Linköping Studies in Science and Technology
Dissertation No. 1375

Integrated Code Generation

by

Mattias Eriksson

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

Linköping 2011

ISBN 978-91-7393-147-2
ISSN 0345-7524
Dissertation No. 1375

Electronic version available at:
`http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-67471`
Printed by LiU-Tryck, Linköping 2011.

Abstract

Code generation in a compiler is commonly divided into several phases: instruction selection, scheduling, register allocation, spill code generation, and, in the case of clustered architectures, cluster assignment. These phases are interdependent; for instance, a decision in the instruction selection phase affects how an operation can be scheduled. We examine the effect of this separation of phases on the quality of the generated code. To study this we have formulated optimal methods for code generation with integer linear programming; first for acyclic code and then we extend this method to modulo scheduling of loops. In our experiments we compare optimal modulo scheduling, where all phases are integrated, to modulo scheduling where instruction selection and cluster assignment are done in a separate phase. The results show that, for an architecture with two clusters, the integrated method finds a better solution than the non-integrated method for 39% of the instances.

Our algorithm for modulo scheduling iteratively considers schedules with increasing number of schedule slots. A problem with such an iterative method is that if the initiation interval is not equal to the lower bound there is no way to determine whether the found solution is optimal or not. We have proven that for a class of architectures that we call transfer free, we can set an upper bound on the schedule length. I.e., we can prove when a found modulo schedule with initiation interval larger than the lower bound is optimal.

Another code generation problem that we study is how to optimize the usage of the address generation unit in simple processors that have very limited addressing modes. In this problem the subtasks are: scheduling, address register assignment and stack layout. Also for this problem we compare the results of integrated methods to the results of non-integrated methods, and we find that integration is beneficial when there are only a few (1 or 2) address registers available.

This work has been supported by The Swedish national graduate school in computer science (CUGS) and Vetenskapsrådet (VR).

Populärvetenskaplig sammanfattning

Processorer som är tänkta att användas i inbyggda system är belastade med motstridiga krav: å ena sidan skall de vara små, billiga och strömsnåla, å andra sidan skall de ha stor beräkningskraft. Kravet att processorn skall vara liten, billig och strömsnål är lättast att uppfylla genom att minimera kiselytan, medan kravet på stor beräkningskraft lättast tillfredställs genom att öka storleken på kiselytan. Denna konflikt leder till intressanta kompromisser, som till exempel klustrade registerbanker, där de olika beräkningsenheterna i processorn bara har tillgång till en delmängd av alla register. Genom att införa sådana begränsningar går det att minska storleken på processorn (det behövs färre kopplingar) utan att göra avkall på beräkningskraften. Men konsekvensen av en sådan design är att det blir svårare att skapa effektiv kod för denna arkitektur.

Att skapa program som skall köras på en processor kan göras på två sätt: antingen skriver man programmet för hand, vilket kräver stor kompetens och tar lång tid, eller så skriver man programmet på en högre abstraktionsnivå och använder en kompilator som översätter programmet till en form som kan köras på processorn. Att använda en kompilator har många fördelar, men en stor nackdel är att prestandan på det resulterande programmet ofta är mycket sämre jämfört med motsvarande program som är skapat för hand av en expert. I den här avhandlingen studerar vi om gapet i kvalitet mellan kompilatorgenererad kod och handskriven kod kan minskas genom att lösa delproblemen i kompilatorn samtidigt.

Det sista steget som görs då en kompilator översätter program till körbar maskinkod kallas för kodgenerering. Kodgenerering delas vanligtvis upp i flera delfaser: instruktionsselektion, schemaläggning, registerallokation, generering av spillkod, och, i de fall där målarkitekturen är klustrad ingår även klusterallokering som en fas. Dessa olika faser är beroende av varandra; till exempel påverkar ett beslut under instruktionsselektionsfasen hur instruktionerna kan schemaläggas. I den här avhandlingen undersöker vi effekterna av fasindelningen. För att kunna studera detta har vi skapat metoder för att generera optimal kod; i våra experiment jämför vi optimal kodgenerering, där alla faser är integrerade och löses som ett problem, med optimal kodgenerering där kodgenereringens faser utförs en åt gången. Resultaten av våra experiment visar att den integrerade metoden hittar bättre lösningar än den icke-integrerade metoden i 39% av fallen för vissa arkitekturer.

Ett annat kodgenereringsproblem som vi studerar är hur man kan optimera användandet av adressgenereringsenheten i enkla processorer där möjligheterna för minnesadressering är begränsade. Detta kodgenereringsproblem kan också delas in i faser: schemaläggning, allokering av adressregister och placering av variabler i minnet. Även för detta problem jämför vi resultaten av optimala, helt integrerade, metoder med resultat av optimala, icke integrerade, metoder. Vi finner att integreringen av faser ofta lönar sig i de fall när det bara finns 1 eller 2 adressregister.

Acknowledgments

I have learned a lot about research during the time that I have worked on this thesis. Most thanks for this goes to my supervisor Christoph Kessler who has patiently guided me and given me new ideas that I have had the possibility to work on in my own way and with as much freedom as anyone can possibly ask for. Thanks also to Andrzej Bednarski, who, together with Christoph Kessler, started the work on integrated code generation with Optimist that I have based much of my work on.

It has been very rewarding and a great pleasure to co-supervise thesis students: Oskar Skoog did great parts of the genetic algorithm in Optimist, and Zesi Cai has worked on extending this heuristic to modulo scheduling. Lukas Kemmer implemented a visualizer of schedules to work with Optimist. Magnus Pettersson worked on how to support SIMD instructions in the integer linear programming formulation. Daniel Johansson and Markus Ålind worked on libraries to simplify programming for the Cell processor (not part of this thesis).

Thanks also to my colleagues at the department of computer and information science, past and present, for creating an enjoyable atmosphere. A special thank you to the ones who contribute to fantastic discussions at the coffee table.

Sid-Ahmed-Ali Touati provided the graphs that were used in the extensive evaluation of the software pipelining algorithms. He was also the opponent at my Licentiate presentation, where he asked many good questions and gave insightful comments. I also want to thank the many anonymous reviewers of my papers for constructive comments that have often helped me to make progress.

Thanks to Vetenskapsrådet (VR) and the Swedish national graduate school in computer science (CUGS) for funding my work.

Finally, thanks to my friends and to my family for encouragement and support. I unexpectedly dedicate this thesis to my two sisters.

Mattias Eriksson
Linköping, April 2011.

Contents

List of Figures

List of Tables

Chapter 1.

Introduction

This chapter gives an introduction to the area of integrated code generation for instruction level parallel architectures and digital signal processors. The main contributions of this thesis are summarized and the thesis outline is presented.

1.1. Motivation

A processor in an embedded device often spends the major part of its lifetime executing a few lines of code over and over again. Finding ways to optimize these lines of code before the device is brought to the market could make it possible to run the application on cheaper or more energy efficient hardware. This fact motivates spending large amounts of time on aggressive code optimization. In this thesis we aim at improving current methods for code optimization by exploring ways to generate provably optimal code (in terms of throughput or code size).

Performance critical parts of programs for *digital signal processors* (DSPs) are often coded by hand because existing compilers are unable to produce code of acceptable quality. If compilers could be improved so that less hand-coding would be necessary this would have several positive effects:

- The cost to create DSP-programs would decrease because the need for highly qualified programmers would be lowered.
- It is often easier to maintain C-code than it is to maintain hardware specific assembler code.

- The portability of programs would increase because less parts of the programs will have to be rewritten for the program to be executable on a new architecture.

1.2. Compilation

A compiler is a program that translates computer programs from one language to another. In this thesis we focus on compilers that translate human readable code, e.g. written in the programming language C, into machine code for processors with *static* instruction level parallelism[1]. For such architectures it is the task of the compiler to find and make the hardware use the parallelism that is available in the source program.

The *front-end* of a compiler is the part which reads the input program and does a translation into *intermediate code* in the form of some *intermediate representation* (IR). This translation is done in steps [ALSU06]:

- First the input program is broken down into tokens. Each token corresponds to a string that has some meaning in the source language, e.g. identifiers, keywords or arithmetic operations. This phase is called *lexical analysis*.

- The second phase, known as *syntactic analysis* is where the token stream is parsed to create tree representations of sequences of the input program. A leaf node in the tree represents a value and the interior nodes represent operations on the values of its children nodes.

- *Semantic analysis* is the third phase, and this is where the compiler makes sure that the input program follows the semantic rules of the source language. For instance, it checks that the types of the values in the parse trees are valid.

[1]In the literature, the acronym ILP is used for both "instruction level parallelism" and "integer linear programming". Since both of these topics are very common in this thesis we have chosen to not use the acronym ILP for any of the terms in the running text.

- The last thing that is done in the front-end is *intermediate code generation*. This is where the intermediate code, which is understood by the back-end of the compiler, is produced. The intermediate code can, for instance, be in the form of three-address code, or in the form of directed acyclic graphs (DAGs), where the operations are similar to machine language instructions.

In this thesis we have no interest in the front-end, except for the intermediate code that it produces. Some argue that the front-end is a solved problem [FFY05]. There are of course still interesting research problems in language design and type theory, but the tricky parts in those areas have little to do with the compiler.

After the analysis and intermediate code generation parts are finished it is time for the back-end. The front-end of the compiler does not need to know anything about the target architecture, it will only know about the source programming language. The opposite is true for the back-end, which knows nothing about the source programming language and only about the target program. The front-end and back-end only need an internal language of communication that is understood by both; this is the intermediate representation. The task of creating the target program is called *code generation*. Code generation is commonly divided into at least three major subtasks which are performed, one at a time, in some sequence. The major subtasks are:

- *Instruction selection* — Select target instructions matching the IR. This phase includes resource allocation.

- *Instruction scheduling* — Map the selected instructions to time slots on which to execute them.

- *Register allocation* — Select registers in which intermediate values are to be stored.

These subtasks are interdependent; the choices made in early subtasks will constrain which choices can be made in subsequent tasks. This means that doing the subtasks in sequence is simpler and less

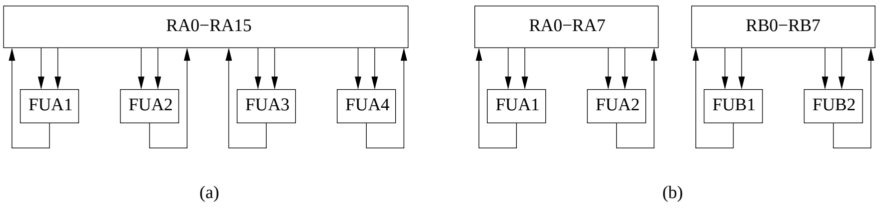


Figure 1.1: (a) Fully connected VLIWs do not scale well as the chip area for register ports is quadratic in the number of ports. (b) Clustered VLIWs limit the number of register ports by only allowing each functional unit to access a subset of the available registers [FFY05].

computationally heavy, but opportunities are missed compared to solving the subtasks as one large integrated problem. Integrating the phases of the code generator gives more opportunities for optimization at the price of an increased size of the solution space; there is a combinatorial explosion when decisions in all phases are considered simultaneously.

1.2.1. Instruction level parallelism

In this thesis we are particularly interested in code generation for *very long instruction word* (VLIW) architectures [Fis83]. For VLIW processors the issued instructions contain multiple operations that are executed in parallel. This means that all instruction level parallelism is static, i.e. the compiler (or assembler level programmer) decides which operations are going to be executed at the same point in time.

Processors that are intended to be used in embedded systems are burdened with conflicting objectives: on one hand, they must be small, cheap and energy efficient, and on the other hand they must have good computational power. To make the device small, cheap and energy efficient we want to minimize the chip area. But the requirement on computational power is easiest to satisfy by increasing the chip area. This conflict leads to interesting compromises, like clustered register banks, clustered memory banks, only very basic addressing modes, etc. These properties makes the architecture more irregular and this leads to more complicated code generation compared to code generation for general purpose architectures. Specifically, these irregularities make the interdependences between the phases of the code generation stronger.

We are interested in *clustered* VLIW architectures in which the functional units of the processor are limited to using a subset of the available registers [Fer98]. The motivation behind clustered architectures is to reduce the number of data paths and thereby making the processor use less silicon and be more scalable. This clustering makes the job of the compiler even more difficult since there are now even stronger interdependences between the phases of the code generation. For instance, which instruction (and thereby also functional unit) is selected for an operation influences to which register the produced value may be written (see Section 2.5 for a more detailed description of the phase ordering problem). Figure 1.1 shows an illustration of a clustered VLIW architecture.

1.2.2. Addressing

Advanced processors often have convenient addressing modes such as *register-plus-offset* addressing. This means that variables on the stack can be referenced with an offset from a frame pointer. While these kinds of addressing modes are convenient, they must be implemented in hardware and this increases the chip size. So, for the smallest processors the complicated addressing modes are not an option.

Therefore, these very small embedded processors use a cheaper method of addressing: address generation units (AGUs). The idea is that the AGU has a dedicated register for pointing to locations in the memory. And each time this address register is read it can, at the same time, be post-incremented or post-decremented by a small value. This design, with post-increment and post-decrement, leads to an interesting problem during code generation: how can the variables be placed in memory so that the address generation unit can be used as often as possible? We want to minimize the number of times an address register has to be explicitly loaded with a new value since this both increases code size and execution time. The problem of how to lay out the variables in the memory is known as the *simple* *offset assignment* problem in the case when there is a single address register, and the generalized problem to multiple address registers is known as the *general offset assignment* problem.

1.3. Contributions

The message of this thesis is that integrating the phases of the code generation in a compiler is often possible. And, compared to non-integrated code generation, integration of the subtasks often leads to improved results.

The main contributions of the work presented in this thesis are:

1. A fully integrated integer linear programming model for code generation, which can handle clustered VLIW architectures, is presented. To our knowledge, no such formulation exists in the literature. Our model is an extension of the model presented earlier by Bednarski and Kessler [BK06b]. In addition to adding support for clusters we also extend the model to: handle data dependences in memory, allow nodes of the IR which do not have to be covered by instructions (e.g. IR nodes representing constants), and to allow spill code generation to be integrated with the other phases of code generation.

2. We show how to extend the integer linear programming model to also integrate modulo scheduling. The results of this method are compared to the results of a non-integrated method where each subtask computes optimal results. This comparison shows how much is gained by integrating the phases.

3. The comparison of integrated versus non-integrated subtasks of code generation is also done for the offset assignment problem. In this problem the subtasks are: scheduling, stack layout and address generation unit usage.

4. We prove theoretical results on how and when the search space of our modulo scheduling algorithm may be limited from a possibly infinite size to a finite size.

The methods and algorithms that we present in this thesis can be implemented in a real compiler, or in a stand-alone optimization tool. This would make it possible to optimize critical parts of programs. It would also allow compiler engineers to compare the code generated by their heuristics to the optimal results; this would make it possible to identify missed opportunities for optimizations that are caused by unfortunate choices made in the early code generation phases. However, we do not believe that the practical use of the algorithms is the most important contribution of this thesis. The experimental results that we present have theoretical value because they quantify the improvements of the integration of code generation phases.

1.4. List of publications

Much of the material in this thesis has previously been published as parts of the following publications:

- Mattias V. Eriksson, Oskar Skoog, Christoph W. Kessler. Optimal vs. heuristic integrated code generation for clustered VLIW architectures. *SCOPES '08: Proceedings of the 11th international workshop on Software & compilers for embedded systems.* — Contains an early version of the integer linear programming model for the acyclic case and a description of the genetic algorithm [ESK08].

- Mattias V. Eriksson, Christoph W. Kessler. Integrated Modulo Scheduling for Clustered VLIW Architectures. *HiPEAC-2009 High-Performance and Embedded Architecture and Compilers*, Paphos, Cyprus, January 2009. Springer LNCS. — Includes an improved integer linear programming model for the acyclic case and an extension to modulo scheduling. This paper is also where the theoretical part on optimality of the modulo scheduling algorithm was first presented [EK09].

- Mattias Eriksson and Christoph Kessler. Integrated Code Generation for Loops. *ACM Transactions on Embedded Computing Systems. SCOPES Special Issue.* Accepted for publication.

- Christoph W. Kessler, Andrzej Bednarski and Mattias Eriksson. Classification and generation of schedules for VLIW processors. *Concurrency and Computation: Practice and Experience* 19:2369-2389, Wiley, 2007. — Contains a classification of acyclic VLIW schedules and is where the concept of dawdling schedules was first presented [KBE07].

- Mattias Eriksson and Christoph Kessler. Integrated offset assignment. *ODES-9: 9th Workshop on Optimizations for DSP and Embedded Systems*, Chamonix, France, April 2011.

Many of the experiments presented in this thesis have been rerun after the initial publication to take advantage of the recent improvements of our algorithms. Also the host computers on which the tests were done have been upgraded, and there have been improvements in the integer linear programming solvers that we use. Some experiments have been added that are not previously published.

1.5. Thesis outline

The remainder of this thesis is organized as follows:

- Chapter 2 provides background information that is important to understand when reading the rest of the thesis.

- Chapter 3 contains our integrated code generation methods for the acyclic case: First the integer linear programming model, and then the genetic algorithm heuristic.

- In Chapter 4 we extend the integer linear programming model to modulo scheduling. We also present the search algorithm that we use and prove that the search space can be made finite.

- Chapter 5 contains another integrated code generation problem: integrating scheduling and offset assignment for architectures with limited addressing modes..

- Chapter 6 shows related work in acyclic and cyclic integrated code generation.

- Chapter 7 lists topics for future work.
- Chapter 8 concludes the thesis.
- In Appendix A we show AMPL-listings used for the evaluations in Chapters 4 and 5.

Chapter 2. Background and terminology

This chapter contains a general background of the work in this thesis. The goal here is to only present some of the basic concepts that are necessary for understanding the rest of the thesis. For a more in-depth treatment of general compiling topics for embedded processors we recommend the book by Fisher et al. [FFY05].

In this chapter we focus on the basics; a more thorough review of research that is related to the work that we present in this thesis can be found in Chapter 6.

2.1. Intermediate representation

In a modern compiler there are usually more than one form of *intermediate representations* (IR). A program that is being compiled is often gradually lowered from high-level IRs, such as abstract syntax trees, to lower level IRs such as control flow graphs. High-level IRs have a high level of abstraction; for instance, array accesses are explicit. In the low-level IRs the operations are more similar to machine code; for instance, array accesses are translated into pure memory accesses. Having multiple levels of IR means that analysis and optimizations can be done at the most appropriate level of abstraction.

In this thesis we assume that the IR is a directed graph with a low level of abstraction. Each node in the graph is a simple operation such as an addition or a memory load. A directed edge between two nodes, u and v, represents a dependence meaning that operation v can not be started before operation u is finished. The graph must be acyclic if it represents a basic block, but cycles can occur if the graph represents the operations of a loop where there are loop-carried dependences.

2.2. Instruction selection

When we generate code for the target architecture we must select instructions from the target architecture instruction set for each operation in the intermediate code. This mapping from operations in the intermediate code to target code is not simple; there is usually more than one alternative when a target instruction is to be selected for an operation in the intermediate code. For instance, an addition in intermediate code could be executed on any of several functional units as a simple addition, or it can in some circumstances be done as a part of a multiply-and-add instruction.

The instruction selection problem can be seen as a pattern matching problem. Each instruction of the target architecture correspond to one or more *patterns*. Each pattern consists of one or more *pattern nodes*. For instance a multiply-and-add instruction has a pattern with one multiply node and one addition node. The pattern matching problem is to select, given the available patterns, instructions such that every node in the IR-graph is covered by exactly one *pattern node*. This will be discussed in more detail in Chapter 3.

If a cost is assigned each target instruction, the problem of minimizing the accumulated cost when mapping all operations in intermediate code represented by a DAG is NP-complete.

We assume that the IR-graph is *fine-grained* with respect to the instruction set in the sense that each instruction can be represented as a DAG of IR-nodes and it is never the case that a single node in the graph needs more than one target instruction to be covered.

2.3. Scheduling

Another main task of code generation is scheduling. When scheduling is done for a target architecture that has no instruction level parallelism it is simply the task of deciding in which order the operations will be issued. However, when instruction level parallelism is available, the scheduling task has to take this into consideration and produce a schedule that can utilize the multiple functional units in a good way.

One important goal of the scheduling is to minimize the number of intermediate values that need to be kept in registers; if we get too many intermediate values, the program will have to temporarily save values to memory and later load them back into registers; this is known as *spilling*. Already the task of minimizing spilling is NP-complete, and when we consider instruction level parallelism the size of the solution space increases even more.

2.4. Register allocation

A value that is stored in a register in the CPU is much faster to access than a value that is stored in the memory. However, there is only a limited number of registers available; this means that, during code generation, we must decide which values are going to be stored in registers. If some values do not fit, spill code must be generated.

2.5. The phase ordering problem

A central topic in this thesis is the problem caused by dividing code generation into subsequent phases. As an example consider the dependences between instruction selection and scheduling: if instruction selection is done first and instructions I_a and I_b are selected for operations a and b, respectively, where a and b are operations in the intermediate code, then, if I_a and I_b use the same functional unit, a and b can not be executed at the same time slot in the schedule. And this restriction is caused by decisions taken in the instruction selection phase; there is no other reason why it should not be possible to schedule a and b in the same time slot using a different instruction I'_a that does not use the same resource as I_b. Conversely, if scheduling is done first and a and b are scheduled on the same time slot, then I_a

and I_b cannot use the same functional unit. In this case as well the restriction comes from a decision made the previous phase. Hence, no matter how we order scheduling and instruction selection, the phase that comes first will sometimes constrain the following phase in such a way that the optimal target program is impossible to achieve.

Another example of the phase ordering problem is the interdependences between scheduling and register assignment: If scheduling is done first then the live ranges of intermediate values are fixed, and this will constrain the freedom of the register allocator. And if register allocation is done first, this will introduce new dependences for the scheduler.

We can reduce some effects of the phase ordering problem by making the early phases *aware* of the following phases. For instance, in scheduling we can try to minimize the lengths of each live range. This will reduce the artificial constraints imposed on register allocation, but can never completely remove them. Also, there may be conflicting goals: for instance, when a system has cache memory, we want to schedule loads as early as possible to minimize the effect of a cache miss (assuming that the architecture is stall-on-use [FFY05]), but we must also make sure that live ranges are short enough so that we do not run out of registers [ATJ09].

The effects of the phase ordering problems are not completely understood, and the problem of what order to use is an unsolved problem. We do not know which ordering of the phases leads to the best results, or by how much the results can be improved if the phases are integrated and solved as one problem. This will be the theme of this thesis: understanding how the code quality improves when the phases are integrated compared to non-integrated code generation.

