operator whose functionality could be shared by two adjacent target instructions, and the boundaries between two adjacent covering patterns therefore always coincide with boundaries between IR operations.

From now on, let us assume that our IR is fine-grained. This assumption is realistic for low-level IRs in current compilers, such as L-WHIRL in ORC, or the LCC IR [FH95] that we use in our implementation.

3.3.1. IR-level scheduling

An *IR schedule* of the basic block (IR DAG) is a bijective mapping $S : V \rightarrow \{1, ..., n\}$ describing a linear sequence of the n IR operations in V that is compliant with the partial order defined by E, that is, $(u, v) \in E \Rightarrow S(u) < S(v)$. A *partial IR schedule* of G is an IR schedule of a subDAG of G. A *subDAG* $G' = (V', E \cap (V' \times V'))$ of a DAG G is a subgraph induced by a subset $V' \subseteq V$ of nodes where for each $v' \in V'$ holds that all predecessors of v' in G are also in V'. A partial IR schedule of G can be extended to a (complete) IR schedule of G if it is prefixed to a schedule of the remaining DAG induced by $V - V'$.

3.3.2. Instruction selection

Instruction selection is performed by applying graph pattern matching to the IR DAG. The user specifies one or several patterns for each target instruction;

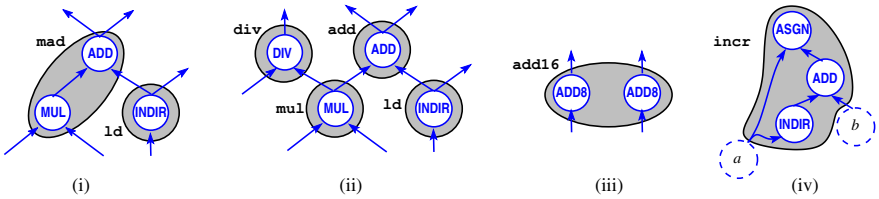

Figure 3.2.: Instruction selection by covering (a part of) an IR DAG by patterns, resulting in a target-level DAG. (i) A tree pattern for linked multiply-add (`mad`) matches the multiplication and the addition operation. (ii) `mad` does not match here because there is another operation (`div`) using the result of the multiplication operation. (iii) A forest pattern, describing a 16-bit SIMD-parallel addition of two 8-bit integers. (iv) A DAG pattern for incrementing a memory location with address a (common subexpression) by integer expression b.

a pattern is a small graph consisting of IR nodes and edges that describes the behavior of the instruction in terms of IR operations and data flow. The patterns may be trees, forests, or DAGs; some examples are given in Figure 3.2.

Covering a set χ of IR DAG nodes with a pattern B_y for an instruction y implies that each DAG node $v \in \chi$ is mapped one-to-one to a node $B_y(v)$ in the pattern with the same operator, and all DAG edges (v, w) between nodes $(v, w) \in \chi^2$ coincide with pattern edges $(B_y(v), B_y(w))$ between the corresponding pattern nodes. There may be additional constraints, such as on values of constants, on type, size, and location of operands, *etc.* Moreover, for the most interesting case $|\chi| > 1$, we usually have the constraint that "interior" values $v \in \chi$, corresponding to the sources of DAG edges (v, w) covered by a pattern edge $(B_y(v), B_y(w))$, must not be referenced by edges (v, u) to DAG nodes $u \notin \chi$, because they will not be exposed by instruction y for use by other instructions selected for u.

An *instruction selection* Y for an IR DAG $G = (V, E)$ and a given instruction set $\mathcal{I}$ is a surjective mapping $V \to \mathcal{I}$ such that each DAG node $v \in V$ is covered exactly once in a pattern for an instruction $y \in \mathcal{I}$. Moreover, for all DAG edges that are not covered by pattern edges and thus represent data flow between instructions, the type, size and storage of the source instruction's result must match the corresponding expectations for the target instruction's operand. By applying an instruction selection Y to a (IR) DAG G, we obtain a *target-level DAG* $\hat{G}$ whose $\hat{n}$ nodes correspond to target instructions and thereby to covered subsets χ of IR DAG nodes, and whose edges are prece-

dence constraints among target instructions and correspond to the IR DAG edges between covered node subsets.

Generally, there are several possible ways to cover each DAG node v and thus many possible instruction selections for a DAG G. In order to keep track of all alternatives, we denote by $\Psi(v)$ the set of all instructions in $\mathcal{I}$ such that v may be covered by a leaf node in a pattern for y. We will use Ψ in our algorithms to narrow the search space for matching instructions. It can be derived automatically by analyzing the patterns specified for the instructions in $\mathcal{I}$. Note that for IR trees, more general preprocessing methods for fast matching of tree patterns have been described in the literature, *e.g.* [HO82].

Example A common example in the DSP world consists in covering a multiplication and a dependent addition IR node with a single MAC (multiply and accumulate) instruction. Moreover, an integer multiplication of an operand by 2 is semantically equivalent to an integer addition of the operand with itself or to a left shift of the operand by 1. Thus, if an IR node v represents an integer multiplication, then $\Psi(v) = \{\texttt{MUL}, \texttt{MAC}, \texttt{ADD}, \texttt{LSH}\}$ contains instructions for integer multiplication, multiply-add, addition, and left shift. Our framework will consider all alternatives, since they may imply different resource usage and latency behavior. ■

3.3.3. Target-level scheduling

A *target schedule* is a mapping s from the set of time slots in all issue units, $\{1, ..., \omega\} \times \mathbb{N}$, to instructions such that $s_{i,j}$ denotes the instruction issued to unit u_i at time slot j. Where no instruction is issued to u_i at time slot j, $s_{i,j}$ is defined as $\Diamond$, meaning that u_i is idle. If an instruction $s_{i,j}$ produces a value that is used by an instruction $s_{i',j'}$, it must hold $j' \geqslant j + \ell(s_{i,j})$. Finally, the resource reservations by the instructions in s must not overlap. The accumulated reservations for all instructions scheduled in s can be described by a *resource usage map* or global reservation table [DSTP75], a boolean matrix RU_s where $RU_s(i,j) = 1$ iff resource i is occupied at time j.

The *reference time* $\rho(s)$ of a target schedule s is defined as follows: Let t denote the most recent clock cycle where an instruction (including explicit NOPs) is issued in s to some functional unit. If there is any fillable slot left on an issue unit at time t, we define $\rho(s) = t$, otherwise $\rho(s) = t + 1$.

The *execution time* $\tau(s)$ of a target schedule s is the number of clock cycles required for executing s, that is,

$$\tau(s) = \max_{i,j}\{j + \ell(s_{i,j}) : \ s_{i,j} \neq \Diamond\}.$$

A target schedule s is *time-optimal* if it takes not more time than any other target schedule for the DAG.

3.4. Classes of Schedules

This notion of time-optimality requires a precise characterization of the solution space of all target schedules that are to be considered by the optimizer. We will now define several classes of schedules according to how they can be linearized and re-compacted into explicitly parallel form by various compaction strategies, and discuss their properties. This classification will allow us to a-priori reduce the search space, and it will also illuminate the general relationship between VLIW schedules and linear schedules for (in-order-issue) superscalar processors.

3.4.1. Greedy Schedules

An important subclass of target schedules are the so-called greedy schedules where each instruction is issued as early as data dependences and available resources allow, given the placement of instructions issued earlier.

In a target schedule s, consider any (non-NOP) instruction $s_{i,j}$ issued on an issue unit u_i at a time $j \in \{0, ..., \rho(s)\}$. Let e denote the earliest issue time of $s_{i,j}$ as permitted by data dependences; thus, $e = 0$ if $s_{i,j}$ does not depend on any other instruction. Obviously, $e \leqslant j$. We say that $s_{i,j}$ is *tightly scheduled* in s if, for each time slot $k \in \{e, e+1, ..., j-1\}$, there were a resource conflict with some instruction issued earlier than time j if $s_{i,j}$ were issued (to u_i) already at time k.

A target schedule s is called *greedy* if all (non-NOP) instructions $s_{i,j} \neq \Diamond$ in s are tightly scheduled. Figure 3.3 (i) shows an example of a greedy schedule.

Any target schedule s can be converted into a greedy schedule without increasing the execution time $\tau(s)$ if, in ascending order of issue time, instructions in s are moved backwards in time to the earliest possible issue time that does not violate resource or latency constraints [CC95].

3.4.2. Strongly Linearizable Schedules and In-order Compaction

A schedule s is called *strongly linearizable* if for all times $j \in \{0, ..., \rho(s)\}$ where at least one (non-NOP) instruction is issued, at least one of these instructions $s_{i,j} \neq \Diamond$, $i \in \{1, ..., \omega\}$, is tightly scheduled.

The definition implies that every greedy schedule is strongly linearizable. The target schedules in Figure 3.3 (i) and (ii) are strongly linearizable. The

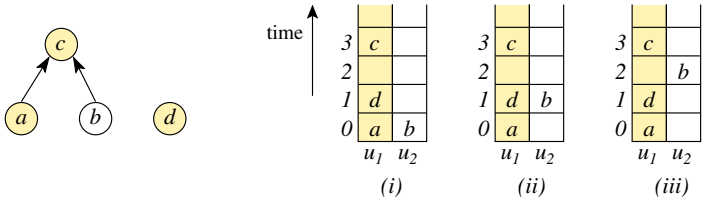

Figure 3.3.: A target-level DAG with four instructions, and three examples for target schedules: (i) greedy, strongly linearizable, (ii) not greedy, strongly linearizable, (iii) not greedy, not strongly linearizable. Instructions a, c and d with latency 3 are to be issued to unit u_1, and b to unit u_2 with $\ell(b) = 1$. We assume no resource conflicts.

schedule in Figure 3.3 (ii) is not greedy because instruction b is not tightly scheduled. The schedule in Figure 3.3 (iii) is not strongly linearizable because instruction b, which is the only instruction issued at time 2, is not tightly scheduled.

Any strongly linearizable schedule s can be emitted for an in-order issue superscalar processor without insertion of explicit NOP instructions to control the placement of instructions in s. We can represent each strongly linearizable schedule s by a sequence $\bar{s}$ containing the instructions of s such that the dynamic instruction dispatcher of an in-order issue superscalar processor, when exposed to the linear instruction stream $\bar{s}$, will schedule instructions precisely as specified in s. For instance, a linearized version of the schedule in Figure 3.3 (ii) is $\langle a, d, b, c\rangle$. We can compute a *linearized version* $\bar{s}$ from s by concatenating all instructions in s in increasing order of issue time, where we locally reorder the instructions with the same issue time j such that a tightly scheduled one of them appears first in $\bar{s}$. A similar construction allows to construct linearized schedules $\bar{s}$ for EPIC/VLIW processors and reconstruct the original schedule s from $\bar{s}$ if the in-order issue policy is applied instruction by instruction.

In the reverse direction, we can reconstruct a strongly linearizable schedule s from a linearized form $\bar{s}$ by a method that we call *in-order compaction*, where instructions are placed on issue units in the order they appear in $\bar{s}$, as early as possible by resource requirements and data dependence, but always in nondecreasing order of issue time. In other words, there is a nondecreasing "current" issue time t such that all instruction words issued at time 1,...,t − 1 are already *closed*, *i.e.*, no instruction can be placed there even if there should be a free slot. The instruction word at time t is currently *open* for further in-

sertion of instructions, and the instruction words for time $t+1$ and later are still unused. The "current" issue time t pointing to the open instruction word is incremented whenever the next instruction cannot be issued at time t (because issue units or required resources are occupied or required operand data are not ready yet), such that the instruction word at time $t+1$ is opened and the instruction word at time t is closed. Proceeding in this way, the "current" issue time t is just the reference time $\rho(s)$ of the schedule s being constructed.

As an example, in Figure 3.3 (iii), we cannot reconstruct the original schedule s from the sequence $\langle a, d, b, c\rangle$ by in-order compaction, as b would be issued in the (after having issued d at time 1) still free slot $s_{2,1}$, instead of the original slot $s_{2,2}$.

Generally there may exist several possible linearizations for a strongly linearizable schedule s, but their in-order compactions will all result in the same schedule s again.

In the context of in-order compaction of strongly linearizable schedules it is thus well-defined to speak about *appending* an instruction y to a schedule s (namely, assigning it an issue slot as early as possible but not earlier than $\rho(s)$, resulting in a new schedule) and about a *prefix* s' of a schedule s (namely the in-order compaction of the prefix $\bar{s}'$ of a suitable linearization $\bar{s}$ of s).

3.4.3. Weakly Linearizable Schedules

Weakly linearizable schedules s, such as that in Figure 3.3 (iii), can only be linearized if explicit NOP instructions are issued to occupy certain units and thereby delay the placement of instructions that are subsequently issued on that unit. In Figure 3.3 (iii), we could reconstruct the schedule *e.g.* from the sequence $\langle a, d, \texttt{NOP1}, b, c\rangle$ where `NOP1` denotes an explicit NOP instruction on unit 1.

As we focus on time optimization in this chapter, it is sufficient to consider greedy schedules when looking for a time-optimal one. We will see later that our optimizer actually takes strongly linearizable schedules into account, for technical reasons. However, non-strongly linearizable schedules such as that in Figure 3.3 (iii) may be superior to greedy or strongly linearizable schedules if register need or energy consumption are the main optimization criteria.

3.4.4. Non-linearizable Schedules

So far, we have not found any good example of a really non-linearizable schedule. However, such schedules may exist for certain architectures if more detailed latency models are considered. For instance, if there were pre-assigned registers, anti-dependences may actually induce negative latencies for certain

combinations of instructions, which means that an optimal schedule may have to schedule an instruction earlier than its predecessor in the dependence graph. Such cases cannot be modeled in a straightforward way if methods based on topological sorting of the dependence graph and in-order compaction are used, as in our case. Here, this is not an issue because we assume nonnegative latencies.

3.5. Advanced Code Obtained by Superoptimization

Note that we generally only consider such target schedules that can be derived from covering the IR with specified patterns for instructions in the given instruction set. Such a collection of patterns can only yield a conservative approximation to the —in general, undecidable— complete set of schedules with the same semantics as the IR. There may thus exist a more involved composition of instructions that computes the same semantic function as defined by the IR DAG in even shorter time, but there is no corresponding covering of the DAG. Finding such an advanced code by exhaustively enumerating arbitrary program fragments and semiautomatically testing or formally verifying them to see whether their behavior is as desired, is known as *superoptimization* [JNR02, Mas87] and beyond the scope of this work. Hence, the terms "(target) schedule" and "optimal" in this thesis are used with respect to the set of target schedules generatable by covering the IR DAG with patterns specified by the user for the instruction set.

3.6. Register allocation

A *register allocation* for a given target schedule s of a DAG is a mapping r from the scheduled instructions $s_{i,j}$ to physical registers such that the value computed by $s_{i,j}$ resides in a register $r(s_{i,j})$ from time slot $j + \ell(s_{i,j})$ and is not overwritten before its last use. For a particular register allocation, its register need is defined as the maximum number of registers that are in use at the same time. A register allocation r is optimal for a given target schedule s if its register need is not higher than that of any other register allocation r' for s. That register need is referred to as the *register need* of s. An optimal register allocation for a given target schedule can be computed in linear time in a straightforward way [Fre74]. A target schedule is *space-optimal* if it uses no more registers than any other target schedule of the DAG.

Chapter 4.

Integrated Optimal Code Generation Using Dynamic Programming

In this chapter we describe our dynamic programming algorithms for determining a time-optimal schedule for regular and irregular architectures. Our dynamic programming algorithms are based on enumeration of topological sortings; we introduce the concepts of selection tree and selection DAG. We also prove compression theorems for generating time-optimal schedules. The compression makes it possible to cope with basic blocks of reasonable sizes. We evaluate our dynamic programming algorithm on various basic blocks taken from real world DSP programs, and for regular and irregular architectures.

4.1. Overview of our Approach

We consider local code generation for in-order issue superscalar processors, regular VLIW processors (with a single general-purpose register file where each register is equally accessible to each instruction), and clustered VLIW processors (with multiple register files and operand residence constraints on parallel execution). We propose an integrated approach based on dynamic programming and problem-specific solution strategies which can deliver optimal or highly optimized code. In principle, our method generates all possible instruction selections and schedules but simultaneously applies compression and pruning techniques to reduce the size of the solution space. The algorithm finds a time-optimal solution among all generated code sequences. In a previous work, Kessler has exemplified that, due to the compression, the

algorithm can solve non-trivial problem instances of finding space-optimal schedules [Kes98]. In order to optimize for execution time on pipelined and VLIW processors, we introduce so-called *time profiles*, structures that summarize resource occupation and latency status of operations under execution at a given point in the code, as far as relevant for future instruction selection and scheduling decisions. As we will see later, partial solutions for the same subset of IR nodes with equal time profiles are comparable, that is, our dynamic programming algorithm needs to keep only one of these (namely a locally optimal one) as building block for the construction of further solutions, which leads to compression of the solution space.

For clustered VLIW architectures, we introduce the concept of *space profiles*, structures that describe the current residence of values that are alive at a given point in the code. Space profiles are used for comparison of partial solutions in a modified dynamic programming algorithm to produce time-optimal code for such architectures. We show that compression by local optimization of partial solutions for the same subset of IR nodes is applicable when their space profiles and time profiles are identical.

We organize the space of partial solutions as a two-dimensional grid spanned by axes for execution time and the number of IR nodes considered. By constructing partial solutions systematically in increasing order of execution time, less promising partial solutions are postponed as far as possible, so that we hopefully never need to explore them: As we will see, the first complete solution constructed by the algorithm is guaranteed to be an optimal one.

4.2. Main Approach to Optimal Integrated Code Generation

For a better explanation of our subsequent dynamic programming algorithms, we start with a simple brute-force algorithm for optimal integrated code generation that is based on exhaustive enumeration of target schedules, each of which can be generated by interleaving topological sorting of the IR DAG nodes with instruction selection and in-order compaction. We begin with the case of a homogeneous register file and then generalize the approach to clustered VLIW architectures.

4.2.1. Interleaved Exhaustive Enumeration Algorithm

The following enumeration algorithm constructs inductively all target schedules for larger and larger subDAGs of an IR DAG G as follows:

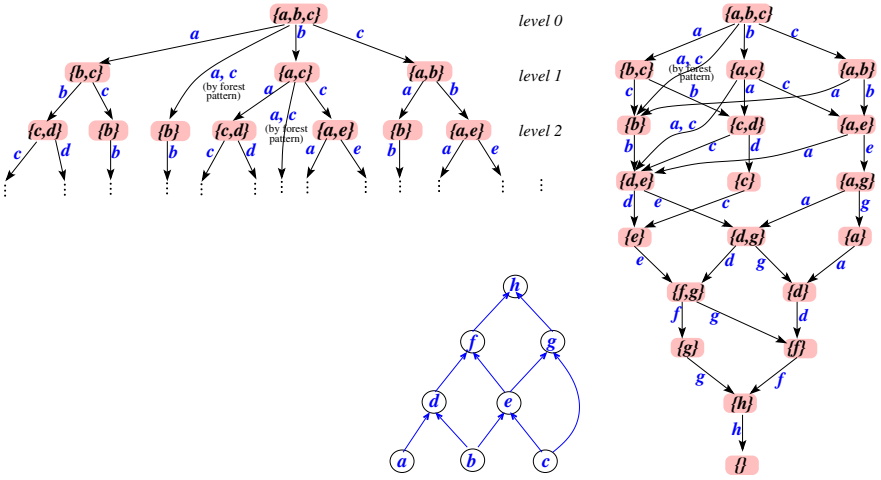

Figure 4.1.: The selection DAG (right hand side) of the example DAG (middle bottom) as a compression of the selection tree (left upper corner) where all selection nodes with the same zero-indegree set could be merged. Schedules are not shown. Note that selection edges for instructions whose pattern covers more than one IR node (here, we assume a forest pattern covering a and c) accordingly skip some levels in the selection DAG.

We start with an initial configuration consisting of a zero-indegree set z equal to the set of DAG leaves and an empty target schedule. The set *scheduled*(z) of already scheduled DAG nodes is thus still empty.

In each step, given a configuration (z, s) with a zero-indegree set z and some target schedule s for the subDAG G_z induced by the already scheduled nodes *scheduled*(z), the algorithm enumerates new target schedules with prefix s by selecting code for, and scheduling, at least one more IR operation in z: For each IR node $v \in z$, we explore all instructions $y \in \Psi(v)$ whose pattern may cover a set χ of non-scheduled operations, including v as a leaf of the pattern. For each v and each possible selection $y \in \Psi(v)$, we arrive at a new configuration by removing the set χ of nodes covered by y from the DAG, determining the resulting zero-indegree set z', and appending the instruction y to the target schedule s by in-order compaction, resulting in a new schedule s'. For each of these possible new configurations (z', s'), the same enumeration strategy is inductively applied, until all DAG nodes have been scheduled.

For each (partial) target schedule s' constructed by the algorithm, we im-

mediately compute its execution time and its register need. Among all enumerated complete target schedules for the entire DAG G, we pick a (time-) optimal one. By a straightforward extension of the algorithm, configurations with (partial) schedules s' whose register need exceeds the number of available physical registers or whose execution time exceeds a specified time limit would be discarded immediately and not considered further in the enumeration.

This strategy generates an enumeration tree whose nodes, also called *selection nodes*, are the configurations, and whose edges, called *selection edges*, denote single selection and scheduling steps made by the above enumeration algorithm, see Figure 4.1 (left hand side) for an example. We call this tree a *selection tree*. The selection nodes can be arranged in *levels*, such that the level of a selection node describing a zero-indegree set z corresponds to the number $|\mathit{scheduled}(z)|$ of already scheduled IR nodes "below" z, and thus selection edges are always directed from lower to higher level selection nodes, where the distance in levels spanned by the selection edge is just the number $|\chi|$ of nodes covered in the corresponding selection step. All selection nodes representing complete target schedules for the entire DAG with n nodes are at level n.

In principle, we could as well first enumerate all possible coverings Y of the DAG and then, for each such covering, enumerate all possible target schedules s (see Figure 4.2); this method would enumerate exactly the same set of target schedules. Likewise, we could enumerate all IR schedules S of the DAG completely and then apply to each S all possibilities for covering contiguous subsequences of operations in S by patterns for corresponding target instructions. Note that each possible covering by some pattern will eventually be possible in this scenario because the IR operations to be covered will eventually be enumerated contiguously in some schedule S. We however prefer interleaving the steps of selecting a "next" IR node to be scheduled and covering it, as this allows for incremental compaction and computation of the accumulated execution time, and for earlier compression and pruning of the solution space, which we will exploit later.

Theorem 4.2.1 *The interleaved enumeration algorithm enumerates all strongly linearizable target schedules for the input DAG.*

Proof Assume there exists a strongly linearizable target schedule t, thus based on instructions obtained by some covering of the IR DAG with specified patterns, that is not enumerated by the algorithm. Consider any linearized version of t, $\bar{t} = \langle y_1, ..., y_{\hat{n}_t} \rangle$. We conceptually compare t to each linearized target schedule $s = \langle x_1, ..., x_{\hat{n}_s} \rangle$ enumerated by the algorithm. By the assumption, for each such s there must be a position $k \in \{1, ..., \min(\hat{n}_t, \hat{n}_s)\}$ such that

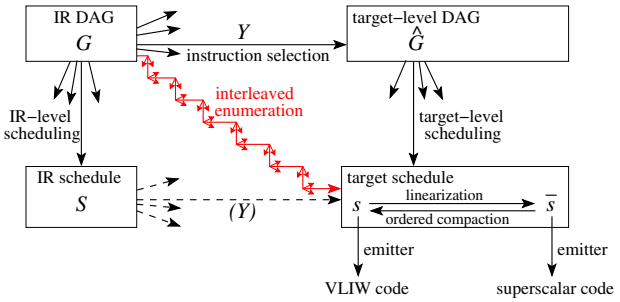

Figure 4.2.: Survey of the interleaved enumeration process.

$y_1 = x_1, \ldots, y_{k-1} = x_{k-1}$ but $y_k \neq x_k$. Now consider the zero-indegree set z in the configuration after y_1,...,y_{k-1} have been scheduled (for $k = 1$, z is just the set of all leaves, as no node has been scheduled yet). Let $z = \{v_1, ..., v_q\}$. The algorithm iterates through all v_i, $i = 1, ..., q$, and for each v_i through all possible patterns in $\Psi(v_i)$, covering v_i (and maybe some other, not yet scheduled, IR nodes). As t is a valid schedule consisting of instructions with specified patterns, and instruction y_k is scheduled after y_{k-1} in t, some pattern for y_k does cover some v_i in z. Hence, as all possibly covering patterns have been given as input to the enumeration algorithm, instruction y_k is in $\Psi(v_i)$ and will be enumerated, contradiction. □

4.2.2. Comparability of Target Schedules

The reader may have noticed that multiple instances of the same zero-indegree set z may occur in the selection tree. For all these instances, the same set *scheduled*(z) of nodes in the same subDAG G_z of G "below" z has been scheduled. This leads to the idea that we could perhaps optimize locally among several partial schedules with equal zero-indegree set instances, merge all these selection nodes to a single selection node and keep just one optimal partial schedule to be used as a prefix in future scheduling steps (see Figure 4.1). By this compression of the solution space, the selection tree becomes a *selection DAG*. In the same way as in the selection tree, the selection nodes in a selection DAG can be partitioned into $n + 1$ levels. An example is shown in Figure 4.1 (right hand side).

We will now define an equivalence relation among target schedules, called *comparability*, such that even with merging of selection nodes corresponding to comparable schedules it is still guaranteed that an optimal solution is enumerated.

The definition of comparability depends on the architecture and the optimization goal. In either case, the comparability definition must fulfill the following condition:

> *Comparable prefix schedule exchange condition (CPSEC):* Any prefix p in a linearized target schedule $\bar{s}$ can be exchanged by any schedule p' that is comparable with p, without affecting the correctness (but possibly the performance) of the resulting schedule.

We will consider three definitions of comparability for increasingly complex architectures in Sections 4.2.3, 4.2.4, and 4.2.5, and prove that these fulfil the CPSEC. A summary is given in Table 4.1.

4.2.3. Comparability I

For very simple RISC architectures with a single, non-pipelined functional unit and unit latency for all instructions, time optimization is not an issue. For minimizing the number of registers used, we define comparability by equality of zero-indegree sets, since there is a one-to-one relationship between a zero-indegree set z and $\mathit{alive}(z)$, the set of (results of) operations that reside in (virtual) registers at the configuration described by z:

$$\mathit{alive}(z) = \{\, u \in \mathit{scheduled}(z) : \exists (u, v) \in E,\ \ v \notin \mathit{scheduled}(z) \,\}$$

The resulting compression of the solution space considerably decreases the optimization time and makes it possible to generate space-optimal schedules for DAGs of reasonable size [Kes98].

Before we generalize this for time optimization, we note a property of the enumeration algorithm with compressed solution space:

Lemma 4.2.2 *If comparability is defined by equality of zero-indegree sets, all possible (with respect to the specified patterns) zero-indegree sets are enumerated even if occurrences of the same zero-indegree set are merged.*

Proof The interleaved enumeration algorithm enumerates all possible zero-indegree sets. For multiple occurrences of the same zero-indegree set in the selection tree, it will enumerate isomorphic subtrees that only differ in the schedules (see Figure 4.1). Hence, no zero-indegree set will be missed if all but one of these subtrees are cut. □

Table 4.1.: Comparability criteria for various optimal code generation goals and processor architectures.

Architecture	Optimization goal	Comparability	Section
single-issue, unit latency	minimum register instruction sequencing	equal zero-indegree sets	4.2.3
pipelined, in-order issue superscalar / VLIW	register-constrained minimum time code generation	equal zero-indegree sets and equal time profiles[1]	4.2.4
clustered VLIW	register-constrained minimum time code generation	equal zero-indegree sets, equal time profiles, and equal space profiles[1]	4.2.5

[1] Time profiles and space profiles will be introduced in Sections 4.2.4 and 4.2.5, respectively.

4.2.4. Comparability II, Time Profiles

For determining time-optimal schedules on single- and multi-issue processors with pipelined units and non-unit latencies and occupation times, the definition of comparability of target schedules involves more factors. We introduce the concept of a *time profile* to represent the occupation and latency status of all issue units and resources at a given time. A time profile of a target schedule s with reference time t records the occupation and latency effects of instructions in s that are under execution at time t and have not yet completed on every resource, that is, they still occupy some resource or their results have not yet been written, which may influence future scheduling decisions. Hence, time profiles are a generalization of resource usage maps that includes latency information, restricted to a small window of recently issued instructions in s.

We will see that, for time optimization for general pipelined VLIW and superscalar processors, two partial schedules are comparable if they have the same zero-indegree set and the same time profile. It will thus be sufficient to keep, among all comparable schedules, only one with least execution time, which results in considerable compression of the solution space.

Time Profiles

Given a target schedule s with reference time $t = \rho(s)$, let X_s denote the set of IR nodes covered by instructions in s that are under execution at time t (including those issued at time t) and that are sources of dependence edges to IR

nodes covered by other instructions[1]. Note that the nodes in X_s correspond to values that are being computed at time t.

For a processor with ω issue units $u_1, ..., u_\omega$ and f resources $U_1, ..., U_f$ with latest occupations after $O_1, ..., O_f$ clock cycles, respectively, we define a *time profile* P for s as a triplet

$$P = (\beta, \lambda, \pi) = \quad ((\beta_1, \ldots, \beta_\omega),\ \lambda, \\ (\pi_{1,0}, \ldots, \pi_{1,O_1-1},\ \pi_{2,0}, \ldots, \pi_{2,O_2-1},\ \ldots,\ \pi_{f,0}, \ldots, \pi_{f,O_f-1}))$$

where $\beta_i \in \{0, 1\}$ indicates whether some instruction has already been issued to issue unit u_i at time t, for $i \in \{1, ..., \omega\}$. $\lambda : X_s \to \mathbb{N}$ maps each value $v \in X_s$ under computation to its residual latency $\lambda(v)$, that is, the remaining number of clock cycles from t until its result becomes available. Hence, for an instruction $s_{i,j}$, issued to u_i at time $j \leqslant t$ and covering IR node v, its result becomes available for instructions issued at time $t + \lambda(v) = j + \ell(s_{i,j})$. The bit matrix entries $\pi_{i,j} \in \{0, 1\}$ specify whether resource i is occupied at time $t + j$, for $i \in \{1, ..., f\}$ and $j \in \{0, ..., O_i - 1\}$.

Example Figure 4.3 illustrates, for the ARM9E processor [ARM02], the time profile for the schedule `...; b=ADD(...); c=LD(b)` with reference time t. Here, $\beta = (0)$, $X_s = \{c\}$, and $\lambda = \{(c \mapsto 1)\}$, *i.e.*, the residual latency of c is 1. As the latest occupation distances O_1, O_2 are 1 on both resources, π is a 2×1 bit matrix. Here, $\pi = (0, 0)$, as no instruction has subscribed to any resource for time t yet. ■

Comparability and Compression with Time Profiles

For in-order issue superscalar and for VLIW architectures (corresponding to our machine model) with an ordinary homogeneous register file, we define comparability of two target schedules s, s′ by equality of the corresponding zero-indegree sets *and* equality of the time profiles obtained for s and s′, respectively.

The following theorem states that this comparability definition fulfills the CPSEC. In other words, zero-indegree set plus time profile contain all the information required to decide about the earliest time slot where the instruction for the node selected next can be scheduled.

[1] Note that in the case where a set χ of multiple IR nodes is covered by a single pattern, only the root node(s) in χ produce value(s) and the others can thus be omitted.

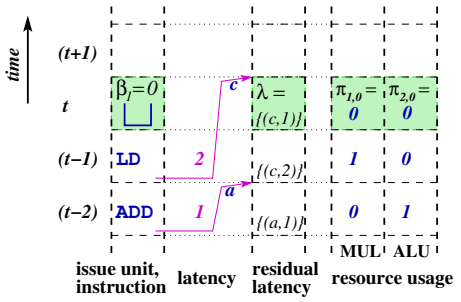

Figure 4.3.: Example of a time profile of a schedule (... a=ADD(...); c=LD(b)) for the ARM9E processor. The symbol ⊔ denotes a fillable slot on an issue unit.

Theorem 4.2.3 *For a DAG* G *and any subDAG* G_z *of* G*, any target schedule* s *of* G_z *that occurs as a prefix in some strongly linearizable schedule of* G *could be replaced by any other target schedule* s' *for the same subDAG* G_z *that has the same time profile* P *as* s*, without violating any data dependences or resource reservations.*

Proof Consider two target schedules s and s' of G_z with the same time profile $P = (\beta, \lambda, \pi)$ but possibly different reference times $t = \rho(s)$ and $t' = \rho(s')$. For any IR node $v \in z$ ready to be scheduled next, and any possible instruction $y \in \Psi(v)$ that can cover v (and thus could be appended to both s and s' by in-order compaction at the reference time or later), the following three situations may occur if y is scheduled:

(1) v is a leaf in G, *i.e.* it does not depend on any operation. Hence, y will be scheduled by in-order compaction as early as the required resources and issue unit are available (but not earlier than the reference time). As s and s' have equal resource usage π from the reference time on forward and the occupied issue slots in β are identical, too, instruction y has to wait equally long after the reference time if appended by in-order compaction to s and s'.

(2) v depends on an operand w computed by some instruction $y = s_{i,j}$ that is still in progress and thus occurs in X_s. As the mapping λ is identical for both s and s' (and thus, $X_s = X_{s'}$), operand w has the same residual latency $\lambda(w)$ relative to the reference time; thus, instruction y must wait equally long for w's result becoming available if appended to both s and s'. The argument for resource and issue unit occupation is the same as in (1).