2.6. Instruction level parallelism

To achieve good performance in a processor it is important that multiple instructions can be running at the same time. One way to accomplish this is to make the functional units of the processor pipelined. The concept of pipelining relies on the fact that some instructions

```
LD ..., R0 ; LD uses the functional unit
NOP        ; LD uses the functional unit
LD ..., R1 ; delay slot
NOP        ; delay slot
NOP        ; delay slot
           ; R0 contains the result
```

Figure 2.1: Example: The occupation time of load is 2, the delay is 3 and the latency is 5. This means that a second load can start 2 cycles after the first load.

use more than one clock cycle to produce the result. If the execution of such an instruction can be divided into smaller steps in hardware, we can issue subsequent instructions before all previous instructions are finished. See Figure 2.1 for an example; the *occupation time* is the minimum number of cycles before the next (independent) instruction can be issued; the *latency* is the number of clock cycles before the result of an instruction is visible; and the *delay* is latency minus occupation time.

Another way to increase the computational power of a processor is to add more functional units. Then, at each clock cycle, we can start as many instructions as we have functional units. The hardware approach to utilize multiple functional units is to add issue logic to the chip. This issue logic will look at the incoming instruction stream and dynamically select instructions that can be issued at the current time slot. This is a very easy way to increase the performance of sequential code. But of course this comes at the price of devoting chip area to the issue logic. Processors that use this technique are known as *superscalar* processors.

A *reservation table* is used to keep track of resource usage. The reservation table is a boolean matrix where the entry in column u and row r indicates that the resource u is used at time r. The concept of a reservation table is used for individual instructions, for entire blocks of code, and for partial solutions.

For an embedded system, using much chip area for issue logic may not be acceptable; it both increases the cost of the processors and their power consumption. Instead we can use the software method for utilizing multiple functional units, by leaving the task of finding instructions to execute in parallel to the compiler. This is done by using *very long instruction words* (VLIW), in which multiple operations may be encoded in a single instruction word, and all of the individual instructions will be issued at the same clock cycle (to different functional units).

2.7. Software pipelining

For a processor with multiple functional units it is important that the compiler can find instructions that can be run at the same time. Sometimes the structure of a program is a limiting factor; if there are too many dependences between the operations in the intermediate code it may be impossible to achieve good utilization of the processor resources. One possibility for increasing the available instruction level parallelism is to do transformations on the intermediate code. For instance, we can unroll loops so that multiple iterations of the loop code are considered at the same time; this means that the scheduler can select instructions from different iterations to run at the same cycle. A disadvantage of loop unrolling is that the code size is increased and that the scheduling problems becomes more difficult because the problem instances are larger.

Another way in which we can increase the throughput of loops is to create the code for a single iteration of the loop in such a way that iteration $i+1$ can be started before iteration i is finished. This method is called *modulo scheduling* and the basic idea is that new iterations of the loop are started at a fixed interval called the *initiation interval* (II). Modulo scheduling is a form of *software pipelining* which is a class of cyclic scheduling algorithms with the purpose of exploiting inter-iteration instruction level parallelism.

When doing modulo scheduling of loops a *modulo reservation table* is used. The modulo reservation table must have one row for each cycle of the resulting kernel of the modulo scheduled loop, this means

Listing 2.1: C-code

```
for(i=0; i < N; i++) {
  sum += A[i] * B[i];
}
```

Listing 2.2: Compacted schedule for one iteration of the loop.

```
L: LD *(R0++), R2
   LD *(R1++), R3
   NOP
   MPY R2, R3, R4
   NOP
   ADD R4, R5, R5
```

Listing 2.3: Software pipelined code.

```
1      LD
2      LD
3      NOP  || LD
4      MPY  || LD
5  L:  NOP  || NOP  || LD
6      ADD  || MPY  || LD       || BR L
7              NOP  || NOP
8              ADD  || MPY
9                      NOP
10                     ADD
```

that the modulo reservation table will have II rows and the entry in column u and row r indicates if resource u is used at time $r + k \cdot II$ in the iteration schedule for some integer k.

2.7.1. A dot-product example

We can illustrate the concept of software pipelining with a simple example: a dot-product calculation, see Listing 2.1. If we have a VLIW processor with 2 stage-pipelined load and multiply, and 1 stage add, we can generate code for one iteration of the loop as in Listing 2.2 (initialization and branching not included). To use software pipelining we must find the fastest rate at which new iterations of the schedule can be started. In this case we note that a new iteration can be started every second clock cycle, see Listing 2.3. The code in instructions 1 to 4 fill the pipeline and is called the *prolog*. Instructions 5 and 6 are the *steady state*, and make up the body of the software pipelined loop, this is sometimes also called *kernel*. And the code in instructions 7 to 10 drain the pipeline and are known as the *epilog*.

In this example the initiation interval is 2, which means that in every second cycle an iteration of the original loop finishes, except for during the prolog. I.e. the throughput of the software pipelined loop approaches $1/II$ iterations per cycle as the number of iterations increases.

Another point that is worth noticing is that if the multiplication and addition had been using the same functional unit then the code in Listing 2.2 would still be valid and optimal, but the software pipelined code in Listing 2.3 would not be valid since, at cycle 6, the addition and multiplication happen at the same time. That means that we would have to increase the initiation interval to 3, which makes the throughput 50% worse. Or we can add a `nop` between the multiplication and the addition in the iteration schedule, which would allow us to keep the initiation interval of 2. But the iteration schedule is no longer optimal; this example shows that it is not always beneficial to use locally compacted iteration schedules when doing software pipelining.

2.7.2. Lower bound on the initiation interval

Once the instruction selection is done for the operations of the loop body we can calculate a lower bound on the initiation interval. One lower bound is given by the available resources. For instance, in Listing 2.2 we use `LD` twice; if we assume that there is only one load-store unit, then the initiation interval can not be lower than 2. A lower bound of this kind is called *resource-constrained minimum initiation interval* (*ResMII*).

Another lower bound on the initiation interval can be found by inspecting cycles in the graph. In our example there is only one cycle: from the add to itself, with an *iteration difference* of 1, meaning that the addition must be finished before the addition in the following iteration begins. In our example, assuming that the latency of `ADD` is 1, this means that the lower bound of the initiation interval is 1. In larger problem instances there may be many more cycles; in the worst case the number of cycles is exponential in the number of nodes. Hence finding the critical cycle may not be possible within reasonable time. Still it may be useful to find a few cycles and calculate the lower bound based on this selection; the lower bound will still be valid, but it may not be the tightest lower bound that can be found. The lower bound on the initiation interval caused by dependence cycles (recurrences) is known as *recurrence-constrained minimum initiation interval* (*RecMII*).

Taking the largest of *RecMII* and *ResMII* gives a total lower bound called *minimum initiation interval* (*MinII*).

2.7.3. Hierarchical reduction

The software pipelining technique described above works well for inner loops where the loop body does not contain conditional statements. If we want to use software pipelining for loops that contain if-then-else code we can use a technique known as *hierarchical reduction* [Lam88]. The idea is that the then and else branches are scheduled individually and then the entire if-then-else part of the graph is reduced to a single node, where the new node represents the union of the then-branch and the else-branch. After this is done the scheduling proceeds as before and we search for the minimum initiation interval, possibly doing further hierarchical reductions. When code is emitted for the loop, any instructions that are concurrent with the if-then-else code are duplicated, one instruction is inserted into the then-branch and the other in the else-branch.

The same technique can be used for nested loops, where the innermost loop is scheduled first and then the entire loop is reduced to a single node, and instructions from the outer nodes can now be executed concurrently with the prolog and epilog of the inner loop. Doing this hierarchical reduction will both reduce code size and improve the throughput of the outer loop.

2.8. Integer linear programming

Optimization problems which have a linear optimization objective and linear constraints are known as *linear programming* problems. A linear programming problem can be written:

$$\begin{aligned} &\min \sum_{j=1}^{n} c_j x_j \\ \text{s.t.}\forall i \in \{1,\ldots,m\},\ &\sum_{j=1}^{n} a_{i,j} x_j \leq b_i \\ \forall j \in \{1,\ldots,n\},\ &\mathbb{R} \ni x_j \geq 0 \end{aligned} \tag{2.1}$$

Where x_j are non-negative solution variables, c_j are the coefficients of the objective function, and $a_{i,j}$ are the coefficients of the constraints. Assuming that the constraints form a finite solution space, an optimal solution, i.e. an assignment of values to the variables, is found on the edge of the feasible region defined by the constraints. An optimal solution to an LP-problem can be found in polynomial time.

If we constrain the variables of the linear programming problem to be integers, we get an *integer linear programming* problem. And finding an optimal solution to an integer linear programming problem is an NP-complete problem, i.e. there is no known algorithm that solves the problem in polynomial time.

Solvers use a technique known as *branch-and-cut* for solving integer linear programming instances. The algorithm works by temporarily relaxing the problem by removing the integer constraint. If the relaxed problem has a solution that happens to be integer despite this relaxation, then we are done, but if a variable in the solution is non-integer, then one of two things can be done: Either non-integral parts of the search space are removed by adding more constraints (called cuts). Or, the algorithm branches by selecting a variable, x_s, that has a non-integer value $\hat{x}_s$ and creating two subproblems, the first subproblem adds the constraint $x \leq \lfloor \hat{x}_s \rfloor$ and the second subproblem adds the constraint $x \geq \lceil \hat{x}_s \rceil$. The branching will create a branch tree

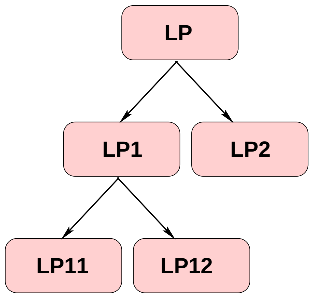


Figure 2.2: A branch tree is used for solving integer linear programming instances. The root node is the original problem instance with the integrality constraints dropped. Branching is done on integer variables that are not integer in the solution of the relaxed problem.

(see Figure 2.2) where the root node of the tree is the original problem, and the non-root nodes are subproblems with added constraints.

If there are many integer variables in the optimization problem, the branching tree will potentially grow huge, and more nodes will be created than the memory of the computer can hold. The branching can be limited by observing the solution of the relaxed problem; if the relaxed problem has an optimum that is worse than the best *integer* solution found so far, then branching from this node can be stopped because it has no potential to lead to an improved solution. Going deeper in the branching tree only adds constraints, and thereby make the optimum worse. If the branch-and-bound algorithm can remove nodes from the branch tree at the same rate as new nodes are added, the memory usage can be kept low. When there are no more nodes left in the branch tree, the algorithm is finished, and if a solution has been found it is optimal.

Using general methods, like integer linear programming, for solving the combinatorial problem instances in a compiler allows to draw upon general improvements and insights. The prize is that it can be hard sometimes to formulate the knowledge in the general way. Another advantage of using integer linear programming is that a mathematically precise description of the problem is generated as a side effect.

```
   R0 = 10
L: ...
   ...
   BDEC L, R0
```

Figure 2.3: The number of iterations is set before the loop begins.

2.9. Hardware support for loops

Some processors have specialized hardware support that makes it possible to execute loop code with very little or no overhead. This hardware makes it possible to have loops without an explicit loop counter. One way of supporting this in hardware is to implement a decrement-and-branch instruction (`BDEC`). The idea is that the number of loop iterations is loaded into a register before the loop begins. Every time the control reaches the end of the loop the `BDEC` instruction is executed, and if the value in the register reaches 0 the branch is not taken, see the example in Figure 2.3.

Taking the hardware support one step further is the special loop instruction (`LOOP`). The instruction `LOOP nrinst, nriter` will execute the next `nrinst` instructions `nriter` times. Some DSPs even have hardware support for modulo scheduling. One such DSP is the Texas Instruments C66x, which has the instructions `SPLOOP` and `SPKERNEL`; let us have a closer look at these instructions by inspecting the code of the example in Figure 2.4 (adapted from the C66x manual [Tex10]): In this example a sequence of words is copied from one place in memory to another. The code in the example starts with loading the number of iteration (8) to the special *inner loop count* register (`ILC`). Loading the ILC needs 4 cycles, so we add 3 empty cycles before the loop begins. `SPLOOP 1` denotes the beginning of the loop and sets the initiation interval to 1. Then the value to be copied is loaded; it is assumed that the address of the first word is in register `A1`, and the destination of the first word is in register `B0` at the entry to the example code. Then the loaded value is copied to the other register bank, this is needed because otherwise there would eventually be a

```
MVC 8, ILC    ;set the loop count register to 8
NOP 3         ;the delay of MVC is 3
SPLOOP 1      ;the loop starts here, the II is 1
LDW *A1++, A2 ;load source
NOP 4         ;load has 4 delay slots
MV.L1X A2, B2 ;transfer data
SPKERNEL 6,0 || STW B2, *B0++
              ;ends the loop and stores the value
```

Figure 2.4: An example with hardware support for software pipelining, from [Tex10].

conflict between the load and store in the loop body (i.e. they would use the same resource). In the last line of code the `SPKERNEL` denotes the end of the loop body and the `STW` instruction writes the loaded word to the destination. The arguments to `SPKERNEL` make it so that the code following the loop will not overlap with the epilog (the "6" means that the epilog is 6 cycles long).

The body of the loop consists of 4 execution packets which are loaded into a specialized buffer. This buffer has room for 14 execution packets; if the loop does not fit in the buffer then some other technique for software pipelining has to be used instead. Using the specialized hardware has several benefits: code size is reduced because we do not need to store prolog and epilog code, memory bandwidth (and energy use) is reduced since instructions do not need to be fetched every cycle, and we do not need explicit branch instructions, which frees up one functional unit in the loop.

2.10. Terminology

2.10.1. Optimality

When we talk about optimal code generation we mean optimal in the sense that the produced target code is optimal when it does all the operations included in the intermediate code. That is, we do not include any transformations on the intermediate code, we assume that all such transformations have been done already in previous phases of the compiler. Integrating all standard optimizations of the compiler in the optimization problem would be difficult. Finding the provably *truly* optimal code can be done with exhaustive search of all programs, and this is extremely expensive for anything but very simple machines (or subsets of instructions) [Mas87].

2.10.2. Basic blocks

A basic block is a block of code that contains no jump instructions and no other jump targets than the beginning of the block. I.e., when the flow of control enters the basic block all of the operations in the block are executed exactly once.

2.10.3. Mathematical notation

To save space we will sometimes pack multiple sum-indices in the same sum-character. We write:

$$\sum_{\substack{x\in A\\ y\in B\\ z\in C}}$$

for the triple sum

$$\sum_{x\in A}\sum_{y\in B}\sum_{z\in C}$$

For modulo calculations we write

$$x \equiv a \pmod{b}$$

meaning that $x = kb + a$, for some integer k. If a is the smallest non-negative integer where this relation is true this is sometimes written $x\%b = a$.

Chapter 3.

Integrated code generation for basic blocks

This chapter describes two methods for integrated code generation for basic blocks. The first method is exact and based on integer linear programming. The second method is a heuristic based on genetic algorithms. These two methods are compared experimentally.

3.1. Introduction

The *back-end* of a compiler transforms intermediate code, produced by the front-end, into executable code. This transformation is usually performed in at least three major steps: *instruction selection* selects which instructions to use, *instruction scheduling* maps each instruction to a time slot and *register allocation* selects in which registers a value is to be stored. Furthermore the back-end can also contain various optimization phases, e.g. modulo scheduling for loops where the goal is to overlap iterations of the loop and thereby increase the throughput. In this chapter we will focus on the basic block case, and in the next chapter we will do modulo scheduling.

3.1.1. Retargetable code generation and Optimist

Creating a compiler is not an easy task, it is generally very time consuming and expensive. Hence, it would be good to have compilers that can be targeted to different architectures in a simple way. One approach to creating such compilers is called *retargetable compiling*

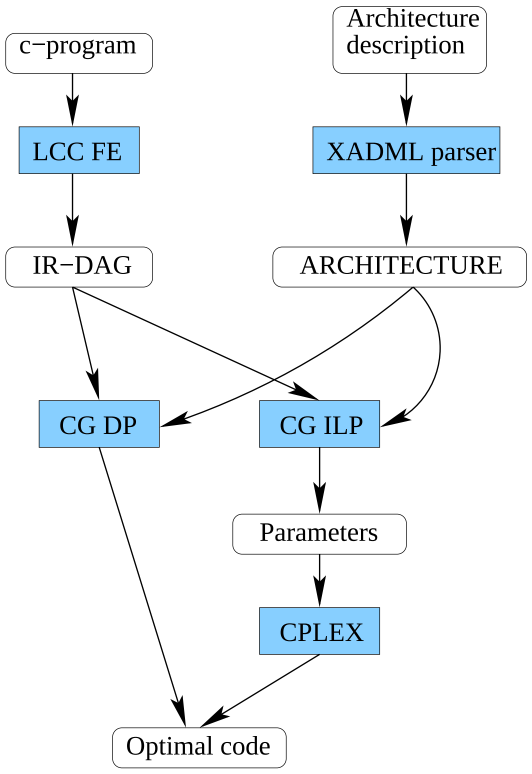


Figure 3.1: Overview of the Optimist compiler.

where the basic idea is to supply an architecture description to the compiler (or to a compiler generator, which creates a compiler for the described architecture). Assuming that the architecture description language is general enough, the task of creating a compiler for a certain architecture is then as simple as describing the architecture in this language.

The implementations in this thesis have their roots in the retargetable Optimist framework [KB05]. Optimist uses a front-end that is based on a modified LCC (Little C Compiler) [FH95]. The front-end generates Boost [Boo] graphs that are used, together with a processor description, as input to a pluggable code generator (see Figure 3.1).

Already existing integrated code generators in the Optimist framework are based on:

- Dynamic programming (DP), where the optimal solution is searched for in the solution space by intelligent enumeration [KB06, Bed06].

- Integer linear programming, in which parameters are generated that can be passed on, together with a mathematical model, to an integer linear programming solver such as *CPLEX* [ILO06], *GLPK* [Mak] or Gurobi [Gur10]. This method was limited to single cluster architectures [BK06b]. This model has been improved and generalized in the work described in this thesis.

- A simple heuristic, which is basically the DP method modified to not be exhaustive with regard to scheduling [KB06]. In this thesis we add a heuristic based on genetic algorithms.

The architecture description language of Optimist is called *Extended architecture description markup language* (xADML) [Bed06]. This language is versatile enough to describe clustered, pipelined, irregular and asymmetric VLIW architectures.

3.2. Integer linear programming formulation

For optimal code generation for basic blocks we use an integer linear programming formulation. In this section we will introduce all parameters, variables and constraints that are used by the integer linear programming solver to generate a schedule with minimal execution time. This model integrates instruction selection (including cluster assignment), instruction scheduling and register allocation. Also, the integer linear programming model is natural to extend to modulo scheduling, as we show in Chapter 4. The integer linear programming model presented here is based on a series of models previously published in [BK06b, ESK08, EK09].

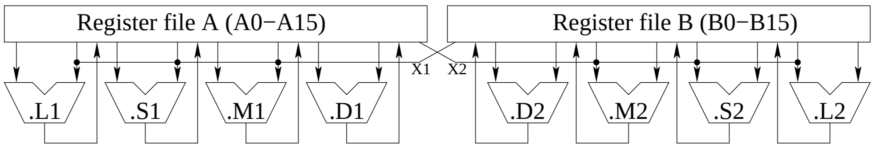


Figure 3.2: The Texas Instruments TI-C62x processor has two register banks with 4 functional units each [Tex00]. The crosspaths X1 and X2 are used for limited transfers of values from one cluster to the other.

3.2.1. Optimization parameters and variables

In this section we introduce the parameters and variables that are used in the integer linear programming model.

Data flow graph

A basic block is modeled as a directed acyclic graph (DAG) $G = (V, E)$, where $E = E_1 \cup E_2 \cup E_m$. The set V contains intermediate representation (IR) nodes, the sets $E_1, E_2 \subset V \times V$ represent dependences between operations and their first and second operand respectively. Other precedence constraints are modeled with the set $E_m \subset V \times V$. The integer parameter Op_i describes operators of the IR-nodes $i \in V$.

Instruction set

The instructions of the target machine are modeled by the set $P = P_1 \cup P_{2+} \cup P_0$ of patterns. P_1 is the set of *singletons*, which only cover one IR node. The set P_{2+} contain *composites*, which cover multiple IR nodes (used e.g. for multiply-and-add which covers a multiplication immediately followed by an addition). And the set P_0 consists of patterns for *non-issue* instructions which are needed when there are IR nodes in V that do not have to be covered by an instruction, e.g. an IR node representing a constant value that needs not be loaded into a register. The IR is low level enough so that all patterns model exactly

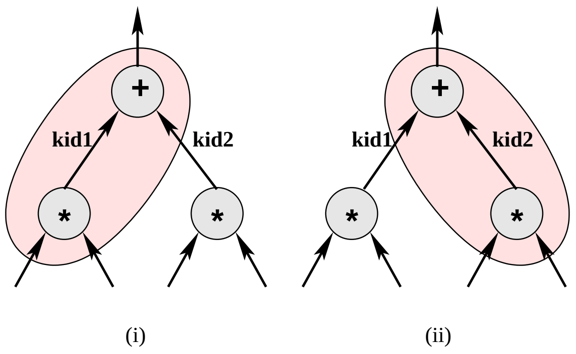


Figure 3.3: A multiply-and-add instruction can cover the addition and the left multiplication (i), or it can cover the addition and the right multiplication (ii).

one (or zero in the case of P_0) instructions of the target machine. When we use the term *pattern* we mean a pair consisting of one instruction and a set of IR-nodes that the instruction can implement. I.e., an instruction can be paired with different sets of IR-nodes and a set of IR-nodes can be paired with more than one instruction.

Example 1. *On the TI-C62x DSP processor (see Figure 3.2) a single addition can be done with any of twelve different instructions (not counting the multiply-and-add instructions):* `ADD.L1`*,* `ADD.L2`*,* `ADD.S1`*,* `ADD.S2`*,* `ADD.D1`*,* `ADD.D2`*,* `ADD.L1X`*,* `ADD.L2X`*,* `ADD.S1X`*,* `ADD.S2X`*,* `ADD.D1X` *or* `ADD.D2X`*.*

For each pattern $p \in P_{2+} \cup P_1$ we have a set $B_p = \{1, \ldots, n_p\}$ of generic nodes for the pattern. For composites we have $n_p > 1$ and for singletons $n_p = 1$. For composite patterns $p \in P_{2+}$ we also have $EP_p \subset B_p \times B_p$, the set of edges between the generic pattern nodes. Each node $k \in B_p$ of the pattern $p \in P_{2+} \cup P_1$ has an associated operator number $OP_{p,k}$ which relates to operators of IR nodes. Also, each $p \in P$ has a *latency* L_p, meaning that if p is scheduled at time slot t the result of p is available at time slot $t + L_p$.

Example 2. *The multiply-and-add instruction is a composite pattern and has $n_p = 2$ (one node for the multiplication and another for the add). When performing instruction selection in the DAG the multiply-and-add covers two nodes. In Figure 3.3 there are two different ways to use multiply-and-add.*

Resources and register sets

We model the resources of the target machine with the set $\mathcal{F}$ and the register banks with the set $\mathcal{RS}$. The binary parameter $U_{p,f,o}$ is 1 iff the instruction with pattern $p \in P$ uses the resource $f \in \mathcal{F}$ at time step o relative to the issue time. Note that this allows for multiblock [KBE07] and irregular reservation tables [Rau94]. $\mathcal{R}_r$ is a parameter describing the number of registers in the register bank $r \in \mathcal{RS}$. The *issue width* is modeled by ω, i.e. the maximum number of instructions that may be issued at any time slot.

For modeling transfers between register banks we do not use regular instructions (note that transfers, like spill instructions, do not cover nodes in the DAG). Instead we let the integer parameter $LX_{r,s}$ denote the latency of a transfer from $r \in \mathcal{RS}$ to $s \in \mathcal{RS}$. If no such transfer instruction exists we set $LX_{r,s} = \infty$. And for resource usage, the binary parameter $UX_{r,s,f}$ is 1 iff a transfer from $r \in \mathcal{RS}$ to $s \in \mathcal{RS}$ uses resource $f \in \mathcal{F}$. See Figure 3.2 for an illustration of a clustered architecture.

Note that we can also integrate spilling into the formulation by adding a virtual register file to $\mathcal{RS}$ corresponding to the memory, and then have transfer instructions to and from this register file corresponding to stores and loads, see Figure 3.4.

Lastly, we have the sets $PD_r, PS1_r, PS2_r \subset P$ which, for all $r \in \mathcal{RS}$, contain the pattern $p \in P$ iff p stores its result in r, takes its first operand from r or takes its second operand from r, respectively.

Solution variables

The parameter $t_{\max}$ gives the last time slot on which an instruction may be scheduled. We also define the set $T = \{0, 1, 2, \ldots, t_{\max}\}$,

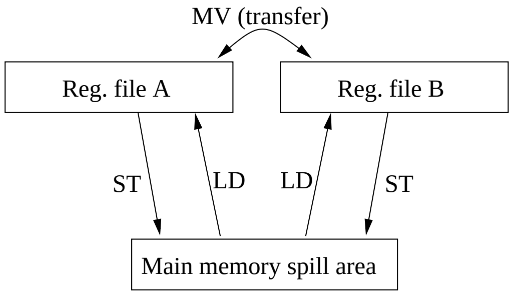


Figure 3.4: Spilling can be modeled by transfers. Transfers out to the spill area in memory correspond to store instructions and transfers into registers again are load instructions.

i.e. the set of time slots on which an instruction may be scheduled. For the acyclic case $t_{\max}$ is incremented until a solution is found.

So far we have only mentioned the parameters that describe the optimization problem. Now we introduce the solution variables which define the solution space. We have the following binary solution variables:

- $c_{i,p,k,t}$, which is 1 iff IR node $i \in V$ is covered by $k \in B_p$, where $p \in P$, issued at time $t \in T$.
- $w_{i,j,p,t,k,l}$, which is 1 iff the DAG edge $(i, j) \in E_1 \cup E_2$ is covered at time $t \in T$ by the pattern edge $(k, l) \in EP_p$ where $p \in P_{2+}$ is a composite pattern.
- $s_{p,t}$, which is 1 iff the instruction with pattern $p \in P_{2+}$ is issued at time $t \in T$.
- $x_{i,r,s,t}$, which is 1 iff the result from IR node $i \in V$ is transfered from $r \in \mathcal{RS}$ to $s \in \mathcal{RS}$ at time $t \in T$.
- $r_{rr,i,t}$, which is 1 iff the value corresponding to the IR node $i \in V$ is available in register bank $rr \in \mathcal{RS}$ at time slot $t \in T$.

We also have the following integer solution variable:

- τ is the first clock cycle on which all latencies of executed instructions have expired.

3.2.2. Removing impossible schedule slots

We can significantly reduce the number of variables in the model by performing soonest-latest analysis on the nodes of the graph. Let $L_{\min}(i)$ be 0 if the node $i \in V$ may be covered by a composite pattern, and the lowest latency of any instruction $p \in P_1$ that may cover the node $i \in V$ otherwise. Let $\text{pre}(i) = \{j : (j,i) \in E\}$ and $\text{succ}(i) = \{j : (i,j) \in E\}$. We can recursively calculate the soonest and latest time slot on which node i may be scheduled:

$$\mathit{soonest}'(i) = \begin{cases} 0 & \text{, if } |\text{pre}(i)| = 0 \\ \max_{j \in \text{pre}(i)}\{\mathit{soonest}'(j) + L_{\min}(j)\} & \text{, otherwise} \end{cases} \tag{3.1}$$

$$\mathit{latest}'(i) = \begin{cases} t_{\max} & \text{, if } |\text{succ}(i)| = 0 \\ \max_{j \in \text{succ}(i)}\{\mathit{latest}'(j) - L_{\min}(i)\} & \text{, otherwise} \end{cases} \tag{3.2}$$

$$T_i = \{\mathit{soonest}'(i), \ldots, \mathit{latest}'(i)\} \tag{3.3}$$

We can also remove all the variables in c where no node in the pattern $p \in P$ has an operator number matching i. We can view the matrix c of variables as a sparse matrix; the constraints dealing with c must be written to take this into account; in the following mathematical presentation $c_{i,p,k,t}$ is taken to be 0 if $t \notin T_i$ for simplicity of presentation.

3.2.3. Optimization constraints

Optimization objective

The objective of the integer linear program is to minimize the execution time:

$$\min \tau \tag{3.4}$$

The execution time is the latest time slot where any instruction terminates. For efficiency we only need to check for execution times for instructions covering an IR node with out-degree 0, let $V_{\text{root}} = \{i \in V : \nexists j \in V,\ (i,j) \in E\}$:

$$\forall i \in V_{\text{root}},\ \forall p \in P,\ \forall k \in B_p,\ \forall t \in T, \quad c_{i,p,k,t}(t + L_p) \leq \tau \tag{3.5}$$

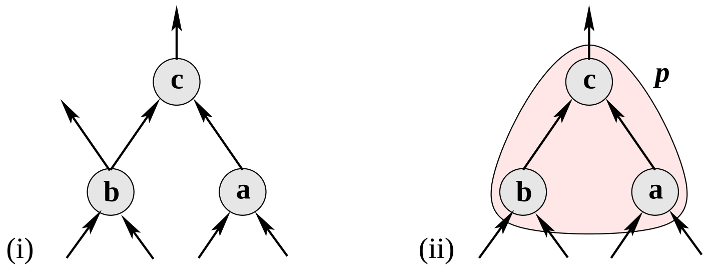


Figure 3.5: (i) Pattern p can not cover the set of nodes since there is another outgoing edge from b, (ii) p covers nodes a, b, c.

Node and edge covering

Exactly one instruction must cover each IR node:

$$\forall i \in V, \quad \sum_{\substack{p \in P \\ k \in B_p \\ t \in T}} c_{i,p,k,t} = 1 \tag{3.6}$$

Equation 3.7 sets $s_{p,t} = 1$ iff the composite pattern $p \in P_{2+}$ is used at time $t \in T$. This equation also guarantees that either all or none of the generic nodes $k \in B_p$ are used at a time slot:

$$\forall p \in P_{2+},\ \forall t \in T,\ \forall k \in B_p, \quad \sum_{i \in V} c_{i,p,k,t} = s_{p,t} \tag{3.7}$$

An edge within a composite pattern may only be used in the covering if there is a corresponding edge (i, j) in the DAG and both i and j are covered by the pattern, see Figure 3.5:

$$\begin{gathered}\forall (i,j) \in E_1 \cup E_2,\ \forall p \in P_{2+},\ \forall t \in T,\ \forall (k,l) \in EP_p, \\ 2w_{i,j,p,t,k,l} \le c_{i,p,k,t} + c_{j,p,l,t}\end{gathered} \tag{3.8}$$

If a generic pattern node covers an IR node, the generic pattern node and the IR node must have the same operator number:

$$\forall i \in V,\ \forall p \in P,\ \forall k \in B_p,\ \forall t \in T, \quad c_{i,p,k,t}(Op_i - OP_{p,k}) = 0 \tag{3.9}$$

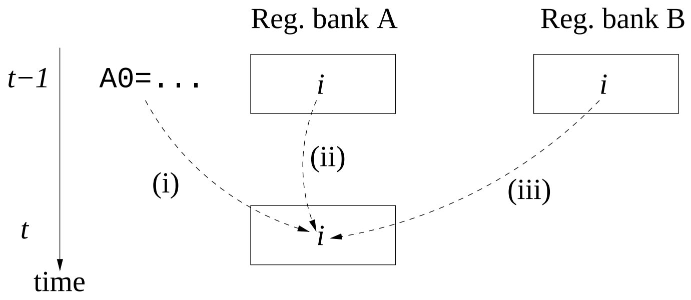


Figure 3.6: A value may be live in a register bank A if: (i) it was put there by an instruction, (ii) it was live in register bank A at the previous time step, and (iii) the value was transferred there by an explicit transfer instruction.

Register values

A value may only be present in a register bank if: it was just put there by an instruction, it was available there in the previous time step, or just transfered to there from another register bank (see visualization in Figure 3.6):

$$\forall rr \in \mathcal{RS},\ \forall i \in V,\ \forall t \in T,$$
$$r_{rr,i,t} \leq \sum_{\substack{p \in PD_{rr} \cap P \\ k \in B_p}} c_{i,p,k,t-L_p} + r_{rr,i,t-1} + \sum_{rs \in \mathcal{RS}} \left(x_{i,rs,rr,t-LX_{rs,rr}} \right) \tag{3.10}$$

The operand to an instruction must be available in the correct register bank when it is used. A limitation of this formulation is that composite patterns must read all operands and store its result in the same register bank:

$$\forall (i,j) \in E_1 \cup E_2,\ \forall t \in T,\ \forall rr \in \mathcal{RS},$$
$$r_{rr,i,t} \geq \sum_{\substack{p \in PD_{rr} \cap P_{2+} \\ k \in B_p}} \left(c_{j,p,k,t} - \sum_{(k,l) \in EP_p} w_{i,j,p,t,k,l} \right) \tag{3.11}$$

Internal values in a composite pattern must not be put into a register (e.g. the product in a multiply-and-add instruction):

$$\forall rr \in \mathcal{RS}, tp \in T, tr \in T, p \in P_{2+},\ \forall (k,l) \in EP_p,\ \forall (i,j) \in E_1 \cup E_2,$$
$$r_{rr,i,tr} \leq 1 - w_{i,j,p,tp,k,l} \tag{3.12}$$

If they exist, the first operand (Equation 3.13) and the second operand (Equation 3.14) must be available when they are used:

$$\forall (i,j) \in E_1,\ \forall t \in T,\ \forall rr \in \mathcal{RS}, \quad r_{rr,i,t} \geq \sum_{\substack{p \in PS1_{rr} \cap P_1 \\ k \in B_p}} c_{j,p,k,t} \tag{3.13}$$

$$\forall (i,j) \in E_2,\ \forall t \in T,\ \forall rr \in \mathcal{RS}, \quad r_{rr,i,t} \geq \sum_{\substack{p \in PS2_{rr} \cap P_1 \\ k \in B_p}} c_{j,p,k,t} \tag{3.14}$$

Transfers may only occur if the source value is available:

$$\forall i \in V,\ \forall t \in T,\ \forall rr \in \mathcal{RS}, \quad r_{rr,i,t} \geq \sum_{rq \in \mathcal{RS}} x_{i,rr,rq,t} \tag{3.15}$$

Non-dataflow dependences

Equation 3.16 ensures that non-dataflow dependences are fulfilled. For each point in time t it must not happen that a successor is scheduled at time t or earlier if the result of the predecessor only becomes available at time $t+1$ or later:

$$\forall (i,j) \in E_m,\ \forall t \in T, \quad \sum_{p \in P} \sum_{t_j=0}^{t} c_{j,p,1,t_j} + \sum_{p \in P} \sum_{t_i = t - L_p + 1}^{t_{\max}} c_{i,p,1,t_i} \leq 1 \tag{3.16}$$

Writing the precedence constraints like this is a common technique when using integer linear programming for scheduling. This formulation leads to a tight polytope, see for instance [GE93].

Resources

We must not exceed the number of available registers in a register bank at any time:

$$\forall t \in T,\ \forall rr \in \mathcal{RS},\quad \sum_{i \in V} r_{rr,i,t} \leq R_{rr} \tag{3.17}$$

Condition 3.18 ensures that no resource is used more than once at each time slot:

$$\forall t \in T,\ \forall f \in \mathcal{F},$$
$$\sum_{\substack{p \in P_{2+} \\ o \in \mathbb{N}}} U_{p,f,o} s_{p,t-o} + \sum_{\substack{p \in P_1 \\ i \in V \\ k \in B_p}} U_{p,f,o} c_{i,p,k,t-o} + \sum_{\substack{i \in V \\ (rr,rq) \in (\mathcal{RS} \times \mathcal{RS})}} UX_{rr,rq,f}\, x_{i,rr,rq,t} \leq 1 \tag{3.18}$$

And, lastly, Condition 3.19 guarantees that we never exceed the issue width:

$$\forall t \in T,\quad \sum_{p \in P_{2+}} s_{p,t} + \sum_{\substack{p \in P_1 \\ i \in V \\ k \in B_p}} c_{i,p,k,t} + \sum_{\substack{i \in V \\ (rr,rq) \in (\mathcal{RS} \times \mathcal{RS})}} x_{i,rr,rq,t} \leq \omega \tag{3.19}$$

This ends the presentation of the integer linear programming formulation. The next section introduces a heuristic based on genetic algorithms. And in Section 3.4 the two approaches are compared experimentally.

3.3. The genetic algorithm

The previous section presented an algorithm for optimal integrated code generation. Optimal solutions are of course preferred, but for large problem instances the time required to solve the integer linear program to optimality may be too long. For these cases we need a heuristic method. Kessler and Bednarski present a variant of list scheduling [KB06] in which a search of the solution space is performed

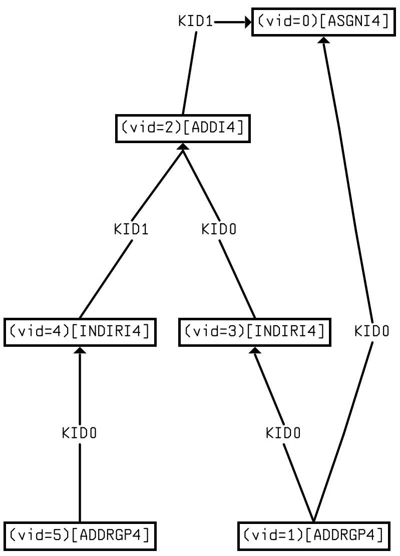


Figure 3.7: A compiler generated DAG of the basic block representing the calculation $a = a + b$;. The vid attribute is the node identifier for each IR node.

for one order of the IR nodes. The search is exhaustive with regard to instruction selection and transfers but not exhaustive with regard to scheduling. We call this heuristic HS1. The HS1 heuristic is fast for most basic blocks but often does not achieve great results. We need a better heuristic and bring our attention to genetic algorithms.

A genetic algorithm [Gol89] is a heuristic method which may be used to search for good solutions to optimization problems with large solution spaces. The idea is to mimic the process of natural selection, where stronger individuals have better chances to survive and spread their genes.

The creation of the initial population works similarly to the HS1 heuristic; there is a fixed order in which the IR nodes are considered and for each IR node we choose a random instruction that can cover the node and also, with a certain probability, a transfer instruction for one of the alive values at the reference time (the latest time slot on which an instruction is scheduled). The selected instructions are appended to the partial schedule of already scheduled nodes. Every new instruction that is appended is scheduled at the first time slot larger than or equal to the reference time of the partial schedule, such that all dependences and resource constraints are respected. (This is called in-order compaction, see [KBE07] for a detailed discussion.)

From each individual in the population we then extract the following genes:

- The order in which the IR nodes were considered.
- The transfer instructions that were selected, if any, when each IR node was considered for instruction selection and scheduling.
- The instruction that was selected to cover each IR node (or group of IR nodes).

Example 3. *For the DAG in Figure 3.7, which depicts the IR DAG for the basic block consisting of the calculation $a = a + b$; we have a valid schedule:*

```
LDW .D1 _a, A15 || LDW .D2 _b, B15
NOP ; Latency of a load is 5
NOP
NOP
NOP
ADD .D1X A15, B15, A15
MV .L1 _a, A15
```

with a TI-C62x [Tex00] like architecture. From this schedule and the DAG we can extract the node order $\{1, 5, 3, 4, 2, 0\}$ *(nodes 1 and 5 represent symbols and do not need to be covered). To this node order we have the instruction priority map* $\{1 \rightarrow \text{NULL}, 5 \rightarrow \text{NULL}, 3 \rightarrow \text{LDW.D1}, 4 \rightarrow \text{LDW.D2}, 2 \rightarrow \text{ADD.D1X}, 0 \rightarrow \text{MV.L1}\}$. *And the schedule has no explicit transfers, so the transfer map is empty.*

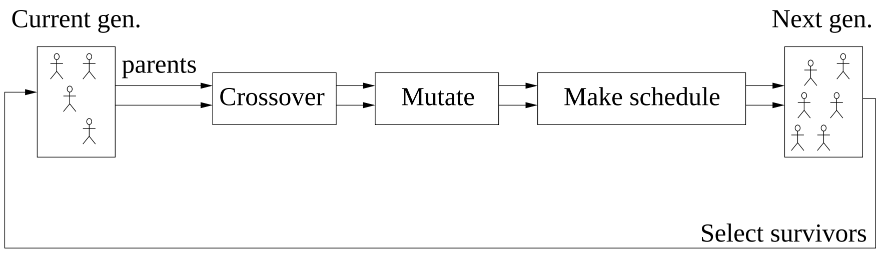


Figure 3.8: The components of the genetic algorithm.

3.3.1. Evolution operations

All that is needed to perform the evolution of the individuals are: a fitness calculation method for comparing the quality of the individuals, a selection method for choosing parents, a crossover operation which takes two parents and creates two children, methods for mutation of individual genes and a survival method for choosing which individuals survive into the next generation. Figure 3.8 shows an overview of the components of the genetic algorithm.

Fitness

The fitness of an individual is the execution time, i.e. the time slot when all scheduled instructions have terminated (cf. τ in the previous section).

Selection

For the selection of parents we use *binary tournament*, in which four individuals are selected randomly and the one with best fitness of the first two is selected as the first parent and the best one of the other two for the other parent.

Crossover

The crossover operation takes two parent individuals and uses their genes to create two children. The children are created by first finding a crossover point on which the parents *match*. Consider two parents, p_1 and p_2, and partial schedules for the first n IR nodes that are selected, with the instructions from the parents instruction priority and transfer priority maps. We say that n is a *matching point* of p_1 and p_2 if the two partial schedules have the same pipeline status at the reference time (the last time slot for which an instruction was scheduled), i.e. the partial schedules have the same pending latencies for the same values and have the same resource usage for the reference time and future time slots. Once a matching point is found, doing the crossover is straight forward; we simply concatenate the first n genes of p_1 with the remaining genes of p_2 and vice versa. Now these two new individuals generate valid schedules with high probability. If no matching point is found we select new parents for the crossover. If there are more than one matching point, one of them is selected randomly for the crossover.

Mutation

Next, when children have been created they can be mutated in three ways:

1. Change the positions of two nodes in the node order of the individual. The two nodes must not have any dependence between them.

2. Change the instruction priority of an element in the instruction priority map.

3. Remove a transfer from the transfer priority map.

Survival

Selecting which individuals survive into the next generation is controlled by two parameters to the algorithm.

1. We can either allow or disallow individuals to survive into the next generation, and

2. selecting survivors may be done by truncation, where the best (smallest execution time) survives, or by the roulette wheel method, in which individual i, with execution time τ_i, survives with a probability proportional to $\tau_w - \tau_i$, where τ_w is the execution time of the worst individual.

We have empirically found the roulette wheel selection method to give the best results and use it for all the following tests.

3.3.2. Parallelization of the algorithm

We have implemented the genetic algorithm in the Optimist framework and found by profiling the algorithm that the largest part of the execution time is spent in creating individuals from gene information. The time required for the crossover and mutation phase is almost negligible. We note that the creation of the individuals from genes is easily parallelizable with the master-slave paradigm. This is implemented by using one thread per individual in the population which performs the creation of the schedules from its genes, see Figure 3.9 for a code listing showing how it can be done. The synchronization required for the threads is very cheap, and we achieve good speedup as can be seen in Table 3.1. The tests are run on a machine with two cores and the average speedup is close to 2. The reason why speedups larger than 2 occur is that the parallel and non-parallel algorithm do not generate random numbers in the same way, i.e., they do not run the exact same calculations and do not achieve the exact same final solution. The price we pay for the parallelization is a somewhat increased memory usage.

3.4. Experimental evaluation

The experimental evaluations were performed either on an Athlon X2 6000+, 64-bit, dual core, 3 GHz processor with 4 GB RAM, or on a Intel Core i7 950, quad core, 3.07 GHz with 12 GB RAM. The genetic algorithm implementation was compiled with gcc at optimization level

```
class thread_create_schedule_arg{
public:
  Individual ** ind;
  Population * pop;
  int nr;
  sem_t * start, * done;
};

void Population::create_schedules(individuals_t& _individuals)
{
  /* Create schedules in parallel */
  for(int i = 0; i < _individuals.size(); i++){
    /* pack the arguments in arg[i] (known by slave i) */
    arg[i].ind = &(_individuals[i]);
    arg[i].pop = this;
    /* let slave i begin */
    sem_post(arg[i].start);
  }
  /* Wait for all slaves to finish */
  for(int i=0; i < _individuals.size(); i++){
    sem_wait(arg[i].done);
  }
}

/* The slave runs the following */
void * thread_create_schedule( void * arg )
{
  int nr =  ((thread_create_schedule_arg*)arg)->nr;
  std::cout << "Thread " << nr << " starting" << std::endl;
  while(1){
    sem_wait(((thread_create_schedule_arg*)arg)->start);
    Individual **ind = ((thread_create_schedule_arg*)arg)->ind;
    Population *pop = ((thread_create_schedule_arg*)arg)->pop;
    pop->create_schedule(*ind, false, false, true)
    sem_post(((thread_create_schedule_arg*)arg)->done);
  }
}
```

Figure 3.9: Code listing for the parallelization of the genetic algorithm.

Popsize	t_s(s)	t_p(s)	Speedup
8	51.9	27.9	1.86
24	182.0	85.2	2.14
48	351.5	165.5	2.12
96	644.8	349.1	1.85

Table 3.1: The table shows execution times for the serial algorithm (t_s), as well as for the algorithm using pthreads (t_p). The tests are run on a machine with two cores (Athlon X2).