(3) v depends on an operand w computed by some instruction $y = s_{i,j}$ that has already retired, *i.e.*, w's result is already available, at the reference time

or earlier. Hence, w is no longer contained in X_s. As the time profiles of s and s′ are equal, w is not contained in $X_{s'}$ either. Even though the respective instructions covering w, their latencies, issue units and resource occupations may differ in s and s′, this does no longer affect the issue time of y, because in-order compaction cannot schedule y earlier than the reference time. As the issue slot availability β and resource occupation π are also identical for both s and s′ (see case (1)), the issue time of y found by in-order compaction will be, relative to the respective reference time, the same for s and s′. □

Having established comparability by Theorem 4.2.3, we can now easily derive the corresponding compression theorem, which forms the basis of the dynamic programming algorithm. For this, we need the following observation:

Lemma 4.2.4 *If comparability is defined by equality of zero-indegree sets and time profiles, all possible (wrt. specified patterns) combinations of zero-indegree set and time profile are enumerated even if occurrences of configurations with the same zero-indegree set and time profile are merged.*

The proof is similar to that of Lemma 4.2.2.

Theorem 4.2.5 *For determining a time-optimal schedule of a DAG* G*, it is sufficient to keep, for any subDAG* G_z*, just one time-optimal target schedule* s *among all those target schedules* s′ *for the same subDAG* G_z *that have the same time profile* P*, and to use this* s *as a prefix for all target schedules that could be created from these target schedules* s′ *by a subsequent selection step.*

Proof It remains to show that merging configurations with comparable schedules and and local optimization among these does not lead to a situation where all optimal target schedules for G could be missed. By Lemma 4.2.4, all combinations of zero-indegree set and time profile that are enumerated by the interleaved enumeration algorithm will also be enumerated if merging is applied. By Theorem 4.2.3, we could exchange any prefix schedule s in a given schedule s_G for DAG G by a comparable one without conflicting with data dependences or resource allocations. If s and s′ have equal reference time (and thus execution time), it is thus safe to drop any of these. Where s and s′ differ in their reference time (and thus in their execution time), it is safe to drop the slower one. The optimality of those target schedules for the entire DAG that are finally kept when the enumeration with compression and local optimization has run to completion, follows by induction over all subDAGs of G. We will formally show this in Theorem 4.2.6, after having introduced the dynamic programming algorithm. □

Dynamic Programming Algorithm

The compression Theorem 4.2.5 yields directly a dynamic programming algorithm for time optimization, resulting from applying compression and local optimization in the interleaved enumeration algorithm. By keeping the interleaved enumeration order, compression is applied as early as possible. Among all configurations (z, s) (with s implying its time profile P) being enumerated, we will keep only one configuration for each pair (z, P), namely one with locally optimal target schedule, and store it in a data structure called *extended selection nodes (ESnodes)*. Hence, we attach to each ESnode (z, P) a locally optimal target schedule s^*. The dynamic programming algorithm inductively constructs all ESnodes as follows:

The algorithm starts with a single ESnode, consisting of a zero-indegree set z containing all leaves in the DAG G, and an empty time profile (all issue units are free and there are no dependences from operations that are still under execution, and no reserved resources). An empty schedule is associated with this ESnode.

From now on, we must make sure that, for each ESnode (z, P) with associated schedule s^*, the algorithm proceeds to a successor configuration $(z', ...)$, using s^* as an optimal prefix, only after having considered all configurations that are comparable to s^*. A partial order of constructing ESnodes that preserves this constraint is the order implied by set inclusion of the sets $\mathit{scheduled}(z)$ of already scheduled IR nodes. More conservative but easier to implement is constructing all ESnodes strictly in increasing order of IR level, because every selection step covers at least one IR operation. Hence, we apply the following expansion method to all ESnodes (z, P) level-wise (*i.e.*, for increasing IR schedule lengths $l = |\mathit{scheduled}(z)|$) until the entire DAG G has been scheduled:

For each ESnode (z, P) at the current level l, the algorithm considers, as before, all possible new configurations (z', s') by covering any $v \in z$ with any matching pattern for some instruction y, and determining the resulting target schedules s' by appending y to s by incremental in-order compaction. Each s' is a target schedule for the subDAG $G_{z'}$ that includes G_z and the nodes covered by the pattern chosen for y. Let P' denote the time profile for s'. For each new configuration (z', s'), one of the following three cases can occur:

Case 1: No configuration with a comparable schedule was enumerated yet, so we keep s' in a new ESnode (z', P').

Case 2: There exists an ESnode (z', P') with a comparable, previously enumerated schedule s^* for $G_{z'}$ whose execution time is not worse than that of s'. In this case, s' is discarded. (In the case of equal execution time, we may break ties according to secondary optimization goals such as register need.)

Case 3: s' was better than the associated s^*, then s^* is replaced in the ESnode (z', P') by s'.

This is continued level-wise, until all ESnodes at level n have been expanded completely and thus all possibilities for selection and scheduling of the last IR node in G have also been considered. Finally, we scan the schedules s^* associated with all ESnodes $(\emptyset, ...)$ at level $n+1$ and pick an optimal one. We claim that this schedule is optimal for the entire DAG.

Theorem 4.2.6 *For each strongly linearizable target schedule* s *of DAG* G *with time profile* P*, there exists, after termination of the dynamic programming algorithm above, a comparable target schedule* s^* *(not necessarily different from* s*) kept in the ESnode* $(\emptyset, P)$ *such that the execution time of* s^* *does not exceed that of* s*.*

Proof by induction over all possible subDAGs G_z, ordered by set inclusion of their node sets *scheduled*(z).

When the algorithm starts, z is the set of all leaves, *scheduled*$(z) = \emptyset$, and the target schedule associated with the initial ESnode is the empty schedule with execution time 0, hence the theorem holds for the initial case.

For the general case, consider any subDAG $G_z = (V_z, E_z)$ of G implied by the set $V_z =$ *scheduled*(z). Assume that the induction hypothesis holds for all subDAGs $G_{z'} = (V_{z'}, E_{z'})$ implied by a proper subset $V_{z'} =$ *scheduled*$(z') \subsetneq V_z$. Now assume that there is a target schedule s of G_z with time profile P such that s is faster than the schedule s^* ($\tau(s) < \tau(s^*)$) associated with ESnode (z, P) after the algorithm has finished its work, that is, after it has explored all its alternatives to generate code for G_z. Consider any linearization $\bar{s} = \langle y_1, ..., y_q \rangle$ of s. Instruction y_q scheduled last covers a set χ of IR nodes in *scheduled*(z). Consider $V' = V_z - \chi$. As y_q was scheduled last, the nodes in χ have no successors in V'. Hence, there exists a unique zero-indegree set z' with $V_{z'} = V'$. Let $G_{z'}$ denote the subDAG of G_z induced by $V_{z'}$. Consider the prefix schedule $s' = \langle y_1, ..., y_{q-1} \rangle$ of s, and let P' denote the time profile of s'. s' is a target schedule for $G_{z'}$. By Lemma 4.2.4, some ESnode (z', P') is eventually created by the dynamic programming algorithm, as the configuration (z', s') is enumerated by the interleaved enumeration algorithm. By the construction of the dynamic programming algorithm, the schedule s'^* that is associated with ESnode (z', P') is comparable to s' (note that s' may even be identical to s'^*). By the induction hypothesis, s'^* is optimal for (z', P'); in particular, the execution time of s'^* is not worse than that of s'. As y_q is a legal selection covering χ and can be scheduled next after any target schedule for $V_{z'}$, the schedule $\tilde{s}$ built by appending y_q to s'^* is eventually enumerated by our algorithm and compared to the local optimum s^* associated with (z, P). The

execution time $\tau(\tilde{s})$ cannot be worse than that of s. But this contradicts the assumption that the target schedule that is finally stored in ESnode (z, P) be worse than s, which completes the proof of the induction step. □

Again, the algorithm can easily be extended by immediately discarding computed configurations with target schedules s' whose register need or execution time exceeds predefined limits.

Finally, we have to explain why the dynamic programming algorithm uses in-order compaction and thereby considers the entire class of strongly linearizable schedules—as stated above, it would have been sufficient, for pure time optimization, if the interleaved enumeration algorithm limited its scope of enumeration to greedy schedules, by applying greedy compaction. Incremental greedy compaction inserts an instruction y into an existing schedule s as early as allowed by operand latencies and resource occupations of instructions in s, such that y might then actually be inserted in a quite early free slot in s that is out of the scope of the time profile of s, *i.e.*, there may be no residual latency of y at the reference time and no resource occupation by y may appear in the π entries of the time profile of the resulting new schedule s'. In this case, comparability as defined above would no longer be sufficient to decide which schedule to keep for future expansion. Comparable schedules usually differ in parts not covered by the time profile. Hence, we may have thrown away some comparable schedule where greedy compaction could later have inserted some instruction while this might not necessarily be possible with the schedule kept, and thus an optimal schedule might have been missed. If in-order compaction is applied, such insertions outside the scope of the time profile cannot occur.

4.2.5. Comparability III, Space Profiles

Now we consider clustered VLIW architectures with multiple register files. We will see that we can apply the same compression mechanism as above also in this case, provided that we define comparability by equality of zero-indegree sets, time profiles, and *space profiles*, where a space profile describes where the values in *alive*(z) reside, and when they reside there. For a formal definition, we need to introduce the concepts of *register classes* and *residence classes*.

Registers and Residences

We are given a DSP architecture with k registers $R_1, \ldots, R_k$ and κ different (data) memory modules $M_1, \ldots, M_\kappa$. Standard architectures with a monolithic memory have $\kappa = 1$ memory module.

A value can reside simultaneously in several *residence places*, for instance, in one or several registers and/or in one or several memory modules. For simplicity, we assume that the capacity of each register is one value[2]. Furthermore, we assume the capacity of a memory module to be unbounded.

Let $\mathcal{R} = \{R_1, \ldots, R_k\}$ denote the set of register names and $\mathcal{M} = \{M_1, \ldots, M_\kappa\}$ the set of memory module names. Then, $\mathcal{RM} = \mathcal{R} \cup \mathcal{M}$ denotes the set of all *residence places*.

The set of the residence places where a certain value v resides at a certain point of time t is called the *residence* of v at time t.

An instruction takes zero or more operands (called operand 1, operand 2 *etc.* in the following) and generates zero or more results (called result 1, result 2 *etc.* in the following). For each instruction y and each of its operands q and results q', the instruction set defines the set of possible operand residence places, $Res_o(y, q) \subseteq \mathcal{RM}$, and the set of possible result residence places, $Res_r(y, q')$.

Example Table 4.2 shows a few example instructions with constraints on operand and result residence for the TI-C62x DSP processor, a load-store architecture with two register files `A` and `B` and a single memory module. Note that arithmetic operations such as `MPY` can also take one operand from the other cluster via the corresponding cross path, see Figure 3.1. This cross path is modeled as a resource, as it can be used at most once per time slot. An intra-cluster direct load (`LDW`) instruction expects an address value as operand 1, which should reside in some register of the same cluster; the value being loaded actually resides in memory. In order to model memory dependences, we assume that the loaded value is implicitly given as operand 2 of a LOAD instruction. The loaded value is then written to a register in the same cluster. Similarly, a store instruction (`STW`) takes an address value as operand 1 and a register holding the value to be stored as operand 2, and creates a value residing in memory, which we denote as implicitly given result to model memory dependences. We summarize all instructions that move values directly between different memory modules and register classes as *transfer instructions*. In this sense, `LDW` and `STW` are transfer instructions as well. ■

[2]Note that this is not generally the case for most DSPs. For instance, for the TI-C62x family DSPs, the ADD2 instruction performs two 16-bit integer additions on upper and lower register halves with a single instruction. A corresponding generalization of our framework is left for future work.

Table 4.2.: Examples of compute and data transfer instructions for TI-C62x.

instruction (with example operands)			constraints on residence result 1	operand 1	operand 2	meaning
ABS	.L1	A1, A2	A	A	–	absolute value computation on unit L1
MPY	.M2	B3, B1, B2	B	B	B	intra-cluster multiplication on M2
MPY	.M2X	A1, B1, B2	B	A	B	multiplication on M2, using cross path A → B
MV	.S1X	A1, B2	B	A	–	inter-cluster copy register to register
LDW	.D2	*B1, B2	B	B	(mem)	load a register from memory
STW	.D1	B1, *A2	(mem)	A	B	store a register to memory

Register Classes, Residence Classes, and Versatility

We derive general relationships between registers and register classes by analyzing the instruction set.

For two different residence places R_i and R_j in $\mathcal{RM}$, we denote by $R_i \leqslant R_j$ (read: R_j is at least as *versatile* as R_i) when for all instructions y, $R_i \in Res_o(y, q) \Rightarrow R_j \in Res_o(y, q)$ for all operands q and $R_i \in Res_r(y, q') \Rightarrow R_j \in Res_r(y, q')$ for all results q'. In other words, wherever R_i can be used as operand or result, one may use R_j as well. We denote $R_i \equiv R_j$ for $R_i \leqslant R_j \wedge R_j \leqslant R_i$.

For a given set $\mathcal{I}$ of instructions of the target processor, a *register class* is a maximum-size subset of $\mathcal{R}$, containing registers that can, in all instructions in $\mathcal{I}$, be used interchangeably as operand 1, as operand 2, as result 1, *etc.*, respectively. $\mathcal{I}$ could be the entire instruction set, or may be narrowed to the set of instructions that are applicable at a certain scheduling situation.

Note that register classes are just the equivalence classes of the "equally versatile" relation ($\equiv$).

Example For the TI-C62x (see Figure 3.1), there are two register classes A and B consisting of 16 registers each, each one connected to a common memory. ■

A similar concept was proposed by Rau *et al.* [RKA99]; their access-equivalent register sets correspond to our register classes.

Following the generalization of registers to residences, we obtain the straightforward generalization of register classes to *residence classes*, where the residence class of a memory module residence is just that memory module name itself. Note that residence class is a static characterization for a particular hardware architecture. Let $\mathcal{RC}$ denote the set of all residence classes.

The thesis considers residence classes that do not overlap, which is a characteristic feature of clustered VLIW processors.

We assume that the residence places of global program variables (in memory modules and/or registers) are given, while we will optimize the residence classes of temporary variables (usually, in registers) for results computed by the instructions, along with the necessary data transfer operations. A method for optimizing the placement of global variables in clustered VLIW processors has been described by Terechko *et al.* [TTG+03].

Interleaved Enumeration with Generation of Transfer Instructions

As a first step to integrated code generation for clustered VLIW architectures, we extend the interleaved enumeration algorithm by automatic generation of data transfer operations.

Beyond enumerating all possibilities for choosing the next $v \in z$ and all possible instructions $y \in \Psi(v)$ covering v, resulting in a new configuration (z', s'), we now also enumerate all possibilities for choosing any alive value $v \in \mathit{alive}(z)$ and all possibilities of generating a possible transfer instruction for v to a residence class where v is not yet residing or under way at time $\rho(s)$, also resulting in a new configuration (z, s''). Note that there is no progress in the level, *i.e.* in scheduled IR nodes, but the resulting schedule and maybe the execution time will account for the additional transfer instructions. Note also that these resulting configurations are purely speculative, as it is not known yet whether there will be any need of v in that residence class at any time. If a transfer later turns out to be unnecessary and only occupied resources that could have been used better for other instructions, this will be reflected in inferior execution time, and schedules built from this one will eventually be discarded as suboptimal. On the other hand, if v is not present in a certain residence class when it is needed there as operand of some instruction y', then y' will simply not be selectable and not be enumerated by the algorithm—another configuration where v will be in the right place at the right time will eventually be enumerated, too.

Obviously this adds another level of combinatorial explosion to the interleaved enumeration algorithm, but there seems to be a large potential for compression, too. We will now define the comparability function that allows us to derive a dynamic programming algorithm for clustered VLIW architectures.

Space Profiles

Consider a configuration (z, s) with target schedule s, reference time $t = \rho(s)$, and zero-indegree set z. Let W_s denote the set of values that are, at time t, being copied by a data transfer operation (including transfers issued at time t). A *space profile* for s is a pair $Q = (\gamma, \mu)$, where $\gamma : (\mathit{alive}(z) - X_s) \to 2^{\mathcal{RC}}$ describes in which residence classes each of the already computed values in $\mathit{alive}(z) - X_s$ resides at time t, and $\mu : (W_s \times \mathcal{RC}) \to \mathbb{N},\ (v, RC) \mapsto j$ maps each value v that is being transferred to some residence class RC at time t, to the residual latency j of the transfer, such that the value will be available in RC for use by instructions issued at time $t + j$ or later. Note that a value $w \in X_s$ that is still under computation at time t becomes only eligible for data transfers issued not earlier than time $t + \lambda(w)$, that is, when w has been computed and is taken up in the γ part of the space profile.

Example For the TI-C62x, consider an IR addition c with operands a and b, such as in Figure 3.3, at the scheduling situation given by $z = \{c\}$, with $\mathit{alive}(\{c\}) = \{a, b\}$. Assume that a resides in register `A1` and b in register `B2`,

that is, the operands are in distinct register files, and the space profile entries are $\gamma = \{a \mapsto A,\ b \mapsto B\}$ and $\mu = \emptyset$. To perform a cluster-local addition on register file B, a data transfer of value a to that register file is necessary, that is, `MV .L2X A1,B3`. The transfer latency is one clock cycle. At the issue time t of the transfer, we have $\mu : \{(a, B) \mapsto 1\}$, meaning that a will be available in register file B for instructions issued at time $t + 1$ or later.

Alternatively, IR node c could have been covered with a TI-C62x instruction that does both a data transfer (using the cross path `X2` from A to B, see Figure 3.1) and computation within a single clock cycle, such as `ADD .L2X A1, B1,B3`, provided that the cross path is not yet occupied at that time. ■

We define comparability of target schedules by equality of zero-indegree sets, time profiles, and space profiles. The following theorem shows that this fulfills the CPSEC and creates the basis for the subsequent dynamic programming algorithm:

Theorem 4.2.7 *For determining a time-optimal schedule for a clustered VLIW architecture, it is sufficient to keep just one locally optimal target schedule* s *among all those target schedules* s' *for the same subDAG* G_z *that have the same time profile* P *and the same space profile* Q*. For any target schedule that could be created from these target schedules* s' *by a subsequent selection step, we could use* s *as a prefix without increasing the execution time.*

Proof Consider two target schedules s and s' of G_z with equal time profile $P = (\beta, \lambda, \pi)$ and equal space profile $Q = (\gamma, \mu)$ (thus $W_s = W_{s'}$) but potentially different reference times t and t', respectively. For the selectability of an instruction y covering the next $v \in z$, each operand v_q, $q = 1, 2, ...$, of y must be available in the respective residence class $RC \in Res_o(y, q)$ expected by y (a similar argument holds for the selectability of transfer instructions, too). Either, operand v_q is already available (*i.e.*, $v_q \in (alive(z) - X_s)$ and $RC_q \in \gamma(v_q)$), or the latency of a not yet finished computation ($\lambda(v_q) > 0$) or a transfer instruction ($\mu(v_q, RC_q) > 0$) has to be awaited, or a transfer of v_q to RC_q has not been issued (*i.e.*, neither $RC_q \in \gamma(v_q)$ nor $v_q \in W_s$). Hence, for both s and s', each operand q will be or become available in RC_q at the same relative time $t + j$ and $t' + j$ for some offset $j \geqslant 0$ from the respective reference time, or it will not be available and thus y not be selectable in either case. If selectable, instruction y will be scheduled by in-order compaction as early as possible, but not earlier than time $t + j$ or $t' + j$, respectively. As the subscriptions to resources (π) from the reference time on forward are identical in both cases, instruction y will be issued at the same relative time $t + j'$ and $t' + j'$, respectively, for some distance $j' \geqslant j$. A similar argument holds for scheduling selectable transfer instructions. This establishes comparability of

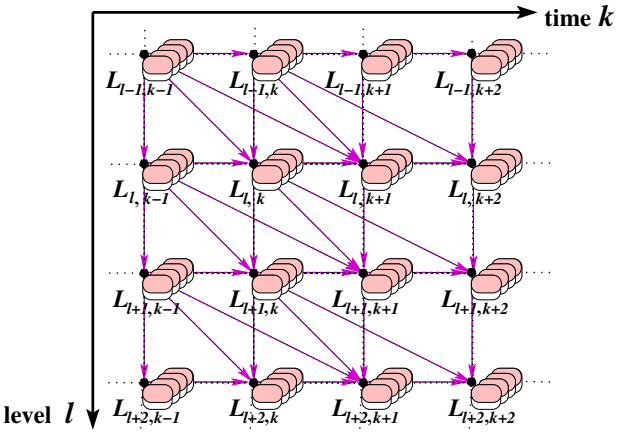

Figure 4.4.: Structuring the space of partial solutions as a two-dimensional grid.

s and s′. Hence, it is sufficient to keep one schedule with lower (reference) time. □

Dynamic Programming Algorithm for Clustered VLIW Architectures

We apply Theorem 4.2.7 to the interleaved enumeration algorithm with generation of transfer instruction of previous subsection, identifying ESnodes by triplets (z, P, Q) consisting of a zero-indegree set z, a time profile P, and a space profile Q. We obtain a new dynamic programming algorithm for the case of clustered VLIW architectures, which generalizes the algorithm of Section 4.2.4. A detailed description will be given in Section 4.3.3.

4.3. Improvement of the Dynamic Programming Algorithms

4.3.1. Structuring of the Solution Space

We structure the solution space as a two dimensional grid, as shown in Figure 4.4. The grid axes are the level (*i.e.*, the number of scheduled IR nodes), and the execution time. Every grid entry $L_{l,k}$ is a list of ESnodes containing a schedule s^* with level l and execution time k.

Let $maxLat = \max_{y \in I} \ell(y)$ denote the maximum latency of an instruction

(including transfer instructions), and *maxPatSz* denote the maximum number of IR nodes in a pattern for some instruction. Appending an instruction y that covers a set χ of DAG nodes to an existing schedule s increases the execution time of the resulting target schedule s' by at least 0 and at most *maxLat* cycles, and the level by at least 1 and at most *maxPatSz* nodes. In contrast, appending a transfer instruction to s does not increase the level because it does not cover any IR node (*i.e.*, $\chi = \emptyset$). Hence, if s was stored in some ESnode in $L_{l,k}$, the resulting ESnode for s' will be stored in exactly one of $L_{l+|\chi|,k}$, $L_{l+|\chi|,k+1}$, ..., $L_{l+|\chi|,k+maxLat}$. Seen from the perspective of the new schedule s' with level l' and time k', all ESnodes with schedules s from which s' could be built by appending an instruction will be located in the rectangular subgrid $L_{l'-maxPatSz,k'-maxLat}$, ..., $L_{l'-maxPatSz,k'}$, ..., $L_{l',k'-maxLat}$, ..., $L_{l',k'}$.

As a first consequence, this observation narrows the search space for all possible candidates for ESnodes with schedules that could be comparable to s', to the lists $L_{l',k'-maxLat}$, ..., $L_{l',k'+maxLat}$, provided that compression is consistently applied immediately after each enumeration step.

A generalization to a three-dimensional solution space structure by adding an axis for register need (or a higher-dimensional grid with one register need axis per register class for clustered VLIW architectures) is possible, and a similar observation of limits on the possible changes in register need can be made [Kes98].

For searching within each list $L_{l,k}$, we use hashing over the ESnodes' zero-indegree sets, time and space profiles.

4.3.2. Changed Order of Construction and Early Termination

The fact that adding an instruction never decreases time and never decreases the level (see the arrows in Figure 4.4) also implies precedence constraints among grid points (l, k) for the construction of partial solutions. Hence, we can change the order of construction as far as possible such that the more promising solutions will be considered first, while the less promising ones are set aside and reconsidered only if all the initially promising alternatives finally turn out to be suboptimal. This means that we will proceed along the time axis as first optimization goal, while the axes for length (and maybe register need) have secondary priority.

A complete IR schedule ends at level n (after all IR operations are scheduled). This allows us to optimize the look-up for final solutions by simply checking $L_{n,k}$ in the solution space after all configurations with execution time k have been expanded. If there is any ESnode in $L_{n,k}$, its associated schedule s^* is time-optimal, and we can stop.

4.3.3. Putting the Pieces Together: Time-optimal Code Generation for Clustered VLIW Architectures

An instruction y that may match a node v is only *applicable* (and thus, selectable) in a certain selection step if its operands are available in the "right" residence classes. Further instructions may become applicable after applying one or several data transfer instructions that broaden the residence of some values. For now, we do not generate transfers to memory classes.

According to Theorem 4.2.7, it is sufficient to keep among all target schedules with equal zero-indegree set, equal time profiles and equal space profiles, one schedule s^* with shortest execution time, which we store as an attribute $\eta.s^*$ of an ESnode $\eta = (z, P, Q)$.

The overall algorithm, *clustered_timeopt*, is depicted in Figure 4.5 and Figure 4.6. Initially, the solution space L, structured as two-dimensional grid, contains a single ESnode with the DAG leaves as zero-indegree set z_0 (lines 4–5). The initial ESnode has an empty schedule (line 5) with execution time 0 and level 0 (line 6). The outer loop over k (line 8) progresses along the execution time axis of the solution space. The second axis of progress is the level l (line 10). The algorithm progresses in a wave front manner by first generating all schedules that can be built from locally optimal schedules $\eta.s^*$ stored in every ESnode in $L_{l,k}$, by selecting a zero-indegree node v and an instruction $y \in \Psi(v)$, and appending y to s^* by incremental in-order compaction (lines 12–18) for each ESnode. This generates new ESnodes, which are then compared to previously computed ESnodes and inserted in the solution space by the function *update_solution_space*, given in Figure 4.6. Before the algorithm proceeds to the next grid point, it tries all possibilities for inserting transfer instructions for all already available, alive values to all possible register classes RC where that value is not yet present (*i.e.*, RC is not in $\gamma(v)$) or under way (*i.e.*, (v, RC) is not in the domain of μ) (lines 20–23). The function *update_solution_space* is called again (line 24) because new ESnodes are generated, as profiles and maybe execution time are affected even though there is no progress in the IR level l. Then the algorithm proceeds to the next level. After all ESnodes at time t have been processed, the algorithm progresses on the time axis. The algorithm continues until the first nonempty list $L_{n,k}$ of ESnodes at level n is considered, which means that all IR nodes have been scheduled. This is checked by function *checkstop* (lines 51–54 in Figure 4.6). As there was no nonempty ESnode at level n at a time value encountered earlier, this solution must be optimal, and we can stop.

The structuring and traversal order of the solution space allows us to optimize the memory consumption of the optimization algorithm. As soon as we have processed all ESnodes from list $L_{l,k}$, we can safely remove $L_{l,k}$ completely

```
1:  L_{l,k} ← new empty list of ESnodes, for all l, k;
2:  int maxtime ← 0;

3:  function clustered_timeopt ( DAG G with n nodes and set z_0 of leaves)
4:   η_0 ←new ESnode(z_0, P_0, Q_0);
5:   η_0.s* ← empty schedule;
6:   L_{0,0}.insert(η_0);
7:   // outer loop: over time axis
8:   for k from 0 to infinity do
9:      checkstop(k);  // terminate if a solution with time k exists
10:     for level l from 0 to n − 1 do
11:        for all η = (z, P, Q) ∈ L_{l,k} do
12:           let t denote the reference time of the schedule s* stored in η
13:           for all ν ∈ z do
14:              for all instructions y ∈ Ψ(ν) that are now selectable,
15:                    given z, time profile P and space profile Q do
                       // instruction selection:
16:                 let χ = { IR nodes covered by the pattern for instruction y };
17:                 z' ← selection(z, χ);  // new zero-indegree set
18:                 s' ← incr_in-order_compaction(s, y);  // new schedule
19:                 update_solution_space(η, l, k, z', s', |χ|);
                    // enter new partial solution
20:           for all already available alive values ν ∈ alive(z) − X_s do
                 consider the space profile Q = (γ, μ):
21:              for all possible transfers T of ν to a register class RC ∉ γ(ν)
                       where (ν, RC) ∉ domμ do
                    // suppress transfers to residence classes where ν
                    // already resides or to where ν is already under way
22:                 if T selectable then
23:                    s' ← incr_in-order_compaction(s, T);  // new schedule
24:                    update_solution_space(η, l, k, z, s', 0);
25:        free L_{l,k} and all ESnodes inside;  // no longer needed
26:  end function clustered_timeopt
```

Figure 4.5.: Dynamic programming algorithm for time-optimal code generation, taking data transfers into account.

```
27:  function update_solution_space ( ESnode η = (z, P, Q),
28:      level l, time k,
29:      zero-indegree set z′, target schedule s′,
30:      integer p (progress in level) )
31:   if register need of s′ exceeds resources then
32:      discard s′;
33:      return;
34:   P′ ← time profile of s′;
35:   Q′ ← space profile for s′;
36:   k′ ← τ(s′); // execution time of s′
37:   η′ ← new ESnode (z′, P′, Q′) with associated schedule s′;
38:   for all L_{l+p,j} with k ⩽ j ⩽ maxtime do // note that maxtime ⩽ k+maxLat
39:      if η″ ← L_{l+p,j}.lookup(z′, P′, Q′) exists then
40:         break;
41:   if η″ = (z′, P′, Q′) exists then
42:      // j denotes the execution time of the schedule stored in η″
43:      if k′ < j then
44:         L_{l+p,j}.remove(η″); // as it was suboptimal
45:         L_{l+p,k′}.insert(η′); // improved solution
46:      else forget η′;
47:   else L_{l+p,k′}.insert(η′); // first solution found for (z′, P′, Q′)
48:      maxtime ← max(maxtime, k′); // update search scope for lookup
      // now check whether a new solution was entered with current time k:
49:   if l + p = n then checkstop(k);
50:  end function update_solution_space

51:  function checkstop(k)
52:   if L_{n,k}.nonempty() then
53:      exit with solution η.s* for some η ∈ L_{n,k};
54:  end function checkstop
```

Figure 4.6.: The algorithm of Figure 4.5 continued.

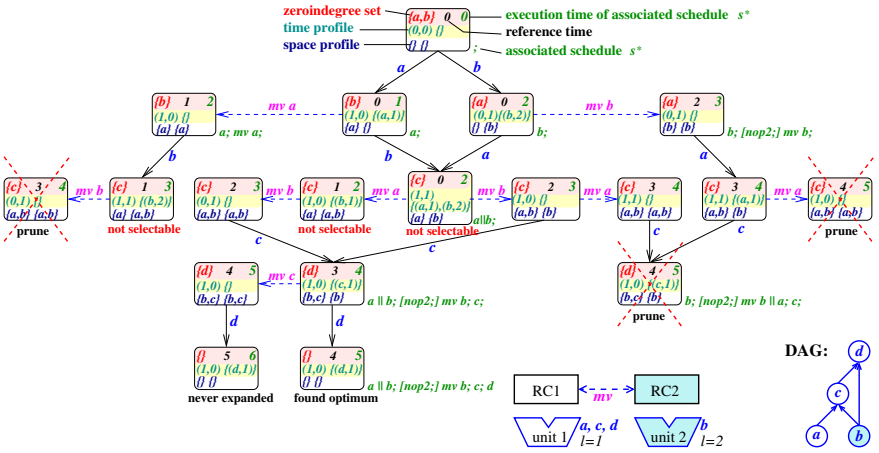

Figure 4.7.: Solution space (extended selection DAG) for the example DAG in the right lower corner.

because they will never be looked up again (line 25).

Function *update_solution_space* first checks the register need of a generated schedule s' and discards it immediately if available resources are exceeded (lines 31–33). Otherwise, the algorithm derives the execution time, time profile and space profile for s' and creates a new ESnode with s' as associated schedule (lines 34–37). Then it tests if the new node is to be inserted into the solution space or not. For that, the algorithm searches for an ESnode η'' with the same zero-indegree set, time and space profile (lines 38–40). If there is no such node, the new ESnode is inserted (line 47). If a comparable ESnode exists (line 43) but has a worse execution time j, then the "old" ESnode η'' was found to be suboptimal and is replaced by the current one, which is inserted in $L_{l+p,k'}$ instead (line 45). Otherwise the new solution s' was no improvement, and we discard it (line 46).

4.3.4. Example

Figure 4.7 illustrates the resulting extended selection DAG for the example DAG shown in the lower right corner of Figure 4.7. The example assumes a target processor with two issue units. Instructions that could be selected for DAG node b have latency 2 and are to be issued to unit u_2, those for all other DAG nodes to u_1 with latency 1. For simplicity, we assume that the processor has only two residence classes, namely register files RC1 and RC2, which are

attached to units u_1 and u_2, respectively.

For better readability, each node of Figure 4.7 is organized into three layers. The top layer of each node represents the zero-indegree set z, the reference time t, and the execution time τ of the associated schedule s^*. For example, the top node contains the set of DAG leaves as zero-indegree set, $\{a, b\}$, and the execution time is 0, as the initial schedule is empty. The second layer represents parts of the time profile: the issue unit usage (β) and the residual latencies (λ) at time t, where the latter is written as set of pairs of value and residual latency. Initially, all functional units are empty and the reference time is 0. The bottom layer shows the γ part of the space profile, *i.e.* the mapping of already available, alive values to residence classes; the μ part is not shown. Initially, the nodes reside in none of the residence classes. We could have started with a preset residence profile, but here we assume for simplicity that the leaves are constants and do not need to reside in any residence class. We also display, for some ESnodes, the associated schedule s^* in pseudoassembler format at their right lower corner.