`-O2`. The integer linear programming solver is Gurobi 4.0 with AMPL 10.1. All shown execution times are measured in wall clock time.

The target architecture that we use for all experiments in this chapter is a slight variation of the TI-C62x (we have added division instructions which are not supported by the hardware). This is a VLIW architecture with dual register banks and an issue width of 8. Each bank has 16 registers and 4 functional units: L, M, S and D. There are also two cross cluster paths: X1 and X2. See Figure 3.2 for an illustration.

3.4.1. Convergence behavior of the genetic algorithm

The parameters to the GA can be configured in a large number of ways. Here we will only look at 4 different configurations and pick the one that looks most promising for further tests. The configurations are:

- GA0: All mutation probabilities 50%, 50 individuals and no parents survive.
- GA1: All mutation probabilities 75%, 10 individuals and parents survive.
- GA2: All mutation probabilities 25%, 100 individuals and no parents survive.
- GA3: All mutation probabilities 50%, 50 individuals and parents survive.

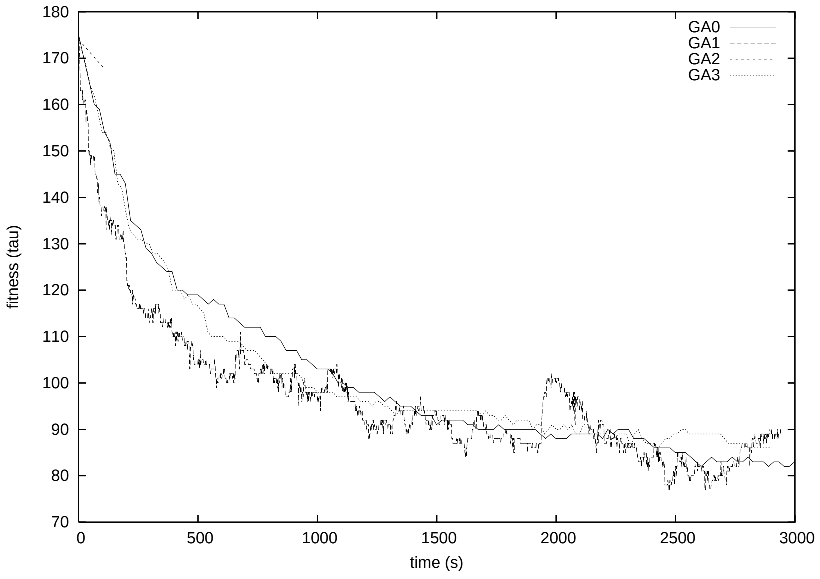


Figure 3.10: A plot of the progress of the best individual at each generation for 4 different parameter configurations. The best result, $\tau = 77$, is found with GA1 — 10 individuals, 75% mutation probability with the parents survive strategy. For comparison, running a random search 34'000 times (corresponding to the same time budget as for the GA algorithms) only gives a schedule with fitness 168. (Host machine: Athlon X2.)

The basic block that we use for evaluating the parameter configurations is a part of the inverse discrete cosine transform calculation in Mediabench's [LPMS97] mpeg2 decoding and encoding program. It is one of our largest and hence very demanding test cases and contains 138 IR-nodes and 265 edges (data flow and memory dependences).

Figure 3.10 shows the progress of the best individual in each generation for the four parameter configurations. The first thing we note is that the test with the largest number of individuals, GA2, only runs for a few seconds. The reason for this is that it runs out of memory. We also observe that GA1, the test with 10 individuals and 75% mutation probabilities achieves the best result, $\tau = 77$. The GA1 progress is more bumpy than the other ones, the reason is twofold; a low number of individuals means that we can create a larger number of generations in the given time and the high mutation probability means that the difference between individuals in one generation and the next is larger.

A more stable progress is achieved by the GA0 parameter set where the best individual rarely gets worse from one generation to the next. If we compare GA0 to GA1 we would say that GA1 is risky and aggressive, while GA0 is safe. Another interesting observation is that GA0, which finds $\tau = 81$, is better than GA3, which finds $\tau = 85$. While we can not conclude anything from this one test, this supports the idea that if parents do not survive into the next generation, the individuals in a generation are less likely to be similar to each other and thus a larger part of the solution space is explored.

For the rest of the evaluation we use GA0 and GA1 since they look most promising.

3.4.2. Comparing optimal and heuristic results

We have compared the heuristics HS1 and GA to the optimal integer linear programming method for integrated code generation on 81 basic blocks from the Mediabench [LPMS97] benchmark suite. The basic blocks were selected by taking all blocks with 25 or more IR nodes from the mpeg2 and jpeg encoding and decoding programs. The largest basic block in this set has 191 IR nodes.

The result of the evaluation is summarized in Figure 3.11, and details are found in Tables 3.3, 3.4 and 3.5. The first column in the tables, $|G|$, shows the size of the basic block, the second column, BB, is a number which identifies a basic block in a source file. The next 6 columns show the results of: the HS1 heuristic (see Section 3.3), the GA heuristic with the parameter sets GA0 and GA1 (see Section 3.4.1) and the integer linear program execution (see Section 3.2).

The execution time ($t(s)$) is measured in seconds, and as expected the HS1 heuristic is fast for most cases. The execution times for GA0 and GA1 are approximately 4 minutes for all tests. The reason why not all are identical is that we stopped the execution after 1200 CPU seconds, i.e. the sum of execution times on all 4 cores of the host processor, or after the population had evolved for 20000 generations. All the integer linear programming instances were solved in less than 30 seconds, and in many cases this method is faster than the HS1 heuristic. This is not completely fair since the integer linear programming solver can utilize all of the cores and the multithreading of the host processor, while the heuristic is completely sequential. Still, from these results we can conclude that the dynamic programming based approach does not stand a chance against the integer linear programming approach in terms of solution speed.

A summary of the results is shown in Figure 3.11; all DAGs are solved by the integer linear programming method and the produced results are optimal. The largest basic block that is solved by the integer linear programming method contains 191 IR nodes. After presolve 63272 variables and 63783 constraints remain, and the solution time is 30 seconds[1]. The time to optimally solve an instance does not only depend on the size of the DAG, but also on other characteristics of the problem, such as the amount of instruction level parallelism that is possible.

[1]The results of the integer linear programming model have significantly improved since our first publication [ESK08]. At that time many of the instances could not be solved at all. The reason behind the improvements is that we have improved the model, we use a new host machine, and we use Gurobi 4.0 instead of CPLEX 10.2.

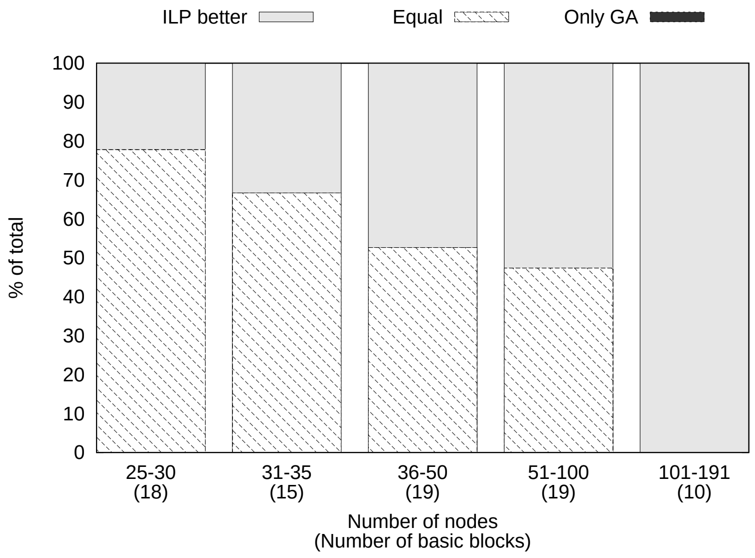


Figure 3.11: Stacked bar chart showing a summary of the comparison between the integer linear programming and the genetic algorithm for basic blocks: *ILP better* means that integer linear programming produces a schedule that is shorter than the one that GA produces, *Equal* means the schedules by ILP and GA have the same length, and *Only GA* means that GA finds a solution but integer linear programming fails to do so (this was common until recently). The integer linear programming method always produces an optimal result if it terminates. (Host machine: Intel i7, 12 GB RAM.)

Δ_{opt}	GA0	GA1
0	43	39
1	10	10
2	1	3
3	4	0
4	1	3
5	1	0
6	3	6
7	1	2
8	2	1
9	0	0
≥ 10	15	17

Table 3.2: Summary of results for cases where the optimal solution is known. The column GA0 and GA1 shows the number of basic blocks for which the genetic algorithm finds a solution Δ_{opt} clock cycles worse than the optimal.

Table 3.2 summarizes the results for the cases where the optimum is known. We note that GA0 and GA1 finds the optimal solution in around 50% of the cases. On average GA0 is 8.0 cycles from the optimum and GA1 is 8.2 cycles from optimum. But it should also be noted that the deviation is large, for the largest instance the GA0 is worse by 100 cycles.

		HS1		GA0	GA1	ILP	
$\|G\|$	BB	t(s)	τ	τ	τ	t(s)	τ
idct — inverse discrete cosine transform							
27	01	1	19	15	15	1	15
34	14	1	31	23	23	1	23
47	00	64	56	27	27	2	27
120	04	292	146	84	82	13	32
138	17	263	166	109	94	26	35
spatscal — spatial prediction							
35	33	1	60	25	25	2	24
44	51	1	57	33	33	1	29
46	71	1	57	40	40	2	39
predict — motion compensated prediction							
25	228	2	26	19	19	1	19
27	189	3	31	19	19	1	19
30	136	1	58	53	53	1	13
30	283	5	41	18	19	1	18
30	50	1	43	42	42	1	16
34	106	1	50	48	48	1	17
35	93	1	51	40	40	1	18
35	94	1	51	40	40	1	18
36	107	1	50	44	44	1	17
36	265	18	45	22	21	1	21
45	141	4	50	27	27	1	25
quantize — quantization							
26	120	1	38	20	20	1	19
26	145	1	37	19	19	1	18
30	11	1	38	25	25	1	25
31	29	1	42	27	27	1	26
transfrm — forward/inverse transform							
25	122	1	35	21	21	1	21
26	160	1	36	15	15	1	15
36	103	1	47	26	26	1	20
36	43	1	47	26	26	1	20
37	170	1	48	31	31	2	31

Table 3.3: Experimental results for the HS1, GA0, GA1 and ILP methods of code generation. In the t columns we find the execution time of the code generation and in the τ columns we see the execution time of the generated schedule. The basic blocks are from the mpeg2 program.

		HS1		GA0	GA1	ILP	
$\|G\|$	BB	t(s)	τ	τ	τ	t(s)	τ
jcsample — downsampling							
26	94	1	41	19	19	1	19
31	91	1	42	29	29	1	29
34	143	1	40	29	29	1	29
58	137	1	89	39	42	2	38
84	115	30	135	43	48	4	40
85	133	1	142	72	72	5	72
109	111	27	186	74	49	7	67
jdcolor — colorspace conversion							
30	34	1	46	22	22	1	22
32	83	1	47	23	23	1	23
36	33	1	62	39	39	1	39
38	30	1	59	32	32	2	32
38	82	1	63	40	40	2	40
45	79	1	71	32	32	2	32
jdmerge — colorspace conversion							
39	63	1	69	30	30	1	30
53	89	1	89	37	37	2	37
55	110	1	100	41	41	2	41
55	78	1	100	41	41	2	41
62	66	1	103	41	41	3	41
62	92	1	103	42	42	3	41
jdsample — upsampling							
26	100	1	34	22	22	1	22
28	39	1	45	28	28	1	28
28	79	1	45	28	28	1	28
30	95	1	50	30	30	1	30
33	130	1	43	20	20	1	20
47	162	1	68	34	34	2	34
52	125	1	78	23	25	2	23
jfdctfst — forward discrete cosine transform							
60	06	1	70	39	39	2	39
60	20	1	70	39	39	2	39
76	02	2	87	40	43	13	37
76	16	2	87	40	43	13	37

Table 3.4: Experimental results for the HS1, GA0, GA1 and ILP methods of code generation. In the t columns we find the execution time of the code generation and in the τ columns we see the execution time of the generated schedule. The basic blocks are from the jpeg program (part 1).

		HS1		GA0	GA1	ILP	
$\|G\|$	BB	t(s)	τ	τ	τ	t(s)	τ
jfdctint — forward discrete cosine transform							
74	06	13	81	34	32	4	28
74	20	13	81	36	34	4	28
78	02	2	88	43	44	10	38
80	16	2	89	42	46	10	39
jidctflt — inverse discrete cosine transform							
31	03	1	44	19	19	1	19
142	30	9	171	73	86	12	52
164	16	47	189	95	88	16	59
jidctfst — inverse discrete cosine transform							
31	03	1	44	19	19	1	19
44	30	1	61	20	21	2	19
132	43	7	160	76	78	10	64
162	16	46	196	99	98	20	68
jidctint — inverse discrete cosine transform							
31	03	1	44	19	19	1	19
44	30	1	61	20	20	2	19
166	43	275	204	132	119	17	64
191	16	576	226	168	160	30	68
jidctred — discrete cosine transform reduced output							
27	06	1	38	18	18	1	18
32	63	1	43	16	16	1	16
35	78	1	63	38	38	1	38
40	23	1	55	18	19	2	18
47	70	1	64	40	40	2	40
79	56	4	101	38	44	4	30
92	32	8	116	45	52	5	45
118	14	30	145	62	68	10	48

Table 3.5: Experimental results for the HS1, GA0, GA1 and ILP methods of code generation. In the t columns we find the execution time of the code generation and in the τ columns we see the execution time of the generated schedule. The basic blocks are from the jpeg program (part 2).

Chapter 4.

Integrated code generation for loops

Many computationally intensive programs spend most of their execution time in a few inner loops. Thus it is important to have good methods for code generation for those loops, since small improvements per loop iteration can have a large impact on overall performance for whole programs.

In Chapter 3 we introduced an integer linear program formulation for integrating instruction selection, instruction scheduling and register allocation. In this chapter we will show how to extend that formulation to also do modulo scheduling for loops. In contrast to earlier approaches to optimal modulo scheduling, our method aims to produce provably optimal modulo schedules with integrated cluster assignment and instruction selection.

An extensive experimental evaluation is presented in the end of this chapter. In these experiments we compare the results of our fully integrated method to the results of the non-integrated method.

4.1. Extending the model to modulo scheduling

Software pipelining [Cha81] is an optimization for loops where the iterations of the loop are pipelined, i.e. subsequent iterations begin executing before the current one has finished. One well known kind of software pipelining is *modulo scheduling* [RG81] where new iterations of the loop are issued at a fixed rate determined by the *initiation interval* (*II*). For every loop the initiation interval has a lower bound

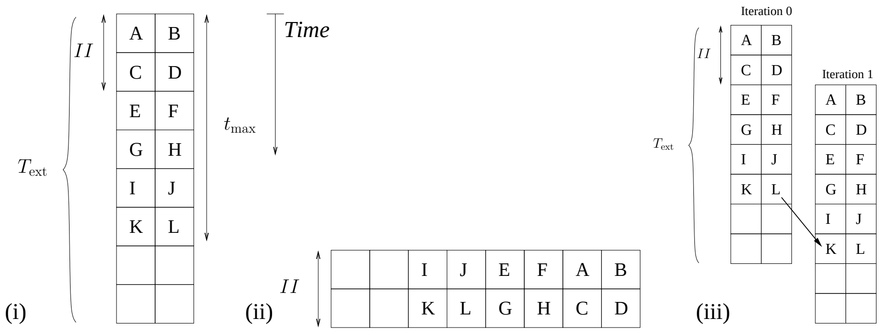


Figure 4.1: An acyclic schedule (i) can be rearranged into a modulo schedule (ii), A-L are target instructions in this example. (iii) T_{ext} has enough time slots to model the extended live ranges. Here $d_{\max} = 1$ and $II = 2$ so any live value from Iteration 0 can not live after time slot $t_{\max} + II \cdot d_{\max}$ in the iteration schedule.

$MinII = \max(ResMII, RecMII)$, where *ResMII* is the bound determined by the available resources of the processor, and *RecMII* is the bound determined by the critical dependence cycle in the dependence graph describing the loop body, see Section 2.7 for definitions of these lower bounds. Methods for calculating *RecMII* and *ResMII* are well documented in e.g. [Lam88].

We note that a kernel can be formed from the schedule of a basic block by scheduling each operation modulo the initiation interval, see (i) and (ii) in Figure 4.1. The modulo schedules that we create have a corresponding *iteration schedule*, and by the *length* of a modulo schedule we mean the number of schedule slots ($t_{\max}$) of the iteration schedule. We also note that, since an iteration schedule must also be a valid basic block schedule, creating a valid modulo schedule only adds constraints compared to the basic block case.

First we need to model loop carried dependences by adding a distance to edges: $E_1, E_2, E_m \subset V \times V \times \mathbb{N}$. The element $(i, j, d) \in E$ represents a dependence from i to j which spans over d loop iterations. Obviously the graph is no longer a DAG since it may contain cycles. The only thing we need to do to include loop distances in the model is to change $r_{rr,i,t}$ to: $r_{rr,i,t+d\cdot II}$ in Equations 3.11, 3.13 and 3.14, and modify Equation 3.16 to:

$$\forall (i,j,d) \in E_m,\ \forall t \in T_{\text{ext}},\quad \sum_{p\in P} \sum_{t_j=0}^{t-II\cdot d} c_{j,p,1,t_j} + \sum_{p\in P} \sum_{t_i=t-L_p+1}^{t_{\max}+II\cdot d_{\max}} c_{i,p,1,t_i} \leq 1 \tag{4.1}$$

The initiation interval *II* must be a parameter to the integer linear programming solver. To find the best (smallest) initiation interval we must run the solver several times with different values of the parameter. A problem with this approach is that it is difficult to know when an optimal *II* is reached if the optimal *II* is not *RecMII* or *ResMII*; we will get back to this problem in Section 4.2.

The slots on which instructions may be scheduled are defined by $t_{\max}$, and we do not need to change this for the modulo scheduling extension to work. But when we model dependences spanning over loop iterations we need to add extra time slots to model that variables may be alive after the last instruction of an iteration is scheduled. This extended set of time slots is modeled by the set $T_{\text{ext}} = \{0, \ldots, t_{\max} + II \cdot d_{\max}\}$ where $d_{\max}$ is the largest distance in any of E_1 and E_2. We extend the variables in $x_{i,r,s,t}$ and $r_{rr,i,t}$ so that they have $t \in T_{\text{ext}}$ instead of $t \in T$, this is enough since a value created by an instruction scheduled at any $t \leq t_{\max}$ will be read, at latest, by an instruction $d_{\max}$ iterations later, see Figure 4.1(iii) for an illustration.

4.1.1. Resource constraints

The constraints in the previous section now only need a few further modifications to also do modulo scheduling. The resource constraints of the kind $\forall t \in T,\ \text{expr} \leq \text{bound}$ (Constraints 3.17–3.19) are modified to:

$$\forall t_o \in \{0, 1, \ldots, II-1\},\quad \sum_{\substack{t\in T_{\text{ext}}:\\ t\equiv t_o (\text{mod } II)}} \text{expr} \leq \text{bound}$$

For instance, Constraint 3.17 becomes:

$$\forall t_o \in \{0, 1, \ldots, II - 1\},\ \forall rr \in \mathcal{RS},\quad \sum_{i \in V} \sum_{\substack{t \in T_{\text{ext}}: \\ t \equiv t_o (\text{mod } II)}} r_{rr,i,t} \leq R_{rr} \tag{4.2}$$

Inequality 4.2 guarantees that the number of live values in each register bank does not exceed the number of available registers. If there are overlapping live ranges, i.e. when a value i is saved at t_d and used at $t_u > t_d + II \cdot k_i$ for some integer $k_i > 1$ the values in consecutive iterations can not use the same register for this value. We may solve this e.g. by doing modulo variable expansion [Lam88]. An example of how modulo variable expansion is used is shown in Figure 4.2. In this example the initiation interval is 2 and the value d is alive for 4 cycles. Then, if iteration 0 and iteration 1 use the same register for storing d, the value from iteration 0 will be overwritten before it is used. One solution is to double the size of the kernel and make odd and even iterations use different registers.

Another issue that complicates the software pipelining case compared to the basic block case is that limiting the number of live values at each point alone is not enough to guarantee that the register allocation will succeed. The circular live ranges may cause the register need to be larger than the maximum number of live values (*maxlive*). *Maxlive* is a lower bound on the register need, but Rau et al. [RLTS92] have shown that this lower bound is very often achievable in practice[1]. In the cases where the lower bound is not achievable, the body of the loop can always be unrolled until the register need is equal to *maxlive*; this was proven by Eisenbeis et al. [ELM95].

4.1.2. Removing more variables

As we saw in Section 3.2.2 it is possible to improve the solution time for the integer linear programming model by removing variables whose values can be inferred.

[1]In their experiments over 90% of the instances had a register need that was the same as *maxlive*. And in almost all of the remaining cases the register need was 1 larger than *maxlive*.

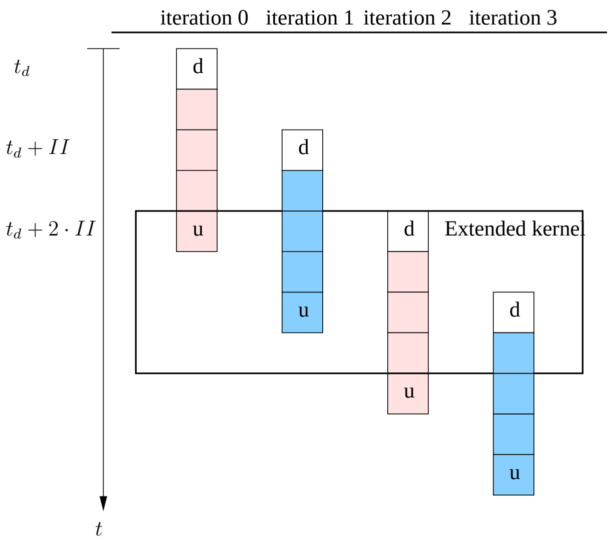


Figure 4.2: In this example we need an extended kernel. The value d is alive for 4 clock cycles, $II = 2$, $k_a = 1$. Even and odd iterations must use different registers for storing d.

Now we can take loop-carried dependences into account and find improved bounds:

$$soonest(i) = \max \left\{ \begin{array}{l} soonest'(i), \\ \max_{\substack{(j,i,d) \in E \\ d \neq 0}} \left(soonest'(j) + L_{\min}(j) - II \cdot d\right)\} \end{array} \right\} \tag{4.3}$$

$$latest(i) = \max \left\{ \begin{array}{l} latest'(i), \\ \max_{\substack{(i,j,d) \in E \\ d \neq 0}} \left(latest'(j) - L_{\min}(i) + II \cdot d\right)\} \end{array} \right\} \tag{4.4}$$

With these new derived parameters we create

$$T_i = \{soonest(i), \ldots, latest(i)\} \tag{4.5}$$

that we can use instead of the set T for the t-index of variable $c_{i,p,k,t}$. I.e., when solving the integer linear program, we do not consider the variables of c that we know must be 0.

```
**Input:** A graph of IR nodes $G = (V, E)$, the lowest possible initiation interval $MinII$, and the architecture parameters.
**Output:** Modulo schedule.
$MaxII = t_{\text{upper}} = \infty$;
$t_{\max} = MinII$;
while $t_{\max} \leq t_{\text{upper}}$ do
  Compute $soonest'$ and $latest'$ with the current $t_{\max}$;
  $II = MinII$;
  while $II < \min(t_{\max}, MaxII)$ do
    solve integer linear program instance;
    if solution found then
      if $II == MinII$ then
        return solution; //This solution is optimal
      fi
      $MaxII = II - 1$ ; //Only search for better solutions.
    fi
    $II = II + 1$
  od
  $t_{\max} = t_{\max} + 1$
od
```

Figure 4.3: Pseudocode for the integrated modulo scheduling algorithm.

Equations 4.3 and 4.4 differ from Equations 3.1 and 3.2 in two ways: they are not recursive and they need information about the initiation interval. Hence, *soonest'* and *latest'* can be calculated when $t_{\max}$ is known, before the integer linear program is run, and *soonest* and *latest* can be parameters that are calculated at solution time.

4.2. The algorithm

Figure 4.3 shows the algorithm for finding a modulo schedule; this algorithm explores a two-dimensional solution space as depicted in Figure 4.4. The dimensions in this solution space are number of schedule slots ($t_{\max}$) and kernel size (II). Note that if there is no

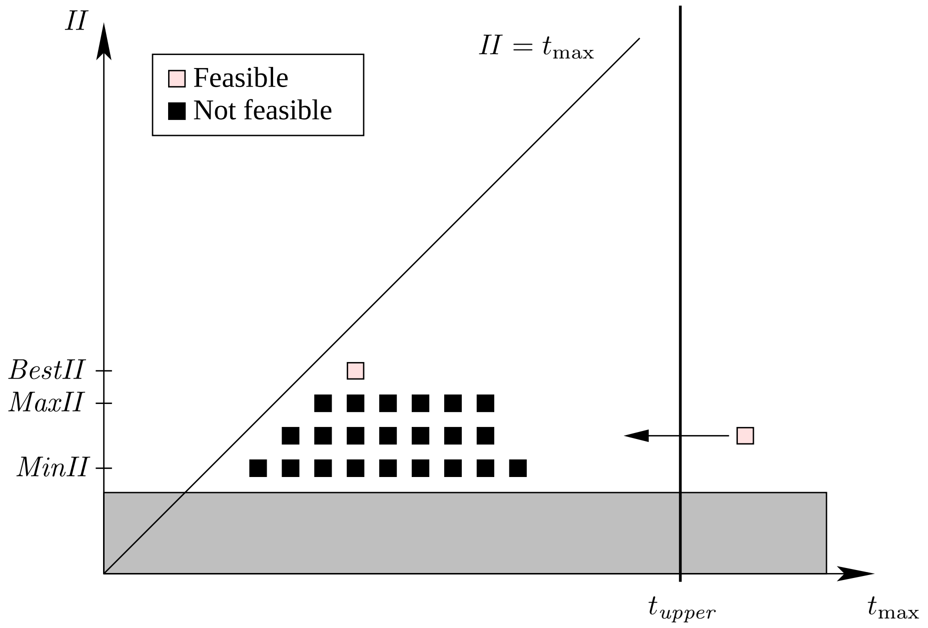


Figure 4.4: This figure shows the solution space of the algorithm. *BestII* is the best initiation interval found so far. For some architectures we can derive a bound, t_{upper}, on the number of schedule slots, $t_{\max}$, such that any solution to the right of t_{upper} can be moved to the left by a simple transformation.

solution with initiation interval *MinII* this algorithm never terminates (we do not consider cases where $II > t_{\max}$). In the next section we will show how to make the algorithm terminate with optimal result also in this case.

There are many situations where increasing $t_{\max}$ will lead to a lower II. One such example is shown in Figure 4.5. Other more complex examples occur on clustered architectures when the register pressure is high and increasing $t_{\max}$ will allow values to be transferred.

A valid alternative to this algorithm would be to set $t_{\max}$ to a fixed sufficiently large value and then solve for the minimal II. A problem with this approach is that the solution time of the integer linear program increases superlinearly with $t_{\max}$. Therefore we find

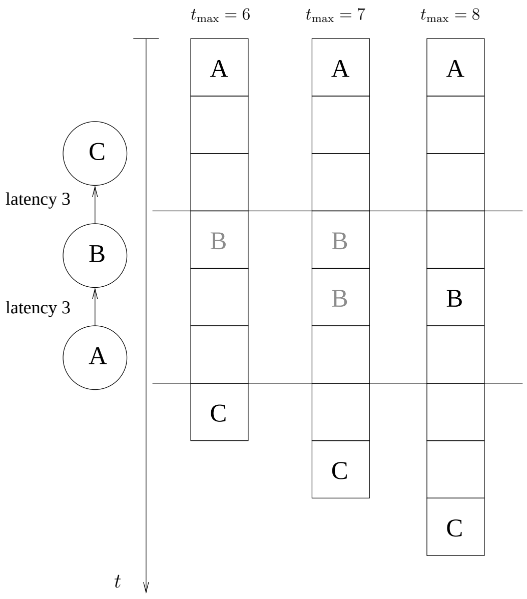


Figure 4.5: An example illustrating the relation between II and $t_{\max}$. The three nodes in the graph can only be covered by instructions which use the same resource, i.e. they can not be scheduled at the same time slot. When $t_{\max} = 6$ and $t_{\max} = 7$ there does not exist a valid modulo schedule with $II = 3$, but when we increase $t_{\max}$ to 8, there is an optimal modulo schedule with $II = 3$.

that beginning with a low value of $t_{\max}$ and increasing it iteratively works best.

Our goal is to find solutions that are optimal in terms of throughput, i.e. to find the minimal initiation interval. An alternative goal is to also minimize code size, i.e. $t_{\max}$, since large $t_{\max}$ leads to long prologs and epilogs to the modulo scheduled loop. In other words: the solutions found by our algorithm can be seen as *pareto optimal* solutions with regards to throughput and code size where solutions with smaller code size but larger initiation intervals are found first.

4.2.1. Theoretical properties

In this section we will have a look at the theoretical properties of Algorithm 4.3 and show how the algorithm can be modified so that it finds optimal modulo schedules in finite time for a certain class of architectures.

Definitions and schedule length

First, we need a few definitions:

Definition 4. *We say that a schedule s is* dawdling *if there is a time slot $t \in T$ such that (a) no instruction in s is issued at time t, and (b) no instruction in s is running at time t, i.e. has been issued earlier than t, occupies some resource at time t, and delivers its result at the end of t or later [KBE07].*

Definition 5. *The* slack window *of an instruction i in a schedule s is a sequence of consecutive time slots on which i may be scheduled without interfering with another instruction in s. And we say that a schedule is n-*dawdling *if each instruction has a slack window of at most n positions.*

Definition 6. *We say that an architecture is* transfer free *if all instructions except NOP must cover a node in the IR-graph. I.e., no extra instructions such as transfers between clusters may be issued unless they cover IR nodes. We also require that the register file sizes of the architecture are unbounded.*

It is obvious that the size of a non-dawdling schedule has a finite upper bound. We formalize this in the following lemma:

Lemma 7. *For a transfer free architecture every non-dawdling schedule for the data flow graph (V, E) has length*

$$t_{\max} \leq \sum_{i \in V} \hat{L}(i)$$

where $\hat{L}(i)$ is the maximal latency of any instruction covering IR node i (composite patterns need to replicate $\hat{L}(i)$ over all covered nodes).

Proof. Since the architecture is transfer free only instructions covering IR nodes exist in the schedule, and each of these instructions is active at most $\hat{L}(i)$ time units. Furthermore we never need to insert dawdling NOPs to satisfy dependences of the kind $(i, j, d) \in E$; consider the two cases:

(a) $t_i \leq t_j$: Let $L(i)$ be the latency of the instruction covering i. If there is a time slot t between the point where i is finished and j begins which is not used for another instruction then t is a dawdling time slot and may be removed without violating the lower bound of j: $t_j \geq t_i + L(i) - d \cdot II$, since $d \cdot II \geq 0$.

(b) $t_i > t_j$: Let $L(i)$ be the latency of the instruction covering i. If there is a time slot t between the point where j ends and the point where i begins which is not used for another instruction this may be removed without violating the upper bound of i: $t_i \leq t_j + d \cdot II - L(i)$. ($t_i$ is decreased when removing the dawdling time slot.) This is where we need the assumption of unlimited register files, since decreasing t_i increases the live range of i, possibly increasing the register need of the modulo schedule (see Figure 6.1 for such a case).

□

And if each operation in the schedule has a finite slack window no larger than n, the length of the schedule has the upper bound defined by the following corollary:

Corollary 8. *An n-dawdling schedule for the data flow graph (V, E) has length*

$$t_{\max} \leq \sum_{i \in V} (\hat{L}(i) + n - 1)$$

Now we can formalize the relation between the initiation interval and schedule length. The following lemma shows when the length of a modulo schedule can be shortened. Theorem 10 shows a property on the relation between the initiation interval and the schedule length.

Lemma 9. *If a modulo schedule s with initiation interval II has an instruction i with a slack window of size at least $2II$ time units, then s can be shortened by II time units and still be a modulo schedule with initiation interval II.*

Proof. If i is scheduled in the first half of its slack window the last II time slots in the window may be removed and all instructions will keep their position in the modulo reservation table. Likewise, if i is scheduled in the last half of the slack window the first II time slots may be removed. $\square$

Theorem 10. *For a transfer free architecture, if there does not exist a modulo schedule with initiation interval $\tilde{II}$ and $t_{\max} \leq \sum_{i \in V}(\hat{L}(i) + 2\tilde{II} - 1)$ there exists no modulo schedule with initiation interval $\tilde{II}$.*

Proof. Assume that there exists a modulo schedule s with initiation interval $\tilde{II}$ and $t_{\max} > \sum_{i \in V}(\hat{L}(i) + 2\tilde{II} - 1)$. Also assume that there exists no modulo schedule with the same initiation interval and $t_{\max} \leq \sum_{i \in V}(\hat{L}(i) + 2\tilde{II} - 1)$. Then, by Lemma 7, there exists an instruction i in s with a slack window larger than $2\tilde{II} - 1$ and hence, by Lemma 9, s may be shortened by $\tilde{II}$ time units and still be a modulo schedule with the same initiation interval. If the shortened schedule still has $t_{\max} > \sum_{i \in V}(\hat{L}(i) + 2\tilde{II} - 1)$ it may be shortened again, and again, until the resulting schedule has $t_{\max} \leq \sum_{i \in V}(\hat{L}(i) + 2\tilde{II} - 1)$. $\square$

And now, finally, we can make the search space of the algorithm finite by setting a limit on the schedule length.

Corollary 11. *We can guarantee optimality in the algorithm in Section 4.2 for transfer free architectures if, every time we find an improved II, we set* $t_{\text{upper}} = \sum_{i \in V}(\hat{L}(i) + 2(II - 1) - 1)$.

Increase in register pressure caused by shortening

Until now we have assumed that the register file sizes are unbounded. Now we show how to allow bounded register file sizes by adding another assumption. The new assumption is that all loop carried dependences have distance no larger than 1.

Lemma 12. *If there is a true data dependence* $(b, a, d) \in E$ *and* a *precedes* b *in the iteration schedule then the number of dawdling time slots between* a *and* b *is bounded by*

$$\omega_{a,b} \leq II \cdot d - L_b$$

where L_b *is the latency of the instruction covering* b*.*

Proof. The precedence constraint dictates

$$t_b + L_b \leq t_a + II \cdot d \tag{4.6}$$

If there are $\omega_{a,b}$ dawdling time slots between a and b in the iteration schedule then

$$t_b \geq t_a + \omega_{a,b} \tag{4.7}$$

Hence

$$t_a + \omega_{a,b} \leq t_b \leq t_a + II \cdot d - L_b \Rightarrow \omega_{a,b} \leq II \cdot d - L_b$$

□

Corollary 13. *If* $d_{\max} \leq 1$ *then any transformation that removes a block of II dawdling time slots from the iteration schedule will not increase the register pressure of the corresponding modulo schedule with initiation interval II.*

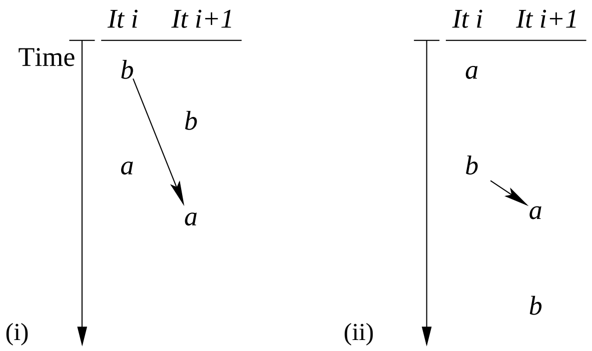


Figure 4.6: Two cases: (i) b precedes a and (ii) a precedes b in the iteration schedule.

Proof. Consider every live range $b \to a$ that needs a register. First we note that the live range is only affected by the transformation if the removed block is between a and b.

If b precedes a in the iteration schedule (see Figure 4.6(i)) then removing a block of II nodes between b and a can only reduce register pressure.

If a precedes b in the iteration schedule (see Figure 4.6(ii)) then, by Lemma 12, assuming $L_b \geq 1$, there does not exist a removable block of size II between a and b in the iteration schedule. □

With these observations we can change the assumption of unbounded register file sizes in Definition 6. The new assumption is that all loop carried dependences have distances smaller than or equal to 1. Furthermore, we can limit the increase in register pressure caused by removing a dawdling II-block:

Corollary 14. *Given an iteration schedule for a data flow graph $G = (V, E)$ the largest possible increase in register pressure of the modulo schedule with initiation interval II caused by removing dawdling blocks of size II is bounded by*

$$R_{increase} \leq \sum_{\substack{(b,a,d)\in E \\ d>1}} (d-1)$$

```
; it 0, it 1
  a
  b
  NOP
        a
        b
        NOP
```

Figure 4.7: The latency of the last instruction in the iteration schedule makes it so that the lowest possible II (3) is larger than $t_{\max}$.

Proof. Consider a live range $b \rightarrow a$ with loop carried distance $d > 1$. By Lemma 12 there are at most

$$\left\lfloor \frac{II \cdot d - L_b}{II} \right\rfloor < d - 1$$

blocks of size II between a and b in the iteration schedule if a precedes b (if b precedes a there can be no increase in register pressure with the same reasoning as above). □

4.2.2. Observation

There are cases when the best possible II is larger than $t_{\max}$. Consider $E = \{(a, b, 0), (b, a, 1)\}$, $L_a = 1, L_b = 2$, $t_{\max} = 2$. The best possible II is 3, see Figure 4.7, however, cases such as this one are not very interesting because software pipelining does not increase the throughput compared to local compaction, and a solution with the optimal II will be found when $t_{\max}$ is increased.

4.3. Experiments

The experiments were run on a computer with an Intel core i7 950 processor with 12 GB RAM. The integer linear programming solver is Gurobi 4.0 with Ampl 10.1.

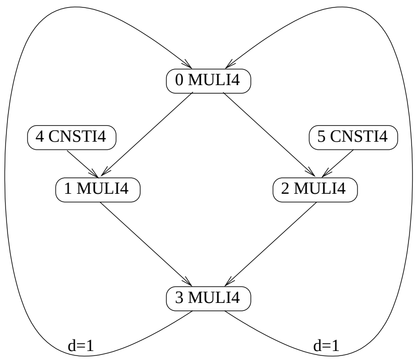


Figure 4.8: *II* is larger than *MinII*. This loop body consists of 4 multiplications. The edges between Node 3 and Node 0 are loop carried dependences with distance 1.

4.3.1. A contrived example

Let us consider an example that demonstrates how Corollary 11 can be used. Figure 4.8 shows a graph of an example program with four multiplications. Consider the case where we have a non-clustered architecture with one functional unit which can perform pipelined multiplications with latency 2. Clearly, for this example we have $RecMII = 6$ and $ResMII = 4$, but an initiation interval of 6 is impossible since IR-nodes 1 and 2 can not be issued at the same clock cycle. When we run the algorithm we quickly find a modulo schedule with initiation interval 7, but since this is larger than *MinII* the algorithm can not determine that it is an optimal solution. Now we can use Corollary 11 to find that an upper bound of 18 can be set on $t_{\max}$. If no improved modulo schedule is found where $t_{\max} = 18$ then the modulo schedule with initiation interval 7 is optimal. This example is solved to optimality in a few seconds by our algorithm.

4.3.2. Dspstone kernels

Table 4.1 shows the results of our experiments with the algorithm from Section 4.2. We used 5 target architectures, all variations of the Texas Instruments TI-C62x DSP processor [Tex00]:

Architecture	*MinII*	*II*	$t_{\max}$	time(s)	To
single	5	5	17	36	-
trfree	3	3	16	41	-
mac	3	3	15	51	-
mac_tr	3	3	15	51	-
mac_tr_spill	3	3	15	55	-

(a) dotp, $|V| = 14$

Architecture	*MinII*	*II*	$t_{\max}$	time(s)	To
single	6	6	19	45	-
trfree	3	3	16	33	-
mac	3	3	16	63	-
mac_tr	3	3	16	65	-
mac_tr_spill	3	3	16	69	-

(b) FIR, $|V| = 20$

Architecture	*MinII*	*II*	$t_{\max}$	time(s)	To
single	8	8	12	15	-
trfree	4	6	10	23	13
mac	4	6	10	65	12
mac_tr	4	6	10	1434	11
mac_tr_spill	4	6	10	2128	11

(c) n_complex_updates, $|V| = 27$

Architecture	*MinII*	*II*	$t_{\max}$	time(s)	To
single	9	9	14	20	-
trfree	5	5	13	33	-
mac	5	5	13	63	-
mac_tr	5	5	13	92	-
mac_tr_spill	5	5	13	191	-

(d) biquad_N, $|V| = 30$

Architecture	*MinII*	*II*	$t_{\max}$	time(s)	To
single	18	21	35	179	39
trfree	18	19	31	92	72
mac	17	19	31	522	35
mac_tr	17	20	29	4080	30
mac_tr_spill	17	20	29	4196	30

(e) IIR, $|V| = 38$

Table 4.1: Experimental results with 5 DSPSTONE kernels on 5 different architectures.

- single: single cluster, no MAC, no transfers and no spill[2],
- trfree: 2 clusters, no MAC, no transfers and no spill,
- mac: 2 clusters, with MAC, no transfers and no spill,
- mac_tr: 2 clusters, with MAC and transfers, no spill,
- mac_tr_spill: 2 clusters, with MAC, transfers and spill.

The kernels are from the DSPSTONE benchmark suite [uVSM94] and the dependence graphs were generated by hand. The columns marked *II* show the found initiation interval and the columns marked $t_{\max}$ show the schedule length. The IR does not contain branch instructions, and the load instructions are side effect free (i.e. no auto increment of pointers).

The time limit for each individual integer linear program instance was set to 3600 seconds and the time limit for the entire algorithm was set to 7200 seconds. If the algorithm timed out before an optimal solution was found the largest considered schedule length is displayed in the column marked To. We see in the results that the algorithm finds optimal results for the `dotp`, `fir` and `biquad_N` kernels within minutes for all architectures. For the `n_complex_updates` kernel an optimal solution for the single cluster architecture is found and for the other architectures the algorithm times out before we can determine if the found modulo schedule is optimal. Also for the `iir` kernel the algorithm times out long before we can rule out the existence of better modulo schedules.

We can conclude from these experiments that while the algorithm in Section 4.2 *theoretically* produces optimal results for transfer free architectures with the t_{upper} bound of Corollary 11, it is not realistic to use for even medium sized kernels because of the time required to solve big integer linear programming instances. However, in all cases,

[2]Strictly speaking, spilling is not part of the architecture. In our modelling of the problem, however, spilling is activated by adding a “fake” register bank representing the spill area in the memory. Therefore, we consider it a property of the target architecture.

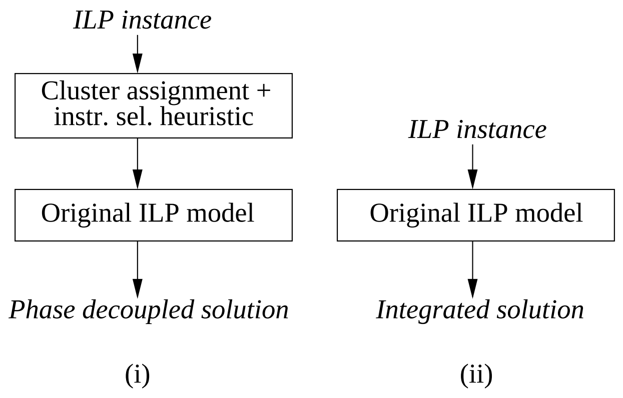


Figure 4.9: By applying a separate algorithm for instruction selection and cluster assignment as a preprocessing step to the ILP solver (i) we can quantify the improvement due to fully integrating the code generation phases (ii).

the algorithm finds a schedule with initiation interval within 3 of the optimum.

4.3.3. Separating versus integrating instruction selection

In this section we will investigate and quantify the difference in code quality between our integrated code generation method and a method where instruction selection and cluster assignment are not integrated with the other phases. One way to do this is to first use a separate algorithm for cluster assignment and instruction selection (see Figure 4.9). In this way we could define values for some of the solution variables in the integrated integer linear programming model and then solve the restricted instance. This would allow for a comparison of the achieved initiation interval for the less integrated method to the more integrated one. In this section we describe the details of how we did this and show the results of an extensive evaluation.