In Figure 4.7, the ESnodes are only grouped according to their level (increasing from top to bottom). The dashed arrows represent transfer instructions inserted by the algorithm; these make no progress in level. The crossed ESnodes are pruned by the algorithm. ESnodes marked "not selectable" are not expanded further because their space profile does not match the requirements of the candidate instructions that could be selected there.

The second ESnode in the last row contains the optimal solution, which takes 5 clock cycles, and contains a transfer instruction (move) inserted by the algorithm. Table 4.3 gives the complete schedule: for each time slot it shows the instruction word, the status of the functional units and the residences of alive values. Note that node d is alive at the exit of the basic block (time slot 5) and resides in RC1.

4.3.5. Heuristic Pruning of the Solution Space

Despite considerable compression of the solution space, the combinatorial complexity of the dynamic programming algorithm remains high. For larger problem instances, we therefore enhance the algorithm with a heuristic that limits the number of configurations generated at each intermediate stage. This heuristic pruning considerably reduces computation time and still generates highly optimized code, which however is no longer guaranteed to be optimal.

For architectures with homogeneous register files, the heuristic limits the number of generated variants at each selection step of the algorithm (line 13 in Figure 4.5). This is controlled by an integer value N, that only considers at most N candidates from the zero-indegree set z for selection. If $N = 1$,

Table 4.3.: Final optimal solution for the example of Figure 4.7. The first column indicates the issue time slots in the schedule. The second column represents the instruction word (linearized form), and the third column shows the mapping to issue units. Symbol ◊ indicates that the issue unit is idle. `nop2` denotes a NOP on unit 2 to fill the delay slot of computing b.

t	instruction word	issue unit occupation u_1	u_2	values resident in RC1	RC2
0	a ‖ b	a	b	–	–
1	[nop2]	◊	◊ (delay)	a	–
2	mov b	◊	mov	a	b
3	c	c	◊	a,b	b
4	d	d	◊	b,c	b
(5)	–	◊	◊	d	–

we obtain a variant of list scheduling where the next node v is selected randomly among all the candidates in the zero-indegree set z. Observe that we still consider all possible coverings of that candidate node v. This is necessary to guarantee that at least one successor configuration will be found. If we additionally limit the selection among all possible target instructions ($\Psi(v)$) to a subset, the algorithm may fail to find a solution, which is particularly common in the case of clustered VLIW architectures where not all operands may be available in the expected residence classes.

In case of clustered VLIW architectures, we additionally control the number of transfer instructions inserted after each selection step. This is achieved in the similar manner as described above, where at most M transfers are issued (line 20 in Figure 4.5).

4.3.6. Beyond the Basic Block Scope

An *extended basic block* [Muc97] is a set of basic blocks connected by control flow edges that form a tree, that is, there are no join points of control flow (except maybe for the root block of the tree). If we process the basic blocks in topological order for generating code, we can simply propagate the outgoing time and space profile of a basic block B (after having scheduled the branch instruction) forward to all control flow successors of B as ingoing time and space profile, and continue code generation for each successor independently.

For the successor block, we thus start the algorithm with that ingoing time and space profile instead of blank profiles. Note that we do not consider moving IR operations across basic block boundaries, in contrast to global scheduling techniques such as trace scheduling [Fis81], and we do not consider pattern matching across branches either.

If there were join points of control flow, we would instead lose precision because normally the ingoing profiles do not match. In the context of backward branches, some ingoing profiles may not even be known in advance. Extending our method to entire loops is an issue of future work.

4.4. Implementation and Evaluation

We implemented a prototype compiler called OPTIMIST with the above dynamic programming algorithms for integrated code generation. We use LCC [FH95] as C front-end, and the rest of the system is implemented in C++. The target processor is specified in our XML-based architecture description language xADML (see Chapter 8). The specification contains structural information of the target processor: issue units, resources, register sets and memory modules grouped in residence classes. It specifies the instruction set architecture in terms of DAG patterns, reservation tables and residence constraints. Finally, for clustered architectures it lists the set of transfer instructions that are to be considered by the optimizer.

Table 4.4 shows the time requirements for finding a time-optimal schedule on a collection of basic blocks taken from the TI-C64x benchmarks and FreeBench. We considered only basic blocks of the benchmarks with sizes greater or equal to 10 nodes. The evaluation was performed for the ARM9E (in ARM mode) and TI-C62x processors. In order to study the effect of clustering on optimization times and the complexity of the solution space, we added a single-cluster variant of the TI-C62x, with a single register file and an issue width of four. The measurements have been performed on a Linux PC with 1.6 GHz AMD processor and 1.5 GB RAM. Column BB refers to the corresponding basic block in the benchmark. Columns 4–6, 7–9 and 10–12 report results for the ARM9E, single-cluster TI-C62x, and the TI-C62x, respectively. Columns 4, 7 and 10 display the time (in seconds) for finding a time-optimal schedule. Columns 5, 8 and 11 indicate the number of occurrences of comparable ESnodes that are detected by the algorithm and merged into a single node. Column #ESnodes lists the memory requirement by reporting the total number of ESnodes in the solution space. The ratio between the total number of configurations generated by dynamic programming (#merged + #ESnodes) and the number of ESnodes kept (#ESnodes) represents the aver-

age indegree of an ESnode, which gives a rough impression of the compression achieved. For instance, for the ARM processor and basic block 24 of benchmark codebk_srch, we get an ESnode indegree of almost 9 on average. Computations that did not produce a result within 6 hours or exceeded the available memory appear in the table as dashes.

Table 4.5 gives results for the ARM9 and a single-cluster version of TI-C62x, and Table 4.6 for TI-C62x, using the heuristic pruning method H_M^N that limits the number of candidates v considered from the zero-indegree set to N and, for TI-C62x, the number of data transfers issued to M, per configuration expanded by the algorithm. Note that $M > 2$ makes no sense for the TI C62x, as at most 2 transfers can run in parallel.

We observe that the single-cluster variant of C62x leads to higher optimization times and larger solution spaces, because its larger issue width, more resources, and deeper pipelines yield more variation in time profiles. Medium-sized problems with up to 40 nodes can be solved optimally.

When comparing the single-cluster variant to the double-cluster C62x, we observe a dramatic increase in optimization time and space requirements, in spite of the higher compression rate achieved by space profiles. This demonstrates the effect of the additional transfer instructions. Most problems with more than 20 nodes cannot be solved optimally.

The heuristic pruning methods H_M^N show a much larger impact of the number N of zero-indegree candidates considered on both code quality and optimization time and space requirements, compared to the limit M on transfers. In many cases, time-optimal or almost time-optimal code is found already for $N = 2$, in the cases where we could verify this using the dynamic programming algorithms.

Table 4.4.: Optimization time and space for various basic blocks from the TI-C64x DSP benchmarks.

			ARM9E-ARM mode			TI-C62x single-cluster			TI-C62x		
benchmark	BB	size	t[s]	#merged	#ESnodes	t[s]	#merged	#ESnodes	t[s]	#merged	#ESnodes
analyzer	51	34	98.9	546250	62965	12623.2	2547538	630132	—	—	—
analyzer	203	31	56.5	309673	49833	14.1	34559	29780	—	—	—
bmmse	4	14	0.1	489	165	0.1	523	375	10.9	15215	7042
bmmse	10	17	0.3	2558	582	0.8	5349	2413	6404.2	860455	204056
bmmse	11	14	0.1	768	234	0.2	825	562	56.8	38938	15557
codebk_srch	12	15	0.1	595	236	0.2	658	678	163.4	83220	33667
codebk_srch	16	21	0.7	6880	1512	8.3	47665	19613	—	—	—
codebk_srch	20	23	2.2	20380	3723	45.0	240079	76005	—	—	—
codebk_srch	24	42	178.5	783854	98580	—	—	—	—	—	—
fir_vselp	6	23	0.5	4479	1215	4.4	20483	10331	—	—	—
fourinarow	6	29	0.2	862	561	1.2	4976	2214	10.0	23599	9203
irr	4	21	0.4	4526	1016	5.8	40477	12036	—	—	—
irr	7	38	2.1	8753	3259	38.0	87666	47233	—	—	—

Table 4.5.: Impact on optimization time and code quality of heuristic for ARM9E and single-cluster C62x.

			ARM9E-ARM mode							TI-C62x single-cluster						
			$H^{N=1}$		$H^{N=2}$		$H^{N=3}$		OPT	$H^{N=1}$		$H^{N=2}$		$H^{N=3}$		OPT
benchmark	BB	size	t[s]	τ[cc]	t[s]	τ[cc]	t[s]	τ[cc]	τ[cc]	t[s]	τ[cc]	t[s]	τ[cc]	t[s]	τ[cc]	τ[cc]
analyzer	51	34	0.1	21	0.7	15	8.2	15	15	0.1	43	1.5	18	58.3	18	18
analyzer	59	45	0.1	27	1.1	18	30.3	18	—	0.1	55	3.1	29	87.9	22	—
bmmse	0	53	0.1	41	1.4	33	57.2	33	—	0.1	53	44.2	24	—	—	—
bmmse	3	141	0.2	159	303.2	132	—	—	—	0.5	218	—	—	—	—	—
codebk_srch	10	57	0.1	66	3.3	58	—	—	—	0.1	75	73.4	50	—	—	—
codebk_srch	20	23	0.1	23	0.1	17	0.7	17	17	0.1	36	0.3	23	5.1	18	14
codebk_srch	24	42	0.1	43	0.7	33	8.1	33	33	0.1	67	2.8	37	266.4	31	—
irr	7	38	0.1	48	0.2	41	0.8	41	41	0.1	56	0.8	35	7.2	31	30
minerr	5	88	0.1	105	41.6	93	—	—	—	0.3	107	1393.9	59	—	—	—

Table 4.6.: Computation time and code quality with heuristic pruning, for the TI-C62x.

benchmark	BB	size	$H^{N=1}_{M=1}$ t[s]	$H^{N=1}_{M=1}$ τ[cc]	$H^{N=1}_{M=2}$ t[s]	$H^{N=1}_{M=2}$ τ[cc]	$H^{N=2}_{M=1}$ t[s]	$H^{N=2}_{M=1}$ τ[cc]	$H^{N=3}_{M=1}$ t[s]	$H^{N=3}_{M=1}$ τ[cc]	$H^{N=2}_{M=2}$ t[s]	$H^{N=2}_{M=2}$ τ[cc]	OPT τ[cc]
bmmse	4	14	0.1	18	0.1	18	0.5	9	1.2	9	1.0	9	9
bmmse	10	17	0.4	23	0.5	23	4.2	14	5.2	9	4.5	14	9
bmmse	11	14	0.3	18	0.4	18	1.3	10	4.8	9	1.5	10	9
codebk_srch	12	15	0.1	27	0.1	27	1.3	19	4.6	15	1.9	19	15
codebk_srch	20	23	0.2	34	0.2	34	16.9	21	5816.2	19	24.0	21	—
codebk_srch	25	11	0.1	16	0.1	16	0.1	11	0.2	7	0.2	11	7
codebk_srch	29	10	0.1	13	0.1	13	0.1	7	0.1	7	0.2	7	7
codebk_srch	31	15	0.1	15	0.1	15	0.1	14	0.2	14	0.1	14	14
codebk_srch	34	19	0.2	31	0.2	31	12.4	17	21.7	16	21.1	17	—
codebk_srch	42	14	0.1	19	0.1	19	0.9	14	2.1	14	1.6	14	14
fir_vselp	10	17	0.2	24	0.5	24	2.2	15	6.8	15	11.0	15	—
irr	4	21	0.1	27	0.1	27	1.2	13	19.8	13	1.5	13	—
mac_vselp	3	23	0.4	35	0.9	35	20.3	26	155.9	26	56.9	26	—
vec_sum	3	17	0.2	29	0.6	29	5.1	15	35.2	15	30.1	15	15

Chapter 5.

Energy Aware Code Generation

Optimal integrated code generation is a challenge in terms of problem complexity, but it provides important feedback for the resource-efficient design of embedded systems and is a valuable tool for the assessment of fast heuristics for code generation. In this chapter we present a method for energy optimal integrated code generation for generic VLIW processor architectures that allows to explore trade-offs between energy consumption and execution time.

5.1. Introduction to Energy Aware Code Generation

Power dissipation in embedded systems is of serious concern especially for mobile devices that run on batteries. There are various approaches in embedded processor design that aim at reducing the energy consumed, and most of these have strong implications for power-aware code generation.

Voltage scaling reduces the power consumption by reducing voltage and clock frequency. The processor can be switched to such a power-saving mode in program regions where speed is of minor importance, such as waiting for user input. This technique is only applicable to coarse-grained regions of the program because transitions between modes have a non-negligible cost.

Clock gating denotes hardware support that allows to switch off parts of a processing unit that are not needed for a certain instruction. For instance, for integer operations, those parts of the processor that only deal with floatingpoint arithmetics can be switched off. Deactivation and reactivation require some (small) additional amount of energy, though. This feature for fine-grained power management requires optimizations by the code generator that avoid long sequences of repeated activations and deactivations by closing up instructions that largely use the same functional units. The method described in this chapter will take this feature into account.

Pipeline gating reduces the degree of speculative execution and thus the utilization of the functional units. And there are several further hardware design techniques that exploit a trade-off between speed and power consumption. See [BBS+00] for an overview. Other factors that influence power dissipation are rather a pure software issue:

Memory accesses contribute considerably to power dissipation. Any power-aware code generator must therefore aim at reducing the number of memory accesses, for instance by careful scheduling and register allocation to minimize spill code, or by cyclic register allocation techniques for loops, such as register pipelining [SSWM01].

Switching activities on buses at the bit level are also significant. In CMOS circuits, power is dissipated when a gate output changes from 0 to 1 or vice versa. Hence, bit toggling on external and internal buses should be reduced. Hardware design techniques such as low-power bus encoding may be useful if probability distributions of bit patterns on the buses are given [Ped01]. However, the code generator can do much more. For instance, the bit patterns for subsequent instruction words (which consist of opcodes, register addresses and immediates) should not differ much, *i.e.* have a small Hamming distance. This implies constraints for instruction selection and for register assignment as well as for the placement of data in memory. Moreover, the bit patterns of the instruction addresses (*i.e.*, the program counter value) do matter. Even the contents of the registers accessed have an influence on power dissipation. Beyond the Hamming distance, the *weight* (*i.e.* the number of ones in a binary word) may have an influence on power dissipation – positive or negative.

Instruction decoding and execution may take more or less energy for different instructions. This is a challenge for instruction selection if there are multiple choices. For instance, an integer multiplication by 2 may be replaced by a left shift by one or by an integer addition. Often, multiplication has a higher base cost than a shift operation [MSW01]. Different instructions may use different functional units, which in turn can influence the number of unit activations/deactivations. Hence, resource allocation is a critical issue. In the multiplication example, if the adder is already "warm" but the shifter and multiplier are "cold", *i.e.* not used in the preceding cycle, the power-aware choice would be to use the adder, under certain conditions even if another addition competes for the adder in the same time slot.

Execution time in terms of the number of clock cycles taken by the program is directly related to the energy consumed, which is just the integral of power dissipation over the execution time interval. However, there is no clear correlation between execution time and *e.g.* switching activities [STD94]. There are further trade-offs such as between the number of cycles needed and the number of unit activations/deactivations: Using a free but "cold" functional unit

increases parallelism and hence may reduce execution time, but also increases power dissipation. Such effects make this issue more tricky than it appears at a first glance.

In order to take the right decisions at instruction selection, instruction scheduling (including resource allocation) and register allocation (including register assignment), the code generator needs a *power model* that provides quite detailed information about the power dissipation behavior of the architecture, which could be given as a function of the instructions, their encoding, their resource usage, their parameters, and (if statically available) their address and the data values accessed. On the other hand, such a power model should abstract from irrelevant details and thus allow for a fast calculation of the expected power dissipation for a given program trace. Ideally, a power model should be applicable to an entire class of architectures, to enhance retargetability of a power-aware code generator.

The information for a power model can be provided in two different ways: by simulation and by measurements. Simulation-based approaches take a detailed description of the hardware as input and simulate the architecture with a given program at the microarchitectural level, cycle by cycle to derive the energy consumption. Examples for such simulators are SimplePower [YVKI00] and Wattch [BTM00]. In contrast, measurement-based approaches assume a small set of factors that influence power dissipation, such as the width of opcodes, Hamming distances of subsequent instruction words, activations of functional units, *etc.*, which are weighted by originally unknown parameters and summed up to produce the energy prediction for a given program trace. In order to calibrate the model for a given hardware system, an ampèremeter is used to measure the current that is actually drawn for a given test sequence of instructions [TMW94]. The coefficients are determined by regression analysis that takes the measurements for different test sequences into account. Such a statistical model is acceptable if the predicted energy consumption for an arbitrary program differs from the actual consumption only by a few percent. Recently, two more detailed, measurement-based power models were independently developed by Lee *et al.* [LEM01] and by Steinke *et al.* [SKWM01], for the same processor, the ARM7TDMI; they report on energy predictions that are at most 2.5% and 1.7% off the actual energy consumption, respectively. We will use an adapted version of their power model as a basis for the energy optimizations described in this chapter.

Higher level compiler optimization techniques may address loop transformations, memory allocation, or data layout. For instance, the memory layout of arrays could be modified such that consecutive accesses traverse the element addresses in a gray code manner. If available, some frequently accessed variables could be stored in small on-chip memory areas [MSW01]. In this

work, we assume that such optimizations are already done and we focus on the final code generation step.

Not all factors in a power model are known at compile time. For instance, instruction addresses may change due to relocation of the code at link or load time, and the values residing in registers and memory locations are generally not statically known, even though static analysis could be applied to predict *e.g.* equality of values at least in some cases.

We present a method for energy-aware integrated local code generation that allows to explore trade-offs between energy consumption and execution time for a generic VLIW processor architecture. Our framework can be applied in two ways: It can determine power-optimal code (for a given power model) and it can optimize for execution time given a user-specified energy budget for (parts of) the program. An integrated approach to code generation is necessary because the subproblems of instruction selection, instruction scheduling, resource allocation and register allocation depend on each other and should be considered simultaneously. This combined optimization problem is a very hard one, even if only the basic block scope is considered (see Chapter 2). Fortunately, the application program is often fixed in embedded systems, and the final production run of the compiler can definitely afford large amounts of time and space for optimizations. Finally, an optimal solution is of significance for the design of energy-efficient processors because it allows to evaluate the full potential of an instruction set design. Furthermore knowing the optimal solution allows to evaluate the quality of fast heuristics, as we will show later.

5.2. Power Model

We adopt a simple power model [MSW01, LEM01] that largely follows the measurement-based power models described above, which we generalize in a straightforward way for VLIW architectures. Our model assumes that the contribution of every instruction y to the total energy consumption consists of the following components: (i) a base cost $bcost(y)$ that is independent of the context of the instruction, (ii) an overhead cost $ohcost(y, y')$ that accounts for inter-instruction effects with the instruction y' that precedes y in the same field of the instruction word, such as bit toggling in the opcode fields, and (iii) an activation/deactivation cost ac_i that is paid if functional unit U_i is activated or deactivated.

5.3. Energy-optimal Integrated Code Generation

As we mentioned in Chapter 3 we focus on code generation for basic blocks and extended basic blocks [Muc97] where the data dependences among the IR operations form a directed acyclic graph (DAG) $G = (V, E)$. n denotes the number of IR nodes in the DAG.

Some instructions issued at time $t \leqslant \rho(s)$ may not yet terminate at time $\rho(s)$, which means that the time slots $\rho(s)+1, ..., \tau(s)$ in s must be padded by NOPs to guarantee a correct program. In order to account for the energy contribution of these trailing NOP instructions we transform the base cost of all instructions y to cover successive NOPs: $basecost(y) = bcost(y) - bcost(\texttt{NOP})$. In particular, the transformed $basecost(\texttt{NOP})$ is 0. Thus the base cost for target schedule s is:

$$E_{bc}(s) = \tau(s) \cdot bcost(\texttt{NOP}) + \sum_{\sigma^s_{i,j} \neq \texttt{NOP}} basecost(\sigma^s_{i,j}).$$

The overhead cost for a target schedule s can be calculated as follows:

$$E_{oh}(s) = \sum_{i=1}^{\omega} \sum_{j=1}^{\tau(s)} ohcost(\sigma^s_{i,j}, \sigma^s_{i,j-1})$$

The activation/deactivation cost of s is:

$$E_{act}(s) = \sum_{i=1}^{f} \sum_{j=1}^{\tau(s)} ac_i \cdot \delta'(s_{i,j}, s_{i,j-1})$$

where

$$\delta'(a, b) = \begin{cases} 0 \text{ if} (a \neq \bot \wedge b \neq \bot) \vee (a = \bot \wedge b = \bot), \text{ and} \\ 1 \text{ otherwise.} \end{cases}$$

Then the total energy cost for s is $E(s) = E_{bc}(s) + E_{oh}(s) + E_{act}(s)$.

5.4. Power Profiles

A *power profile* $\Pi(s) = (s_{t-1,1}, ..., s_{t-1,\omega}, a_1, ..., a_f)$ for a target schedule s at reference time $t = \rho(s)$ contains the instructions $s_{t-1,k}$ issued in each slot k of the next-to-last instruction word in s at time $t-1$, and the *activity status* $a_i \in \{0, 1\}$ in the last filled slot of unit U_i in s, that is, at time t if some instruction or a definite NOP was already scheduled to unit U_i at time t, and $t-1$ otherwise.

The power profile thus stores all the information that may be necessary to determine the impact of scheduling steps at time t on power dissipation: the activity status of all functional units says whether a unit must be activated or deactivated, and the information about the preceding instructions allows to calculate inter-instruction power effects for instructions to be selected and scheduled at time t. Given the power profile and a new instruction y to be appended to the current target schedule s, the new power profile can be calculated incrementally.

We can thus associate with every target schedule s for G_z the accumulated energy $E_z(s)$ that was consumed by executing s from time 1 to $\tau(s)$ according to our power model. The goal is of course to find a target schedule s for the entire basic block that minimizes $E_\emptyset(s)$. If we optimize for energy only (and thus ignore the time requirements), two target schedules s_1 and s_2 for the same subDAG G_z are comparable with respect to their energy consumption if they have the same power profile, as they could be used interchangeably as a prefix in future scheduling decisions. Hence, the following compression theorem allows us to apply our dynamic programming framework to energy optimization:

Theorem 5.4.1 *For any two target schedules for the same subDAG* G_z, s_1 *with reference time* $t_1 = \rho(s_1)$ *and* s_2 *with reference time* $t_2 = \rho(s_2)$, *where* $\Pi(s_1) = \Pi(s_2)$, *the* s_i *with higher accumulated energy consumption* $E_z(s_i)$ *can be thrown away, that is, needs not be considered in further scheduling steps, without losing optimality. If* $E_z(s_1) = E_z(s_2)$, *either* s_1 *or* s_2 *can be thrown away without losing optimality.*

The theorem follows from the fact that all information that is not stored in the power profile, such as instructions issued at a time earlier than $t-1$ or units active at a time earlier than $t-1$, has no influence on the power dissipation in cycle t (reference time) according to our power model.

5.5. Construction of the Solution Space

For energy-only optimization, an *extended selection node* (ESnode for short) $\eta = (z, \Pi)$ is characterized by a zero-indegree set z and a power profile $\Pi = \Pi(s)$ for some target schedule s of G_z that is locally energy-optimal among all target schedules for G_z with that power profile. The ESnode stores s as an attribute.

Figure 5.1 represents the whole solution space for the matched DAG represented on the left side that computes `y=x*x`. The target architecture has two functional units U_1 and U_2 with latency $\ell = (2, 1)$ and activation cost

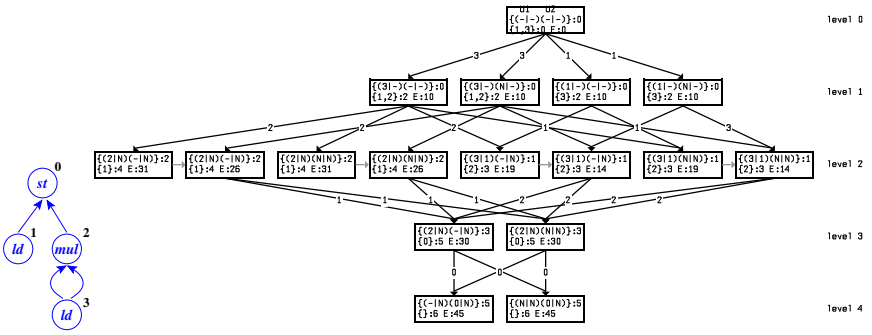

Figure 5.1.: Solution space for squaring on a 2-issue architecture with 2 functional units $\ell = (2,1)$, with base costs: 4 for st, 3 for others, 1 for NOP, and instruction overhead of 0.

$ac = (6,5)$. The base cost for node 0 is 4, 3 for others, and 1 for NOP. Node 0 is to be executed on unit U_2 and other nodes on either U_1 or U_2. ESnodes are grouped according to their level (length of the IR schedule). The first row of each ESnode represents the power profile and its reference time $\rho(s)$. For example, for the leftmost ESnode at level 1, $\{(3|-)(-|-)\}$ means that node 3 (ld) matches instruction 3 and is scheduled on unit U_1 at the current time slot. The status of unit U_2 is empty (denoted by –). In other ESnodes, N denotes an explicit NOP inserted by the dynamic programming algorithm. Remark that we show explicitly the occupation status of each functional unit at reference time $t = \rho(s)$ and $t-1$. Edges are annotated with nodes selected in the selection step of topological sorting (see Section 4.2.1). Edges that fall into a single ESnode indicate that the resulting partial solution nodes are equivalent. Unlabeled edges (horizontal) show an improvement of a partial solution, *i.e.* a partial ESnode is constructed that dissipates less energy than an equivalent ESnode that was already in the solution space; the resulting ESnode is the one with least energy dissipation. The second row in each ESnode represents the zero-indegree set (*e.g.*, 1,2: nodes 1 and 2), followed by the total execution time in terms of number of clock cycles (2 for the example node). Finally, we show the energy required for the partial schedule (10).

We structure the solution space as a two-dimensional grid L, spanned by an energy axis and a length axis. In order to obtain a discrete solution space we partition the energy axis into intervals $[k\cdot\Delta E, (k+1)\cdot\Delta E[$ of suitable size ΔE and normalize the lower bounds of the intervals to the integers $k = 0,1,2,\ldots$ Grid entry $L(l,E)$ stores a list of all ESnodes that represent IR schedules of length l and accumulated energy consumption E. This structure supports efficient

retrieval of all possible candidates for comparable partial solutions. When constructing the solution space, we proceed along the energy axis as driving axis, as energy is supposed to be the main optimization goal, while the length axis has secondary priority. The grid structure allows, by taking the precedence constraints for the construction of the partial solutions into account, to change the order of construction as far as possible such that the more promising solutions will be considered first while the less promising ones are set aside and reconsidered only if all the initially promising alternatives finally turn out to be suboptimal.

This is based on the property that the accumulated energy consumption never decreases if another instruction is added to an existing schedule. Most power models support this basic assumption; otherwise a transformation as in [KB02] can be applied to establish monotonicity for the algorithm.

5.6. Heuristics for Large Problem Instances

The structuring and traversal order of the solution space allows us to optimize the memory consumption of the optimization algorithm. Once all partial solutions in an ESnode (E, l) have been expanded it can be removed, as it will never be looked up again. Our algorithm for finding an energy-optimal schedule appears to be practical up to size 30, see Table 5.1.

Large DAGs require heuristic pruning of the solution space to cope with the combinatorial complexity. As a first attempt we control the number of variants generated from a scheduling situation, *i.e.* the number of ESnodes produced at each selection step. Instead of generating all possible selections we stop after N variants. Increasing the value of N results in better schedules with a slight computation time overhead. Using this heuristic significantly decreases computation times, that still present exponential behavior, and results in highly optimized code quality within 10% to optimal. We additionally implemented list scheduling (LS) and simulated annealing (SA) heuristics. For the results obtained with LS heuristic we observe an overhead of 173% on average and for SA 55%. This significant overhead for both heuristics is caused by that they do not consider using an already "warm" functional unit, nor the long delays for certain instructions resulting in switching on and off functional units often.

Table 5.1 shows the time requirements for finding an energy optimal schedule of our energy-only optimization algorithm on a collection of basic blocks taken from handwritten example programs and DSP benchmarks. Measurements have been performed on a Linux PC with 1.6GHz AMD processor (Athlon) and 1.5GB RAM. Column BB refers to a basic block among different

Table 5.1.: Influence of heuristics on computation time and code quality.

BB	OPT		H1(s)		H5(s)		H10(s)		H25(s)		LS		SA	
	t(s)	(eU)	t(s)	o(%)	t(s)	o(%)	t(s)	o(%)	t(s)	o(%)	(eU)	o(%)	(eU)	o(%)
bb1 (22)	221.7	94	1.2	5	5.2	0	5.6	0	11.1	0	237	152	111	18
bb2 (22)	42.0	89	1.7	38	5.1	13	9.4	11	19.1	0	237	166	145	63
bb3 (22)	59.7	86	1.2	9	4.3	0	5.8	0	15.4	0	243	183	136	58
bb4 (23)	18.0	83	3.1	20	3.4	14	6.4	12	9.8	0	248	199	134	61
bb5 (25)	17.2	102	1.9	17	5.0	0	7.8	0	14.2	0	274	169	153	50
bb6 (25)	113.0	94	1.9	18	8.8	0	12.7	0	36.7	0	259	176	152	62
bb7 (25)	16.6	102	1.8	17	5.2	0	7.8	0	14.0	0	274	169	158	55
bb8 (27)	560.0	101	2.7	34	12.2	10	23.3	0	78.9	0	277	174	152	50
bb9 (30)	112.4	112	2.6	20	12.8	9	22.3	0	53.8	0	304	171	180	60
bb10 (30)	8698.4	118	4.0	14	24.7	0	62.2	0	319.5	0	309	162	191	62
bb11 (32)	6031.5	113	5.0	30	27.1	11	73.2	9	336.0	9	311	175	173	53
bb12 (32)	21133.0	110	5.0	20	32.6	0	94.8	0	557.0	0	296	169	172	56
bb13 (33)	5054.0	125	5.0	18	37.3	0	75.6	0	350.8	0	349	179	198	58
bb14 (33)	4983.8	125	5.1	17	35.8	0	75.2	0	345.3	0	349	179	203	62
bb15 (40)	—	—	12.1	—	121.3	—	374.5	—	2353.0	—	398	—	270	—
bb16 (41)	—	—	13.2	—	161.2	—	511.2	—	3506.5	—	418	—	270	—
bb17 (44)	—	—	10.4	—	126.9	—	369.3	—	2240.5	—	365	—	263	—

benchmark programs, where the size is indicated in parenthesis. The second column reports the time in seconds for finding an energy-optimal schedule and its corresponding energy dissipation in energy unit (eU). If the algorithm run out of the time quantum (6 hours) it was interrupted, and indicated in the table by a dash. Columns 3-6 represent computation times for different values of N (1, 5, 10 and 25) and energy overheads compared to the optimal solution found in the optimal search. Finally, columns LS and SA indicate the energy dissipation and overhead obtained with a naive list scheduling (LS) and simulated annealing (SA) heuristics.

5.7. Possible Extensions

In principle, our framework can be extended to almost any power model, although this may affect the performance of our algorithm. For instance, we plan to study the effect of register assignment on energy consumption. In that case, we need to solve, for each partial solution, a register assignment problem for a partial interference graph and a partial register flow graph. Such algorithms exist in the literature [CP95, KSMS02] and could be adapted for our purposes.

5.8. Related Work

Lee, Lee *et al.* [LLHT00] focus on minimizing Hamming distances of subsequent instruction words in VLIW processors. They show that their formulation of power-optimal instruction scheduling for basic blocks is $\mathcal{NP}$-hard, and give a heuristic scheduling algorithm that is based on critical-path scheduling. They also show that for special multi-issue VLIW architectures with multiple slots of the same type, the problem of selecting the right slot *within* the same long instruction word can be expressed as a maximum-weight bipartite matching problem in a bipartite graph whose edges are weighted by negated Hamming distances between microinstructions of two subsequent long instruction words.

Lee, Tiwari *et al.* [LTMF95] exploit the fact that for a certain 2-issue Fujitsu DSP processor, a time-optimal target schedule is actually power-optimal as well, as there the unit activation/deactivation overhead is negligible compared to the base power dissipation per cycle. They propose a heuristic scheduling method that uses two separate phases, greedy compaction for time minimization followed by list scheduling to minimize inter-instruction power dissipation costs. They also exploit operand swapping for commutative operations

(multiplication).

Toburen *et al.* [TCR98] propose a list scheduling heuristic that could be used in instruction dispatchers for superscalar processors such as the DEC Alpha processors. The time behavior and power dissipation of each functional unit is looked up in an xADML-like description (see Chapter 8) of the processor. The list scheduler uses a dependence level criterion to optimize for execution time. Microinstructions are added to the current long instruction word unless a user-specified power threshold is exceeded. In that case, the algorithm proceeds to the next cycle with a fresh power budget.

Su *et al.* [STD94] focus on switching costs and propose a postpass scheduling framework that breaks up code generation into subsequent phases and mixes them with assembling. First, tentative code is generated with register allocation followed by pre-assembling. The resulting assembler code contains already information about jump targets, symbol table indices *etc.*, thus a major part of the bit pattern of the final instructions is known. This is used as input to a power-aware postpass scheduler, which is a modified list scheduling heuristic that greedily picks that instruction from the zero-indegree set that currently results in the least contribution to power dissipation. The reordered code is finally completed with a post-assembler.

Chapter 6.