Figure 4.10 shows the components of the software pipelining algorithm. The integrated software pipelining model (d) takes a set of instructions, which are to be picked from, for each IR-node. The de-

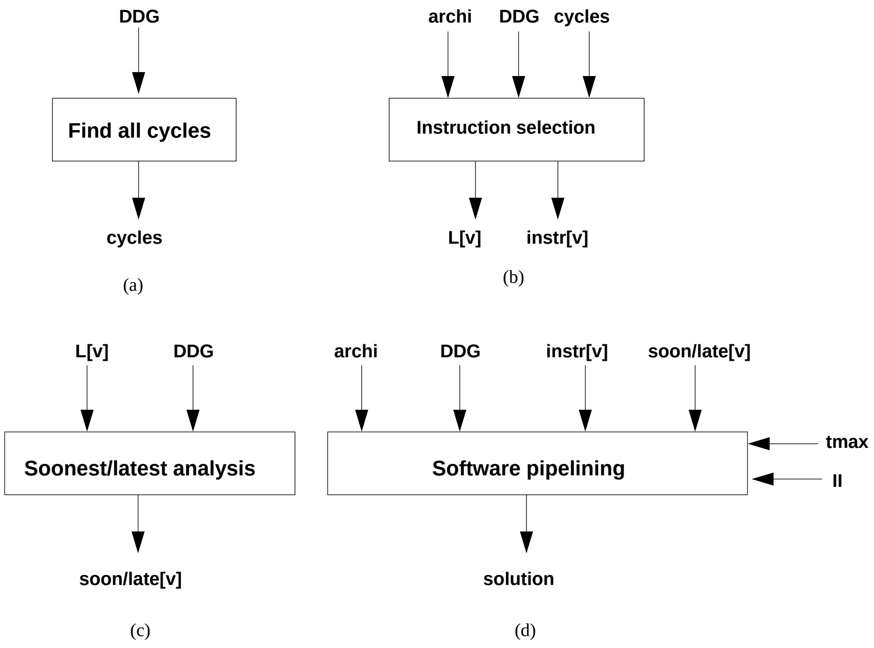


Figure 4.10: The components of the separated method for software pipelining.

fault set is the one which contains all instructions that can cover the operation that the IR-node represents. So making instruction selection separated is just a matter of supplying a set with only one instruction for each IR-node. Soonest-latest analysis (c) was described in Section 4.1.2, here it is extended so that the exact latency of each node of the graph can be given as input. This is useful for the cases where instruction selection has already been done. If the latencies are not known before the software pipelining phase we must use conservative values for these latencies. The integer linear programming model for the instruction selection phase (b) is basically the same model as the fully integrated one, but with the scheduling and register constraints removed and with a modified objective function that solves for minimal *MinII*. The result of the cycle finding part (a) is used when calculating *RecMII*.

The instruction selection phase

For instruction selection we use a smaller version of the integer linear programming model for modulo scheduling. The constraints regarding scheduling and register allocation were stripped out and Constraints 3.6–3.9 are kept. To ensure that the instruction selection will work we add a constraint saying that for all edges $(a, b, d) \in E_1 \cup E_2$, if a writes to rr and b reads from $rs \neq rr$ there must be a transfer for this edge. Then we can bound *ResMII* by the following constraint (here with a simplified presentation):

$$\forall f \in \mathcal{F}, \quad \mathit{ResMII} \geq \sum_{i \in V} \text{instr. covering } i \text{ uses } f + \sum_{y \in \text{transfers}} y \text{ uses } f \tag{4.8}$$

When we calculate the lower bound on the initiation interval we need to take two things into account: the available resources and the cycles of the data dependence graph. We implemented Tarjan's algorithm for listing all cycles in a graph [Tar73][3]. We ran this algorithm once on all data dependence graphs to generate all cycles; the time limit of the algorithm was set to 10 seconds[4], and if the limit was exceeded we stored all cycles found so far. For the cases where the enumeration algorithm exceeded the time limit, we had already found several thousand cycles. Continuing beyond this point may lead to finding cycles that are more critical, which could tighten the lower bound *RecMII*.

A cycle is represented by a set of all IR-nodes that are in it. Every cycle has a distance sum, that is the sum of distances of the edges that make up the cycle. Let $\text{Cyc}(V, E)$ be the set of cycles defined by the graph with nodes V and edges E. And let $\text{dist}(C)$ be the distance sum of cycle $C \in \text{Cyc}(\text{V}, \text{E})$. Let $\mathit{intern}[p] = \{a : \exists b, (a, b) \in E_p\}$ denote

[3]Tarjan's algorithm has time complexity $O(|V||E|\text{nrcyc})$. There are more efficient algorithms, e.g. Johnson's algorithm $O((|V| + |E|)\text{nrcyc})$ [Joh75], but Tarjan's algorithm is easier to implement.

[4]These 10 seconds are not included in the reported solution times in the evaluation section.

the set of inner nodes in pattern p, then *RecMII* can be bounded by

$$\forall C \in \mathrm{Cyc}(V,E), \quad \sum_{\substack{n\in C\\ p\in P\\ k\in(B_p - intern[p])\\ t\in T}} L_p c_{n,p,k,t} \leq \mathit{RecMII} \cdot \mathrm{dist}(C) \quad (4.9)$$

In the instruction selection phase the most important objective is to minimize the lower bound on the initiation interval. However, just minimizing the lower bound is not enough in many cases: when *MinII* is defined by *RecMII* and *ResMII* is significantly lower than *RecMII* the instruction selection solution may be very unbalanced with regards to resource allocation of the functional units. To address this problem we make the instruction selection phase minimize *MinII* as a first objective, and with this minimal *MinII* also minimize *ResMII*. Assuming that $\mathit{MinII} < M$ (we used $M = 100$ in our experiments) this can be expressed as:

$$\min \; M \cdot \mathit{MinII} + \mathit{ResMII} \quad (4.10)$$

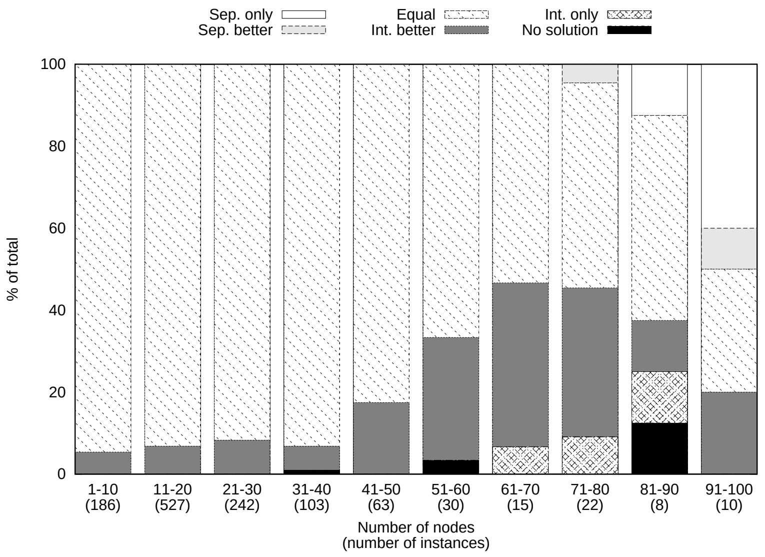
subject to:

$$\mathit{MinII} \geq \mathit{ResMII} \quad (4.11)$$

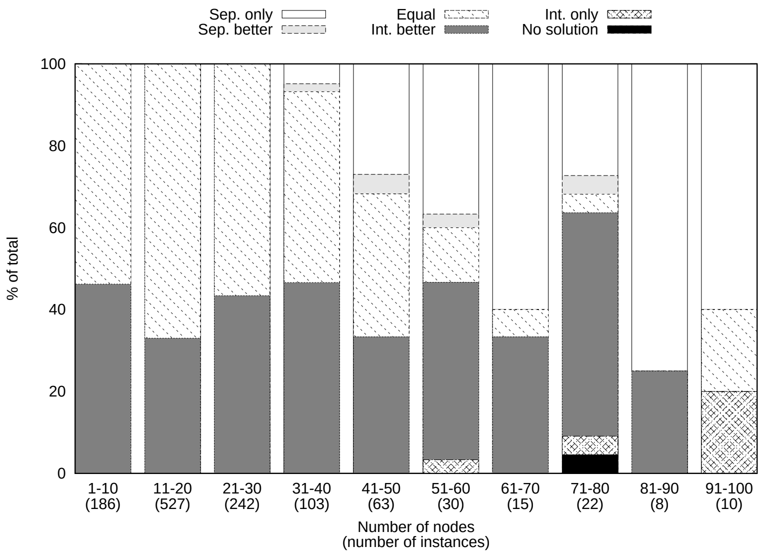
$$\mathit{MinII} \geq \mathit{RecMII} \quad (4.12)$$

The advantage of adding *ResMII* to the objective function of the minimization problem is that resources will be used in a more balanced way for the cases where the *MinII* is dominated by *RecMII*. This better balance makes the scheduling step easier, but we have observed that the more complicated objective function in the instruction selection phase leads to a much increased memory usage. Even though the increased memory usage of the instruction selection phase makes the machine run out of memory the results are on average better than when the simple objective function is used.

The experiments use data dependence graphs generated by the st200cc compiler with optimization level `-O3`. Input to the compiler are programs from the Spec2000, Spec2006, Mediabench and FFMPEG benchmarks [Tou09]. For solving the integer linear programming instances we use Gurobi 4.0 running on an Intel Core i7 950.



(a) Single cluster.



(b) Double cluster.

Figure 4.11: Results of comparison between the separated and fully integrated version for the two architectures. The time limit is 4 minutes.

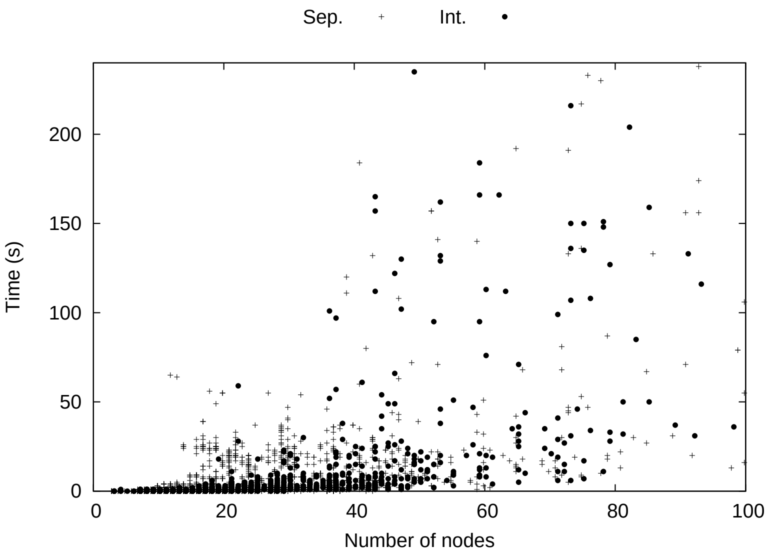
The primary optimization goal is to minimize the initiation interval. After removing all graphs which include instructions that are too specialized for our model (e.g. shift-add instructions) and all duplicate graphs 1206 graphs remain. We have used two target architectures: a single cluster and a double cluster variant of TI-C62x. The double clustered variant has multiply-and-add instructions and includes transfers and spill instructions. The single cluster variant does not have composites, transfers or spill instructions.

Figure 4.11 summarizes the comparison between the integrated and the separated method, both with a time limit of 4 minutes per instance. For the single-cluster architecture (a) we see that the cases where the integrated method is better than the separated one are rare; it only happens for 9% of the instances. The reason for this is that it is relatively easy to find a good instruction selection for this simple architecture, hence not much is gained by integrating instruction selection. When we look at the results for the two-cluster architecture (b) we find that the integrated method beats the separated one in 39% of the instances. It does not seem to matter much what size the graph has; the integrated method beats the separated one in roughly the same proportion of the instances for all sizes of the input graph.

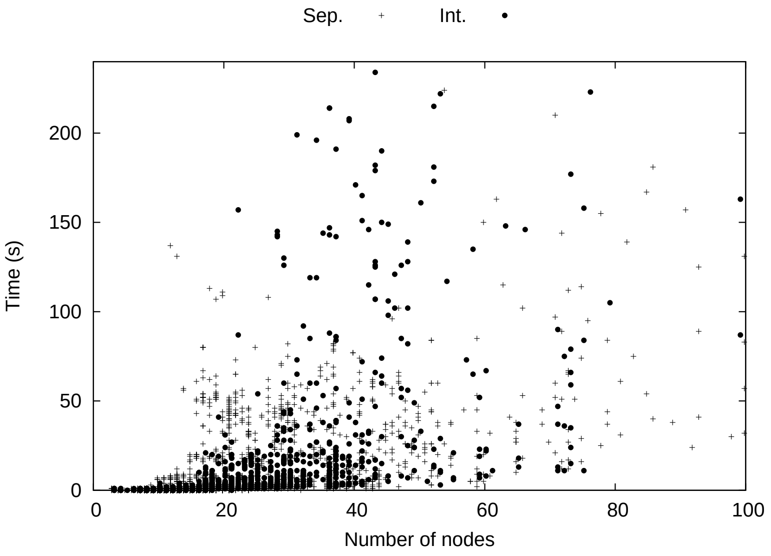
However, as the size of the graphs become larger the increased complexity makes the integrated method more likely to time out before finding any solution at all, while the simpler, separated method manages to finds a solution. The few cases where both methods find a solution, and the separated solution is better, are explained by the fact that the separated method reaches larger $t_{\max}$ before it times out, so it explores a larger part of the solution space in terms of iteration schedule length; increasing the time limit of the integrated method would remove this anomaly.

Figure 4.12 shows scatter-plots of the solution times for each found solution for the single-cluster and two-cluster experiments. It comes as no surprise that it takes less time to find solutions with the separated method than with the integrated one.

Table 4.2 summarizes the results for the instances where the integrated method finds a solution that is better than the one found



(a) Single cluster



(b) Double cluster

Figure 4.12: Scatter-plots showing the time required to find solutions where the optimization terminates in the time limit.

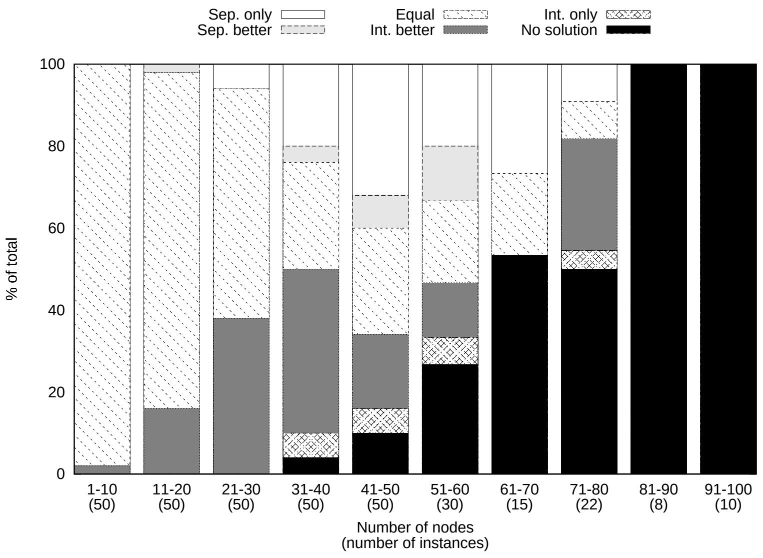


Figure 4.13: Results of comparison between the separated and fully integrated version for the 4-way clustered architecture. The time limit is 2 minutes and the number of instances is limited to 50 in this chart. For the cases where Int. finds a solution that is better than one found by Sep. the average value of *IntII*/*SepII* is 0.84.

	1-cluster	2-cluster	4-cluster(bus)
1-10	.82	.69	.92
11-20	.90	.83	.78
21-30	.89	.89	.86
31-40	.93	.89	.84
41-50	.95	.92	.90
51-60	.88	.86	.78
61-70	.96	.87	-
71-80	.91	.88	.76
81-90	.97	.87	-
91-100	.88	-	-
all	.90	.83	.84

Table 4.2: Average values of *IntII*/*SepII* for the instances where Int. is better than Sep.

by the separated method. The values shown are average values of *IntII*/*SepII*, where *IntII* and *SepII* are the found initiation intervals of the integrated and the separated methods respectively. These values quantify how much better the integrated method is compared to the separated method. For instance: if the size of the kernel is halved then the throughput of the loop is doubled assuming that the number of iterations is large enough (so that the prolog and epilog of the loop are negligible). The average value of *IntII*/*SepII* for the two-cluster architecture is 0.83, i.e. the average asymptotic speedup for the loops is $1/0.83 = 1.20$.

Figure 4.13 shows the results of an experiment with a four-cluster version of the TI-C62x architecture where the two cross paths are replaced by a single bus. For this test we decreased the time limit to 2 minutes and limited the number of instances to 50 per size range. Now, because of the lower time limit and more complicated architecture, both methods perform worse compared to in the two-cluster experiment. There are quite many instances for which both methods fail to find a solution, these must be solved by a cheaper heuristic. An interesting result is that the ratio of instances where Int beats Sep is smaller compared to in the two-cluster case. One explanation for this is that replacing the cross paths with a bus reduces the possibilities for Int to optimize the use of transfers that makes integration of phases beneficial.

4.3.4. Live range splitting

Perhaps the biggest drawback of software pipelining is the increase in code size of the loop code. The code size increase is caused both by modulo variable expansion and by adding prolog- and epilog-code. If we are lucky the prolog and epilog can be executed in parallel with the code surrounding the loop, but the unrolled code caused by modulo variable expansion has no such opportunities.

Some architectures have hardware support for handling overlapping lifetimes; this hardware is called *rotating register files*. But such support in hardware is costly, and there are software techniques that can significantly reduce code size expansions e.g. by Llosa et al. [LF02] and by Stotzer and Leiss [SL09].

One technique for reducing modulo variable expansion is to make the live ranges as short as possible, i.e. schedule instructions close to successors or predecessors. Another technique is *live range splitting*, where copy instructions are inserted so that a live range uses multiple registers to make sure that conflicts are avoided.

These two techniques can be integrated into our integer linear programming model by adding the constraint:

$$\begin{gathered}\forall rr \in \mathcal{RS}, \forall i \in V, \forall t_0, \\ \sum_{t=t_0}^{t_0+II} r_{rr,i,t} \leq II + \sum_{t=t_0}^{t_0+II-1} x_{rr,rr,t}\end{gathered} \tag{4.13}$$

The interpretation of this constraint is that no value (i) may be alive for $II+1$ time slots in the same register bank (rr), unless it was copied during the first II time slots. Adding this constraint can lead to a worse II compared to not having it, but modulo variable expansion would not be necessary.

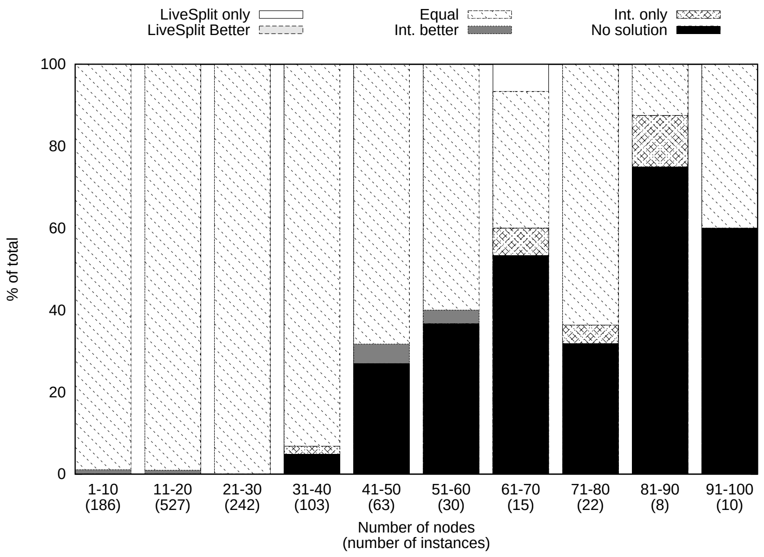


Figure 4.14: Comparison between the integrated algorithm using live range splitting, and the integrated algorithm without live range splitting. Time limit is 4 minutes.

Figure 4.14 shows the effects of using the live range splitting described in Inequality 4.13. We can see that adding the constraint for live range splitting affects the *II* optimization results only slightly; the normal integrated algorithm is only better than the live range splitting algorithm in 1% of the instances. The average increase in solution time is 16%.

It is also possible to set an upper bound on the unrolling caused by modulo variable expansion. This is done by allowing a value to be live for longer than *II* cycles; we set the maximum unroll factor to *MU* and modify Inequality 4.13 to the following:

$$\forall rr \in \mathcal{RS}, \forall i \in V, \forall t_0,$$
$$\sum_{t=t_0}^{t_0+II\cdot MU} r_{rr,i,t} \leq MU \cdot II + \sum_{t=t_0}^{t_0+MU\cdot II-1} x_{rr,rr,t} \tag{4.14}$$

This will constrain the solution so that a value can only be live for $II \cdot MU$ cycles before it has to be copied or transfered to another register bank.

Chapter 5. Integrated offset assignment

One important part of generating code for DSP processors is to make good use of the address generation unit (AGU). In this chapter we divide the code generation into three parts: (1) scheduling, (2) address register assignment, and (3) storage layout. The goal is to find out if solving these three subproblems as one big integrated problem gives better results compared to when scheduling or address register assignment is solved separately. We present optimal dynamic programming algorithms for both integrated and non-integrated code generation for DSP processors.

In our experiments we find that integration of the subproblems is beneficial, especially when the AGU has 1 or 2 address registers available. We also find that integrating address register assignment and storage layout gives slightly better results than integrating scheduling and storage layout. I.e. address register assignment has a larger impact on the end result than scheduling has.

5.1. Introduction

Digital signal processors often include an *address generation unit* (AGU) for calculating memory addresses. These AGUs have a register file containing pointers into memory, and every time one of these pointers is read it can, at the same time, be incremented or decremented by a small value. For a compiler to generate fast and compact code for such an architecture good utilization of the AGU is a necessity.

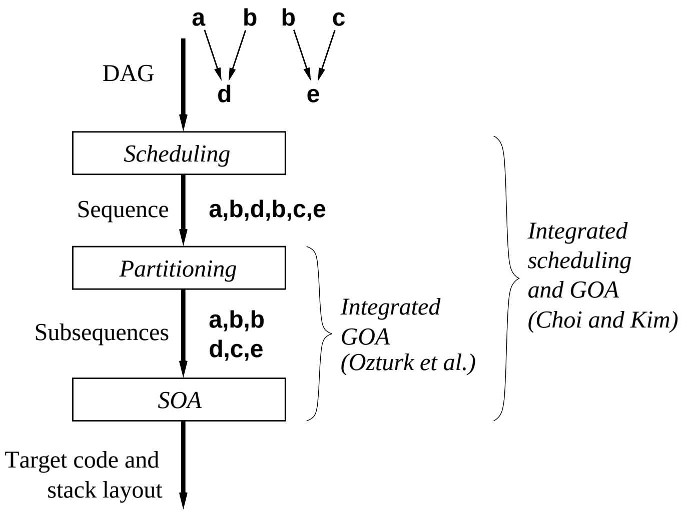

a
b
b
c
d
e
DAG
Scheduling
Sequence
a,b,d,b,c,e
Partitioning
Subsequences
a,b,b
d,c,e
SOA
Target code and stack layout
Integrated GOA (Ozturk et al.)
Integrated scheduling and GOA (Choi and Kim)

Figure 5.1: The subproblems of integrated offset assignment. In this chapter we study how much can be gained by solving all subproblems at the same time. Choi and Kim have presented an heuristic algorithms for the fully integrated problem [CK02]. Özturk et al. have solved GOA (partitioning and SOA) optimally [OKT06].