Exploiting DAG Symmetries

Our dynamic programming algorithms presented in Chapter 4 and 5 require a large amount of time and memory. Measurements show that the time and space requirements of the algorithm increase exponentially with the size of the problem instance. This chapter presents an optimization technique that exploits during processing a specific property of DAGs, which we call *partial-symmetry*, to reduce time and space usage.

We observe that numerous DSP applications have DAGs that are partial-symmetric according to our definition. Exploiting the partial-symmetry property shows a significant reduction in time and memory usage for real world DSP and handwritten example programs.

6.1. Motivation

Our algorithm presented in Section 4.3.3 for irregular register sets is applicable to small, but still not trivial problem instances of size up to 20 instructions.

This chapter introduces a technique for pruning the solution space as early as possible, based on an idea presented by Chou *et al.* [CC95] for optimal instruction scheduling. Chou *et al.* establish and exploit equivalence relations among instructions of a basic block to cope with the combinatorial explosion of the solution space resulting from a branch-and-bound optimization approach. Similarly, we exploit partial-symmetry in our framework to reduce time and space usage. Informally, for time optimization two DAG nodes are *partial-symmetric* if interchanging them and their corresponding DAG successors in a given schedule does not make any difference for the final execution time.

To illustrate the partial-symmetry property let us consider the following C statement:

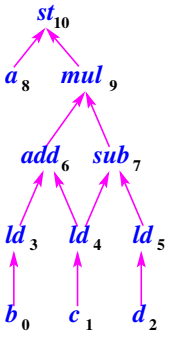

Figure 6.1.: IR-DAG for C statement `a=(b+c)*(c-d)`.

```
a=(b+c)*(c-d);
```

which corresponding DAG is shown in Figure 6.1. The DAG contains 11 nodes numbered from 0 to 10. There are 12 partial-symmetric pairs of nodes that occur during exploring all possibilities for scheduling the DAG on a simple load-store architecture with one arithmetic unit with unit latency and one multiplier with a two clock cycles latency. Let us assume that the arithmetic unit performs addition and subtraction, and the multiplier is used for the multiplication and LOAD/STORE operations. For instance, at a given scheduling situation where nodes 3 and 5 are possible candidates for being selected next, we observe that the execution time of the complete schedule remains the same indifferently if we select node 3 or 5 first.

We define an equivalence relation for operators that increases the number of possible partial-symmetries. Such an extension is target dependent and requires additional information. For instance, it is possible that pair (6, 7) may be considered as equivalent although the nodes represent different operators, *i.e.* addition (ADD) and subtraction (SUB).

We observed that several benchmarks for DSPs exhibit a high degree of partial-symmetry in terms of the defined equivalence relation. Table 6.1 reports the measurements of the number of symmetries detected in the Texas Instruments "comp_bench" benchmark set for the example architecture introduced above. We also added some examples of basic linear algebra computations. The first column gives the name of the benchmark file, the second refers to the basic block that is being optimized, and the third indicates the size of the DAG, *i.e.* the number of IR nodes. The number of partial-symmetries, shown in the last column, gives the number of pairs of nodes in the DAG that occur to be equivalent during scheduling.

Table 6.1.: Evaluation of the number of partial-symmetries in standard DSP benchmarks for an architecture with two functional units with latencies 1 and 2.

Name	BB	IR nodes	#symmetries
codebk_srch.c	28	23	116
codebk_srch.c	33	17	6
cplx.c	0	14	182
cplx.c	2	22	4469
dot.c	4	17	64
fir_vselp.c	7	25	139
fir_vselp.c	10	19	0
matrixcopy.c	4	18	0
scalarprod.c	3	17	64
summatrix.c	3	17	23
sumvector_un.c	2	17	8
vecsum_c.c	6	20	69

6.2. Solution Space Reduction

In this section we describe our optimization technique that exploits partial-symmetries in DAGs and improve our previous dynamic programming algorithm for determining a time-optimal schedule.

6.2.1. Exploiting the Partial-symmetry Property

To be able to handle larger problem instances we investigate a property that we call partial-symmetry as a base for early pruning of the solution space.

In [CC95] Chou *et al.* define an equivalence relation, based on symmetry properties of DAGs to prune the solution space of their branch-and-bound method. The main difference with our work is that Chou *et al.* do not merge (selection) nodes but perform enumeration on the selection tree. Additionally, they address the problem as a separate phase of code generation, *i.e.* instruction scheduling at the target level after instruction selection was done. In our framework, we address phases concurrently. Thus the selection of a target instruction for an IR node v is performed only when v is selectable, *i.e.* all predecessors of v have already been scheduled. In the problem definition given in this chapter, this does not make a difference since for simplicity we do not consider data locality and assume that all operands are available at the right

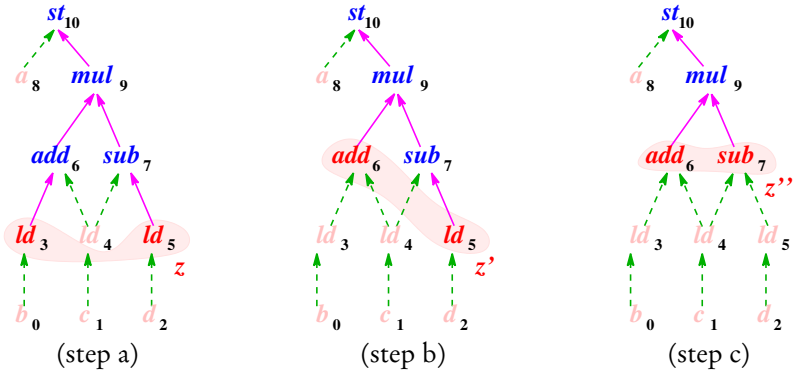

Figure 6.2.: Example DAG for C statement `a=(b+c)*(c-d);` under processing.

places. Generally, in our approach the problem occurs on a higher abstraction level, that is, on the IR level and not on target level as in Chou *et al.* [CC95].

We say that two IR nodes u and v that belong to a zero-indegree set z are equivalent if exchanging u and v and their respective successors in the final optimal schedule does not modify the execution time, the time profile at the exit of the basic block, nor the register need. That is, we could transform every schedule S_u of G_z starting with u into a schedule S_v starting with v by interchanging equivalent nodes with their "mirror" nodes, such that S_v has the same time and space behavior.

Figure 6.2 illustrates one possible scheduling process of the statement of Figure 6.1, that starts with the zero-indegree set containing nodes 3 and 5 (step a), marked by light gray area. Light gray nodes belong to *scheduled*(z), and black nodes are nodes above the zero-indegree set. The example starts with selecting node 3, which results in the new zero-indegree set consisting of nodes 6 and 5 (step b). We show another possible subsequent selection step that consists in choosing node 5, resulting in a new zero-indegree set consisting of nodes 6 and 7 (step c). Thus, we obtain a part of the schedule that contains the sequence of nodes 3 and 5. We would obtain the same result in terms of execution time and time profile shape if we started at (step a) by selecting first node 5 then 3. We remark in this example that the mirror node of node 3 is node 5.

6.2.2. Instruction Equivalence

An instruction y_i is *equivalent* to instruction y_j if both are executed on the same functional unit. We assume here that instructions that are executed on the same functional unit require the same number of clock cycles to be completed. Thus, appending instruction y_i to a schedule s with time profile P results in a schedule s' with time profile P'. We obtain an equivalent time profile P'', *i.e.* with the same shape and the same completion time as P', by appending y_j to s.

6.2.3. Operator Equivalence

The set of functional units on which an operation $op(v)$ of a DAG node v may be computed is denoted by $F(v) = \{U_i \mid \exists y \in \Psi(v), y \textit{ is being executed on } U_i\}$. We say that two opcodes $op(u)$ and $op(v)$ are *operator-equivalent*, denoted $u \equiv_{op} v$, if and only if $F(u) = F(v)$.

Example Let us illustrate operator-equivalence on the example of the C statement `a=(b+c)*(c-d);` whose DAG is shown in Figure 6.1. The scheduling is performed for a target processor with two functional units U_1, U_2 and a load/store unit. U_1 computes addition (`add`) and subtraction (`sub`). Multiplication (`mul`) and load (`ld`) are performed on unit U_2. Thus, operations (`sub`) and (`add`) (node 7 and 6) are equivalent, and we denote $node(7) \equiv_{op} node(6)$. ■

6.2.4. Node Equivalence

Let us consider a scheduling situation given by an ESnode $\eta = (z, P, t)$ with current zero-indegree set z, time profile P and reference point t of P in $\eta.schedule$. Let u and v be two nodes in z. Let $p_1, p_2, \ldots, p_k$ (resp. $q_1, q_2, \ldots, q_k$) denote the parents of u (resp. v). For simplicity we assume that inner DAG nodes are either binary or unary. An extension to higher node degrees is straightforward. We denote by *other_child*(p_i) the other child of p_i (that is not u). If the number of parents of u and v differs, u and v cannot be equivalent. Let $t(v)$ be the issue time of the instruction computing v. We denote by

$$\theta(u) = t + \max_{i \in \{1,2\}} \{0, t(c_i) + \ell_{\hat{k}(i)}\}$$

the *earliest schedule time* of u, where c_i is the i-th child of u still in the time profile P computed by functional unit $U_{\hat{k}(i)}$. $t(c_i)$ represents the issue time of node c_i.

We say u and v are *equivalent*, denoted $u \equiv_{sym} v$, if $\theta(u) = \theta(v)$ and $u \equiv_{upw} v$, where:

$u \equiv_{upw} v$ (upwards equivalence) if:

(1) u and v are identical, or

(2) u and v are roots of the DAG and the IR operations of u and v are operator-equivalent, denoted as $u \equiv_{op} v$, or

(3) there is a permutation $\tilde{q}$ of $q_1, q_2, \ldots, q_k$ such that $p_i \equiv_{upw} \tilde{q}_i$ for all $i \in 1, 2, \ldots, k$ and *other_child*$(p_i) \equiv_{dwn}$ *other_child*$(\tilde{q}_i)$ and $u \equiv_{op} v$.

$u \equiv_{dwn} v$ (downwards equivalence) if:

(1) u and v are in the zero-indegree set z, $u \equiv_{op} v$ and $\theta(u) = \theta(v)$, or

(2) u and v are in *alive*(z), for u and v the number of parents that are not in *scheduled*(z) are the same, and the predecessors of u and v are no longer in the time profile P, or

(3) u and v are unary, $u \equiv_{op} v$ and their children are downwards equivalent, or

(4) u and v are binary and $u \equiv_{op} v$ and:
either $u.lc \equiv_{dwn} v.lc$ and $u.rc \equiv_{dwn} v.rc$,
or $u.lc \equiv_{dwn} v.rc$ and $u.rc \equiv_{dwn} v.lc$,
where lc denotes the left and rc the right child.

The above relation $\equiv_{sym}$ is reflexive, symmetric, and transitive, thus it defines an equivalence relation.

We remark that for equivalent nodes u and v, u and v are mirror nodes, and the p_i and $\tilde{q}_i$ are mirrors of each other. If $u, v \in z$ are equivalent then any pattern (tree, forest, DAG) containing u (and maybe some of u's successors) has a mirror pattern containing v (and the mirrors of those successors of u).

Example We consider the DAG of Figure 6.1 at a given scheduling situation where $z = \{3, 5\}$, *scheduled*$(z) = \{0, 1, 2, 4, 8\}$, $6 \equiv_{op} 7$ and $3 \equiv_{op} 5$ represented in Figure 6.2 (step a). The following induction rules show that nodes 3 and 5 are equivalent where the numbers in parentheses indicate the matching rule of the equivalence relation.

$$\frac{\dfrac{\text{true}}{4 \in \textit{alive}(z)} \quad \dfrac{\text{true}}{0 = 0}}{4 \equiv_{dwn} 4} \; (2)$$

$$\dfrac{\dfrac{\text{true}}{6 \equiv_{\text{op}} 7} \wedge \dfrac{\dfrac{\text{true}}{3,5 \in z} \wedge \dfrac{\text{true}}{3 \equiv_{\text{op}} 5} \wedge \dfrac{\text{true}}{\theta(3) = \theta(5)}\ (1)}{3 \equiv_{\text{dwn}} 5} \dfrac{\text{true}}{\wedge 4 \equiv_{\text{dwn}} 4}\ (4)}{6 \equiv_{\text{dwn}} 7}$$

$$\dfrac{\dfrac{\dfrac{\text{true}}{9 \equiv_{\text{upw}} 9} \wedge \dfrac{\text{true}}{6 \equiv_{\text{dwn}} 7} \wedge \dfrac{\text{true}}{9 \equiv_{\text{op}} 9}}{6 \equiv_{\text{upw}} 7} \dfrac{\text{true}}{\wedge 4 \equiv_{\text{dwn}} 4} \wedge \dfrac{\text{true}}{6 \equiv_{\text{op}} 7}\ (3)}{3 \equiv_{\text{upw}} 4}$$

$$\dfrac{\dfrac{\text{true}}{3 \equiv_{\text{upw}} 4} \wedge \dfrac{\text{true}}{\theta(3) = \theta(5)}}{3 \equiv_{\text{sym}} 5}$$

■

The equivalence relation allows us to reduce (in the best case) the solution space by a factor up to two for each pair of equivalent nodes. That is, whenever two nodes u and v at a given scheduling step are simultaneously present in z and are equivalent, the algorithm needs consider only one sequence that starts either with u or v, but not both of them.

6.2.5. Improved Dynamic Programming Algorithm

In this chapter we consider regular architectures, and thus we first adapt the *clustered_timeopt* algorithm from Section 4.3.3 by removing the part of the algorithm that concern transfers between different residence classes (see Figure 4.5, lines 20–24) and call it *timeopt*. To perform symmetry evaluation, we modify the loop **for all** $v \in z$ (see Figure 4.5, line 13) such that a node $v \in z$ is skipped if it is equivalent to any other preceding node $v' \in z$. The modified part of the algorithm is given in Figure 6.3.

Note that node equivalence is not a static property of the DAG but (partially) depends on the other nodes in z and on the relevant history of scheduling decisions, *i.e.* the time profile. This means that an individual equivalence test must be run at each selection step, and therefore we must be careful that testing overhead does not outweigh the gains by compression due to found equivalences.

```
function timeopt ( DAG G with n nodes and set z_0 of leaves)
η_0 ←new ESnode(z_0, P_0);
η_0.s* ← empty schedule;
L_{0,0}.insert(η_0);
...
        for all ν_i ∈ z = {ν_1, ν_2, ..., ν_q} do
            if ∃j ∈ {1, 2, ..., i − 1} : θ(ν_j) = θ(ν_i) AND ν_j ≡_upw ν_i then
                skip ν_i
            else
                z' ← selection(z, χ);   // new zero-indegree set
                ...
            fi
        od
    ...
end function timeopt
```

Figure 6.3.: Modified algorithm that exploits the equivalence relation for determining a time-optimal schedule.

6.3. Implementation and Results

At the present time we implemented a simple version of the equivalence test algorithm of Section 6.2.4 which takes exponential time. In Chapter 10 we discuss possible alternatives as part of future work. We evaluated our method with two architectures:

(1) single-issue with four functional units with latencies 1, 1, 2 and 2.

(2) two-issue with three functional units with latencies 1, 2 and 2.

The benchmarks have been computed on a PC machine, with a 1.6GHz Athlon processor and 1.5GB RAM.

Table 6.2 shows the gain in terms of time by exploiting symmetries in different DSP programs. The first column gives the name of the benchmark, the second indicates the basic block that is being optimized, and the third gives the size of the basic block in term of number of DAG nodes. Column GT1 (resp. GT2) gives the gain in time for Architecture (1) (resp. Architecture (2)) by exploiting symmetries. Columns $\#sy_1$ and $\#sy_2$ indicate the number of symmetries that occur during computation for each architecture. We show in columns t_1 and t_2 computation times for the algorithm without exploiting symmetries.

		Architecture 1			Architecture 2		
	$\|V\|$	GT1[%]	$\#sy_1$	$t_1(s)$	GT2[%]	$\#sy_2$	$t_2(s)$
(a)	23	-13.89	220	0.72	-8.62	30	0.58
(b)	22	65.76	7042	43.66	52.21	5306	23.98
(c)	25	2.41	763	0.83	8.62	920	1.16
(d)	33	22.72	18761	190.44	29.38	52872	1599.80
(e)	22	36.48	11137	25.25	35.94	28796	168.02
(f)	25	-14.95	185	2.81	-6.95	136	1.87
(g)	44	24.32	60257	5748.33	19.96	97728	11106.23
(h)	30	-3.62	641	246.01	-0.48	2220	284.20
(i)	40	0.20	14888	6818.14	0.17	39248	13190.48
(j)	25	9.52	1195	0.84	23.33	1317	1.2
(k)	33	36.23	25111	187.61	39.22	66507	1557.99
(l)	22	-16.05	10	0.81	-10.91	29	0.55
(m)	30	-31.31	766	2.14	-23.32	716	1.93
(n)	27	-21.33	90	3.00	-19.31	139	2.02
(o)	32	-25.41	670	18.97	-20.94	1051	16.33
(p)	32	-2.64	4908	170.08	-2.66	4896	339.38

Table 6.2.: Time gain by exploiting symmetries in DSP benchmarks.

We observe that the gains are significant for larger problem instances that take long time to compute. The highest gain is 66% for a complex multiplication (benchmark (b), architecture (1)). In general loop unrolling increases significantly the amount of symmetric cases. However, for summatrix benchmarks (l)–(o) the gain is negative, *i.e.* in that situation we suffer from the overhead of computing node equivalence even if the unrolling factor for the main loop is 4. The same case occurs for benchmark (p).

Figure 6.4 and Figure 6.5 illustrate the relation between the amount of potential symmetries and the gain in time for architecture (1) and (2) respectively. The no-symm (resp. symm) bars indicate the computation time for algorithm without (resp. with) exploiting the symmetry property. For each benchmark we show the overhead in identifying symmetries (sym-id) and the amount of symmetries (#symm) reported by the framework that occurs during computation. The left y-axis indicates time in seconds, and the right y-axis the number of symmetries. The name of benchmarks correspond to the names of Table 6.2.

Generally, a high number of symmetries decreases the computation time[1]. For small problem instances the overhead for computing symmetries is higher than the gain. It is also unexpected for matrix multiplication unrolled once (benchmark (i)) that the computation time does not decrease significantly despite the large number of symmetric situation is high. In that case most of the symmetric situations occur at the end of computation when the solution space is large. Thus, the overhead for identifying symmetries and the gain are small.

[1] We observe that exploiting symmetries decreases or uses the same amount of memory, but never increases it.

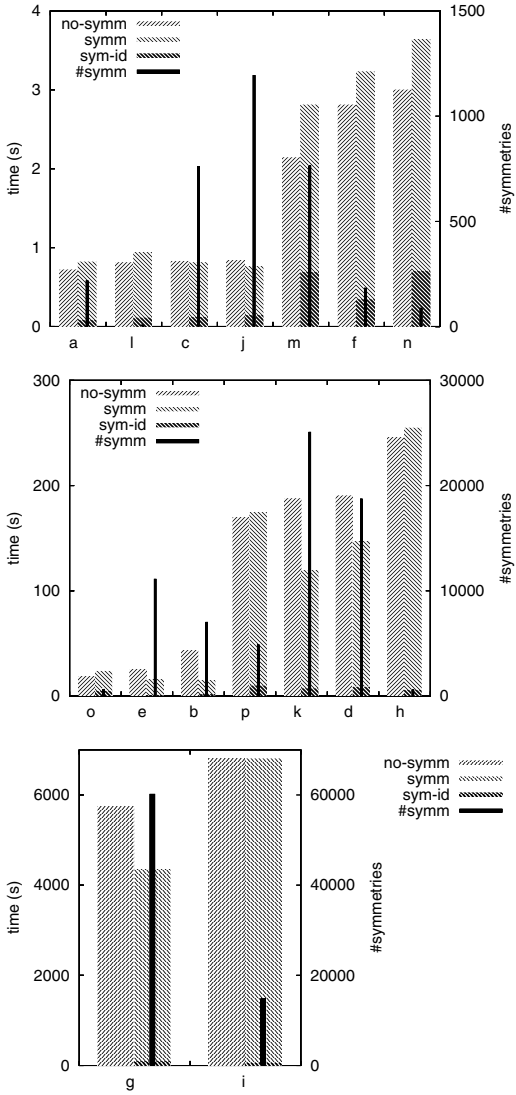

Figure 6.4.: Time results for Architecture (1) illustrating the computation time (left y-axis) without exploiting the symmetry property (no-symm), computation time with symmetry property (symm), symmetry overhead (sym-id) and the amount of symmetries (#symm) reported on the right y-axis.

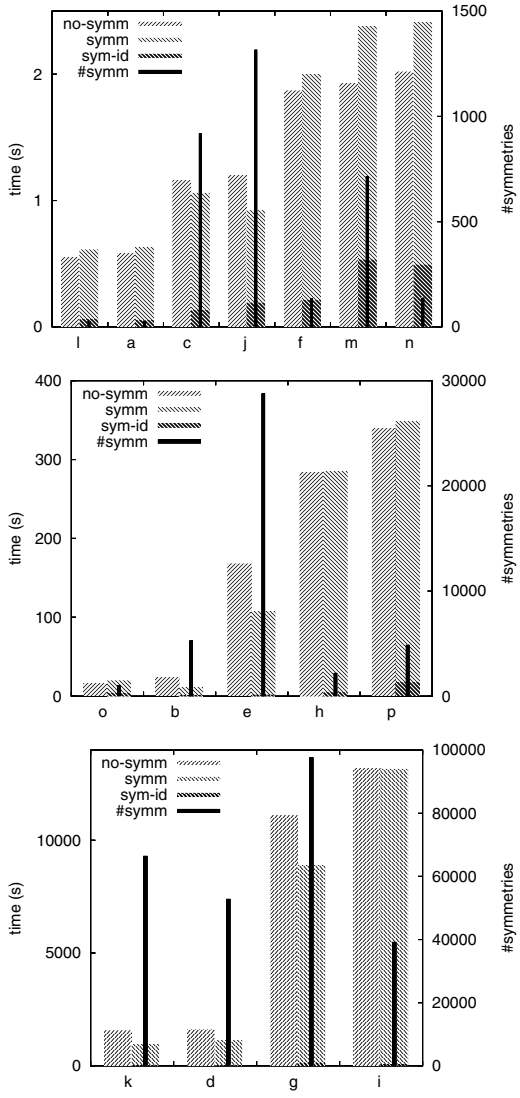

Figure 6.5.: Time results for Architecture (2) using the same notation as in Figure 6.4.

Chapter 7.
Integer Linear Programming Formulation

To our knowledge there is only one Integer Linear Programming (ILP) formulation in the literature, by Wilson [WGB94], that fully integrates all steps of code generation, *i.e.* instruction selection, register allocation and instruction scheduling, on the basic block level. We give in this chapter an improved version of this ILP formulation that also covers VLIW processors. Moreover, our ILP formulation does no longer require preprocessing the basic block's data flow graph to support instruction selection. We also evaluate and compare our ILP formulation with our DP method on a VLIW processor.

7.1. Introduction

We consider the problem of optimal integrated code generation for instruction-level parallel processor architectures such as VLIW processors. Integrated code generation solves simultaneously, in a single optimization pass, the tasks of instruction selection, instruction scheduling including resource allocation and code compaction, and register allocation.

In Chapter 4, we developed a dynamic programming approach and implemented it in our retargetable framework called OPTIMIST [KB05]. However, there may be further general problem solving strategies that could likewise be applied to the integrated code generation problem. In this chapter, we consider the most promising of these, *integer linear programming* (ILP).

Integer linear programming (ILP) is a general-purpose optimization method that gained much popularity in the past 15 years due to the arrival of efficient commercial solvers and effective modeling tools. In the domain of compiler back-ends, it has been used successfully for various tasks in code generation, most notably for instruction scheduling. Wilken *et al.* [WLH00] use ILP

for instruction scheduling of basic blocks which allows, after preprocessing the basic block's data flow graph, to derive optimal solutions for basic blocks with up to 1000 instructions within reasonable time.

The formulations that search for time-optimal schedules integrating instruction scheduling and resource allocation are either *time-based* or *order-based*. In time-based formulations the main decision variable indicates the time slot when an operation is to be started. In order-based formulations the decision variable represents the flow of the hardware resources among intermediate operations (resource flow).

Gebotys [GE91] formulates a time-based formulation that integrates instruction scheduling, resource allocation and computes time-optimal schedules. Leupers and Marwedel [LM97] provide a time-based ILP formulation for code compaction of a given instruction sequence with alternative instruction encodings. Zhang [Zha96], Chang *et al.* [CCK97] and Kästner [Käs00a] provide order-based and/or time-based ILP formulations for the combination of instruction scheduling with register allocation.

We know of only one ILP formulation in the literature that addressed all three tasks simultaneously, which was proposed by Wilson *et al.* [WMGB93, WGHB94]. However, their formulation is for single-issue architectures only. Furthermore, their proposed model assumes that the alternatives for pattern matching in instruction selection be exposed explicitly for each node and edge of the basic block's data flow graph (DFG), which would require a preprocessing of the DFG before the ILP problem instance can be generated.

In this chapter we provide an ILP formulation that fully integrates all three phases of code generation and extends the machine model used by Wilson *et al.* by including VLIW architectures. Moreover, our formulation does no longer need preprocessing of the DFG.

7.2. The ILP Formulation

In this section we first introduce various variables and parameters and then provide the ILP formulation for fully integrated code generation. The reader can find the complete ILP model in the Appendix B, specified in AMPL language[1]. However, first we introduce some notations that we use in the formulation.

[1] Mathematical programming language mainly used for solving operational optimization problems. `http://www.ampl.com`

7.2.1. Notations

We use uppercase letters to denote parameters and constants provided to the ILP formulation (model). Lowercase letters denote solution variables and indexes.

Indexes i and j denote nodes of the DFG. We reserve indexes k and l for instances of nodes composing a given pattern. t is used for time index. We use the common notation $|X|$ to denote the cardinality of a set (or pattern) X.

As usual, instruction selection is modeled as a general pattern matching problem, covering the DFG with instances of patterns that correspond to instructions of the target processor. The set of patterns B is subdivided into patterns that consist of a single node, called *singletons* (B''), and patterns consisting of more than one node, with or without edges (B'). That is, $B = B' \cup B''$ such that $\forall p \in B', |p| > 0$ and $\forall p \in B'', |p| = 1$.

In the ILP formulation that follows, we provide several instances of each non-singleton pattern. For example, if there are two locations in the DFG where a multiply and accumulate pattern (MAC) is matched, these will be associated with two different instances of the MAC pattern, one for each possible location. We require that each pattern instance be matched at most once in the final solution. As a consequence, the model requires to specify a sufficient number of pattern instances to cover the DFG G. For singleton patterns, we only need a single instance. This will become clearer once we have introduced the coverage equations where the edges of a pattern must correspond to some DFG edges.

7.2.2. Solution Variables

The ILP formulation uses the following solution variables:

- $c_{i,p,k,t}$ a binary variable that is equal to 1, if a DAG node i is covered by instance node k of pattern p at time t. Otherwise the variable is 0.
- $w_{i,j,p,k,l}$ a binary variable that is equal to 1 if DFG edge (i, j) is covered by a pattern edge (k, l) of pattern $p \in B'$.
- $s_{p,t}$ a binary variable that is set to 1 if a pattern $p \in B'$ is selected and the corresponding instruction issued at time t, and to 0 otherwise.
- $r_{i,t}$ a binary variable that is set to 1 if DFG node i must reside in some register at time t, and 0 otherwise.
- τ an integer variable that represents the execution time of the final schedule.

We search for a schedule that minimizes the total execution time of a basic block. That is, we minimize τ.

In the equations that follow, we use the abbreviation $c_{i,p,k}$ for the expression $\sum_{\forall t \in 0..T_{max}} c_{i,p,k,t}$, and s_p for $\sum_{\forall t \in 0..T_{max}} s_{p,t}$.

7.2.3. Parameters to the ILP Model

The model that we provide is sufficiently generic to be used for various instruction-level parallel processor architectures. At present, the ILP model requires the following parameters:

Data flow graph:

- G index set of DFG nodes
- E_G index set of DFG edges
- OP_i operation identifier of node i. Each DFG node is associated with an integer value that represents a given operation.
- OUT_i indicates the out-degree of DFG node i.

Patterns and instruction set:

- B' index set of instances of non-singleton patterns
- B'' index set of singletons (instances)
- E_p set of edges for pattern $p \in B'$
- $OP_{p,k}$ operator for an instance node k of pattern instance p. This relates to the operation identifier of the DFG nodes.
- $OUT_{p,k}$ is the out-degree of a node k of pattern instance p.
- L_p is an integer value representing the latency for a given pattern p. In our notation, each pattern is mapped to a unique target instruction, resulting in unique latency value for that pattern.

Resources:

- F is an index set of functional unit types.
- M_f represents the amount of functional units of type f, where $f \in F$.

- $U_{p,f}$ is a binary value representing the connection between the target instruction corresponding to a pattern (instance) p and a functional unit f that this instruction uses. It is 1 if p requires f, otherwise 0.

- W is a positive integer value representing the issue width of the target processor, *i.e.*, the maximum number of instructions that can be issued per clock cycle.

- R denotes the number of available registers.

- T_{max} is a parameter that represents the maximum execution time budget for a basic block. The value of T_{max} is only required for limiting the search space, and has no impact on the final result. Observe that T_{max} must be greater than (or equal to) the time required for an optimal solution, otherwise the ILP problem instance has no solution.

The rest of the section provides the ILP model for fully integrated code generation for VLIW architectures. First, we give equations for covering the DFG G with a set of patterns, *i.e.* the instruction selection. Secondly, we specify the set of equations for register allocation. Currently, we address regular architectures with general purpose registers, and thus only check that the register need does not exceed the amount of physical registers at any time. Next, we address scheduling issues. Since we are working on the basic block level, only flow (true) data dependences are considered. Finally, we assure that, at any time, the schedule does not exceed available resources, and that the instructions issued simultaneously fit into a long instruction word, *i.e.*, do not exceed the issue width.

7.2.4. Instruction Selection

Our instruction selection model is suitable for tree-based and directed acyclic graph (DAG) data flow graphs. Also, it handles patterns in the form of tree, forest, and DAG patterns.

The goal of instruction selection is to cover all nodes of DFG G with a set of patterns. For each DFG node i there must be exactly one matching node k in a pattern instance p. Equation (7.1) forces this full coverage property. Solution variable $c_{i,p,k,t}$ records for each node i which pattern instance node covers it, and at what time. Beside full coverage, Equation (7.1) also assures a requirement for scheduling, namely that for eace DFG node i, the instruction corresponding to the pattern instance p covering it is scheduled (issued) at

some time slot t.

$$\forall i \in G, \sum_{p \in B} \sum_{k \in p} c_{i,p,k} = 1 \tag{7.1}$$

Equation (7.2) records the set of pattern instances being selected for DFG coverage. If a pattern instance p is selected, all its nodes should be mapped to distinct nodes of G. Additionally, the solution variable $s_{p,t}$ carries the information at what time t a selected pattern instance p is issued.

$$\forall p \in B', \forall t \in 0..T_{max}, \sum_{i \in G} \sum_{k \in p} c_{i,p,k,t} = |p| s_{p,t} \tag{7.2}$$

If a pattern instance p is selected, each pattern instance node k maps to exactly one DFG node i. Equation (7.3) considers this unique mapping only for selected patterns, as recorded by the solution variables s.

$$\forall p \in B', \forall k \in p, \sum_{i \in G} c_{i,p,k} = s_p \tag{7.3}$$

Equation (7.4) implies that all edges composing a pattern must coincide with exactly the same amount of edges in G. Thus, if a pattern instance p is selected, it should cover exactly $|E_p|$ edges of G. Unselected pattern instances do not cover any edge of G. Remark that in our model each pattern instance is distinct, and that we further assume that there are enough pattern instances available to fully cover a particular DFG.

$$\forall p \in B', \sum_{(i,j) \in E_G} \sum_{(k,l) \in E_p} w_{i,j,p,k,l} = |E_p| s_p \tag{7.4}$$

Equation (7.5) assures that a pair of nodes constituting a DFG edge covered by a pattern instance p corresponds to a pair of pattern instance nodes. If we have a match ($w_{i,j,p,k,l} = 1$) then we must map DFG node i to pattern instance node k and node j to pattern instance node l of pattern instance p.

$$\forall (i,j) \in E_G, \forall p \in B', \forall (k,l) \in E_p, \; 2w_{i,j,p,k,l} \leqslant c_{i,p,k} + c_{j,p,l} \tag{7.5}$$

Equation (7.6) imposes that instructions corresponding to a non-singleton pattern (instance) p are issued at most once at some time t (namely, if p was selected), or not at all (if p was not selected).

$$\forall p \in B', \; s_p \leqslant 1 \tag{7.6}$$

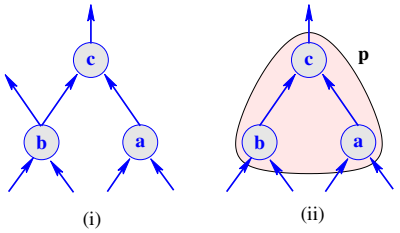

Figure 7.1.: Pattern coverage restrictions: (i) Pattern p cannot cover the set of nodes since there is an outgoing edge from b, (ii) pattern p covers the set of nodes $\{a, b, c\}$.