The basic problem of finding the best storage layout given an access sequence and a single address register was first studied by Bartley [Bar92]. He shows how to transform the access sequence into an undirected graph where the nodes represent variables and an edge between two nodes is given a weight equal to the number of times the nodes are neighbors in the access sequence. Finding a storage layout that maximizes the utilization of auto-increment/decrement is equivalent to finding a maximal Hamiltonian path in the access graph. Finding a maximal Hamiltonian path is an NP-complete problem; Bartley presented a greedy heuristic that has since been improved by Liao et al. [LDK+96], Leupers and Marwedel [LM96] and by Ali et al. [AEBS08]. This basic problem with 1 address register and an increment/decrement range of +1/-1 is called the *simple offset assignment* problem (SOA).

A generalization to *general offset assignment* (GOA), where there are multiple address registers in the AGU, has been presented by Liao et al. [LDK+96]. They propose a heuristic method to solve GOA by reducing it to one SOA instance per address register. Each of the SOA-instances handles accesses to a disjoint subset of the variables in the access sequence. This heuristic was improved by Leupers and Marwedel [LM96] by finding good ways to partition the access sequence into subsequences.

Gebotys presents a fast minimum cost circulation technique (MCC) for optimal address register assignment for a given memory layout and access sequence [Geb97]. She concludes that memory layout only has minor impact on the final quality of the generated code when address register assignment is solved optimally. This conclusion is not supported by the experimental evaluation by Huynh et al. where a range of GOA-heuristics are evaluated with the MCC-technique [HABT07].

Leupers have presented a comparison of a range of different SOA-heuristics [Leu03]. One of the conclusions is that the performance differences of the SOA-heuristics is surprisingly small for real-life problem instances. He also finds that comparisons of SOA-algorithms using random access sequences only give a coarse performance evaluation, and that the results are not always the same when real-life problem instances are used.

Udayanarayanan and Chakrabarti have presented a GOA-heuristic that performs modify register optimization as a final step in the code generation [UC01]. Their evaluation shows that, when a modify register is available, the impact of storage layout is not an important factor to the final quality of the solution.

Özturk et al. have presented an optimal integer linear programming formulation for GOA with modify registers. Their experiments show that all instances in their experiment are solved to optimality within a few minutes with a good integer linear programming solver [OKT06].

Atri et al. explore how commutativity of operations in the SOA-problem can be used to reduce the number of edges in the access graph, allowing for better SOA-solutions [ARK00].

Rao and Pande have presented heuristic methods for SOA which uses commutativity, associativity and distributivity of operations to

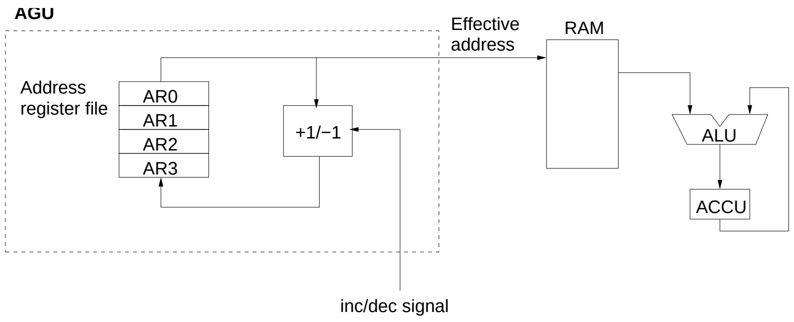


Figure 5.2: The architecture that we consider has an address generation unit with a number of address registers (4 in this figure). The ALU has a single accumulator register.

reorder the access sequence. The heuristics are also extended to GOA and experiments show a static code size reduction (of whole programs) of 2% on average compared to previous heuristics [RP99].

Choi and Kim have described a heuristic algorithm that integrates scheduling and GOA. Their algorithm works in two steps: first an initial schedule and storage assignment in computed; second this solution is iteratively improved by reordering the access sequence and redoing the storage assignment [CK02]. This heuristic is compared to the one presented by Leupers and Marwedel [LM96] and shows good improvements. In Section 5.3 we compare our optimal algorithm to this heuristic by Choi and Kim.

In this chapter we study the question of how important scheduling and address register assignment is for the offset assignment problem. We do this by comparing optimal solutions, in which scheduling and address register assignment are integrated, to solutions of non-integrated code generation. See Figure 5.1 for an illustration of different levels of integration.

5.1.1. Problem description

In the first part we assume a simple accumulator-based processor where all instructions have unit latency and occupation time (see

Figure 5.2). The architecture has:

- one ALU with an accumulator register, and
- one AGU with $k \geq 1$ address registers; an address register can be incremented or decremented by 1 each time it is read.

This kind of setup is common in real-world DSPs; variations of it can be found in e.g. Texas Instruments-C2x [Tex88].

A problem instance is a basic block for which we want to generate code. The basic block is represented by a directed acyclic graph (DAG) $G = (V, E)$. Each node $x \in V$ represents an operation on the accumulator register. An edge $(u, v) \in E$ represents either true data dependences or some other precedence constraint.

The goal of the code generation is to minimize the number of explicit assignments to address registers; we will use the term *cost* for this. Minimizing the *cost* will both minimize run time and code size of the produced code.

5.2. Integrated offset assignment and scheduling

In this section we will introduce an algorithm for integrated offset assignment based on dynamic programming. The method works by starting with an empty solution which is extended by selecting a new node to schedule, and for this node also select an address register to use for the memory access.

Assume that we have k address registers in the AGU and that we have n nodes to schedule. Then we can define what we mean by a partial solution:

Definition 15. *A partial solution P of length l consists of:*

- *A schedule $S = (s_1, s_2, \ldots, s_l)$, $l \leq n$, where s_t is the node that is scheduled at level t, $1 \leq t \leq l$.*
- *Address register usage: r_t^i is the variable that address register i points to when it was last used (at slot t or earlier); if address register i has not been used yet, we can assume that r_t^i points to every variable, $0 \leq i < k$, $0 \leq t < n$.*

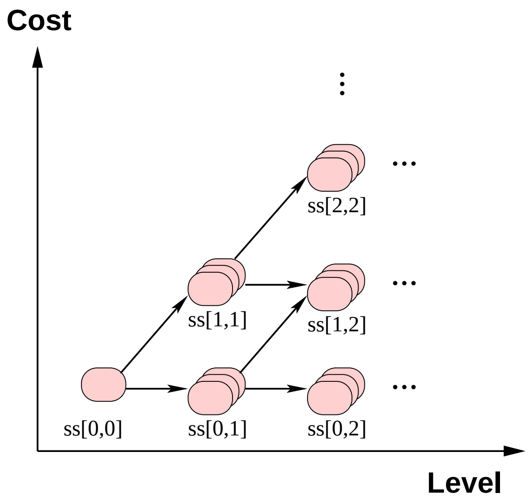


Figure 5.3: Structure of the solution space (*ss*) used in the DP-algorithm. The algorithm expands partial solutions with smallest cost first.

- *A set* NG *of neighbor pairs, if* $(v, w) \in$ NG *the variables* v *and* w *are neighbors on the stack.*

Example 16. *Figure 5.4(a) shows a DAG representing the basic block:*

```
D = A + B;
E = B + C;
```

Assuming that we have a single address register (`AR0`*) Figure 5.4(b) shows a partial solution of length 3. In this partial solution the node sequence is* $S = (1, 0, 2)$ *which means that first* `B` *is loaded into the accumulator register, then* `A` *is added to the accumulator, and then the result is stored in* `D`*. The contents of the set NG implies that* `A` *is neighbor to both* `B` *and* `D`*, this is satisfied when the stack is laid out as in Figure 5.4(c). The corresponding assembler code for the partial solution is shown in Figure 5.4(d); note that the last instruction*

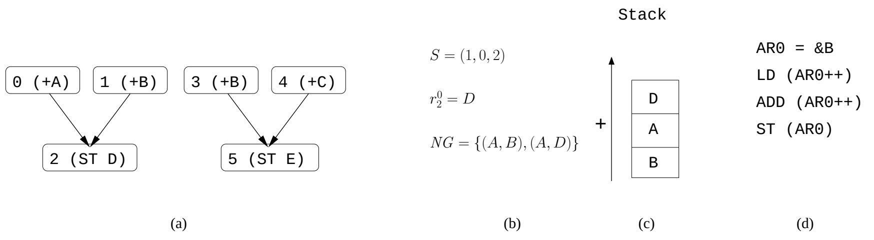


Figure 5.4: A partial solution.

`ST (AR0)` *is not yet finalized, a post-increment or decrement may be added as the partial solution is extended.*

5.2.1. Fully integrated GOA and scheduling

The dynamic programming algorithm in Figure 5.5 works by enumerating the partial solutions in a two dimensional solution space shown in Figure 5.3. The first dimension is *cost* and second dimension (*level*) is the number of scheduled nodes. Initially, the solution space (*ss*) contains a single empty partial solution with *cost* 0 and *level* 0 (lines 0-1). The outer loop over *cost* progresses along the cost axis of the solution space. And the loop over *level* progresses over number of scheduled nodes in the DAG.

The algorithm proceeds by, for each partial solution at the current cost and level, expanding this partial solution by selecting an unscheduled node from the DAG. The node that is selected must be ready, i.e. all its predecessors must have already been selected (line 9). If the algorithm was to enumerate all possible solutions it would be necessary to create a new partial solution at the next level, by using each of the available address registers, however if, in a partial solution, an address register already points to the variable of the selected node, then it is enough to consider this address register only when expanding the partial solution, see Lemma 17.

The function extend(*cur*, *reg*, *v*) creates a copy of the current partial solution (*cur*) and then extends it by using register *reg* for accessing the variable *v*.

Algorithm: DP-RS.
Method: Dynamic programming in two dimensions: *cost* and number of scheduled nodes (*level*).
Input: A DAG representing a basic block and the set of available address registers ($\mathcal{AR}$).
Output: Schedule, storage layout and address register assignment.

```
 0: initialize $ss[cost,level]$ to empty lists for all $cost$ and $level$
 1: push ($ss[0, 0]$, an empty partial solution)
 2: for $cost$ from 0 to infinity do
 3:     for $level$ from 0 to infinity do
            // keep only one in every group of comparable solutions
 4:         prune($ss[cost, level]$)
 5:         while ( $ss[cost, level]$ not empty )
 6:             $cur$= pop ($ss[cost, level]$)
 7:             if ($cur$ is a complete solution)
 8:                 return $cur$
 9:             foreach ($n \in$ selectable nodes in $cur$)
10:                 $n_v$ = the variable of node $n$
11:                 Case 1: A $reg \in \mathcal{AR}$ points to $n_v$:
12:                     push ($ss[cost, level+1]$, extend($cur$, $reg$, $n_v$))
13:                 Case 2: No $reg \in \mathcal{AR}$ points to $n_v$:
14:                     foreach ($reg \in \mathcal{AR}$)
15:                         Case 2A: $reg$ points to a neighbor of $n_v$:
16:                             push ($ss[cost, level+1]$ extend($cur$, $reg$, $n_v$))
17:                         Case 2B: $reg$ points to $v$ that is not a neighbor of $n_v$:
18:                             $new$= extend($cur$, $reg$, $n_v$)
19:                             if (make_neighbor($new$, $v$, $n_v$))
20:                                 push ($ss[cost, level+1]$, $new$)
21:                             push ($ss[cost+1, level+1]$, extend($cur$, $reg$, $n_v$))
```

Figure 5.5: Dynamic programming algorithm.

In Case 2 (line 13) all possibilities for extending the current partial solution are exhausted.

The function make_neighbor (line 19) tries to make v and n_v neighbors in *new*. The function returns true if it is successful. The reason for a failure is either that one of the variables already has two neighbors, or that making the variables neighbors would create a cycle.

The algorithm terminates when the current partial solution (line 6) is a complete solution, i.e. all of the nodes in the DAG have been scheduled. Termination can be done earlier; if a new partial solution is created that is complete, and has the same cost as the current partial solution (i.e. on lines 12, 16, 18 or 20, but not on line 21) then this solution is optimal and the algorithm can exit. This early exit strategy is not included in the presentation in Figure 5.5 to avoid cluttering the presentation.

5.2.2. Pruning of the solution space

The partial solutions are constructed systematically in increasing order of cost. This means that expanding less promising partial solutions is postponed as much as possible. However, even with this postponing it is obvious that any algorithm based on enumeration will suffer from a combinatorial explosion for large instances. We can delay the explosion of the number of partial solutions in the solution space if we can identify partial solutions, a and b, such that a is *at least as good as* b in the sense that a best final solution b^* to which b is a partial solution cannot be better than a best final solution a^* to which a is a partial solution. Then the partial solution b can be removed without affecting the optimality of the enumeration. If both a is at least as good as b and b is at least as good as a, we say that a and b are *comparable*.

One obvious case where two partial solutions are comparable is when all of the following are true:

- The last element in the schedules are equal.
- The same set of nodes has been scheduled.

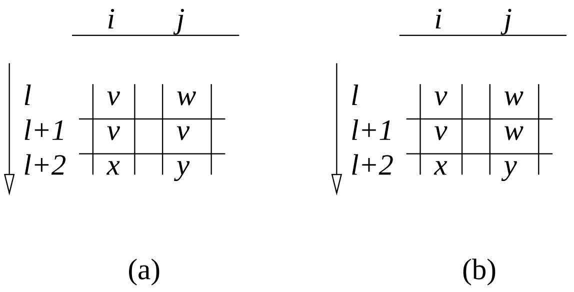


Figure 5.6: Using a register that already has the correct value is always good. If register i already points to the next variable v in the access sequence then for any solution that uses another register j (a), there exists a solution that uses register i that does not have a higher cost (b).

- The set of variables that the address registers point to are the same.
- The NG sets of both partial solutions are equal.

Before the algorithm expands nodes of a certain cost and level we can do a pruning step (see Figure 5.5, line 4). This pruning can be done by sorting the partial solutions and then traversing the sorted list and keeping only one in each group of comparable solutions.

Another way in which we can delay the combinatorial explosion is to avoid doing a complete enumeration when it is not necessary. Consider a situation where we expand a partial solution in which an address register already points to the next variable in the access sequence (Case 1, line 11, in Figure 5.5), we can use this address register and do not need to make additional partial solutions where another register is used. Formally:

Lemma 17. *Let P be a partial solution of length l in which $r^i_l = v$ and $r^j_l = w \neq v$. Assume that the next access is to variable v. Let P^i be a partial solution that uses address register i for this access to v and let P^j be a partial solution that uses address register j for this* *access to v. Then for any extension of P^j to a final solution there exists an extension to P^i that is at least as good.*

Proof. Assume that P^j is extended to a final solution P^{j*} (see Figure 5.6 (a)) which is better than any final solution to which P^i (see Figure 5.6 (b)) is a partial solution.

In P^{j*} we have $r_l^i = v$, $r_{l+1}^i = v$, $r_l^j = w$, and $r_{l+1}^j = v$, let $r_{l+2}^i = x$ and $r_{l+2}^j = y$. Now, there are two cases:

- If $y = v$ then create P^{i*} as a copy of P^{j*} but with $r_{l+1}^j = w$. Now the only difference in cost between P^{j*} and P^{i*} can occur if we must reassign address register j somewhere between levels l and $l+2$, in P^{j*} we have the transition $w \rightarrow v \rightarrow v$ and in P^{i*} the transition is $w \rightarrow w \rightarrow v$, i.e. they have the same cost.

- If $y \neq v$ we must have $x = v$ (the contents of two address registers can not change at the same time), and we can reduce this problem to the case $y = v$ by making $P^{j*'}$ a copy of P^{j*} and swap the values of r_k^i and r_k^j for all $k \geq l+2$ (i.e. take the P^{j*} solution and make it use address register j instead of i at time slot $l+2$).

In other words: there always exists a final solution in which address register i is used for the access to variable v that is at least as good as P^{j*}. □

So, we have shown that if one address register already points to a variable that we wish to access we can use this address register and not care about any other solution in which another address register is used for this access. For the other case, when there is no register that points at the next variable n_v in the access sequence we must exhaustively extend the current partial solution using each of the registers $reg \in \mathcal{AR}$. If reg does not point to a neighbor of n_v then we always extend the partial solution without making the two variables neighbors, and if it is possible, also add one partial solution where the two variables are made neighbors.

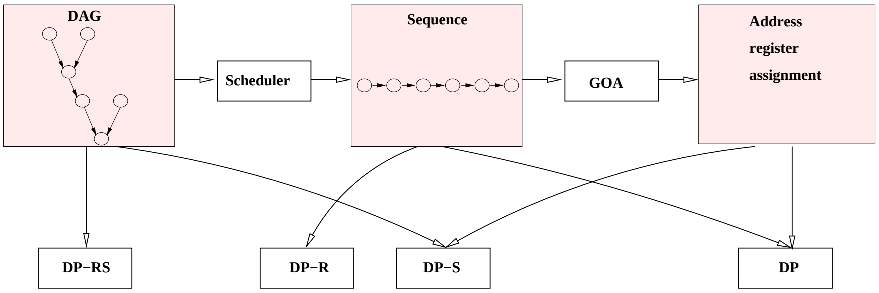


Figure 5.7: The algorithm flow.

5.2.3. Optimal non-integrated GOA

One question that we want to answer here is how much can be gained by integrating the parts of the code generation. To find the answer to this question we need to run experiments where we compare optimal integrated code generation to optimal non-integrated code generation. In this subsection we will present the non-integrated algorithms that we have used for the experiments: DP-R, which does scheduling separately, DP-S, which does address register assignment separately, and DP, which does all parts separately.

DP-R works exactly like DP-RS (Figure 5.5) except that the nodes have already been scheduled. I.e. only one node is selectable on Line 9 in Figure 5.5.

DP-S integrates scheduling, just like DP-RS does. But it forces all partial solutions to conform to the address register assignment found by `solve_goa` [LM96]. I.e. on lines 11, 13 and 14 in Figure 5.5 '$reg \in \mathcal{AR}$' is replaced by '$reg =$ ARX', where ARX is the address register used for variable n_v in the solution of a `solve_goa` run. Note that since scheduling is integrated we cannot simply partition the access sequence and solve k independent subproblems because the sub-sequences will not be independent of each other.

The most basic dynamic programming algorithm, DP, does not integrate any of the phases; all it does is calculate the optimal stack layout with a fixed schedule and a fixed address register assignment.

See Figure 5.7 for a sketch of the different parts of DP, DP-R, DP-S and DP-RS.

5.2.4. Handling large instances

We have observed that the large problem instances often do not succeed because the dynamic programing solver runs out of memory. We can work around this problem by counting the current total number of partial solutions, and when we reach a maximal number of partial solutions we switch to an enumeration strategy that prioritizes extending partial solutions with higher levels, i.e. partial solutions that are nearer to completion. Then the solver program will not run out of memory, but obviously we may lose some opportunities for pruning comparable partial solutions. In other words the computation time probably increases but the memory usage is limited.

5.2.5. Integrated iterative improvement heuristic

Figure 5.8 shows the algorithm Naive-it by Choi and Kim [CK03]. The algorithm is based on iterative improvement and it internally uses the function solve_goa from [LM96]. This algorithm is very fast compared to our dynamic programming algorithms and we will see in our experimental evaluation that it finds results that are not far from the optimum. Using a cheap algorithm to compare our optimal methods to makes it possible to assess how many of our test instances are trivial to solve.

5.3. Experimental evaluation

The experiments were run on an Intel i7 3.06 GHz, each instance had a time limit of 1 hour, and a memory limit of 10GB. In total we have 400 instances divided by sequence length into 4 classes (100 instances in each class). The sequence lengths are 10-14, 15-19, 20-29 and 30+. The source of the instances is Offsetstone[1]. However, the

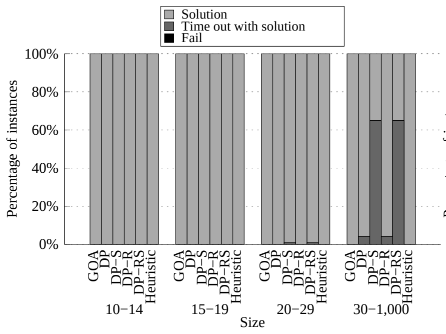
[1]The Offsetstone benchmark suite is available from http://address-code-optimization.org (2010).

Algorithm: Heuristic.
Method: Integrated scheduling and offset assignment by iterative improvement.
Input: A DAG representing a basic block with *nrop* operations.
Output: Schedule, storage layout and address register assignment.

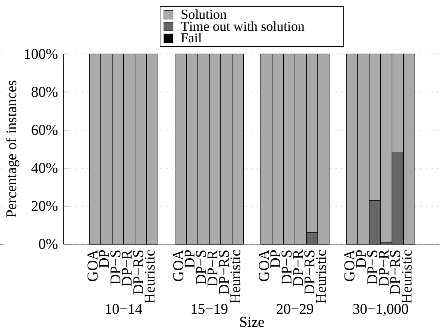
```
$s$ = generate_initial_sequence()
while ($best\_cost$ is improved)
    for $i$ from 0 to $nrop$- 1
        $best\_cost$= solve_goa($s$)
        $j^* = i$
        for $j$ from 0 to $nrop$- 1
            if (can_not_swap($s$, $i$, $j$))
                continue // move to the next operation in the j-loop
            swap($s$, $i$, $j$)
            $new\_cost$= solve_goa($s$)[a]
            if ($new\_cost$< $best\_cost$)
                $j^* = j$
                $best\_cost$= $new\_cost$
            swap($s$, $j$, $i$) // undo the swap.
        swap($s$, $i$, $j^*$) // do the best swap.
```

[a]Use commutativity in Operation i, if this is beneficial.

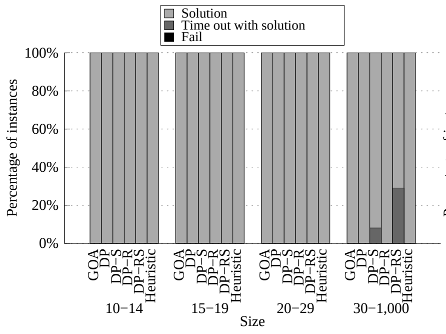
Figure 5.8: Our implementation of the heuristic algorithm Naive-it [CK03] for integrated offset assignment. The algorithm is included here for completeness.