Equation (7.7) checks that the IR operators of DFG (OP_i) corresponds to the operator $OP_{p,k}$ of node k in the matched pattern instance p.

$$\forall i \in G, \forall p \in B, \forall k \in p, \forall t \in 0..T_{max}, c_{i,p,k,t}(OP_i - OP_{p,k}) = 0 \qquad (7.7)$$

Our model forbids, as is the case for almost all architectures, to access a partial result that flows along a covered edge and thus appears inside a pattern. Only a value flowing out of a node matched by a root node of the matching pattern is accessible (and will be allocated some register). This situation is illustrated in Figure 7.1. A possible candidate pattern p that covers nodes a, b, and c cannot be selected in case (i) because the value of b is used by another node (outgoing edge from b). On the other hand, the pattern might be selected in case (ii) since the value represented by b is only internal to pattern p.

For that, Equation (7.8) simply checks if the out-degree $OUT_{p,k}$ of node k of a pattern instance p equals the out-degree OUT_i of the covered DFG node i. As nodes in singleton patterns are always pattern root nodes, we only need to consider non-singleton patterns, *i.e.* the set B'.

$$\forall p \in B', \forall (i,j) \in E_G, \forall (k,l) \in p,\ w_{i,j,p,k,l}(OUT_i - OUT_{p,k}) = 0 \qquad (7.8)$$

7.2.5. Register Allocation

Currently we address (regular) architectures with general-purpose register set. We leave modeling of clustered architectures for future work. Thus, a value carried by an edge not entirely covered by a pattern (active edge), requires a

register to store that value. Equation (7.9) forces a node i to be in a register if at least one of its outgoing edge(s) is active.

$$\forall t \in 0..T_{max}, \forall i \in G, \quad \sum_{t_t=0}^{t} \sum_{(i,j)\in E_G} \sum_{p\in B} \left(\sum_{k\in p} c_{i,p,k,t_t} - \sum_{l\in p} c_{j,p,l,t_t} \right) \leqslant N r_{i,t} \quad (7.9)$$

If all outgoing edges from a node i are covered by a pattern instance p, there is no need to store the value represented by i in a register. Equation (7.10) requires solution variable $r_{i,t}$ to be set to 0 if all outgoing edges from i are inactive at time t.

$$\forall t \in 0..T_{max}, \forall i \in G, \quad \sum_{t_t=0}^{t} \sum_{(i,j)\in E_G} \sum_{p\in B} \left(\sum_{k\in p} c_{i,p,k,t_t} - \sum_{l\in p} c_{j,p,l,t_t} \right) \geqslant r_{i,t} \quad (7.10)$$

Finally, Equation (7.11) checks that register pressure does not exceed the number R of available registers at any time.

$$\forall t \in 0..T_{max}, \sum_{i\in G} r_{i,t} \leqslant R \quad (7.11)$$

7.2.6. Instruction Scheduling

The scheduling is complete when each node has been allocated to a time slot in the schedule such that there is no violation of precedence constraints and resources are not oversubscribed. Since we are working on the basic block level, we only need to model the true data dependences, represented by DFG edges. Data dependences can only be verified once pattern instances have been selected, covering the whole DFG. The knowledge of the covered nodes with their respective covering pattern (*i.e.*, the corresponding target instruction) provides the necessary latency information for scheduling.

Besides assuring full coverage, Equation (7.1) constraints each node to be scheduled at some time t in the final solution. We need additionally to check that all precedence constraints (data flow dependences) are satisfied. There are two cases: first, if an edge is entirely covered by a pattern p (inactive edge), the

latency of that edge must be 0, which means that for all inactive edges (i,j), DFG nodes i and j are "issued" at the same time. Secondly, edges (i,j) between DFG nodes matched by different pattern instances (active edges) should carry the latency L_p of the instruction whose pattern instance p covers i. Equations (7.12) and (7.13) guarantee the flow data dependences of the final schedule. We distinguish between edges leaving nodes matched by a multi-node pattern, see Equation (7.12), and the case of edges outgoing from singletons, see Equation (7.13).

$$\forall p \in B',\ \forall (i,j) \in E_G,\ \forall t \in 0..T_{max} - L_p + 1,$$
$$\sum_{k \in p} c_{i,p,k,t} + \sum_{\substack{q \in P \\ q \neq p}} \sum_{t_t=0}^{t+L_p-1} \sum_{k \in q} c_{j,q,k,t_t} \leqslant 1 \quad (7.12)$$

Active edges leaving a node covered by a singleton pattern p carry always the latency L_p of p. Equation (7.13) assures that the schedule meets the latency constraint also for these cases.

$$\forall p \in B'',\ \forall (i,j) \in E_G,\ \forall t \in 0..T_{max} - L_p + 1,$$
$$\sum_{k \in p} c_{i,p,k,t} + \sum_{q \in B} \sum_{t_t=0}^{t+L_p-1} \sum_{k \in q} c_{j,q,k,t_t} \leqslant 1 \quad (7.13)$$

7.2.7. Resource Allocation

A schedule is valid if it respects data dependences and its resource usage does not exceed the available resources (functional units, registers) at any time. Equation (7.14) verifies that there are no more resources required by the final solution than available on the target architecture. Currently in the ILP model we assume fully pipelined functional units with an occupation time of one for each unit, *i.e.* a new instruction can be issued to a unit every new clock cycle. We leave the modeling of intricate pipelines, as part of future work. The first summation counts the number of resources of type f required by instructions corresponding to selected multi-node pattern instances p at time t. The second part records resource instances of type f required for singletons

(scheduled at time t).

$$\forall t \in 0..T_{max},\ \forall f \in F,\ \sum_{\substack{p \in B' \\ u_{p,f}=1}} s_{p,t} + \sum_{\substack{p \in B'' \\ u_{p,f}=1}} \sum_{i \in G} \sum_{k \in p} c_{i,p,k,t} \leqslant M_f \tag{7.14}$$

Finally Equation (7.15) assures that the issue width W is not exceeded. For each issue time slot t, the first summation of the equation counts for multi-node pattern instances the number of instructions composing the long instruction word issued at t, and the second summation for the singletons. The total amount of instructions should not exceed the issue width W, *i.e.*, the number of available slots in a VLIW instruction word.

$$\forall t \in 0..T_{max},\ \sum_{p \in B'} s_{p,t} + \sum_{p \in B''} \sum_{i \in G} \sum_{k \in p} c_{i,p,k,t} \leqslant W \tag{7.15}$$

7.2.8. Optimization Goal

In this chapter we are looking for a time-optimal schedule for a given basic block. The formulation however allows us not only to optimize for time but can be easily adapted for other objective functions. For instance, we might look for the minimum register usage or code length.

In the case of the time optimization goal, the total execution time of a valid schedule can be derived from the solution variables c as illustrated in Equation (7.16).

$$\forall i \in G,\ \forall p \in P,\ \forall k \in p,\ \forall t \in 0..T_{max},\ c_{i,p,k,t} * (t + L_p) \leqslant \tau \tag{7.16}$$

The total execution time is less or equal to the solution variable τ. Looking for a time-optimal schedule, our objective function is

$$\min \tau \tag{7.17}$$

7.3. Experimental Results

First, we provide two theoretical VLIW architectures for which we generate target code. Secondly we describe the experimental setup that we used to evaluate our ILP formulation against our previous DP approach and summarize the results.

7.3.1. Target Architectures

In order to compare OPTIMIST's DP technique to the ILP formulation of Section 7.2, we use two theoretical VLIW target platforms (Case I and Case II) with the following characteristics.
Case I: The issue width is a maximum of two instructions per clock cycle. The architecture has an arithmetic and logical unit (`ALU`). Most ALU operations require a single clock cycle to compute (occupation time and latency are one). Multiplication and division operations have a latency of two clock cycles. Besides the ALU, the architecture has a multiply-and-accumulate unit (`MAC`) that takes two clock cycles to perform a multiply-and-accumulate operation. There are eight general purpose registers accessible from any unit. We assume a single memory bank with unlimited size. A load/store unit (`LS`) stores and loads data in four clock cycles.
Case II: The issue width is of maximum four instructions per clock cycle. The architecture has twice as many resources as in Case I, *i.e.* two arithmetic and logical units, two multiply-and-accumulate units, and two load/store units with the same characteristics.

7.3.2. Experimental Setup

We implemented the ILP data generation module within the OPTIMIST framework. Currently our ILP model addresses VLIW architectures with regular pipeline, *i.e.* functional units are pipelined, but no pipeline stall occurs. We adapted hardware specifications in xADML (see Chapter 8) such that they fit current limitations of the ILP model. In fact, the OPTIMIST framework accepts more complex resource usage patterns and pipeline descriptions expressible in xADML, which uses the general mechanism of reservation tables [DSTP75]. As assumed in Section 7.2, we use for the ILP formulation the simpler model with unit occupation time and latency for each instruction. An extension of the ILP formulation to use general reservation tables is left to future work.

Figure 7.2 shows our experimental platform. We provide a plain C code sequence as input to OPTIMIST. We use LCC [FH95] (within OPTIMIST) as C front-end. Besides the source code we provide the description of the target architecture in xADML language (see Chapter 8). For each basic block, OPTIMIST outputs the assembly code as result. If specified, the framework also outputs the data file for the ILP model of Section 7.2. The data file contains architecture specifications, such as the issue width of the processor, the set of functional units, patterns, *etc.* that are extracted from the architecture description document. It generates all parameters introduced in Section 7.2.3. Finally

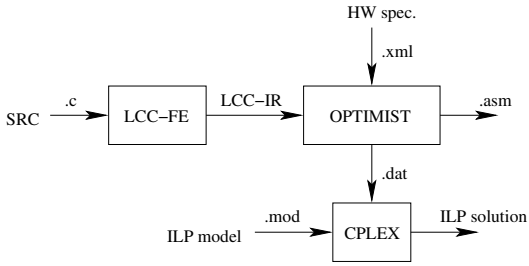

Figure 7.2.: Experimental setup.

we use the CPLEX solver [Inc06] to solve the set of equations.

Observe that for the ILP data we need to provide the upper bound for the maximum execution time in the ILP formulation (T_{max}). For that, we first run a heuristic variant of OPTIMIST that still considers full integration of code generation phases, and provide its execution time (computed in a fraction of a second) as the T_{max} parameter to the ILP data.

7.3.3. Results

We generated code for basic blocks taken from various digital signal processing benchmark programs. We run the evaluation of the DP approach on a Linux (kernel 2.6.13) PC with Athlon 1.6GHz CPU and 1.5GB RAM. The ILP solver runs on a Linux (kernel 2.6.12) PC with Athlon 2.4GHz CPU, 512MB RAM using CPLEX 9.

We should mention a factor that contributes in favor of the ILP formulation. In the OPTIMIST framework we use LCC [FH95] as C front-end. Within our framework we enhanced the intermediate representation with extended basic blocks (which is not standard in LCC, see Section 4.3.6). As consequence, we introduced data dependence edges for resolving memory write/read precedence constraints. In the current ILP formulation we consider only data flow dependences edges. Thus, we instrumented OPTIMIST to remove edges introduced by building extended basic blocks. Removing dependence edges results in DAGs with larger base, *i.e.* with larger number of leaves, and in general a lower height. We are aware that the DP approach suffers from DAGs with a large number of leaves, as OPTIMIST early generates a large number of partial solutions. Further, removing those edges build DFGs that may no longer be equivalent to the original C source code. However, it is still valid to compare the ILP and DP techniques, since both formulations operate on the same intermediate representation.

Table 7.1.: Evaluation of ILP and DP fully integrated code generation approaches for the Case I architecture.

Basic block		|G|	Height	$\|E_G\|$	DP		ILP	
					τ (cc)	t (sec)	τ (cc)	t (sec)
1)	iir filter bb9	10	4	10	10	0.3	10	0.9
2)	vec_max bb8	12	4	12	11	0.6	11	1.3
3)	dijkstra bb19	16	7	15	14	6.6	14	5.6
4)	fir filter bb9	16	3	14	15	61.3	15	7.8
5)	cubic bb16	17	6	16	14	15.0	14	5.7
6)	fir_vselp bb10	17	9	17	16	3.4	16	8.2
7)	matrix_sum loop bb4	17	8	17	16	4.0	16	8.8
8)	scalarprod bb2	17	8	18	17	1.2	17	15.8
9)	vec_sum bb3	17	8	18	16	1.4	16	11.8
10)	matrix_copy bb4	18	7	19	16	4.3	16	12.5
11)	cubic bb4	21	8	23	17	69.8	17	277.7
12)	iir filter bb4	21	6	17	20	3696.4	20	46.5
13)	fir filter bb11	22	6	27	19	89.7	CPLEX	
14)	codebk_srch bb20	23	7	22	17	548.8	17	63.1
15)	fir_vselp bb6	23	9	25	19	40.6	CPLEX	
16)	summatrix_un1 bb4	24	10	28	20	25.4	CPLEX	
17)	scalarprod_un1 bb2	25	10	30	19	14.9	CPLEX	
18)	matrixmult bb6	30	9	35	23	2037.7	AMPL	
19)	vec_sum unrolled bb2	32	10	40	24	810.9	AMPL	
20)	scalarprod_un2 bb2	33	12	42	23	703.1	AMPL	

Table 7.1 reports our results for the Case I architecture. The first column indicates the name of the basic block. The second column reports the number of nodes in the DAG for that basic block. The third and fourth columns give the height of the DAG and the number of edges, respectively. Observe that the height corresponds to the longest path of the DAG in terms of number of DAG nodes, and not to its critical path length, whose calculation is unfeasible since the instruction selection is not yet known. The fifth column reports the amount of clock cycles required for the basic block, and in the sixth column we display the computation time (in seconds) for finding a DP solution. Columns seven and eight report the results for ILP. The computation time for the ILP formulation does not include the time for CPLEX-presolve that optimizes the equations.

In the table we use three additional notations: CPLEX indicates that the ILP solver ran out of memory and did not compute a result. AMPL means that CPLEX-presolve failed to generate an equation system, because it ran out of memory. Where the DP ran out of memory we indicate the entry as MEM.

For all cases that we could check both techniques report the same execution time (τ). It was unexpected to see that the ILP formulation performs quite well and in several cases with an order of magnitude faster than DP. For cases 4), 12) and 14) the DP takes almost eight times, eighty times and nine times respectively longer than the ILP solver to compute an optimal solution. Since we removed the memory data dependence edges (as mentioned earlier) the resulting test cases present two, four and two unrelated DAGs for case 4), 12) and 14) respectively. We know that DP suffers from DAGs with a large number of leaves because a large number of selection nodes is generated already at the first step. For the rest of the test cases, DP outperforms the ILP formulation or has similar computation times. Observe that we reported for cases 3) and 5) that ILP takes shorter time to compute an optimal solution. But if we include the time of CPLEX-presolve, which runs for 7.1s in case 3) and 8.3s in case 5), the ILP times are worse or equivalent. For problems larger than 22 nodes, the ILP formulation fails to compute a solution. For problem instances over 30 nodes, the CPLEX-presolve does not generate equations because it runs out of memory.

Table 7.2 shows the results for the Case II architecture. The notations are the same as for Case I. We added an additional column in the ILP part, denoted t', that reports the ILP computation time when the upper bound T_{max} is derived from a run of a heuristically pruned DP algorithm $H^{N=2}$ described in Section 4.3.5 (this decreases the number of generated equations by providing a value of T_{max} closer to an optimal solution). The time for DP algorithm $H^{N=2}$ run is not included in t'.

Table 7.2.: Evaluation of ILP and DP fully integrated code generation approaches for Case II architecture.

Basic block	|G|	Height	$\|E_G\|$	DP		ILP		
				τ (cc)	t (sec)	τ (cc)	t (sec)	t′ (sec)
1) iir filter bb9	10	4	10	9	0.6	9	1.5	0.4
2) vec_max bb8	12	4	12	10	2.4	10	1.6	0.7
3) dijkstra bb19	16	7	15	14	73.5	14	10.7	4.2
4) fir filter bb9	16	3	14	9	2738.9	9	9.1	2.5
5) cubic bb16	17	6	16	12	1143.3	12	CPLEX	3.8
6) fir_vselp bb10	17	9	17	14	62.1	14	CPLEX	4.9
7) matrix_sum loop bb4	17	8	17	15	90.2	15	CPLEX	10.2
8) scalarprod bb2	17	8	18	15	10.0	—	CPLEX	CPLEX
9) vec_sum bb3	17	8	18	13	11.4	13	CPLEX	4.6
10) matrix_copy bb4	18	7	19	14	89.4	14	AMPL	4.1
11) cubic bb4	21	8	23	16	8568.7	—	AMPL	CPLEX
12) iir filter bb4	21	6	17	—	MEM	12	AMPL	7.4
13) fir filter bb11	22	6	27	—	MEM	—	AMPL	CPLEX
14) codebk_srch bb20	23	7	22	—	MEM	—	AMPL	CPLEX
15) fir_vselp bb6	23	9	25	16	7193.9	—	AMPL	AMPL

Also for the second case, ILP and DP yield the same execution time τ for solutions obtained with both approaches, as expected. For the cases 4) and 12) DP performs worse than ILP. For the case 12) DP runs out of memory, whereas the ILP could compute a solution within 7.4s if T_{max} is close enough to the optimum. Case II results show that it is beneficial to spend time to minimize T_{max}. We could gain about four additional nodes for the ILP problem size. For Case II, if the ILP compute a solution it outperforms the DP.

Chapter 8.

xADML: An Architecture Specification Language

We provide a retargetable code generation framework for various hardware architectures. Therefore, we have developed a structured hardware description language called *Extended Architecture Description Mark-up Language* (xADML), based on Extensible Mark-up Language (XML). The structure of the processor, as far as relevant for the generation of optimal code, is specified in xADML.

In this chapter we provide xADML specifications for parameterizing the OPTIMIST framework and retarget it to a specific target architecture.

8.1. Motivation

Providing a compiler for newly developed processor architectures is a time and money consuming task in industry. New compilers may reuse existing parts of existing frameworks, such as front-ends, high level optimizations modules, *etc.* But the back-end usually needs to be rewritten to a large extent. Writing a specific back-end for each new hardware is not an option where the time-to-market is short. Instead, a specification of the target processor in a high level description language is provided to either automatically generate a back-end or parameterize a generic back-end of a compiler framework (see Section 2.8).

In this chapter we present a hardware description language called xADML, *eXtensible Architecture Description Mark-up Language*, that we use as parametrization language for our retargetable code generation framework OPTIMIST. xADML allows the OPTIMIST framework to be retargeted efficiently to different architectures. We used xADML in the OPTIMIST framework for

very different processor types: a multi-issue clustered VLIW architecture TI-C62x, a multi-issue Motorola MC56K DSP processor [Yon06], and a single issue processors, ARM9E [Lan05].

Additionally, we build variants of these processors to evaluate the impact of the architecture complexity on the dynamic programming algorithms.

The motivation for building a "home" hardware language, rather than using an existing one, was to be able to start with a simple and low complexity language and progressively extend it during the work.

8.2. Notations

In the rest of this chapter we use following notations for the specification language: ellipses (`...`) are only used for brevity and are not part of the hardware specification. Further, we denote by `[string]` a string of alphanumeric characters. The string starts with an alphabetic or numeric character and may contain spaces. `[integer]` represents a positive or zero integer value and `[integer*]` a strictly positive integer.

8.3. xADML: Language Specifications

An xADML document is divided into five parts, or sections, that can be specified in any order. The five parts of the documents are:

- Declaration of hardware resources, such as registers, memory modules and resources that are part of reservation tables of target instructions.
- Definition of generic patterns that are mapped to given target instructions in the instruction set definition part.
- Instruction set definition, which provides the set of target instructions for the processor.
- For irregular architectures, copy instructions are declared in the transfer section.
- Finally, global formatting of the output can be parameterized in the formating part.

An xADML specification contains thus both a structural and behavioral description of a target processor. In the following subsections we describe in detail the different parts of xADML depicted in Figure 8.1.

```
<?xml version="1.0" encoding="iso-8859-1"?>

<architecture name="[string]" version="[string]">
  <!-- Hardware resources -->
  ...
  <!-- Patterns -->
  ...
  <!-- Instruction set -->
  ...
  <!-- Transfer instructions -->
  ...
  <!-- Formating facilities -->
  ...
</architecture>
```

Figure 8.1.: Structure of xADML specification document.

Our architecture description language is based on XML. As xADML is a standard XML document, it begins with the version and encoding information definition. This is necessary for XML parsers and tools. In our framework implementation we use Xerces-C[1], an XML parser. As any XML document, xADML can be viewed as a tree structure. The specification tree is rooted at node `architecture` (see Figure 8.1).

Both attributes of the `architecture` node are optional.

- `name` records the name of the architecture specified in the xADML document.
- `version` represents the version of the specification file. This allows the user to record different variants of the specification, if for instance the hardware is changing during the design phase.

8.4. Hardware Resources

The resource section of an xADML document enumerates hardware resources available for the computations. xADML allows to specify registers, residence classes and any kind of resource that is part of the reservation table.

[1] Xerces-C++ parser is a project hosted at the Apache XML homepage, `http://xml.apache.org/xerces-c/`.

```
...
<!-- Hardware resources -->
<issue_width>[integer*]</issue_width>
<registers> ... </registers>
<residences> ... </residences>
<resources> ... </resources>
...
```

Figure 8.2.: Resources part of the xADML specification document.

```
...
<registers>
  <reg id="[string]" size="[integer*]"/>
  ...
</registers>
...
```

Figure 8.3.: Declaration of data registers. Each register has a name and an optional size.

8.4.1. Issue Width

We specify the issue width (ω, see Section 3.2), within the `issue_width` node, as depicted in Figure 8.2. The issue width is a strictly positive integer, representing the number of instructions that can be issued in the same clock cycle.

8.4.2. Registers

The `registers` node defines all registers of the architecture available for computation. xADML does not include set of control registers, such as the program counter register (PC) or status registers. Each register entry `reg` (Figure 8.3) has two attributes:

- `id` represents the name of the register. Those names are used in the format part when emitting assembler code. Each register requires an `id` attribute.
- `size` an optional attribute that specifies the size, in terms of bits.

8.4.3. Residences

In the `residences` node we find the set of all residence classes (see Section 4.2.5) derived from the instruction set of the target architecture. For the residence classes formed by the set of registers, xADML declares all registers

```
...
<residences>
  <residence id="[string]">
    <reg id="[string]"/>
    ...
  </residence>
  ...
  <residence id="[string]"/>
  ...
</residences>
...
```

Figure 8.4.: Definition of residence classes. A residence class may be composed of registers, or be of unlimited size.

```
...
<resources>
  <resource id="[string]"/>
  ...
</resources>
...
```

Figure 8.5.: Declaration of resources for the reservation table of target instructions. Resources declared in this section of an xADML document represent the column entries of reservation tables.

composing that class (see Figure 8.4, first residence declaration). Observe that the name of a register that composes a residence class must first appear in the `registers` node. For residence classes of infinite size, such as memory modules, xADML declares simply a name for that class (see Figure 8.4, second declaration).

8.4.4. Resources

The `resources` node includes all other type of resources required by the instruction set. For instance, functional units, buses *etc.* appear in this part of the document. Note that registers and residence classes are not part of the `resources` node. xADML models resources using the general principle of reservation tables [DSTP75]. Each target instruction requires a particular resource at a specific time. The `resources` part of the specification simply enumerates all available resources. Each resource represents a column entry in the reservation table. The particular time of resource usage is defined in the

reservation table (see Section 8.6.3).

Defining the set of all resources of the reservation table ahead allows to report errors in the cases where a resource is used, but undeclared. In the specification document each resource requires a unique name.

8.5. Patterns

The node `patterns` allows to declare generic patterns where each of them is mapped to a single target instruction in the instruction set section. Hence, a pattern may be reused across several target instructions. We see patterns as subDAGs of the intermediate representation (IR). In our framework the IR is low enough, so that there is no such IR node that requires at least two target instructions to compute its operation.

The attribute `id` of node `pattern`, required for each pattern, is a unique identifier of the pattern being specified. This identifier is referred to when being associated with a target instruction in the instruction set part of the specification.

Each pattern is composed of nodes and directed edges. A `node` node has an optional `id` attribute. This attribute represents the reference, or the symbolic name of that node, that we can refer to within the pattern. It is a unique identifier within the pattern.

Each node of a pattern represents an instance of an IR node. We map pattern nodes to IR nodes with the `poper` construct, where the obligatory attribute `id` indicates the IR node. Further, the `arity` attribute specifies the number of descendants of the node[2]. Children of an IR node are specified via `kid` nodes. A kid node has a required `nr` attribute, representing the child number of the parent node. The number of children must match the arity specified in the parent node. The `kid` node has an optional `id` attribute used as reference, or symbolic name if specified. Each `kid` node builds implicitly a data dependence edge from child `kid` to parent `poper`. A `kid` node may have further descendant `node` nodes. Figure 8.6 illustrates the `patterns` section of xADML.

To simplify the process of writing specifications, xADML allows to specify generic patterns using the `<or>...</or>` construct to provide different alternatives for similar patterns that are mapped to the same target instruction.

Example As en example of a generic pattern with alternatives let us consider a DAG pattern on the right hand side of Figure 8.7. Triangular shaped nodes represent operands (*i.e.* subDAGs that are reduced to a given residence class). Circular shaped nodes mark IR nodes. Node `c/d` shows that there are two

[2] In the LCC-IR [FH95], each node has at most two descendants.

```
...
<!-- Patterns -->
<patterns>
  <pattern id="[string]">
    <node id="[string]">
      <poper id="[string]" arity="[integer]"/>
      <kid nr="[integer]" id="[string]">
        <node id="[string]">
          ...
        </node>
      </kid>
      ...
    </node>
  </pattern>
  ...
</patterns>
...
```

Figure 8.6.: In this section we declare all generic patterns where each pattern is mapped to a single processor target instruction.

possibilities of IR operators that will match (`c` or `d`). This alternative is expressed with the `<or>...</or>` construct illustrated on the left hand side of Figure 8.7. For instance, if the operator of node `e` is an IR assignment and the operator of the `c/d` node an addition (2 bytes integer addition for `c` and 4 bytes addition for `d`), the pattern represents a post-increment computation. ■

Although XML is a tree based representation, our definition of patterns in xADML is generic enough to allow users to specify patterns in forms of trees, DAGs or forests.

8.6. Instruction Set

An instruction in xADML represents an association between an IR pattern (often a single IR node) and a machine target instruction that performs that computation. For complex target instructions that correspond to a set of IR nodes (tree, DAG or forest patterns) xADML first defines IR patterns in `patterns` section and then associates a pattern with a given target instruction in the `instruction_set` section.

The association between a single IR node and a target instruction differs slightly from the association of a pattern and a target instruction. We call the

```
...
<!-- Patterns -->
<patterns>
  <pattern id="alt_sample">
    <node>
      <poper id="e" arity="2"/>
      <kid nr="0" id="a"/>
      <kid nr="1">
        <node>
          <or>
            <poper id="c" arity="2"/>
            <poper id="d" arity="2"/>
          </or>
          <kid nr="0" id="a"/>
          <kid nr="1" id="b"/>
        </node>
      </kid>
    </node>
  </pattern>
  ...
</patterns>
...
```

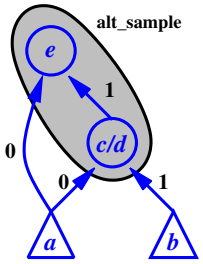

Figure 8.7.: Example of DAG pattern with two alternatives for an instance node specified using the <or>...</or> construct.

```
...
<!-- Instruction set -->
<instruction_set>
  <!-- One-to-one mapping -->
  <instruction id="[string]"> ... </instruction>
  ...
  <!-- Pattern mapping -->
  <pattern id="[string]"> ... </pattern>
  ...
</instruction_set>
...
```

Figure 8.8.: Instruction set is composed of one-to-one mapping and pattern instructions.

```
...
<!-- One-to-one mapping -->
<instruction id="[string]">
  <target id="[string]" dest1="[string]"
                        src1="[string]" src2="[string]">
    <!-- Resource usage, formating and conditions -->
  </target>
  ...
</instruction>
...
```

Figure 8.9.: Defining a target instruction and associating its operands with their residence classes in a one-to-one mapping.

first case a *one-to-one mapping*, and the second a *pattern mapping* (see Figure 8.8). Both mappings share common constructs for emitting code, condition checking and definition of reservation table.

8.6.1. One-to-one Mapping

For a one-to-one mapping between an IR operation and a target instruction, the `instruction` node simply identifies the IR node and specifies the associated target instruction. An `instruction` node has an `id` attribute that identifies the IR operation to associate with a target instruction. The target instruction is specified in the `target` node, a child node of the `instruction` node (see Figure 8.9). The `target` node contains formating information, resource requirement and condition checking, which are describe later in Section 8.6.3.

We remark that it is common that an IR operation has several possible semantically equivalent target instructions that differ, for instance, in resource usage (see Section 8.9 for an example). Thus an `instruction` node may contain several `target` nodes.

A target node has the following attributes:

- `id`, an optional attribute that records the name of the matching rule. It is mostly used for debugging and checking purposes.
- `dest1` indicates to which residence class the instruction outputs the result. For instructions that do not produce any result, the attribute is unspecified.
- `src1` represents the residence class of the first operand of the instruction. For leaf IR nodes, this field is unspecified.

```
...
<!-- Pattern mapping -->
<pattern id="[string]">
  <ptarget id="[string]" dest1="[string]">
    <op id="[string]"><id>[string]</id></op>
    ...
    <!-- Resource usage, formating, conditions and DD edges -->
  </ptarget>
  ...
</instruction>
...
```

Figure 8.10.: Associating operands of an instruction the their residence classes in a pattern mapping.

- `src2` specifies the residence class of the second operand if any. For leaf and unary nodes it is not defined.

8.6.2. Pattern Mapping

The mapping between a pattern and a target instruction is accomplished in the `pattern` node within the instruction set section. Node `pattern` has a required `id` attribute that refers to a pattern defined in the `patterns` section. In a similar way as for the `instruction` node, a `pattern` node has one or more child nodes, here called `ptarget`. Each `ptarget` node corresponds to a target instruction associated with the pattern. A `ptarget` node has an optional attribute, `dest1`, that indicates the residence class where the result of the computation corresponding to a pattern is stored if any.

Operands of a pattern are referred to through their symbolic names (*i.e.* references) that are defined in the pattern definition. Each operand node is associated to a residence class as follows:

- `<op id="[string]">` refers to the symbolic name of the node.
- `<id>[string]</id>` indicates the residence class of the operand that is referred above.
- `</op>` ends the mapping for the node.

There are as many `op` nodes as operands defined in the pattern (see Section 8.9).

```
...
<!-- Resource usage -->
<cycle_matrix [string]="[integer*]" ...>
  <cycle_matrix [string]="[integer*]" ...>
  ...
  </cycle_matrix>
</cycle_matrix>
...
```

Figure 8.11.: Modeling of reservation tables in xADML: each `cycle_matrix` construct starts a new layer of resources usage on top of a previously defined one.

8.6.3. Shared Constructs of Instruction and Pattern Nodes

In this section we describe common constructs shared by `target` and `ptarget` nodes. Each of these constructs is optional within a pattern or one-to-one mapping.

Resource Usage

A target instruction uses hardware resources to perform a desired computation. The resource usage of that instruction is specified in xADML via the reservation table (called cycle_matrix), that is, a matrix where columns represent resources and rows time slots when a given resource is used by the instruction.

We use the hierarchical structure of XML to build reservation tables. A reservation table is defined using the construct `cycle_matrix`. The attributes of a `cycle_matrix` node refer to resources defined within the `resources` section. Only resources that are used by the instruction appear as attributes.

Each new entry (descendant) defines a new row (*i.e.* a new relative time slot) of resource usage that is pushed on top of the latest time of a resource utilization. To define empty rows, xADML provides a `delay` node, whose required attribute `d` specifies the number of time slots that are unused in the reservation table. Figure 8.12 illustrates how to specify a complex reservation table in xADML.

The representation of the reservation tables in xADML is not unique. For that, we could have restricted that each row can increment by a single clock cycle the reservation table height. In this case it would be unnecessary to specify the number of cycles that a resource is busy. But for architectures with deep pipelines the proposed syntax decreases considerably the length of the reservation table specification.

```
<cycle_matrix R1="2" R2="1">
  <delay d="1"/>
  <cycle_matrix R1="1" R3="1"/>
</cycle_matrix>
```

	R_1	R_2	R_3
$t+3$	×		×
$t+2$			
$t+1$	×		
t	×	×	

Figure 8.12.: An instruction uses three resources for its execution: R_1, R_2, and R_3. Its reservation table is illustrated on the right hand side, with the relative time of each resource usage. The left hand side specifies the corresponding xADML definition of the reservation table.

Formating

Nodes `target` and `ptarget` have an optional child `format` that formats the output, if any, for the emitter[3]. It is often the case that the data to be emitted is embedded within IR nodes. Thus the hardware language must provide access functions to retrieve required information from IR nodes. Our compilation framework OPTIMIST provides such access functions.

The format construct is defined as:

```
...
<format>[string]</format>
...
```

where the string represents the format to be output.

Each IR node of the instruction or pattern is matched to a symbolic name, for instance `src1` in the case of one-to-one mapping instruction. Symbolic names that refer to IR nodes are escaped in the `format` node by enclosing them with curly brackets. If the residence class of the symbolic name is built of registers, the name is replaced with a given register when the emitter outputs the format. Otherwise, the emitter emits any information stored in the IR node, using the access functions[4].

Example Let us define a load instruction (`ld`), that takes an address, representing a memory location of a four bytes integer value and loads that value in a register. Let us assume a single memory module called `MEM`, and a set of eight general purpose registers `REG = {r0, r1, ..., r7}`. A possible definition of such an instruction in xADML could be:

[3] Usually the emitter outputs assembly code as target language, but it could be any other format.

[4] OPTIMIST framework uses LCC as `C` front-end, and thus each node is of `C` type `struct node`, as described in [FH95].

```
<instruction id="INDIRI4">
  <target id="load" dest1="REG" src1="MEM">
    ...
    <format>ld {dest1},@{src1.SYM0.X.NAME};</format>
    ...
  </target>
</instruction>
```

If we assume that this instruction is selected, it may produce the following output: `ld r3,@i;`. The symbolic name `dest1` is replaced by an allocated register from the set of registers in `REG`. The symbolic name `src1` is replaced with the name stored in the IR node that matches the source operand of the instruction. To access that name OPTIMIST provides access functions capable of queering IR nodes. In this example the name is accessed via the `.SYM0.X.NAME` access function, which points to the field `node->syms[0]->x.name` of `struct node` structure (see [FH95]). That field records the name of a global variable that is mapped to a memory location. The rest of the string is copied as plain text. ■

Conditions

The `condition` construct allows to check for a certain condition that need to be fulfilled for a pattern to match. A condition contains plain C expressions (enclosed within the `test` construct) that are evaluated when a given pattern is tried for a match. If the expression evaluates to true, then the pattern is selected, otherwise it is skipped.

Within the condition, xADML provides `and` and `or` constructs to build boolean expressions. This is provided only for XML readability, since these constructs can be expressed in plain C as well. If the logical construct is unspecified, then boolean expressions are assumed to be connected with a logical `and`. Observe that C expressions may contain a set of control characters of XML. In this case it is necessary to specify the expression using the XML escape mode that does not interpret characters, but simply copies them as plain text. It starts with the character sequence `<!CDATA[` and ends with sequence `]]>`.

Boolean expressions in the `condition` construct access IR nodes referenced by their symbolic names in a given match. Each symbolic name points to a data structure representing an IR node. In OPTIMIST this corresponds to the C `struct node` of LCC (see [FH95] for further details).

Example Let us illustrate a possible condition expression. One possibility consists in using plain XML format that expresses logical constructs with `and`

and `or` constructs, as illustrated in the upper example of Figure 8.13. Observe that we need to escape the pointer indirection of the C expression, and this is why it is enclosed within a CDATA construct. The equivalent C representation is depicted in the lower part of Figure 8.13, and uses the CDATA construct as well.

■

8.6.4. Data Dependence Edges

The standard representation of an XML tree cannot express additional data dependence edges that occur in DAG-based IRs. xADML provides a construct `ddep` that adds a data dependence edge into a pattern to resolve precedence constraints. The edge is directed from the source (`src`) to the destination (`dest`), and is specified as:

```
<ddep src="[string]" dest="[string]"/>
```

where the strings represent symbolic names of nodes composing a pattern.

The need for adding additional edges appeared when we enhanced LCC basic blocks to extended basic blocks (see Section 4.3.6).

8.7. Transfer Instructions

For irregular architectures, such as clustered VLIW processors, some data may need to be copied from one residence class to another to satisfy availability of operands in the "right" residence classes (see Section 4.3.3). Such transfers are not part of the source code of an application, but are inserted by the dynamic programming algorithm.

The transfer instructions are specified in the `transfer` section of the xADML document, using the `target` nodes. The `target` node of the transfer section is exactly the same construct as the `target` node of one-to-one instruction mapping (see Section 8.6.1), except that in the transfer section, a `target` node has exactly one source `src1` and one destination `dest1`, as depicted in Figure 8.14. Source `src1` indicates the residence class of a value computed by IR subDAG rooted at matched node. The destination `dest1` indicates to which residence class this value should be copied. Also, a `target` node includes all the common nodes of Section 8.6.3.

A `target` node has an optional `id` attribute that is used for debugging. It allows to identify which transfer instructions have been selected in the final code.

```
...
<!-- Conditions -->
<condition>
  <or>
    <and>
      <test><![CDATA[{dest1}->syms[0]!=NULL]]></test>
      <test><![CDATA[
        {dest1}->syms[0]->sclass==lcc::STATIC]]></test>
    </and>
    <or>
      <test><![CDATA[
        {src1}->syms[0]->scope==lcc::CONSTANTS]]></test>
      <test><![CDATA[{src1}->syms[0]->type==lcc::INT]]></test>
    </or>
  </or>
<condition>
...
```

The above expression can be equivalently specified as:

```
...
<!-- Conditions -->
<condition>
  <test><![CDATA[
    (({dest1}->syms[0]!=NULL) &&
     ({dest1}->syms[0]->sclass==lcc::STATIC))||
    (({src1}->syms[0]->scope==lcc::CONSTANTS)||
     ({src1}->syms[0]->type==lcc::INT))
  ]]></test>
</condition>
...
```

Figure 8.13.: Two variants of specifying a condition in xADML: the upper specification uses XML constructs to express a logical expression, whereas the lower formulation uses C syntax.

```
...
<!-- Transfers -->
<transfer>
  <target id="[string]" dest1="[string]" src1="[string]">
    ...
  </target>
  ...
</transfer>
...
```

Figure 8.14.: Modeling of transfers in xADML.

```
...
<!-- Formating facilities -->
<slot_row_prefix>[string]</slot_row_prefix>
<slot_row_suffix>[string]</slot_row_suffix>
<slot_row_separator>[string]</slot_row_separator>
<nop_instruction>[string]</nop_instruction>
<implicit_nops>[YES|NO]</implicit_nops>
...
```

Figure 8.15.: Formating specification for the general output format included in xADML.

8.8. Formating Facilities

We intend to have a free format of the output (usually assembler code). Each instruction contains a node `format` that indicates how it should be output. For single-issue architectures this is usually sufficient. However, for multi-issue platforms, we need to format each slot of an instruction word. Currently we use the following formating nodes (see Figure 8.15) that indicate how an instruction word should be formated.

- `slot_row_prefix` specifies a string that is printed before the first instruction is output. It might be a special character such as tabulation or space. If the node is unspecified or left empty, nothing is printed in front of first instruction of an instruction word.

- `slot_row_suffix` indicates the character to output at the end of an instruction word. Usually we want to have a new-line character (`\n`).

- `slot_row_separator` encodes, if necessary, a separator between different instructions of an instruction word.

- `nop_instruction` specifies the name of a NOP instruction. Many assembler languages use a macro definition for NOP instructions.
- `implicit_nops` can have a value of YES or NO. If the flag is set to YES, it is unnecessary to explicitly write a NOP instruction in the assembler file. If set to NO, then each NOP instruction is output if it is required, as for architecture with delay slots.

8.9. Examples

In this section we provide two examples of situations that often occur in xADML specification. The first example illustrates how to declare semantically equivalent instructions that match the same pattern. The second example shows how to associate each pattern operand node with a residence class.

8.9.1. Specification of Semantically Equivalent Instructions

In *mutation scheduling* semantically equivalent instructions are considered for a same computation. In xADML alternatives are specified in separate `target` node constructs. For instance, Figure 8.16 illustrates an xADML specification of an integer multiplication of an operand by another operand that is a power of two. The operation can be performed by either using a left shift or a multiplication operation.

For the multiplication, we do not need to check if the second operand `src2` is a power of two. Thus we do not need a condition construct. In the case of left shift, the operation is equivalent only if the second operand is a power of two. This is checked in the condition, which first checks if the operand is an integer data type, and then that it is a power of two. Observe that the emit part (`format` node) emits the number of bits to be shifted, and not the initial second operand value.

8.9.2. Associating Pattern Operand Nodes with Residence Classes

In this example we associate the pattern of Figure 8.7 to a target instruction, `SAMPLE`.

We assume that operands must come from residence class `REG`, the set of general purpose registers, and the result is written to the same residence class for the pattern to be selected. Figure 8.17 illustrates how pattern `alt_sample` is associated to a target instruction `SAMPLE` that takes three comma-separated arguments. The first argument is the destination register, represented by `dest1`.

```
<instruction id="MULI4">
  <target id="MPY .M1" dest1="RA" src1= RA" src2="CONST">
    <cycle_matrix M1="1"/>
    <format>MPY .M1 {src1},#{src2},{dest1}</format>
  </target>

  <target id="MPY .M1" dest1= RA" src1="RA" src2="CONST">
    <cycle_matrix S1="1"/>
    <condition>
      <and>
        <test>IS_INTEGER({src2})</test>
        <test>IS_POW_OF_2({src2})</test>
      </and>
    </condition>
    <format>SHL .S1 {src1},#LOG_2({src2}),{dest1}</format>
 </target>
</instruction>
```

Figure 8.16.: Semantically equivalent target instructions for a pattern are simply enumerated using the `target` construct.

The second argument corresponds to common subexpression in our example node `a`, and the third corresponds to node `b`. We assume, for completeness, that the target instruction uses resource `R1` for two clock cycles to perform the `SAMPLE` instruction.

In most code generation framework, the back-ends work on an IR in forms of trees. One of the motivation for writing our own specification language concerns the specification of DAG patterns.

8.10. Other Architecture Description Languages

xADML uses ideas from different architecture description languages. Here we give credits to those languages that influenced the xADML design.

The nML [Fre93] specification language uses an elegant and compact representation of the instruction set. Instructions of a target processor exhibit some common properties that can be grouped in a hierarchical representation. The identification (by the user who specifies the architecture) of such properties reduces considerably the size of the specification file, thus possible errors. In nML an instruction specification has several attributes:

- *action* indicates the semantics of the target instruction as C code with

```
<pattern id="alt_sample">
  <ptarget id="SAMLPLE_id" dest1="reg">
    <op id="a"><id>reg</id></op>
    <op id="b"><id>reg</id></op>
    <cycle_matrix R1="2"/>
    <format>SAMPLE {dest1},{a},{b}</format>
  </ptarget>
</pattern>
```

Figure 8.17.: Association of pattern `alt_sample` to the SAMPLE target instruction.

some additional DSP-related operators.

- *image* indicates the binary form of the target instruction being specified.
- *syntax* encodes the assembly mnemonic of the target instruction.

From an nML specification, it is possible to derive different tools, such as simulator or code generator, making the language suitable for software hardware co-design. In nML the conflicts among resources are specified implicitly. Basically, the user is required to specify all the legal combinations of target instructions. Another approach, adopted in ISDL (described below) is to specify illegal combinations. Depending on the target architecture the first or the second variant may be more appropriate. Further, nML seems to be primarily designed for rather simple single issue architecture. nML was successfully used in the CHESS retargetable code generation framework (see Section 9.3.2).

The *Instruction Set Description Language* (ISDL) [HHD97] used in the AVIV framework (see Section 9.3.1) includes several features of nML. However, ISDL was specially designed for VLIW processors. Similarly to xADML, ISDL is organized in different sections:

- The *instruction word format* section defines how an instruction word is subdivided into subfields. This provide a general (tree) representation of instruction words of a VLIW processor.
- The *global definitions* section provides global definitions that are used in other sections. The purpose of global definitions is to facilitate the generation of assembler from ISDL specification.
- The *storage resources* section lists storage resources, such as registers and memories available for user programs.

- The *assembly syntax* section describes each operation composing a VLIW instruction. Each instruction may be associated with a cost, such as the number of clock cycles, or the instruction size.
- The *constraints* section enumerates combinations of disallowed combinations of operations or instructions. Thus the compiler can check (at any time) if the generated sequence of target instructions is legal.
- The *optimization information* section provides hints to the compiler to improve the global code quality of the final code.

As in nML, ISDL requires to specify legal combinations of instructions of an instruction word. But contrary to nML, ISDL also constrains the code generator on a set of forbidden combinations, rather than enumerating all (possible) legal ones.

The Maril [BHE91] architecture language, which is primarily designed for RISC processors, presents most of ISDL's features, but cannot be used for VLIW processors. An interesting concept of Maril consists in including a section that represents calling conventions for a particular processor, *i.e*. models the run time behavior.

A Maril specification file is divided into four sections:

- *declarations* define hardware resources, such as registers, buses, memory banks, pipeline stages *etc*.
- *run-time model* or *compiler writer virtual machine* specifies the runtime model of the processor. The model is relatively simple, but it is a step towards the formalization of calling conventions.
- *instructions* enumerates processor target instructions. We describe this section in more details below.
- *mapping the IR* specifies how to map IR operations to target instructions.

The instruction section of Maril is subdivided into five parts:

- *mnemonic and operands* specify the assembly instruction of an operand.
- *data type constraints* indicates expected data types for an instruction.
- *tree-pattern definition* describes the tree patterns for the code generator's pattern-matcher.
- *reservation table* specifies the set of resources that the specified instruction uses at each clock cycle.

- *instruction metrics* specify the cost, latency and delay slots of an instruction.

Maril is used in the Marion [BHE91] system, which used an early version of LCC [FH95] as C front-end. We use the same front-end in the OPTIMIST framework [KB05], but we operate on DAGs and not on trees, as in the case of Marion.

The *Target Description Language* (TDL) [Käs00c] was designed for post-pass optimization. It is part of the PROPAN system [Käs00b], a retargetable framework for code optimization at assembly level. In a post-pass optimizer, the source language is assembly code and the output is (optimized) assembly code for the same architecture, or for a different one. TDL is subdivided in several sections:

- *resource* specifies the available resources of the target processor. It enumerates registers, memory modules, functional units and also caches. In a TDL specification, all resources could be used in parallel (simultaneously). As this is not usually the case, restrictions on resource usage are provided in the constraints section.
- *instruction set* is the central part of the specification where the user provides the set of target instructions with their timing behavior and resource requirement. The instruction also provides its semantics in register transfer language (RTL) [Käs00c] to be able to derive a cycle accurate simulator.
- *constraints* represent a set of boolean rules that restrict the instruction-level parallelism and resource usage according to the architecture.
- *assembly* represents the assembly mnemonic of each instruction. This section also includes assembly directives to preserve the semantics of source during code transformations.

The PROPAN post-pass optimization framework has been successfully used *e.g.* for ADSP-2106X SHARC and Philips TriMedia TM1000. Now commercialized (aiPop) by AbsInt GmbH, Saarbrücken.

Chapter 9.

Related Work

Many research groups have addressed code generation issues for a long time, and have achieved high code quality for regular architectures. However, the recent emergence of the electronics market that involves DSP processors constitutes a large pool of irregular architectures, for which standard techniques result in unsatisfying code quality.

In this chapter we classify other work on optimizing code generation for irregular target processors. We differentiate between integrated and decoupled approaches.

9.1. Decoupled Approaches

There are various methods that are able to solve exactly one of the subproblems of code generation for reasonable basic block sizes.

9.1.1. Optimal Solutions

Aho and Johnson [AJ76] use a linear-time dynamic programming algorithm to determine an optimal schedule of expression trees for a single-issue, unit-latency processor. The addressed processor class has a homogeneous register set and allows multiple addressing modes, fetching operands either from registers or directly from memory.

Chou and Chung [CC95] enumerate all possible target schedules to find an optimal one. They propose methods to prune the enumeration tree based on structural properties of the DAG such as a symmetry relation. Their algorithm works fine for basic blocks with up to 30 instructions, but instruction selection and residences are not considered. Further, they do not consider a more compressed representation of the solution space, such as our selection DAG.

Ertl and Krall [EK91] generate an optimal schedule by specifying the instruction scheduling problem as a constraint logic programming instance. A time-optimal schedule is obtained for small and medium-size basic blocks.

Wilken *et al.* [WLH00] present a set of transformations of the DAG that help to derive an advanced integer linear programming (ILP) formulation for the instruction scheduling phase. First, they show that the basic formulation of the ILP of a given problem leads to unacceptable compile time. Then, providing transformations of the DAG, they decrease the complexity of the ILP instance, and the resulting formulation can be solved in acceptable time for basic block with up to 1000 instructions.

Additionally, Wilken *et al.* derive special optimization techniques based on the shape of the DAG. They show that an hourglass-shaped DAG can be scheduled by applying a divide-and-conquer method, where a globally optimal solution is the concatenation of optimal solutions for each subDAG, below and above the articulation node. Note that an explicit application of this simplification is not necessary in our DP approach. Our DP algorithms exploit automatically such a structural property of the DAG. In fact, for regular architectures an hourglass-shaped DAG results in an hourglass-shaped selection DAG. For cases where data locality is an issue, concatenation of two optimal solutions for each subDAG does not necessarily result in an optimal global solution.

Yang *et al.* [YWL89] describe an optimal scheduling algorithm for a special architecture where all instructions require one time unit, and all functional units are identical. Generating an optimal schedule, even for such an architecture, is still $\mathcal{NP}$-complete.

Vegdahl [Veg92] proposes a dynamic programming algorithm for time-optimal scheduling that uses a similar compression strategy as described in Section 4.2.1 for combining all partial schedules of the same subset of nodes. In contrast, he does not exploit any symmetry characteristic of DAGs. He first constructs the entire selection DAG, which is not leveled in his approach, and then applies a shortest path algorithm. Contrary to Vegdahl, we take the time and space requirements of the partial schedules into account immediately when constructing the corresponding selection node. Hence, we need to construct only those parts of the selection DAG that could still lead to an optimal schedule. Further, instruction selection and residences are not considered in [Veg92].

9.1.2. Heuristic Solutions

In the literature various heuristic methods for code generation have been proposed. Heuristic methods are practical for industrial code generators since the

complexity is significantly reduced compared to exact solutions. In general, they can produce effective results within limited time and space, but they do not provide optimality information.

Methods for instruction selection are generally based on tree pattern matching [GG78] and dynamic programming.

Graph coloring [Ers71, CAC+81] methods are still the state-of-the-art for global register allocation. In order to allocate registers using the graph coloring method, the register allocation phase is usually performed after instruction scheduling, because the live range of the program values have to be known in order to be able to build the live range interference graph. In the interference graph, nodes correspond to live ranges, and there is an undirected edge between two nodes if they overlap in time and consequently cannot use the same register. Thus register allocation consists in coloring the interference graph with N colors, where N is the number of registers. A graph coloring is a decoration of the nodes with colors such that connected nodes do not have the same color.

The list scheduling algorithm is popular heuristic for local instruction scheduling [Muc97]. List scheduling determines a schedule by topological sorting. The edges of the DAG are annotated by weights, which correspond to latencies. In critical path list scheduling, the priority for selecting a node from the zero-indegree set is based on the maximum-length path from that node to any root node. The node with the highest priority is chosen first. There exist extensions to list scheduling which take more parameters into account, such as data locality for example.

9.2. Integrated Code Generation

The state-of-the-art in optimizing code generation frameworks for irregular architectures is based on ILP. The problem is formulated as an integer program, which is given to a solver. Solving an integer program is $\mathcal{NP}$-complete, hence exact solutions can be produced only for small problem instances. For larger problems the exactness is abandoned, but still a high code quality is produced.

9.2.1. Heuristic Methods

Leupers [Leu00a, Chap. 4] uses a phase-decoupled heuristic for generating code for clustered VLIW architectures. The mutual interdependence between the partitioning phase (*i.e.*, fixing a residence class for every value) and the

scheduling phase is heuristically solved by an iterative process based on simulated annealing.

9.2.2. Optimal Methods

KÄSTNER [Käs00b] developed a phase coupled postpass optimizer generator that reads in a processor specification described in Target Description Language (TDL) and generates a phase coupled postpass optimizer which is specified as an integer linear program that takes restrictions and features of the target processor into account. An exact and optimal solution is produced, or a heuristic based, if the time limit is exceeded. In this framework, the full phase integration is not possible for larger basic blocks, as the time complexity is too high.

9.3. Similar Solutions

We describe two frameworks that attempt to solve the problem of efficient code generation for irregular architectures in an integrated manner in more detail.

9.3.1. AVIV Framework

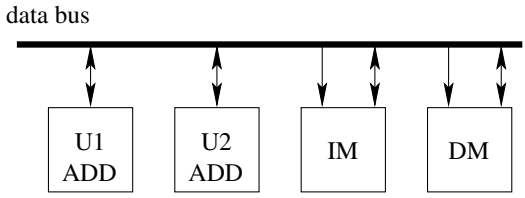

Figure 9.1.: Example of a simple target architecture to illustrate *split-node DAG* data structure.

The optimizing retargetable code generator framework AVIV [HD98] simultaneously addresses, on the basic block level, the problems of instruction selection, resource allocation, and instruction scheduling. The hardware architecture is described in *Instruction Set Description Language* (ISDL) [HHD97].

AVIV transforms the IR DAG into the *split-node DAG*, a graph representing all possible implementations of the IR operations for a given hardware processor. Additionally, the split-node DAG allows to explore instruction parallelism. A split-node DAG is similar to a DAG that represents operations

within a basic block of a source code, but contains two additional types of nodes:

- A *split node* corresponds to an original DAG operation. Split-node children represent all possible alternative implementations of the operation on the target architecture by exploiting the available functional units.
- A *data transfer node* is inserted on the data path if a data transfer between register files is required.

To deal with the combinatorial explosion, the framework uses a branch-and-bound mechanism with an aggressive heuristic for pruning suboptimal solutions as early as possible. The resulting code sequence is "nearly" optimal for small basic blocks with up to 20 instructions.

To illustrate a split-node DAG, let us first define a target architecture with two functional units U_1 and U_2. Both functional units perform addition (`ADD` instruction) and each functional unit contains its own register file. Functional units are connected by a data bus to each other and to instruction and data memory modules. The architecture is represented in Figure 9.1.

Let us consider a basic block that consists only of a single assignment statement `a=b+c+1;`. The source DAG is represented on the left hand side in Figure 9.2. The associated split-node DAG for the simple architecture described above is shown on the right hand side in Figure 9.2. Each split-node (black filled circles) has children that correspond to the original operation executed on different functional units. For example, the computation of `b+c` may be executed either on unit U_1 or U_2. Transfer nodes (boxes) are inserted if a data transfer is required between functional units. Originally, operands `b` and `c` are located in the memory and need to be transferred to the local register file of unit U_1 or U_2.

Subsequent steps in Aviv code generation operate on the split-node DAG, which contains all necessary informations for code generation. The framework receives basic blocks which are connected by a control flow graph, and performs optimization independently on each basic block. Thus Aviv does not perform any global optimizations.

First, the algorithm selects several *functional unit assignments* of least cost for further computation. The cost is evaluated by traversing the split-node DAG in a top-down manner, where the cost function considers the amount of parallelism (associated with a covering) and the number of data transfers required for the assignment. Calculating the cost for all possible assignments is impractical even for basic blocks of small size. To cope with the size of the solution space, the framework selects the most promising assignments based on a heuristic.

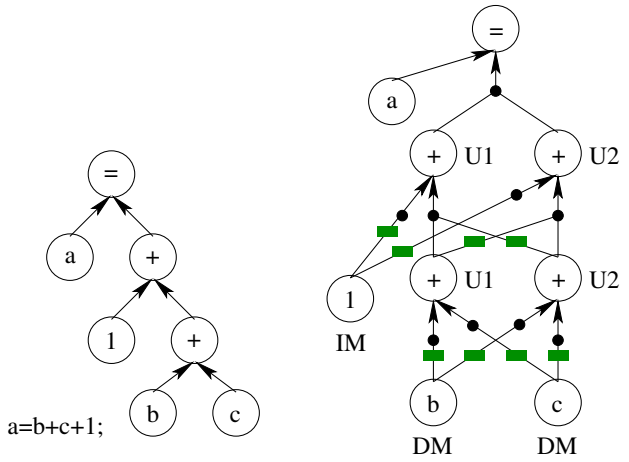

Figure 9.2.: Source DAG (left hand side) and its associated split-node DAG (right hand side).

For selected assignments of lowest cost, the required *transfer* nodes are inserted. This may result in a large number of possibilities for target architectures that present multi-path data transfer. Similarly to assignment selection, the algorithm selects candidates with the lowest cost, where the cost function only takes into account the amount of parallelism.

Next, the algorithm groups possible nodes to be executed in parallel, called cliques. Basically, the grouping is similar to code compaction [Fis81] in VLIW architectures. Once more, generating all possible groups is time consuming. To reduce time for generating cliques, the framework applies a heuristic that limits the distance (in split-node DAG) between candidate nodes for being executed in parallel. Resulting cliques are checked against hardware specification if their instructions can be executed concurrently.

In a bottom-up manner the algorithm covers the split-node DAG starting with selecting the largest clique that does not exceed the amount of available resources and have all their predecessors already scheduled. During the selection process a scheduled clique may contain a set of nodes that belongs to a still unselected group. Therefore, each time a selection is performed, the remaining cliques are shrunk (updated) by removing already scheduled nodes.

Finally, the remaining cliques that exceed the number of available resources are computed. The algorithm selects a candidate node for spilling based on the most needed resource and its number of parents. Then it inserts required spills and loads in the split-node DAG. A schedule obtained by the cliques covering

algorithm satisfies the precedence relation among the nodes, and is thus valid.

The last step consists in performing the detailed register assignment using a graph-coloring method. During the clique selection step, the algorithm guarantees that the amount of required resources does not exceed the amount of available resources. Thus the graph-coloring always finds a feasible register allocation.

We see AVIV as a not really integrated framework, as instruction selection and resource allocation are performed first. Then a phase of alternating scheduling and register allocation produces final code.

9.3.2. CHESS

CHESS [MG95] is retargetable code generation environment that addresses DSPs and Application Specific Integrated Processors (ASIP). The framework is concerned with a special class of target processors, mainly load-store architectures with homogeneous or heterogeneous register set where each instruction is microcoded, thus the computation takes one clock cycle for each instruction.

As input, CHESS takes two descriptions: the source code of the application and a structural and behavioral description of the target processor specified in *nML* [Fre93]. nML provides high level structural information and the instruction set for the target processor. From the two descriptions CHESS derives common internal data structures for code generation, called *control-data flow graph* (CDFG) and *instruction-set graph* (ISG) [MG95].

Additionally to the machine code, the framework provides some statistics that show how well the modeled target processor fits the application. This information is useful in the design loop for an embedded application, where the choice or the design of the target processor is addressed.

As a first step the IR of the program is lowered, such that all operations can be covered by an instruction from the instruction set of the processor. The phases of code selection, register allocation, bit alignment and scheduling are solved in an integrated manner. They are run separately but use a common data structure, so each phase takes into account information from other phases.

Code selection, as in most approaches, is based on pattern matching techniques for DAGs. The patterns are automatically derived from the target processor description in nML. Similarly to our approach, the patterns are identified when processing the DAG, and the algorithm avoids exhaustive enumeration of all possible coverings where unnecessary. The validity of a covering is checked against the ISG [MG95].

Register allocation in CHESS consists of routing data between different instructions to satisfy data locality. The framework considers spilling into memory, transferring data between registers and regeneration (rematerialization) of values. The choice of data transfer, based on the transfer path length, is selected in a branch-and-bound manner.

Finally, global scheduling, which also compacts micro-instructions into instructions (code compaction), uses an adapted list scheduling algorithm.

Chapter 10.

Possible Extensions

In this chapter we present possible topics for the future work in the area of integrated optimal code generation. We give a brief and non-exhaustive description of different topics. In particular, we expect to move beyond the extended basic block level, starting with loop optimization. Loops are hot spots in most DSP applications, where a slight code improvement may result in a considerable performance gain.

10.1. Residence Classes

In Section 4.2.5 we generalized register classes to residence classes, and established the versatility relation. We think of exploiting the versatility relation among residence classes by constructing a lattice of residence classes to narrow the exploration of possible schedules and thus the solution space. However, for irregular architectures residence classes may overlap with each other. For instance, an unary instruction y_i may use the set of registers, $\{r_1, r_2\}$ as source and destination operands, and instruction y_j may use registers $\{r_2, r_3\}$. In such a case, register r_2 belongs simultaneously to two overlapping sets (see also Scholz and Eckstein [SE02]).

Thus the hierarchy in the lattice cannot be constructed in a straightforward manner. Further work is needed to investigate how we should represent and treat overlapping residence classes.

10.2. xADML Extension

xADML was developed in order to allow us to retarget OPTIMIST framework for various target architectures. xADML is based on XML, which makes it convenient to manipulate, since there exist various tools and libraries for

processing XML files. However, the XML tagging principle makes the specification lengthy and difficult to handle if edited by hand. Further, the size of the specification grows rapidly with complex architectures, especially with complex addressing modes, which results in a large number of pattern definitions and associations.

Currently, we are working on a graphical tool to assist users in writing hardware specification [Reh06]. The tool includes a graphical pattern editor, that builds patterns with a drag and drop mechanism. It also includes a graphical tool for definition of reservation tables, in the form of a reservation table matrix representation, where the user simply crosses the resources (columns) used at relative times (rows). A first version of the tool is available at the OPTIMIST home page [KB05].

We also consider providing a Data Type Definition (DTD) for xADML to check the correct syntax of an xADML specification document. The checking is then performed automatically by XML tools.

10.2.1. Parallelization of the Dynamic Programming Algorithms

The dynamic programming algorithms presented in Section 4.2 offer potential for parallel computation. The algorithms structure the solution space, such that different processes can work simultaneously with processing a list of nodes at the same level but different execution time coordinates. We have to take care of the information that is written back to the memory, and thus a shared memory architecture is preferable in a first step. We also consider more distributed parallelization such as cluster machines or even grids.

10.3. Global Code Generation

Local optimization is a first step that we need to pass. The highest potential of optimization in DSP applications can be obtained if optimizing beyond (extended) basic block boundaries.

The main issue concerns loops, which are extensively used in DSP programs, and there is a considerable possible gain. We will, in the near future, consider loop optimization issues based on our dynamic programming approach and ILP as well. We should consider different profiles (*e.g.* time, space, power) as connectors between basic blocks. For instance, at the entry of a basic block we start with a predefined configuration. In the case of loop optimization we may consider loop unrolling and look for a fixed point after a number of iterations in a similar way as Kolson *et al.* [KNDK96] did optimal register assignment for loops.

An alternative could consist in providing information about data locality to the compiler by the programmer, and then observe the resulting code. For each different configuration (profiles), the dynamic algorithm will give an optimal solution for that distribution.

10.4. Software Pipelining

Software pipelining [Muc97] is an optimization technique that tends to increase the throughput of a loop. It consists in overlapping the execution of instructions from successive iterations of a loop and thus decreasing the computation time of the application. Our dynamic programming algorithms do not exploit software pipelining and we consider that issue as part of future work where loop optimization is being addressed.

We could try to extend VEGDAHL's method [Veg92] for software pipelining. First we need to integrate instruction selection and register allocation phases into the algorithm. VEGDAHL considers an operation as a non-compacted instruction. By "instruction" the author means a compacted instruction, such as a VLIW instruction word, where several operations (instructions in our terms) are gathered to form a single (target) instruction. With these properties[1] he builds a data dependence graph (at the operation level) with conflicts and fine-grained dependences.

VEGDAHL's dynamic-programming algorithm for loops builds a complete solution space for a loop body. The loop body is considered with different unroll factors Λ. If $\Lambda = 1$ the scheduling is similar to straight-line code scheduling. For $\Lambda \geqslant 2$ the loop is unrolled Λ times. For these cases the algorithm considers the issue of reentering code. Once the entire solution space is built, the algorithm proceeds to find a shortest cyclic path that corresponds to a most-compacted code sequence.

It remains unclear how far the result is optimal. If optimizing loops for code size, then it is possible to choose the smallest code (in terms of bytes) among the shortest cyclic paths. However, if optimizing for time, the shorter cycle does not necessarily mean shorter execution time.

In order to integrate instruction selection with the compaction phase (and other phases) we might build a variant of Vegdahl's *instruction-set graph*, where nodes instead represent the set of selection nodes (same level and execution time with different time profiles) and edges are annotated with selected nodes in the selection step.

[1] We see Vegdahl's properties as a limitation for applying tree and forest pattern matching techniques, since operations are already matched to a given non-compacted instruction.

This structuring defers (and eventually prunes) expensive partial solutions, which may increase the efficiency of VEGDAHL's algorithm in the same way as it worked out in earlier work on local code optimization. Hence we should try to interleave the shortest-path computation with the construction of the solution space, instead of having two separate phases as proposed by VEGDAHL.

When the algorithm constructs a backward edge, it needs to match the most suitable time profile and particularly the space profile to fit between instructions placed at the boundaries of the loop kernel.

In another alternative we may try to "apply OPTIMIST at two levels" in a two-level nested loop. The outer loop decides upon the set of nodes (from different iterations) that compose the kernel (kernel recognition). The inner loop simply uses the current OPTIMIST technique to schedule the kernel (block).

Figure 10.1 depicts the principle of OPTIMIST^2: the loop body (being optimized) is contained with the rectangular area. Nodes outside the loop body are contained within the `INIT` part (showed at the top left in Figure 10.1 as node v_0, which is a dummy node that allows to describe live-in values and incoming time profile). We (virtually) unroll the loop for showing carried dependences between different loop iterations. Each instance of node v_j of the loop is annotated with its iteration i, v_j^i. OPTIMIST^2 starts by applying OPTIMIST to the initial loop, which produces a valid schedule (see additional remarks later in this section). Then, iteratively the outer loop removes one node from the zero-indegree set out of the current loop body (marked `extract` in Figure 10.1) and inserts the corresponding node from the next iteration of the loop. Then OPTIMIST is applied to the new DAG, which now includes nodes from iteration i and $i+1$.

The outer loop implicitly builds a "kernel tree" (similar to the selection tree) where nodes are kernel DAGs to be scheduled with OPTIMIST. Similarly to the selection tree, we observe that multiple instances of kernel DAGs occur in the kernel tree. We merge those multiple occurrences of the same node into a single node, which results in a DAG of kernel DAGs.

OPTIMIST^2 complexity will certainly increase significantly compared to the complexity of the current (local) OPTIMIST algorithm. We should implement such a method in our prototype and evaluate it. We could use heuristics for the inner loop to speed up the method. Further, we may look for strategies for building and traversing the DAG of kernel DAGs in a structured manner, such as depth-first-search. We intend to rank a kernel with respect to the "appropriateness" for software pipelining. We would like to prune or postpone poor alternatives (if we can identify them) and follow branches first that lead to a quick reduction in computation time of an optimal schedule. The traversal can use a variant of branch-and-bound to favor the best partial solutions.

Additionally, we may specify how far ahead OPTIMIST^2 should consider

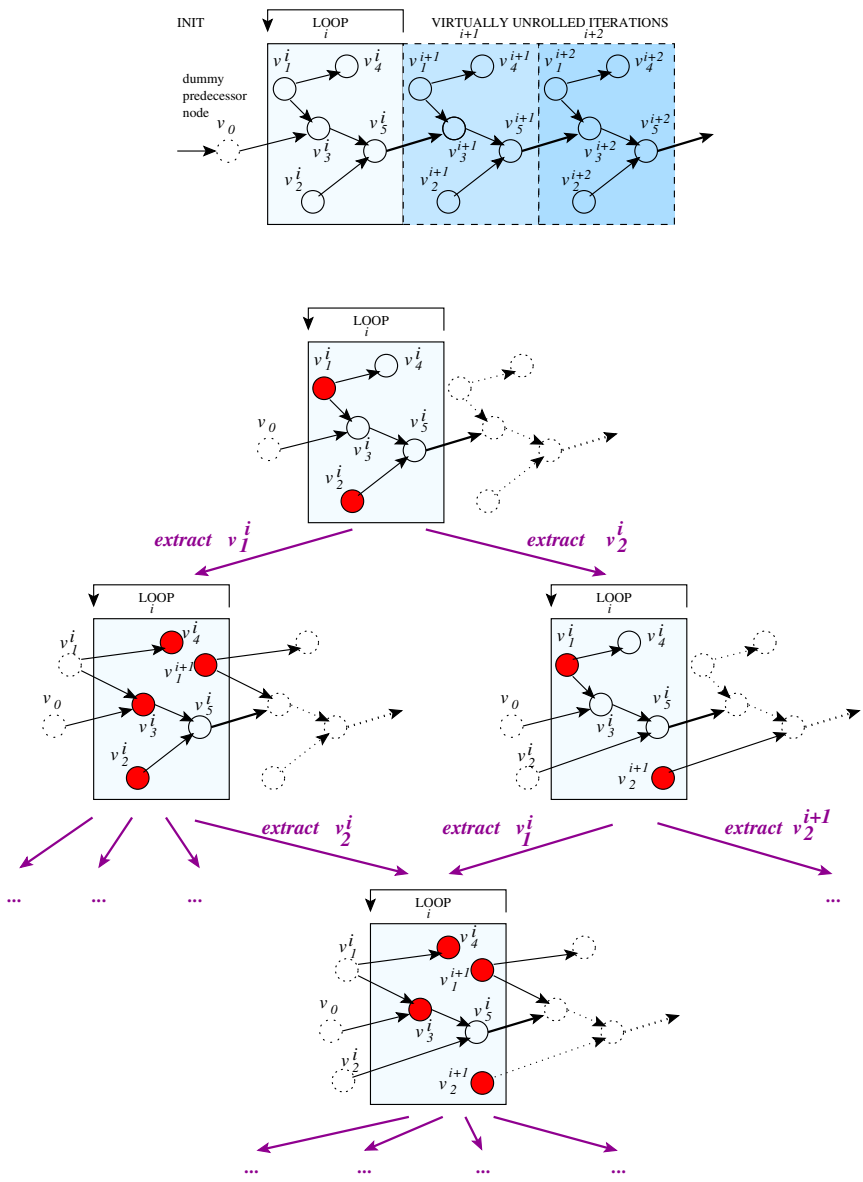

Figure 10.1.: OPTIMIST²: The principle of applying OPTIMIST at two levels.

nodes to enter the current kernel DAG (*i.e.* the *maximum depth* between the current iteration i and the last considered iteration i + d)

Remark that OPTIMIST produces schedules that are valid with respect to data dependences within a single iteration. We need to check (after a schedule is computed) that for each node that emits a loop-carried dependence the latencies are respected.

Known techniques for software pipelining [AJLA95] are usually applied after the instruction selection phase. This is somehow obvious, since software pipelining exploits the fine granularity of instructions to improve the schedule. Furthermore, the integration allows to perform instruction selection across loop iteration boundaries.

Techniques for computing the *initiation interval* (II), that is, the length of the loop kernel, such as cycle enumeration or shortest path computation rely on the instruction selection phase. The advantage of OPTIMIST[2] is that we do not need to estimate the *minimum initiation interval* (MII) and thus not approximate it through an iterative process as done in traditional software pipelining methods[2]. The information should actually be obtained directly from the algorithm.

In the case of OPTIMIST[2] we cannot calculate II ahead the schedule because the instruction selection is performed simultaneously to other phases. However, once we obtained a valid schedule (with a given instruction selection) we can verify if the lower bound for the II is achieved, or if it is achievable or not.

10.5. DAG Characterization

In Chapter 6 we observed cases for which the partial-symmetry does not improve the computation time significantly, even if there is significant number of symmetries. We plan to investigate deeply how to statically characterize DAGs that are likely to benefit from exploiting symmetries. Alternatively, a method for deciding at runtime if exploiting symmetries would pay off, that uses a heuristic estimation based on the shape of the DAG, may also be considered. We also need an extension of the partial-symmetry definition to irregular architectures that takes data locality into consideration.

We can use similar approach for characterizing DAGs statically suitable for either ILP solver or DP.

Further, we plan to develop other pruning techniques to identify alternatives that do not lead to optimal solutions as early as possible. For larger

[2] Current techniques estimate a lower bound for II after the instruction selection has been performed.

problems, we may relax our goal of finding an optimal solution and produce highly optimized, but no longer optimal code.

10.6. ILP Model Extensions

The current ILP formulation presented in Chapter 7 lacks several features available in OPTIMIST framework. In the ILP model we considered and provided a target architecture with characteristics that we modeled. We will consider extending the formulation to handle cluster architectures, such as Veloci-TI DSP variants. For that, we will need to model operand residences (*i.e.*, in which cluster or register set a value is located). This will certainly increase the amount of generated variables and equations and affect ILP performance.

Our ILP formulation is based on a simpler resource usage model that is limited to unit occupation times per functional unit and a variable latency per target instruction. It would be of interest to have a more general model using reservation tables for specifying arbitrary resource usage patterns and complex pipelines, like the one already implemented in OPTIMIST's DP framework.

10.7. Spilling

For now, we do not consider transfers to memory modules. If we did, we could as well explore all ways of spilling values from registers and reloading them at any time. Also, we assume that the register classes have enough capacity to hold all values of interest. However, this is no longer true for small residence classes, as *e.g.* in the case of the Hitachi SH3DSP. Our algorithm is, in principle, able to generate optimal spill code and take this code into account already when selecting instructions and for determining an optimal schedule, and not afterward, where the optimality of a given schedule may be compromised by later insertion of spill code.

On the other hand, taking spilling into consideration may considerably increase the space requirements. We plan to develop further methods for the elimination of uninteresting alternatives for this case.

Chapter 11.

Conclusions

In this work, we developed a framework for integrated code generation with algorithms to optimally solve the main tasks of code generation in a single and fully integrated optimization step for regular and irregular architectures, using dynamic programming and integer linear programming.

We first provided the concept of *time profile* and the compression theorem for regular architectures. The dynamic programming algorithm for superscalar and regular VLIW processors is suitable for small and medium-sized problem instances. Secondly, for irregular architectures we introduced the concept of *space profiles* to describe data locality information and provided the compression theorem for irregular architectures. The dynamic programming algorithm for clustered VLIW processors is applicable to small but not trivial problem instances.

Spilling to memory modules is currently not considered, as we assume that the register classes have enough capacity to hold all values of interest. However, this is no longer true for small residence classes, as *e.g.* in the case of the Motorola MC56K. In principle our algorithm is able to generate optimal spill code and take this code already into account when determining an optimal schedule. On the other hand, taking spill code into consideration may considerably increase the space requirements.

The dynamic programming method is suitable for optimizing for time, space, energy, and mixed goals. We adopted an energy model from the literature and presented a framework for energy-optimal integrated local code generation. We defined a suitable *power profile*, which is the key to considerable compression of the solution space in our dynamic programming algorithm. Our method is generic and not limited to a fixed power model. If more influence factors are to be considered that are known at compile time, it can easily be adapted by modifying the power profile definition accordingly.

In order to address larger problem instances we described an optimization technique that exploits partial symmetries in DAGs for regular architectures. The idea of pruning partial solutions that can be shown to be equivalent to others by analyzing local symmetries in scheduling situations did not lead to substantial improvements because the (moderate) size reduction of the solution space was outweighed by the symmetry analysis time. Partial-symmetry optimization techniques still need further investigations and extensions for irregular architectures.

Large DAGs require heuristic pruning of the solution space to cope with the combinatorial complexity of fully integrated code generation. We showed that limiting the number of alternatives generated from a partial solution nodes (ESnode) produces code with still high quality in much shorter computation times.

Further, in this thesis we provided an integer linear programming formulation for fully integrated code generation for VLIW architectures that includes instruction selection, instruction scheduling and register allocation. We have implemented the generation of data for the ILP model within the OPTIMIST framework and compared it to the dynamic programming approach. Currently, the ILP formulation lacks support for memory dependences and for irregular architecture characteristics, such as clustered register files, complex pipelines, *etc*. We intend to extend the formulation as part of future work.

Finally, we addressed the issue of retargetable code generation. We developed an architecture description language called xADML that is based on XML. We successfully retargeted the OPTIMIST framework to three very different processors: ARM9E a single issue processor, TI-C62x a multi-issue clustered VLIW architecture, and Motorola MC56K a multi-issue DSP processor.

As future work, we need address the problem of overlapping register classes and extend our approach beyond local code generation. Some ideas for extending the DP approach to software pipelining have been described.

Appendix A.

Least-cost Instruction Selection in DAGs is $\mathcal{NP}$-complete

Proebsting proved in 1998 that least-cost instruction selection in DAGs is $\mathcal{NP}$-complete [Pro98], but the proof is unpublished and only available in electronic form at the author's home page. Here follows his proof:

> Producing least-cost instruction selections for DAGs by finding pattern "matches" (or "covers" or "parses") is $\mathcal{NP}$-complete. The trivial proof follows by reducing satisfiability to it. Build a DAG for the formula you want satisfied. Then, have the following rules (all with unit cost):
>
> ```
>  True: VARIABLE
> False: VARIABLE
> False: not(True)
>  True: not(False)
>  True: or(True, True)
>  True: or(True, False)
>  True: or(False, True)
> False: or(False, False)
>  True: and(True, True)
> False: and(True, False)
> False: and(False, True)
> False: and(False, False)
> ```
>
> If you can derive a cover of the DAG that reduces to `True` with a cost exactly equal to the number of nodes, then the formula is satisfiable. Otherwise it is not.

Unless you have a different model of optimality, this proves optimal DAG code generation is NP-complete. Note that it does not rely on the usual complication of (1) number of registers, or (2) asymmetrical instructions.

Note that ERTL [Ert99] showed that least-cost instruction selection in DAGs can be computed in polynomial time for almost all regular processors.

Appendix B.

AMPL Code of ILP Model

In this appendix we provide the integer programming model presented in Chapter 7. The model is provided in AMPL mathematical programming language (see http://www.ampl.com).

```
# Use CPLEX solver
option solver cplex;

# ---------------
# Data flow graph
# ---------------

# DFG nodes
set G;

# DFG edges
set EG within (G cross G);

# Operator of DFG nodes
param OPG {G} default 0;

# Out-degree of DFG nodes
param ODG {G} default 0;

# --------
# Patterns
# --------

# Pattern indexes
set P_prime; # patterns with edges
set P_second; # patterns without edges
```

```
set P := P_prime union P_second;

param n integer > 0;  # Dummy nodes for parameters instances
set PN := 0 .. n - 1; # Generic pattern nodes

# Patterns
set B {P} within PN;
# Pattern edges
set EP {P_prime} within (PN cross PN);
# Operator of patterns
param OPP {P,PN};
# Out-degree of patterns
param ODP {P,PN} default 0;
# Latencies
param L {P};

#----------
# Resources
#----------

# Functional units
set F;
# Maximum FUs
param M {F} integer >0;
# Resource mapping p <-> FU
param U {P,F} binary default 0;

#------------
# Issue width
#------------

# Issue width (omega)
param W integer > 0;
# Number of registers
param R integer > 0;

# ------------------
# Solution variables
# ------------------

var w {EG,P_prime,(PN cross PN)} binary default 0;
var x {EG} binary default 0;

# Maximum time
```

```
param max_t integer > 0;
set T := 0 .. max_t;
var c {G,P,PN,T} binary default 0;
var exec_time integer >= 0;

# Records which patterns (instances) are selected, at which time
var s {P_prime,T} binary default 0;

# Records if a node i requires a register at time t
var r {G,T} binary default 0;

# ---------------------------------------------------------------------

# -----------------
# Optimization goal
# -----------------

minimize Test:
        exec_time;

# Minimize the number of steps
subject to MinClockCycle {i in G, p in P, k in B[p], t in T}:
        c[i,p,k,t] * (t + L[p]) <= exec_time;

# ---------------------
# Instruction selection
# ---------------------

# Each node is covered by at most one pattern
subject to NodeCoverage {i in G}:
        sum{p in P} sum{k in B[p]} sum{t in T} c[i,p,k,t] = 1;

# Record which patterns have been selected at time t
subject to Selected {p in P_prime, t in T}:
        sum{i in G} sum{k in B[p]} c[i,p,k,t] = s[p,t] * card{B[p]};

# Unique coverage:
# each pattern node corresponds to a unique DFG node (if p selected)
subject to Unicity {p in P_prime, k in B[p]}:
        sum{i in G} sum{t in T} c[i,p,k,t] = sum{t in T} s[p,t];

# For each selected pattern,
# assure that all pattern edges matche DFG edges
subject to EdgeCoverage {p in P_prime}:
```

```
        sum{(i,j) in EG} sum{(k,l) in EP[p]} w[i,j,p,k,l]
          = card{EP[p]} * sum{t in T} s[p,t];


# For each pattern edge,
# assure that DFG nodes are matched to pattern nodes
subject to WithinPattern {(i,j) in EG, p in P_prime, (k,l) in EP[p]}:
        2 * w[i,j,p,k,l] <= sum{t in T}(c[i,p,k,t] + c[j,p,l,t]);

# Only one instance of a pattern might be selected at some time t
subject to MaxOneInstance {p in P_prime}:
        sum{t in T} s[p,t] <= 1;

# Operator of DFG and pattern nodes must match
subject to OperatorEqual {i in G, p in P, k in B[p], t in T}:
        c[i,p,k,t] * (OPG[i] - OPP[p,k]) = 0;

# The out-degree of DFG and pattern nodes must match
subject to OutDegree {p in P_prime, (i,j) in EG, (k,l) in EP[p]}:
        w[i,j,p,k,l] * (ODG[i] - ODP[p,k]) = 0;

# -------------------
# Register allocation
# -------------------

# Register pressure
# Set value to register r[i,t] if there is at least one active edge
# at time t
subject to SetReg {t in T, i in G}:
        sum{tt in 0..t} sum{(i,j) in EG} sum{p in P} (
          sum{k in B[p]} c[i,p,k,tt] - sum{l in B[p]} c[j,p,l,tt]
        ) <= r[i,t] * 1000;

# If there are no active edges at time t left, set r[i,t] to zero
subject to ZeroIfNoActiveEdges {t in T, i in G}:
        sum{tt in 0..t} sum{(i,j) in EG} sum{p in P} (
          sum{k in B[p]} c[i,p,k,tt] - sum{l in B[p]} c[j,p,l,tt]
        ) >= r[i,t];

# Check that the number of registers is not exceeded at any time
subject to RegPressure {t in T}:
        sum{i in G} r[i,t] <= R;
```

```
# ----------------------
# Instruction scheduling
# ----------------------

# Precedence constraints for patterns-to (pattern or singleton)
subject to PrecedencePT {p in P_prime, (i,j) in EG, t in T}:
        sum{k in B[p]} c[i,p,k,t] +
        sum{q in P: q != p}
          sum{tt in 0 .. t + L[p] - 1: tt <= max_t}
            sum{k in B[q]} c[j,q,k,tt] <= 1;

# Precedence constraints for singleton-to (pattern or singleton)
subject to PrecedenceST {p in P_second, (i,j) in EG, t in T}:
        sum{k in B[p]} c[i,p,k,t] +
        sum{q in P} sum{tt in 0 .. t + L[p] - 1: tt <= max_t}
            sum{k in B[q]} c[j,q,k,tt] <= 1;

# -------------------
# Resource allocation
# -------------------

# At each scheduling step we should not exceed
# the number of resources
subject to Resources {t in T, f in F}:
        sum{p in P_prime : U[p,f] = 1} s[p,t]
        + sum{p in P_second : U[p,f] = 1}
            sum{i in G} sum{k in B[p]} c[i,p,k,t] <= M[f];

# At each time slot, we should not exced the issue width w
subject to IssueWidth {t in T}:
        sum{p in P_prime} s[p,t]
        + sum{p in P_second} sum{i in G} sum{k in B[p]}
          c[i,p,k,t] <= W;

# ---------------------------------------------------------------------

# ----------------
# Check statements
# ----------------

# Each pattern should be associated with som functional unit
check {p in P}:
        sum{f in F} U[p,f] > 0;
```

References

[AAvS94] Martin Alt, Uwe Assmann, and Hans van Someren. CoSy Compiler Phase Embedding with the CoSy Compiler Model. In Peter A. Fritzson, editor, *5th International Conference, CC'94*, pages 278–293, Edinburgh, U.K., April 1994. LNCS, Springer-Verlag.

[AG01] Andrew W. Appel and Lal George. Optimal Spilling for CISC Machines with Few Registers. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 243–253. ACM Press, 2001.

[AJ76] Alfred V. Aho and S. C. Johnson. Optimal Code Generation for Expression Trees. *Journal of the ACM*, 23(3):488–501, July 1976.

[AJLA95] Vicki H. Allan, Reese B. Jones, Randall M. Lee, and Stephen J. Allan. Software Pipelining. *ACM Comput. Surv.*, 27(3):367–432, 1995.

[AJU77] Alfred V. Aho, S.C. Johnson, and Jeffrey D. Ullman. Code Generation for Expressions with Common Subexpressions. *J. ACM*, 24(1):146–160, January 1977.

[AM95] G. Araujo and S. Malik. Optimal Code Generation for Embedded Memory Non-Homogeneous Register Architectures. *8th International Symposium on System Synthesis (ISSS)*, pages 36–41, 1995.

[AN88] Alexander Aiken and Alexandru Nicolau. Optimal Loop Parallelization. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, pages 308–317. ACM Press, 1988.

[ARM02] ARM Ltd. ARM9E-S^{TM}. Technical Reference Manual ARM DDI0240A Rev. 2, May 2002.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison–Wesley, 1986.

[BBS+00] David M. Brooks, Pradip Bose, Stanley E. Schuster, Hans Jacobson, Prabhakar N. Kudva, Alper Buyuktosunoglu, John-David Wellman, Victor Zyuban, Manish Gupta, and Peter W. Cook. Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors. *Annual IEEE/ACM Int. Symp. Microarchitecture*, pages 26–44, Nov-Dec 2000.

[BDB90] A. Balachandran, D. M. Dhamdhere, and S. Biswas. Efficient Retargetable Code Generation using Bottom-up Tree Pattern Matching. *Computer Languages*, 15(3):127–140, 1990.

[BG89] David Bernstein and Izidor Gertner. Scheduling Expressions on a Pipelined Processor with a Maximal Delay of one Cycle. *ACM Trans. Program. Lang. Syst.*, 11(1):57–66, January 1989.

[BGS93] Denis Barthou, Franco Gasperoni, and Uwe Schwiegelshohn. Allocating Communication Channels to Parallel Tasks. In *Environments and Tools for Parallel Scientific Computing*, volume 6 of *Advances in Parallel Computing*, pages 275–291. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, 1993.

[BHE91] David G. Bradlee, Robert R. Henry, and Susan J. Eggers. The marion system for retargetable instruction scheduling. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, pages 229–240, New York, NY, USA, June 1991. ACM Press.

[BJPR85] David Bernstein, Jeffrey M. Jaffe, Ron Y. Pinter, and Michael Rodeh. Optimal Scheduling of Arithmetic Operations in Parallel with Memory Access. Technical Report 88.136, IBM Israel Scientific Center, Technion City, Haifa (Israel), 1985.

[BL99] Steven Bashford and Rainer Leupers. Phase-Coupled Mapping of Data Flow Graphs to Irregular Data Paths. In *Design Automation for Embedded Systems*, volume 4, pages 1–50, The Netherlands, 1999. Kluwer Academic Publishers.

[BRG89] David Bernstein, Michael Rodeh, and Izidor Gertner. On the Complexity of Scheduling Problems for Parallel/Pipelined

Machines. *IEEE Trans. Comput.*, 38(9):1308–1314, September 1989.

[BS76] J. Bruno and R. Sethi. Code Generation for a One–Register Machine. *J. ACM*, 23(3):502–510, July 1976.

[BSBC95] Thomas S. Brasier, Philip H. Sweany, Steven J. Beaty, and Steve Carr. CRAIG: A Practical Framework for Combining Instruction Scheduling and Register Assignment. In *Proc. Int. Conf. Parallel Architectures and Compilation Techniques (PACT)*, 1995.

[BTM00] David Brooks, Vivek Tiwari, and Margaret Martonosi. WATTCH: A Framework for Architectural-Level Power Analysis and Optimizations. In *Proc. 27th Annual Int. Symp. Computer Architecture*, pages 83–94, June 2000.

[CAC+81] G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins, and P.W. Markstein. Register Allocation via Coloring. *Computer Languages*, 6:47–57, 1981.

[CC95] Hong-Chich Chou and Chung-Ping Chung. An Optimal Instruction Scheduler for Superscalar Processors. *IEEE Transactions on Parallel and Distributed Systems*, 6(3):303–313, 1995.

[CCK97] Chia-Ming Chang, Chien-Ming Chen, and Chung-Ta King. Using Integer Linear Programming for Instruction Scheduling and Register Allocation in Multi-issue Processors. *Computers Mathematics and Applications*, 34(9):1–14, 1997.

[CP95] Jui-Ming Chang and Massoud Pedram. Register Allocation and Binding for Low Power. In *Proc. 32nd Design Automation Conf.* ACM Press, January 1995.

[DSTP75] Edward S. Davidson, Leonard E. Shar, A. Thampy Thomas, and Janak H. Patel. Effective Control for Pipelined Computers. In *Proc. Spring COMPCON75 Digest of Papers*, pages 181–184. IEEE Computer Society Press, February 1975.

[Edq04] Anders Edqvist. High-level Optimizations for OPTIMIST. Master thesis LITH-IDA-EX-04/078-SE, Linköpings universitet, November 2004.

[EGS95] Christine Eisenbeis, Franco Gasperoni, and Uwe Schwiegelshohn. Allocating registers in multiple instruction-issuing processors. In *Proc. Int. Conf. Parallel Architectures and Compilation Techniques (PACT) (PACT'95)*, 1995. Extended version available as technical report No. 2628 of INRIA Roquencourt, France, July 1995.

[EK91] M. Anton Ertl and Andreas Krall. Optimal Instruction Scheduling Using Constraint Logic Programming. In *Proc. 3rd Int. Symp. Programming Language Implementation and Logic Programming (PLILP)*, pages 75–86. Springer LNCS 528, August 1991.

[Ell85] John R. Ellis. *Bulldog: A Compiler for VLIW Architechtures*. PhD thesis, Yale University, 1985.

[EN89] Kemal Ebcioglu and Alexandru Nicolau. A Global Resource-constrained Parallelization Technique. In *Proc. 3rd ACM Int. Conf. Supercomputing*. ACM Press, 1989.

[Ers71] A. P. Ershov. *The Alpha Programming System*. Academic Press, London, 1971.

[Ert99] M. Anton Ertl. Optimal Code Selection in DAGs. In *Proc. ACM SIGPLAN Symp. Principles of Programming Languages*, 1999.

[FH95] Christopher W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Addison-Welsey Publishing Company, 1995.

[FHP92] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a Simple, Efficient Code Generator Generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, September 1992.

[Fis81] Joseph A. Fisher. Trace Scheduling: A General Technique for Global Microcode Compaction. *IEEE Transactions on Computers*, 30(7):478–490, July 1981.

[FR92] Stefan M. Freudenberger and John C. Ruttenberg. Phase Ordering of Register Allocation and Instruction Scheduling. In *Code Generation: Concepts, Tools, Techniques [GG91]*, pages 146–170, 1992.

[Fre74] R.A. Freiburghouse. Register Allocation via Usage Counts. *Comm. ACM*, 17(11):638–642, 1974.

[Fre93] Markus Freerick. The nML Machine Description Formalism. Technical report, TU Berlin CS Dept., July 1993.

[FSF06] Inc. Free Software Foundation. GCC homepage. `http://gcc.gnu.org`, 2006.

[GE91] Catherine H. Gebotys and Mohamed I. Elmasry. Simultaneous Scheduling and Allocation for Cost Constrained Optimal Architectural Synthesis. In *DAC '91: Proceedings of the 28th conference on ACM/IEEE design automation*, pages 2–7, New York, NY, USA, 1991. ACM Press.

[GE92] C. H. Gebotys and M. I. Elmasry. *Optimal VLSI Architectural Synthesis*. Kluwer, 1992.

[GG78] R.S. Glanville and S.L. Graham. A New Method for Compiler Code Generation. In *Proc. ACM SIGPLAN Symp. Principles of Programming Languages*, pages 231–240, January 1978.

[GG91] Robert Giegerich and Susan L. Graham, editors. *Code Generation - Concepts, Tools, Techniques*. Springer Workshops in Computing, 1991.

[GH88] James R. Goodman and Wei-Chung Hsu. Code Scheduling and Register Allocation in Large Basic Blocks. In *Proc. ACM Int. Conf. Supercomputing*, pages 442–452. ACM Press, July 1988.

[GS90] Rajiv Gupta and Mary Lou Soffa. Region Scheduling: An Approach for Detecting and Redistributing Parallelism. *IEEE Trans. on Software Engineering*, 16(4):421–431, April 1990.

[Güt81] Reiner Güttler. Erzeugung optimalen Codes für series–parallel graphs. In *Springer LNCS 104*, pages 109–122, 1981.

[GYZ+99] R. Govindarajan, Hongbo Yang, Chihong Zhang, Jose Nelson Amaral, and Guang R. Gao. Minimum register instruction sequence problem: Revisiting optimal code generation for DAGs. CAPSL Technical Memo 36, Computer Architecture and Parallel Systems Laboratory, University of Delaware, Newark, November 1999.

[HD98] Silvina Hanono and Srinivas Devadas. Instruction Selection, Resource Allocation, and Scheduling in the AVIV Retargetable Code Generator. In *Proceedings of the 35th annual conference on Design Automation Conference*, pages 510–515, San Francisco, California, United States, 1998. ACM Press.

[HFG89] Wei-Chung Hsu, Charles N. Fischer, and James R. Goodman. On the Minimization of Loads/Stores in Local Register Allocation. *IEEE Trans. on Software Engineering*, 15(10):1252–1262, October 1989.

[HG83] John Hennessy and Thomas Gross. Postpass Code Optimization of Pipeline Constraints. *ACM Trans. Program. Lang. Syst.*, 5(3):422–448, July 1983.

[HHD97] George Hadjiyiannis, Silvina Hanono, and Srinivas Devadas. ISDL: An Instruction Set Description Language for Retargetability. In *Proc. Design Automation Conf.*, 1997.

[HHG+95] Wen-mei W. Hwu, Richard E. Hank, David M. Gallagher, Scott A. Mahlke, Daniel M. Lavery, Grant E. Haab, John C. Gyllenhaal, and David I. August. Compiler Technology for Future Microprocessors. *Proc. of the IEEE*, 83(12):1625–1640, December 1995.

[Hit99] Hitachi Ltd. Hitachi SuperH RISC engine SH7729. Hardware Manual ADE-602-157 Rev. 1.0, September 1999.

[HKMW66] L.P. Horwitz, R. M. Karp, R. E. Miller, and S. Winograd. Index Register Allocation. *J. ACM*, 13(1):43–61, January 1966.

[HO82] Christoph M. Hoffmann and Michael J. O'Donnell. Pattern Matching in Trees. *J. ACM*, 29(1):68–95, January 1982.

[Hu61] T. C. Hu. Parallel Sequencing and Assembly Line Problems. *Operations Research*, 9(11):841–848, 1961.

[Inc06] ILOG Inc. CPLEX homepage. `http://www.ilog.com/products/cplex`, 2006.

[JNR02] Rajeev Joshi, Greg Nelson, and Keith Randall. Denali: A goal-directed superoptimizer. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, pages 304–314, 2002.

[KAE01] Krishnan Kailas, Ashok Agrawala, and Kemal Ebcioglu. Cars: A new code generation framework for clustered ilp processors. In *HPCA '01: Proceedings of the Seventh International Symposium on High-Performance Computer Architecture (HPCA'01)*, pages 133–143. IEEE Computer Society, 2001.

[Käs00a] Daniel Kästner. *Retargetable Postpass Optimisations by Integer Linear Programming*. PhD thesis, Universität des Saarlandes, Saarbrücken, Germany, 2000.

[Käs00b] Daniel Kästner. Propan: A Retargetable System for Postpass Optimisations and Analyses. In *ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, Vancouver, CA, June 2000.

[Käs00c] Daniel Kästner. TDL - A Hardware and Assembly Description Language. Technical report, Universität des Saarlandes, Saarbrücken, Germany, 2000.

[KB02] Christoph Kessler and Andrzej Bednarski. Optimal Integrated Code Generation for Clustered VLIW Architectures. In *Proc. ACM SIGPLAN Conf. on Languages, Compilers and Tools for Embedded Systems / Software and Compilers for Embedded Systems, LCTES-SCOPES'2002*, Berlin, Germany, June 2002. ACM Press.

[KB05] Christoph Kessler and Andrzej Bednarski. OPTIMIST. `http://www.ida.liu.se/~chrke/optimist`, 2006.

[Kes98] Christoph W. Kessler. Scheduling Expression DAGs for Minimal Register Need. *Computer Languages*, 24(1):33–53, September 1998.

[Keß00] Christoph W. Kessler. Parallelism and Compilers. Habilitation thesis, FB IV - Informatik, University of Trier, December 2000.

[KG92] Tokuzo Kiyohara and John C. Gyllenhaal. Code Scheduling for VLIW/Superscalar Processors with Limited Register Files. In *Proc. 25th Annual IEEE/ACM Int. Symp. Microarchitecture*, pages 197–201. IEEE Computer Society Press, 1992.

[KNDK96] David J. Kolson, Alexandru Nicolau, Nikil Dutt, and Ken Kennedy. Optimal Register Assignment to Loops for Embedded Code Generation. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 1(2):251–279, 1996.

[KPF95] Steven M. Kurlander, Todd A. Proebsting, and Charles N. Fisher. Efficient Instruction Scheduling for Delayed-Load Architectures. *ACM Trans. Program. Lang. Syst.*, 17(5):740–776, September 1995.

[KRS98] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Code Motion and Code Placement: Just Synonyms? In *Proc. European Symp. Programming*. Springer LNCS, 1998.

[KSMS02] Eren Kursun, Ankur Srivastava, Seda Ogrenci Memik, and Majid Sarrafzadeh. Early Evaluation Techniques for Low Power Binding. In *Proc. Int. Symposium on Low power electronics and design*. ACM Press, August 2002.

[Lam88] Monica Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proc. ACM SIGPLAN Symp. Compiler Construction*, pages 318–328. ACM Press, July 1988.

[Lan05] David Landén. ARM9E Processor Specification for OPTIMIST. Master thesis LiTH-IDA-EX-05/022-SE, Linköpings universitet, February 2005.

[LEM01] Sheayun Lee, Andreas Ermedahl, and Sang Lyul Min. An Accurate Instruction-Level Energy Consumption Model for Embedded RISC Processors. In *ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, pages 1–10. ACM Press, 2001.

[Leu97] Rainer Leupers. *Retargetable Code Generation for Digital Signal Processors*. Kluwer Academic Publishers, 1997.

[Leu00a] Rainer Leupers. *Code Optimization Techniques for Embedded Processors*. Kluwer Academic Publishers, 2000.

[Leu00b] Rainer Leupers. Instruction Scheduling for Clustered VLIW DSPs. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, Philadelphia, PA, October 2000.

[LLHT00] Chingren Lee, Jenq Kuen Lee, TingTing Hwang, and Shi-Chun Tsai. Compiler Optimization on Instruction Scheduling for Low Power. In *Proc. 13th Int. Symposium on System Synthesis*, pages 55–60. ACM Press, 2000.

[LM97] Rainer Leupers and Peter Marwedel. Time-Constrained Code Compaction for DSPs. *IEEE Transactions on VLSI Systems*, 5(1):112–122, 1997.

[LTMF95] Mike Tien-Chien Lee, Vivek Tiwari, Sharad Malik, and Masahiro Fujita. Power Analysis and Low-Power Scheduling Techniques for Embedded DSP Software. In *Proc. 8th Int. Symp. on System Synthesis*, pages 110–115, 1995.

[Mas87] Henry Massalin. Superoptimizer-A Look at the Smallest Program. In *Proc. of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, pages 122–126, Palo Alto, California, United States, 1987. IEEE Computer Society Press.

[MD94] Waleed M. Meleis and Edward D. Davidson. Optimal Local Register Allocation for a Multiple-Issue Machine. In *Proc. ACM Int. Conf. Supercomputing*, pages 107–116, 1994.

[MD99] Waleed M. Meleis and Edward D. Davidson. Dual-Issue Scheduling with Spills for Binary Trees. In *Proc. tenth annual ACM-SIAM symposium on Discrete algorithms*, pages 678 – 686, 1999.

[MG95] Peter Marwedel and Gerd Goossens. *Code Generation for Embedded Processors*. Kluwer, 1995.

[MPSR95] Rajeev Motwani, Krishna V. Palem, Vivek Sarkar, and Salem Reyen. Combining Register Allocation and Instruction Scheduling (Technical Summary). Technical Report TR 698, Courant Institute of Mathematical Sciences, New York, July 1995.

[MSW01] Peter Marwedel, Stefan Steinke, and Lars Wehmeyer. Compilation Techniques for Energy-, Code-Size-, and Run-Time-Efficient Embedded Software. In *Proc. Int. Workshop on Advanced Compiler Techniques for High Performance and Embedded Processors (IWACT)*, 2001.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[Nic84] Alexandru Nicolau. Percolation Scheduling: A Parallel Compilation Technique. Technical Report 85-678, Cornell University, 1984.

[NN95] Steven Novack and Alexandru Nicolau. Mutation Scheduling: A Unified Approach to Compiling for Fine-Grain Parallelism. *Lecture Notes in Computer Science*, 892:16–30, 1995.

[NS04] Rahul Nagpal and Y. N. Srikant. Integrated Temporal and Spatial Scheduling for Extended Operand Clustered VLIW Processors. In *CF'04: Proceedings of the first conference on computing frontiers on Computing frontiers*, pages 457–470. ACM Press, 2004.

[OBC98] Emre Özer, Sanjeev Banerjia, and Thomas M. Conte. Unified Assign and Schedule: A New Approach to Scheduling for Clustered Register File Microarchitectures. In *MICRO 31: Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 308–315. IEEE Computer Society Press, 1998.

[Ped01] Massoud Pedram. Power Optimization and Management in Embedded Systems. In *Proc. Asia South Pacific Design Automation Conference*. ACM Press, January 2001.

[PF91] Todd A. Proebsting and Charles N. Fischer. Linear-Time, Optimal Code Scheduling for Delayed-Load Architectures. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, pages 256–267, June 1991.

[Pro98] Todd A. Proebsting. Least-Cost Instruction Selection for DAGs is NP-Complete, 1998.

[PS93] Krishna V. Palem and Barbara B. Simons. Scheduling Time-Critical Instructions on RISC Machines. *ACM Trans. Program. Lang. Syst.*, 15(4):632–658, 1993.

[PW96] Todd A. Proebsting and Benjamin R. Whaley. One-pass, Optimal Tree Parsing — with or without Trees. In Tibor Gyimothy, editor, *Compiler Construction (CC'96)*, pages 294–308, Linköping, 1996. Springer LNCS 1060.

[Reh06] Andreas Rehnströmer. Xe — A Graphical Editor for Writing xADML Processor Specifications. Master thesis LiTH-IDA-EX-06/006-SE, Linköpings universitet, May 2006.

[RKA99] B. Ramakrishna Rau, Vinod Kathail, and Shail Aditya. Machine-Description Driven Compilers for EPIC and VLIW

Processors. *Design automation for Embedded Systems*, 4:71–118, 1999.

[SE02] Bernhard Scholz and Erik Eckstein. Register Allocation for Irregular Architectures. In *Proceedings of the joint conference on Languages, Compilers and Tools for Embedded Systems (LCTES)*, pages 139–148. ACM Press, 2002.

[Set75] Ravi Sethi. Complete Register Allocation Problems. *SIAM Journal on Computing*, 4:226–248, 1975.

[SKWM01] S. Steinke, M. Knauer, L. Wehmeyer, and P. Marwedel. An Accurate and Fine Grain Instruction-Level Energy Model Supporting Software Optimizations. In *Ifnternational Workshop on Power And Timing Modeling, Optimization and Simulation (PATMOS)*, September 2001.

[SS02] Y. N. Srikant and Priti Shankar, editors. *Architecture Description Languages for Retargetable Compilation, in The Compiler Design Handbook: Optimizations & Machine Code Generation*, chapter 14. CRC Press, 2002.

[SSWM01] Stefan Steinke, R. Schwarz, Lars Wehmeyer, and Peter Marwedel. Low Power Code Generation for a RISC Processor by Register Pipelining. Technical Report 754, University of Dortmund, Dept. of CS XII, 2001.

[STD94] Ching-Long Su, Chi-Ying Tsui, and A.M. Despain. Low Power Architecture Design and Compilation Techniques for High-Performance Processors. In *Proc. Compcon Spring '94, Digest of Papers*, pages 489–498, February 1994.

[SU70] Ravi Sethi and Jeffrey D. Ullman. The Generation of Optimal Code for Arithmetic Expressions. *J. ACM*, 17:715–728, 1970.

[TCR98] M. Toburen, T. Conte, and M. Reilly. Instruction Scheduling for Low Power Dissipation in High Performance Microprocessors. In *Proc. Power Driven Micro-architecture Workshop in conjunction with ISCA'98*, June 1998.

[TMW94] V. Tiwari, S. Malik, and A. Wolfe. Power Analysis of Embedded Software: A First Step Towards Software Power Minimization. In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, volume 2 of *4*, pages 437–445, Dept. of Electr. Eng., Princeton Univ., NJ, USA, 1994.

[TTG+03] Andrei Terechko, Erwan Le Thenaff, Manish Garg, Jos Van Eijndhoven, and Henk Corporaal. Inter-Cluster Communication Models for Clustered VLIW Processors. In *The Ninth International Symposium on High-Performance Computer Architecture (HPCA'03)*, pages 354–364, February 2003.

[Veg92] Steven R. Vegdahl. A Dynamic-Programming Technique for Compacting Loops. In *Proc. 25th Annual IEEE/ACM Int. Symp. Microarchitecture*, pages 180–188. IEEE Computer Society Press, 1992.

[WGB94] Thomas Charles Wilson, Gary William Grewal, and Dilip K. Banerji. An ILP Solution for Simultaneous Scheduling, Allocation, and Binding in Multiple Block Synthesis. In *Proc. of the International Conference on Computer Design (ICCD)*, pages 581–586, 1994.

[WGHB94] Tom Wilson, Gary Grewal, Ben Halley, and Dilip Banerji. An integrated approach to retargetable code generation. In *ISSS '94: Proceedings of the 7th international symposium on High-level synthesis*, pages 70–75. IEEE Computer Society Press, 1994.

[WL01] Jens Wagner and Rainer Leupers. C Compiler Design for a Network Processor. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 20(11):1302–1308, November 2001.

[WLH00] Kent Wilken, Jack Liu, and Mark Heffernan. Optimal Instruction Scheduling Using Integer Programming. *ACM SIGPLAN Notices*, 35(5):121–133, May 2000.

[WMGB93] Thomas Charles Wilson, N. Mukherjee, M.K. Garg, and Dilip K. Banerji. An integrated and accelerated ilp solution for scheduling, module allocation, and binding in datapath synthesis. In *The Sixth International Conference on VLSI Desing*, pages 192–197, January 1993.

[Yon06] Yuan Yongyi. Optimization of Amplifier Code for the Motorola DSP 56367 Processor with OPTIMIST. Master thesis LiTH-IDA-EX-06/032-SE, Linköpings universitet, April 2006.

[YVKI00] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. The Design and Use of SimplePower: A Cycle-Accurate Energy Estimation Tool. In *Proc. 37th Design Automation Conf.*, 2000.

[YWL89] Cheng-I. Yang, Jia-Shung Wang, and Richard C. T. Lee. A Branch-and-Bound Algorithm to Solve the Equal-Execution Time Job Scheduling Problm with Precedence Constraints and Profile. *Computers Operations Research*, 16(3):257–269, 1989.

[Zha96] L. Zhang. *SILP. Scheduling and Allocating with Integer Linear Programming*. PhD thesis, Technische Fakultät der Universität des Saarlandes, Saarbrücken (Germany), 1996.

Index

Department of Computer and Information Science
Linköpings universitet

Dissertations

Linköping Studies in Science and Technology

No 14 **Anders Haraldsson:** A Program Manipulation System Based on Partial Evaluation, 1977, ISBN 91-7372-144-1.

No 17 **Bengt Magnhagen:** Probability Based Verification of Time Margins in Digital Designs, 1977, ISBN 91-7372-157-3.

No 18 **Mats Cedwall:** Semantisk analys av process-beskrivningar i naturligt språk, 1977, ISBN 91-7372-168-9.

No 22 **Jaak Urmi:** A Machine Independent LISP Compiler and its Implications for Ideal Hardware, 1978, ISBN 91-7372-188-3.

No 33 **Tore Risch:** Compilation of Multiple File Queries in a Meta-Database System 1978, ISBN 91-7372-232-4.

No 51 **Erland Jungert:** Synthesizing Database Structures from a User Oriented Data Model, 1980, ISBN 91-7372-387-8.

No 54 **Sture Hägglund:** Contributions to the Development of Methods and Tools for Interactive Design of Applications Software, 1980, ISBN 91-7372-404-1.

No 55 **Pär Emanuelson:** Performance Enhancement in a Well-Structured Pattern Matcher through Partial Evaluation, 1980, ISBN 91-7372-403-3.

No 58 **Bengt Johnsson, Bertil Andersson:** The Human-Computer Interface in Commercial Systems, 1981, ISBN 91-7372-414-9.

No 69 **H. Jan Komorowski:** A Specification of an Abstract Prolog Machine and its Application to Partial Evaluation, 1981, ISBN 91-7372-479-3.

No 71 **René Reboh:** Knowledge Engineering Techniques and Tools for Expert Systems, 1981, ISBN 91-7372-489-0.

No 77 **Östen Oskarsson:** Mechanisms of Modifiability in large Software Systems, 1982, ISBN 91-7372-527-7.

No 94 **Hans Lunell:** Code Generator Writing Systems, 1983, ISBN 91-7372-652-4.

No 97 **Andrzej Lingas:** Advances in Minimum Weight Triangulation, 1983, ISBN 91-7372-660-5.

No 109 **Peter Fritzson:** Towards a Distributed Programming Environment based on Incremental Compilation,1984, ISBN 91-7372-801-2.

No 111 **Erik Tengvald:** The Design of Expert Planning Systems. An Experimental Operations Planning System for Turning, 1984, ISBN 91-7372-805-5.

No 155 **Christos Levcopoulos:** Heuristics for Minimum Decompositions of Polygons, 1987, ISBN 91-7870-133-3.

No 165 **James W. Goodwin:** A Theory and System for Non-Monotonic Reasoning, 1987, ISBN 91-7870-183-X.

No 170 **Zebo Peng:** A Formal Methodology for Automated Synthesis of VLSI Systems, 1987, ISBN 91-7870-225-9.

No 174 **Johan Fagerström:** A Paradigm and System for Design of Distributed Systems, 1988, ISBN 91-7870-301-8.

No 192 **Dimiter Driankov:** Towards a Many Valued Logic of Quantified Belief, 1988, ISBN 91-7870-374-3.

No 213 **Lin Padgham:** Non-Monotonic Inheritance for an Object Oriented Knowledge Base, 1989, ISBN 91-7870-485-5.

No 214 **Tony Larsson:** A Formal Hardware Description and Verification Method, 1989, ISBN 91-7870-517-7.

No 221 **Michael Reinfrank:** Fundamentals and Logical Foundations of Truth Maintenance, 1989, ISBN 91-7870-546-0.

No 239 **Jonas Löwgren:** Knowledge-Based Design Support and Discourse Management in User Interface Management Systems, 1991, ISBN 91-7870-720-X.

No 244 **Henrik Eriksson:** Meta-Tool Support for Knowledge Acquisition, 1991, ISBN 91-7870-746-3.

No 252 **Peter Eklund:** An Epistemic Approach to Interactive Design in Multiple Inheritance Hierarchies,1991, ISBN 91-7870-784-6.

No 258 **Patrick Doherty:** NML3 - A Non-Monotonic Formalism with Explicit Defaults, 1991, ISBN 91-7870-816-8.

No 260 **Nahid Shahmehri:** Generalized Algorithmic Debugging, 1991, ISBN 91-7870-828-1.

No 264 **Nils Dahlbäck:** Representation of Discourse-Cognitive and Computational Aspects, 1992, ISBN 91-7870-850-8.

No 265 **Ulf Nilsson:** Abstract Interpretations and Abstract Machines: Contributions to a Methodology for the Implementation of Logic Programs, 1992, ISBN 91-7870-858-3.

No 270 **Ralph Rönnquist:** Theory and Practice of Tense-bound Object References, 1992, ISBN 91-7870-873-7.

No 273 **Björn Fjellborg:** Pipeline Extraction for VLSI Data Path Synthesis, 1992, ISBN 91-7870-880-X.

No 276 **Staffan Bonnier:** A Formal Basis for Horn Clause Logic with External Polymorphic Functions, 1992, ISBN 91-7870-896-6.

No 277 **Kristian Sandahl:** Developing Knowledge Management Systems with an Active Expert Methodology, 1992, ISBN 91-7870-897-4.

No 281 **Christer Bäckström:** Computational Complexity

of Reasoning about Plans, 1992, ISBN 91-7870-979-2.

No 292 **Mats Wirén:** Studies in Incremental Natural Language Analysis, 1992, ISBN 91-7871-027-8.

No 297 **Mariam Kamkar:** Interprocedural Dynamic Slicing with Applications to Debugging and Testing, 1993, ISBN 91-7871-065-0.

No 302 **Tingting Zhang:** A Study in Diagnosis Using Classification and Defaults, 1993, ISBN 91-7871-078-2.

No 312 **Arne Jönsson:** Dialogue Management for Natural Language Interfaces - An Empirical Approach, 1993, ISBN 91-7871-110-X.

No 338 **Simin Nadjm-Tehrani**: Reactive Systems in Physical Environments: Compositional Modelling and Framework for Verification, 1994, ISBN 91-7871-237-8.

No 371 **Bengt Savén:** Business Models for Decision Support and Learning. A Study of Discrete-Event Manufacturing Simulation at Asea/ABB 1968-1993, 1995, ISBN 91-7871-494-X.

No 375 **Ulf Söderman:** Conceptual Modelling of Mode Switching Physical Systems, 1995, ISBN 91-7871-516-4.

No 383 **Andreas Kågedal:** Exploiting Groundness in Logic Programs, 1995, ISBN 91-7871-538-5.

No 396 **George Fodor:** Ontological Control, Description, Identification and Recovery from Problematic Control Situations, 1995, ISBN 91-7871-603-9.

No 413 **Mikael Pettersson:** Compiling Natural Semantics, 1995, ISBN 91-7871-641-1.

No 414 **Xinli Gu:** RT Level Testability Improvement by Testability Analysis and Transformations, 1996, ISBN 91-7871-654-3.

No 416 **Hua Shu:** Distributed Default Reasoning, 1996, ISBN 91-7871-665-9.

No 429 **Jaime Villegas:** Simulation Supported Industrial Training from an Organisational Learning Perspective - Development and Evaluation of the SSIT Method, 1996, ISBN 91-7871-700-0.

No 431 **Peter Jonsson:** Studies in Action Planning: Algorithms and Complexity, 1996, ISBN 91-7871-704-3.

No 437 **Johan Boye:** Directional Types in Logic Programming, 1996, ISBN 91-7871-725-6.

No 439 **Cecilia Sjöberg:** Activities, Voices and Arenas: Participatory Design in Practice, 1996, ISBN 91-7871-728-0.

No 448 **Patrick Lambrix:** Part-Whole Reasoning in Description Logics, 1996, ISBN 91-7871-820-1.

No 452 **Kjell Orsborn:** On Extensible and Object-Relational Database Technology for Finite Element Analysis Applications, 1996, ISBN 91-7871-827-9.

No 459 **Olof Johansson:** Development Environments for Complex Product Models, 1996, ISBN 91-7871-855-4.

No 461 **Lena Strömbäck:** User-Defined Constructions in Unification-Based Formalisms,1997, ISBN 91-7871-857-0.

No 462 **Lars Degerstedt:** Tabulation-based Logic Programming: A Multi-Level View of Query Answering, 1996, ISBN 91-7871-858-9.

No 475 **Fredrik Nilsson:** Strategi och ekonomisk styrning - En studie av hur ekonomiska styrsystem utformas och används efter företagsförvärv, 1997, ISBN 91-7871-914-3.

No 480 **Mikael Lindvall:** An Empirical Study of Requirements-Driven Impact Analysis in Object-Oriented Software Evolution, 1997, ISBN 91-7871-927-5.

No 485 **Göran Forslund**: Opinion-Based Systems: The Cooperative Perspective on Knowledge-Based Decision Support, 1997, ISBN 91-7871-938-0.

No 494 **Martin Sköld**: Active Database Management Systems for Monitoring and Control, 1997, ISBN 91-7219-002-7.

No 495 **Hans Olsén**: Automatic Verification of Petri Nets in a CLP framework, 1997, ISBN 91-7219-011-6.

No 498 **Thomas Drakengren:** Algorithms and Complexity for Temporal and Spatial Formalisms, 1997, ISBN 91-7219-019-1.

No 502 **Jakob Axelsson:** Analysis and Synthesis of Heterogeneous Real-Time Systems, 1997, ISBN 91-7219-035-3.

No 503 **Johan Ringström:** Compiler Generation for Data-Parallel Programming Langugaes from Two-Level Semantics Specifications, 1997, ISBN 91-7219-045-0.

No 512 **Anna Moberg:** Närhet och distans - Studier av kommunikationsmmönster i satellitkontor och flexibla kontor, 1997, ISBN 91-7219-119-8.

No 520 **Mikael Ronström:** Design and Modelling of a Parallel Data Server for Telecom Applications, 1998, ISBN 91-7219-169-4.

No 522 **Niclas Ohlsson:** Towards Effective Fault Prevention - An Empirical Study in Software Engineering, 1998, ISBN 91-7219-176-7.

No 526 **Joachim Karlsson:** A Systematic Approach for Prioritizing Software Requirements, 1998, ISBN 91-7219-184-8.

No 530 **Henrik Nilsson:** Declarative Debugging for Lazy Functional Languages, 1998, ISBN 91-7219-197-x.

No 555 **Jonas Hallberg:** Timing Issues in High-Level Synthesis,1998, ISBN 91-7219-369-7.

No 561 **Ling Lin:** Management of 1-D Sequence Data - From Discrete to Continuous, 1999, ISBN 91-7219-402-2.

No 563 **Eva L Ragnemalm:** Student Modelling based on Collaborative Dialogue with a Learning Companion, 1999, ISBN 91-7219-412-X.

No 567 **Jörgen Lindström:** Does Distance matter? On geographical dispersion in organisations, 1999, ISBN 91-7219-439-1.

No 582 **Vanja Josifovski:** Design, Implementation and

Evaluation of a Distributed Mediator System for Data Integration, 1999, ISBN 91-7219-482-0.

No 589 **Rita Kovordányi**: Modeling and Simulating Inhibitory Mechanisms in Mental Image Reinterpretation - Towards Cooperative Human-Computer Creativity, 1999, ISBN 91-7219-506-1.

No 592 **Mikael Ericsson:** Supporting the Use of Design Knowledge - An Assessment of Commenting Agents, 1999, ISBN 91-7219-532-0.

No 593 **Lars Karlsson:** Actions, Interactions and Narratives, 1999, ISBN 91-7219-534-7.

No 594 **C. G. Mikael Johansson:** Social and Organizational Aspects of Requirements Engineering Methods - A practice-oriented approach, 1999, ISBN 91-7219-541-X.

No 595 **Jörgen Hansson:** Value-Driven Multi-Class Overload Management in Real-Time Database Systems, 1999, ISBN 91-7219-542-8.

No 596 **Niklas Hallberg:** Incorporating User Values in the Design of Information Systems and Services in the Public Sector: A Methods Approach, 1999, ISBN 91-7219-543-6.

No 597 **Vivian Vimarlund:** An Economic Perspective on the Analysis of Impacts of Information Technology: From Case Studies in Health-Care towards General Models and Theories, 1999, ISBN 91-7219-544-4.

No 598 **Johan Jenvald:** Methods and Tools in Computer-Supported Taskforce Training, 1999, ISBN 91-7219-547-9.

No 607 **Magnus Merkel:** Understanding and enhancing translation by parallel text processing, 1999, ISBN 91-7219-614-9.

No 611 **Silvia Coradeschi:** Anchoring symbols to sensory data, 1999, ISBN 91-7219-623-8.

No 613 **Man Lin:** Analysis and Synthesis of Reactive Systems: A Generic Layered Architecture Perspective, 1999, ISBN 91-7219-630-0.

No 618 **Jimmy Tjäder:** Systemimplementering i praktiken - En studie av logiker i fyra projekt, 1999, ISBN 91-7219-657-2.

No 627 **Vadim Engelson:** Tools for Design, Interactive Simulation, and Visualization of Object-Oriented Models in Scientific Computing, 2000, ISBN 91-7219-709-9.

No 637 **Esa Falkenroth:** Database Technology for Control and Simulation, 2000, ISBN 91-7219-766-8.

No 639 **Per-Arne Persson:** Bringing Power and Knowledge Together: Information Systems Design for Autonomy and Control in Command Work, 2000, ISBN 91-7219-796-X.

No 660 **Erik Larsson:** An Integrated System-Level Design for Testability Methodology, 2000, ISBN 91-7219-890-7.

No 688 **Marcus Bjäreland:** Model-based Execution Monitoring, 2001, ISBN 91-7373-016-5.

No 689 **Joakim Gustafsson:** Extending Temporal Action Logic, 2001, ISBN 91-7373-017-3.

No 720 **Carl-Johan Petri:** Organizational Information Provision - Managing Mandatory and Discretionary Use of Information Technology, 2001, ISBN-91-7373-126-9.

No 724 **Paul Scerri:** Designing Agents for Systems with Adjustable Autonomy, 2001, ISBN 91 7373 207 9.

No 725 **Tim Heyer**: Semantic Inspection of Software Artifacts: From Theory to Practice, 2001, ISBN 91 7373 208 7.

No 726 **Pär Carlshamre:** A Usability Perspective on Requirements Engineering - From Methodology to Product Development, 2001, ISBN 91 7373 212 5.

No 732 **Juha Takkinen:** From Information Management to Task Management in Electronic Mail, 2002, ISBN 91 7373 258 3.

No 745 **Johan Åberg:** Live Help Systems: An Approach to Intelligent Help for Web Information Systems, 2002, ISBN 91-7373-311-3.

No 746 **Rego Granlund:** Monitoring Distributed Teamwork Training, 2002, ISBN 91-7373-312-1.

No 757 **Henrik André-Jönsson:** Indexing Strategies for Time Series Data, 2002, ISBN 917373-346-6.

No 747 **Anneli Hagdahl:** Development of IT-suppor-ted Inter-organisational Collaboration - A Case Study in the Swedish Public Sector, 2002, ISBN 91-7373-314-8.

No 749 **Sofie Pilemalm:** Information Technology for Non-Profit Organisations - Extended Participatory Design of an Information System for Trade Union Shop Stewards, 2002, ISBN 91-7373-318-0.

No 765 **Stefan Holmlid:** Adapting users: Towards a theory of use quality, 2002, ISBN 91-7373-397-0.

No 771 **Magnus Morin:** Multimedia Representations of Distributed Tactical Operations, 2002, ISBN 91-7373-421-7.

No 772 **Pawel Pietrzak:** A Type-Based Framework for Locating Errors in Constraint Logic Programs, 2002, ISBN 91-7373-422-5.

No 758 **Erik Berglund:** Library Communication Among Programmers Worldwide, 2002, ISBN 91-7373-349-0.

No 774 **Choong-ho Yi:** Modelling Object-Oriented Dynamic Systems Using a Logic-Based Framework, 2002, ISBN 91-7373-424-1.

No 779 **Mathias Broxvall:** A Study in the Computational Complexity of Temporal Reasoning, 2002, ISBN 91-7373-440-3.

No 793 **Asmus Pandikow:** A Generic Principle for Enabling Interoperability of Structured and Object-Oriented Analysis and Design Tools, 2002, ISBN 91-7373-479-9.

No 785 **Lars Hult:** Publika Informationstjänster. En studie av den Internetbaserade encyklopedins bruksegenskaper, 2003, ISBN 91-7373-461-6.

No 800 **Lars Taxén:** A Framework for the Coordination of Complex Systems´ Development, 2003, ISBN 91-7373-604-X

No 808 **Klas Gäre:** Tre perspektiv på förväntningar och förändringar i samband med införande av informa-

tionsystem, 2003, ISBN 91-7373-618-X.

No 821 **Mikael Kindborg:** Concurrent Comics - programming of social agents by children, 2003, ISBN 91-7373-651-1.

No 823 **Christina Ölvingson:** On Development of Information Systems with GIS Functionality in Public Health Informatics: A Requirements Engineering Approach, 2003, ISBN 91-7373-656-2.

No 828 **Tobias Ritzau:** Memory Efficient Hard Real-Time Garbage Collection, 2003, ISBN 91-7373-666-X.

No 833 **Paul Pop:** Analysis and Synthesis of Communication-Intensive Heterogeneous Real-Time Systems, 2003, ISBN 91-7373-683-X.

No 852 **Johan Moe:** Observing the Dynamic Behaviour of Large Distributed Systems to Improve Development and Testing - An Emperical Study in Software Engineering, 2003, ISBN 91-7373-779-8.

No 867 **Erik Herzog:** An Approach to Systems Engineering Tool Data Representation and Exchange, 2004, ISBN 91-7373-929-4.

No 872 **Aseel Berglund:** Augmenting the Remote Control: Studies in Complex Information Navigation for Digital TV, 2004, ISBN 91-7373-940-5.

No 869 **Jo Skåmedal:** Telecommuting's Implications on Travel and Travel Patterns, 2004, ISBN 91-7373-935-9.

No 870 **Linda Askenäs:** The Roles of IT - Studies of Organising when Implementing and Using Enterprise Systems, 2004, ISBN 91-7373-936-7.

No 874 **Annika Flycht-Eriksson:** Design and Use of Ontologies in Information-Providing Dialogue Systems, 2004, ISBN 91-7373-947-2.

No 873 **Peter Bunus:** Debugging Techniques for Equation-Based Languages, 2004, ISBN 91-7373-941-3.

No 876 **Jonas Mellin:** Resource-Predictable and Efficient Monitoring of Events, 2004, ISBN 91-7373-956-1.

No 883 **Magnus Bång:** Computing at the Speed of Paper: Ubiquitous Computing Environments for Healthcare Professionals, 2004, ISBN 91-7373-971-5

No 882 **Robert Eklund:** Disfluency in Swedish human-human and human-machine travel booking dialogues, 2004. ISBN 91-7373-966-9.

No 887 **Anders Lindström:** English and other Foreign Linquistic Elements in Spoken Swedish. Studies of Productive Processes and their Modelling using Finite-State Tools, 2004, ISBN 91-7373-981-2.

No 889 **Zhiping Wang:** Capacity-Constrained Production-inventory systems - Modellling and Analysis in both a traditional and an e-business context, 2004, ISBN 91-85295-08-6.

No 893 **Pernilla Qvarfordt:** Eyes on Multimodal Interaction, 2004, ISBN 91-85295-30-2.

No 910 **Magnus Kald:** In the Borderland between Strategy and Management Control - Theoretical Framework and Empirical Evidence, 2004, ISBN 91-85295-82-5.

No 918 **Jonas Lundberg:** Shaping Electronic News: Genre Perspectives on Interaction Design, 2004, ISBN 91-85297-14-3.

No 900 **Mattias Arvola:** Shades of use: The dynamics of interaction design for sociable use, 2004, ISBN 91-85295-42-6.

No 920 **Luis Alejandro Cortés:** Verification and Scheduling Techniques for Real-Time Embedded Systems, 2004, ISBN 91-85297-21-6.

No 929 **Diana Szentivanyi:** Performance Studies of Fault-Tolerant Middleware, 2005, ISBN 91-85297-58-5.

No 933 **Mikael Cäker:** Management Accounting as Constructing and Opposing Customer Focus: Three Case Studies on Management Accounting and Customer Relations, 2005, ISBN 91-85297-64-X.

No 937 **Jonas Kvarnström:** TALplanner and Other Extensions to Temporal Action Logic, 2005, ISBN 91-85297-75-5.

No 938 **Bourhane Kadmiry:** Fuzzy Gain-Scheduled Visual Servoing for Unmanned Helicopter, 2005, ISBN 91-85297-76-3.

No 945 **Gert Jervan:** Hybrid Built-In Self-Test and Test Generation Techniques for Digital Systems, 2005, ISBN: 91-85297-97-6.

No 946 **Anders Arpteg:** Intelligent Semi-Structured Information Extraction, 2005, ISBN 91-85297-98-4.

No 947 **Ola Angelsmark:** Constructing Algorithms for Constraint Satisfaction and Related Problems - Methods and Applications, 2005, ISBN 91-85297-99-2.

No 963 **Calin Curescu:** Utility-based Optimisation of Resource Allocation for Wireless Networks, 2005. ISBN 91-85457-07-8.

No 972 **Björn Johansson:** Joint Control in Dynamic Situations, 2005, ISBN 91-85457-31-0.

No 974 **Dan Lawesson:** An Approach to Diagnosability Analysis for Interacting Finite State Systems, 2005, ISBN 91-85457-39-6.

No 979 **Claudiu Duma:** Security and Trust Mechanisms for Groups in Distributed Services, 2005, ISBN 91-85457-54-X.

No 983 **Sorin Manolache:** Analysis and Optimisation of Real-Time Systems with Stochastic Behaviour, 2005, ISBN 91-85457-60-4.

No 986 **Yuxiao Zhao:** Standards-Based Application Integration for Business-to-Business Communications, 2005, ISBN 91-85457-66-3.

No 1004 **Patrik Haslum:** Admissible Heuristics for Automated Planning, 2006, ISBN 91-85497-28-2.

No 1005 **Aleksandra Tesanovic:** Developing Reusable and Reconfigurable Real-Time Software using Aspects and Components, 2006, ISBN 91-85497-29-0.

No 1008 **David Dinka:** Role, Identity and Work: Extending the design and development agenda, 2006, ISBN 91-85497-42-8.

No 1009 **Iakov Nakhimovski:** Contributions to the Modeling and Simulation of Mechanical Systems with Detailed Contact Analysis, 2006, ISBN 91-85497-43-X.

No 1013 **Wilhelm Dahllöf:** Exact Algorithms for Exact Satisfiability Problems, 2006, ISBN 91-85523-97-6.

No 1016 **Levon Saldamli:** PDEModelica - A High-Level Language for Modeling with Partial Differential Equations, 2006, ISBN 91-85523-84-4.

No 1017 **Daniel Karlsson:** Verification of Component-based Embedded System Designs, 2006, ISBN 91-85523-

79-8.

No 1018 **Ioan Chisalita:** Communication and Networking Techniques for Traffic Safety Systems, 2006, ISBN 91-85523-77-1.

No 1019 **Tarja Susi:** The Puzzle of Social Activity - The Significance of Tools in Cognition and Cooperation, 2006, ISBN 91-85523-71-2.

No 1021 **Andrzej Bednarski:** Integrated Optimal Code Generation for Digital Signal Processors, 2006, ISBN 91-85523-69-0.

Linköping Studies in Information Science

No 1 **Karin Axelsson:** Metodisk systemstrukturering- att skapa samstämmighet mellan informa-tionssystemarkitektur och verksamhet, 1998. ISBN-9172-19-296-8.

No 2 **Stefan Cronholm:** Metodverktyg och användbarhet - en studie av datorstödd metodbaserad systemutveckling, 1998. ISBN-9172-19-299-2.

No 3 **Anders Avdic:** Användare och utvecklare - om anveckling med kalkylprogram, 1999. ISBN-91-7219-606-8.

No 4 **Owen Eriksson:** Kommunikationskvalitet hos informationssystem och affärsprocesser, 2000. ISBN 91-7219-811-7.

No 5 **Mikael Lind:** Från system till process - kriterier för processbestämning vid verksamhetsanalys, 2001, ISBN 91-7373-067-X

No 6 **Ulf Melin:** Koordination och informationssystem i företag och nätverk, 2002, ISBN 91-7373-278-8.

No 7 **Pär J. Ågerfalk:** Information Systems Actability - Understanding Information Technology as a Tool for Business Action and Communication, 2003, ISBN 91-7373-628-7.

No 8 **Ulf Seigerroth:** Att förstå och förändra systemutvecklingsverksamheter - en taxonomi för metautveckling, 2003, ISBN91-7373-736-4.

No 9 **Karin Hedström:** Spår av datoriseringens värden - Effekter av IT i äldreomsorg, 2004, ISBN 91-7373-963-4.

No 10 **Ewa Braf:** Knowledge Demanded for Action - Studies on Knowledge Mediation in Organisations, 2004, ISBN 91-85295-47-7.

No 11 **Fredrik Karlsson:** Method Configuration - method and computerized tool support, 2005, ISBN 91-85297-48-8.

No 12 **Malin Nordström:** Styrbar systemförvaltning - Att organisera systemförvaltningsverksamhet med hjälp av effektiva förvaltningsobjekt, 2005, ISBN 91-85297-60-7.

No 13 **Stefan Holgersson:** Yrke: POLIS - Yrkeskunskap, motivation, IT-system och andra förutsättningar för polisarbete, 2005, ISBN 91-85299-43-X.