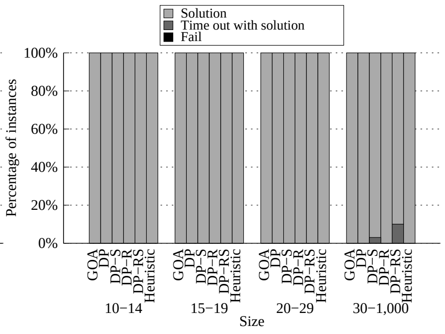
(a) k=1



(b) k=2



(c) k=3



(d) k=4

Figure 5.9: Bar chart showing the success rate of the algorithms for $k = 1$ to 4 address registers. The time limit is sufficiently high so that all algorithms find a solution before they time out.

Offsetstone problems are already scheduled and the graphs are not available, we have therefore created DAGs of the sequences so that the Offsetstone sequence is one possible schedule for the DAG that we create. We have assumed that each operation is commutative and takes two operands. Obviously graphs generated in this way are most likely not equivalent to the original problems. But these DAGs are probably more realistic than randomly generated problems.

The algorithms that are included in the evaluation are:

- GOA, solve_goa from [LM96];
- DP, DP-R and DP-S, described in Section 5.2.3;
- DP-RS, our fully integrated algorithm, described in Section 5.2.1;
- Heuristic, our implementation of Naive-it from [CK03].

Figure 5.9 shows the success rates of the algorithms. When we set the time limit to 1 hour all algorithms find a solution. However, for some of the large instances the dynamic programming algorithms times out before a *guaranteed optimal* solution is found[2]. This case is most common for the larger instances with the fewest number of address registers. The reason for this is that when the number of address registers is large it is easier to find a solution that has no extra explicit assignments of the address registers, i.e. the value of the *cost* variable is 0 in the solution. We can also see in this chart that integrating scheduling is much more expensive than integrating address register assignment.

Figure 5.10 shows the difference in number of clock cycles on average between each algorithm and the optimum for the cases where the optimum is known (i.e. when the fully integrated DP-RS algorithm terminates before the time out is reached). We find that the greatest difference between the integrated and non-integrated algorithms occurs when there is only a single address register (Figure 5.10(a)). In

[2]Note that this is not a contradiction. The solution that is found first can have a *cost* that is one larger than the current cost, meaning that a solution on the current *cost* can still be found.

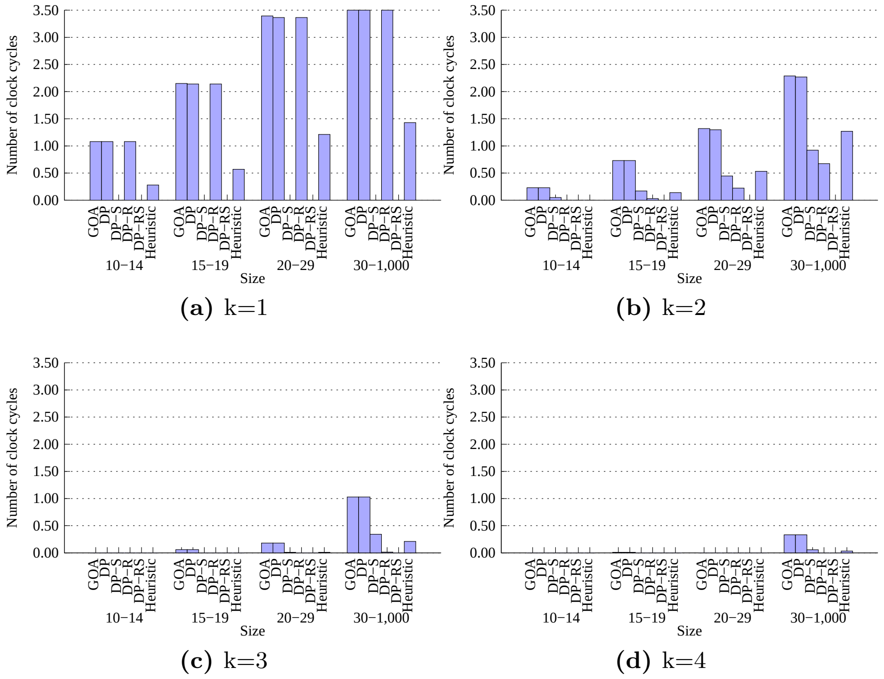


Figure 5.10: Number of extra explicit address register assignments compared to DP-RS (DP-S in (a)).

this case the optimum is found by DP-S since there is no address register assignment. For the largest basic blocks (more than 30 nodes) the average reduction in number of address code operations between the non-integrated DP to the integrated DP-S is 3.5.

Looking at the case with 2 address registers (Figure 5.10(b)) we see that the differences between the algorithms are smaller. Also it is clear that more is gained by integrating register assignment than is gained by integrating scheduling. But both DP-R and DP-S, on average, outperform the Naive-it heuristic for the larger instances but the difference is quite small.

When the number of address registers is increased even more (Fig-

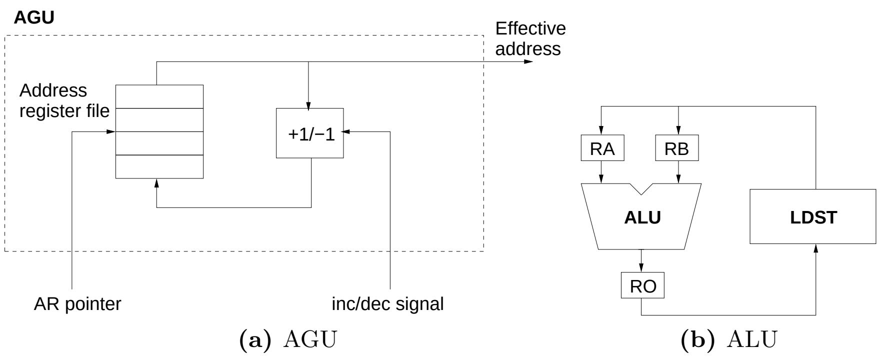


Figure 5.11: The architecture that we consider has an address generation unit (a) with address registers that can be post-incremented/decremented, and a simple arithmetic-logic unit (b) with two in-registers and one out-register.

ure 5.10(c) and (d)) we find that integrated algorithms do not give much better results than the non-integrated ones. I.e. the cheaper algorithms are already quite close to optimal.

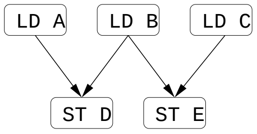
5.4. GOA with scheduling on a register based machine

Until now we have assumed that the target architecture works with a single accumulator register. However, if the target architecture is register based, with two in-registers and a single out-register then some adjustments of the method are necessary.

We assume a simple processor, depicted in Figure 5.11, which has:

- one ALU with two in-registers and one out-register, and
- one AGU with $k \geq 1$ address registers which can be incremented or decremented by 1 each time it is used in a load or store instruction.

At each clock cycle the processor can either load a value into an address register or concurrently do any combination of 1 ALU-operation,



(a)

```
RX = LD A
RY = LD B
R0 = RX + RY
ST D = R0
RX = LD C
R0 = RX + RY
ST E = R0
```

(b)

```
RX = LD A
RY = LD B
R0 = RX + RY ; RX = LD C
R0 = RX + RY ; ST D = R0
ST E = R0
```

(c)

Figure 5.12: Example of GOA scheduling with a register machine. The DAG in (a) can be scheduled as shown in (b), using 7 instructions, or as in (c), using 5 instructions by issuing ALU operations concurrently with load and store instructions.

1 load or store using an address register r, and 1 post-increment or post-decrement of r. We assume that the registers are early-read and late-write and there is no support for register-to-register move, hence all intermediate values must be stored in memory and then be loaded if they are used again. This processor is a simplified variation of the ADSP-2100 processor [Ana90].

The ALU operations are not represented in the DAG, but are instead assumed to be scheduled before the corresponding store. This means that if we schedule a store directly after a load that it depends on one extra clock cycle is added between the load and the store for the ALU operation. Figure 5.12 shows an example of such a situation: If the DAG in (a) is scheduled as in (b) we must add two extra cycles for the ALU-operations to complete in time for the stores. But if we use the schedule in (c) we do not need to add extra cycles since the ALU operations can be done in parallel with a load and a store operation, using the fact that reads from registers are done before writes to registers within a clock cycle. Observe that in general it is not always optimal to place the ALU-operation in the cycle before the corresponding store, hence the explicit placement of the ALU-operation is also an important part of the optimization problem.

The algorithm for this case is the same as the one for the accumulator based architecture, see Figure 5.5, with the addition of handling

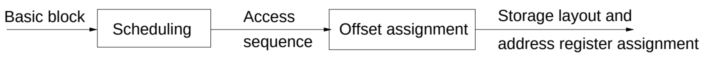


Figure 5.13: The separated algorithm works by first doing scheduling and then solve the offset assignment in a second step.

also the extra in-registers. A node in the DAG is only selectable if it does not cause there to be too many live values. And when a new partial solution is added to the solution space, one additional cycle must be added if a store instruction is scheduled directly after a load instruction that it depends on, because the ALU-operation can not be overlapped with the surrounding operations in this case.

5.4.1. Separating scheduling and offset assignment

To find out in which cases integrating scheduling with offset assignment leads to better code we must compare an optimal integrated algorithm to an optimal algorithm where scheduling is done before offset assignment. In our dynamic programming algorithm (INT) we can separate the phases by first running the algorithm and disregard address computations and offset assignment (see Figure 5.13).

When we have a schedule we can add dependence edges to G, serializing the DAG, and then run INT to solve the offset assignment part of the problem. From now on this separated algorithm is called SEP.

5.4.2. Integer linear programming formulation

We have developed an integer linear programming formulation for the offset assignment problem integrated with scheduling. In addition to making it possible to solve problem instances optimally, creating an integer linear programming model also formalizes the problem that we are studying.

The parameters in our model are: the graph $G = (V, E \cup D)$ where E is the set of true data dependences, D is the set of other data dependences and V is the set of variable accesses. Specifically $V =$

$\{0, \dots, n-1\}$, where n is the number of vertices, the set of schedule slots $T = \{0, \dots, n-1\}$[3], the set of AGU registers AGU $= \{0, \dots, k-1\}$, the set of variables $Var = \{0, \dots, m-1\}$ and the set of stack locations $Loc = \{0, \dots, m-1\}$. There is also a mapping from each node $n \in V$ to the variable that the node represents; the mapping is represented by $n_v = x$, meaning that the node v in the graph corresponds to an access to variable $x \in Var$.

The variables are all binary:

- $S_{t,x,i}$ is 1 iff vertex x is scheduled at slot t using AGU register i.
- $r_{t,i,v}$ is 1 iff AGU register i points to variable v at time slot t.
- $L_{p,v}$ is 1 iff variable v is placed on stack location p.
- $C_{t,i}$ is 1 iff AGU register i must be loaded for the access at time slot t.
- $Calu_t$ is 1 iff it is necessary to add a clock cycle for the ALU operation before time slot t.
- $vl_{t,x}$ is 1 iff the value loaded at vertex x is still alive at time slot t.
- $A_{t,x}$ is 1 iff the ALU-operation of store node x is scheduled at time t.

The objective is to minimize the total cost:

$$\min \sum_{\substack{i \in AGU \\ t \in T}} C_{t,i} + \sum_{t \in T} Calu_t \tag{5.1}$$

First we need some simple constraints saying that all nodes must be selected (5.2), only one node can be selected at a time (5.3), each

[3]The schedule slots are not absolute cycles in the final schedule, but they give an absolute order of the instructions, making the final schedule trivial to construct.

register must point to a single variable at every slot (5.4), each location hosts exactly one variable (5.5) and each variable has exactly one location (5.6):

$$\forall x \in V, \quad \sum_{\substack{i \in AGU \\ t \in T}} S_{t,x,i} = 1 \tag{5.2}$$

$$\forall t \in T, \quad \sum_{\substack{x \in V \\ i \in AGU}} S_{t,x,i} = 1 \tag{5.3}$$

$$\forall t \in T, \forall i \in AGU, \quad \sum_{v \in Var} r_{t,i,v} = 1 \tag{5.4}$$

$$\forall p \in Loc, \quad \sum_{v \in Var} L_{p,v} = 1 \tag{5.5}$$

$$\forall v \in Var, \quad \sum_{p \in Loc} L_{p,v} = 1 \tag{5.6}$$

All stores have exactly one ALU-operation and there may be at most one ALU-operation scheduled at each time slot:

$$\forall d \in V_{\mathrm{ST}}, \quad \sum_{t \in T} A_{t,d} = 1 \tag{5.7}$$

$$\forall t \in T, \quad \sum_{d \in V_{\mathrm{ST}}} A_{t,d} \leq 1 \tag{5.8}$$

For every edge $(x, y) \in E \cup D$, y must not come before x:

$$\forall (x, y) \in E \cup D, \forall t \in T, \quad \sum_{\substack{t' \in T : t' \leq t \\ i \in AGU}} S_{t',y,i} + \sum_{\substack{t' \in T : t' \geq t \\ i \in AGU}} S_{t',x,i} \leq 1 \tag{5.9}$$

The ALU-operation must come after the load instruction and before the store instruction (if the ALU-operation is at the same schedule slot as the store, one cycle must be added, see Equation 5.18):

$$\forall t \in T, \forall (s,d) \in E, \quad \sum_{t'=0}^{t} A_{t',d} + \sum_{i \in AGU} \sum_{t'=t}^{n-1} S_{t',s,i} \leq 1 \tag{5.10}$$

$$\forall t \in T, \forall (s,d) \in E, \quad \sum_{i \in AGU} \sum_{t'=0}^{t} S_{t',d,i} + \sum_{t'=t+1}^{n-1} A_{t',d} \leq 1 \tag{5.11}$$

Address register i can not be used in the schedule unless the register points to the right place:

$$\forall t \in T, v \in Var, i \in AGU, \quad r_{t,i,v} \geq \sum_{x \in V : n_x = v} S_{t,x,i} \tag{5.12}$$

We only allow address register i to take a new value at the same schedule slot where it is used:

$$\begin{gathered} \forall i \in AGU, \forall t \in T, \forall v \in Var, \\ r_{t,i,v} + \sum_{w \in Var : w \neq v} r_{t+1,i,w} - \sum_{x \in V} S_{t,x,i} \leq 1 \end{gathered} \tag{5.13}$$

A vertex $s \in V$ is live in a register at schedule point t if s was loaded before t and an ALU-operation that uses s is scheduled at a time $t' \geq t$:

$$\forall t \in T, \forall (s,d) \in E, \quad vl_{t,s} \geq \sum_{\substack{i \in AGU \\ t' \in T : t' < t}} S_{t',s,i} + \sum_{t' \in T : t' \geq t} A_{t',d} - 1 \tag{5.14}$$

Since our target architecture has only 2 in-registers to the ALU, only 2 vertices may be live at the same time:

$$\forall t \in T, \quad \sum_{s \in V_{\mathrm{LD}}} vl_{t,s} \leq 2 \tag{5.15}$$

And there is only one out-register of the ALU, so all ALU-operations must be consumed by a store before the next ALU-operation:

$$\forall t \in T, \forall (s_1, d_1) \in E, \forall (s_2, d_2) \in E : s_1 \neq s_2 \text{ and } d_1 \neq d_2,$$
$$\sum_{t'=0}^{t-1} A_{t',d_1} + \sum_{i \in AGU} S_{t,d_2,i} + \sum_{i \in AGU} \sum_{t'=t+1}^{n-1} S_{t',d_1,i} \leq 2 \tag{5.16}$$

There must be a load to address register i at time t if the register switches from variable w to variable v unless they are neighbors:

$$\forall t \in T, \forall p \in \mathit{Loc}, \forall v \in \mathit{Var}, \forall w \in \mathit{Var}, \forall i \in AGU,$$
$$C_{t,i} \geq L_{p,v} + r_{t,i,w} + r_{t+1,i,v} - 2 - \sum_{q=p-1}^{p+1} L_{q,w} \tag{5.17}$$

For any edge $(s, d) \in E$, if the ALU operation is scheduled in the same slot as either s or d then we must add an extra clock cycle for doing the ALU operation:

$$\forall t \in T \setminus \{0\}, \forall (s, d) \in E,$$
$$\mathit{Calu}_t \geq \sum_{i \in AGU} (S_{t-1,s,i} + S_{t,d,i}) + A_{t-1,d} + A_{t,d} - 2 \tag{5.18}$$

Lastly, we observe that each stack layout can be mirrored by putting the variables on the stack in reverse order, and the mirrored layout will have the exact same neighbor pairs. Hence we can remove nearly half of the valid solutions by forcing one of the variables to always be on the top half of the stack:

$$\sum_{p \in \mathit{Loc}: p \leq \frac{m}{2}} L_{p,1} = 1 \tag{5.19}$$

We have also reduced the number of variables by the common method of calculating earliest and latest possible schedule slots. This is done in the same way as described in Section 3.2.2.

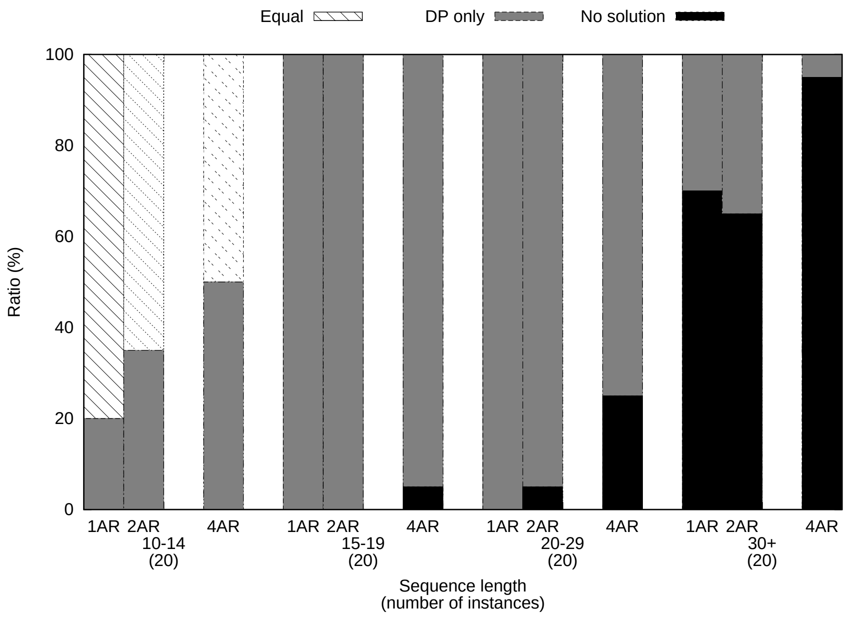


Figure 5.14: Stacked bar chart showing the results of the comparison between ILP and INT. Equal means that both algorithms finds a solution of the same quality. DP only means that the ILP times out and DP is successful.

5.4.3. Experimental evaluation

The experiments were run on an Intel i7 3.06 GHz, each instance had a time limit of approximately 60 seconds, and a memory limit of 4GB. The integer linear programming solver used is CPLEX 10.2.

DP versus integer linear programming

Figure 5.14 shows a stacked bar chart summarizing the results of our comparison between the integer linear programming method (ILP) of Section 5.4.2 and the dynamic programming algorithm (INT) described in Section 5.4. In this experiment we have limited the number

	10-14	**15-19**	**20-29**	**30+**
1AR SEP-INT	.92	1.53	2.36	3.70
2AR SEP-INT	0	.02	.17	.63
4AR SEP-INT	0	0	0	0

Table 5.1: Average cost improvement of INT. compared to SEP. for the cases where both are successful.

of basic blocks per category to 20 and we only consider optimal results (if one of the algorithms has a valid solution when it reaches its time limit, this solution is thrown away since we can not guarantee optimality). The results are a bit surprising: ILP fails for all blocks of size 15 or larger, and for every instance of size smaller than 15, if ILP finds an optimal solution then so does INT. From this test we conclude that the algorithm based on dynamic programming works faster than our method based on integer linear programming because every instances that is solved by ILP is also solved by INT, and in addition some of the instances that are not solved by ILP are solved by INT.

INT versus SEP

Figure 5.15 shows a stacked bar chart summarizing the comparison of our algorithms INT and SEP. For the smallest blocks, which have 10-14 accesses, we find that all instances are solved to optimality in SEP. In these instances the access sequence is very important for SOA; in Table 5.1 we see that INT results in 0.92 lower cost (clock cycles) than SEP on average. On the other hand, for GOA (with 2 and 4 address registers) the scheduling is not important: INT and SEP always result in solutions with the same cost.

The results are similar when the number of accesses is 15-29. It is worth noting that the importance of scheduling increases somewhat for SOA. And for the instances with 20-29 accesses and 2 address registers the number of cases where INT beats SEP is not negligible

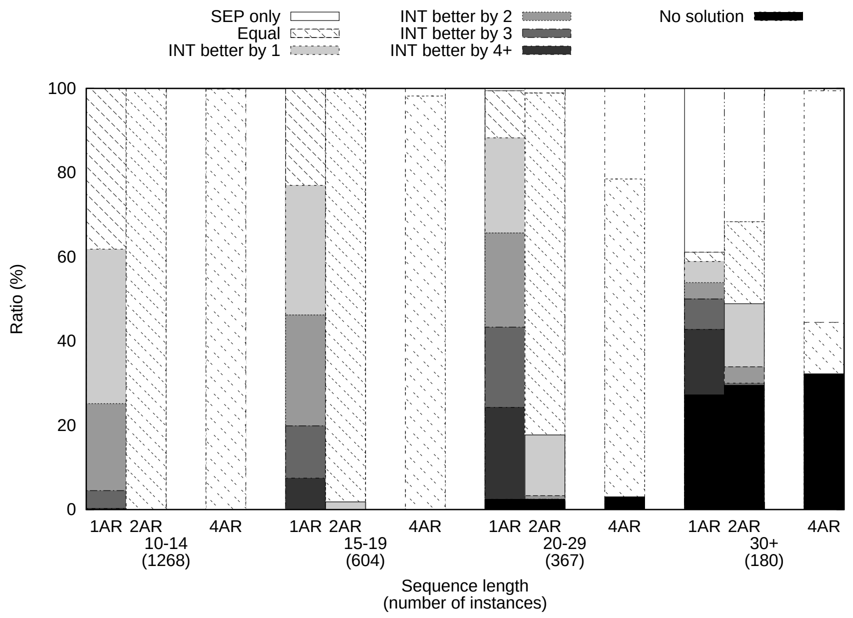


Figure 5.15: Stacked bar chart showing the results of the comparison between INT and SEP.

(15%). We also note that INT times out in 22% of the cases with 4 address registers.

For the largest instances, with more than 30 accesses, the results are noticeably different. The reason for this is that many of the instances are becoming intractable and can not be solved to optimality by our algorithms. For the cases where the algorithm finishes the results suggest that the integration of scheduling now becomes even more important: in Table 5.1 we see that, for the cases where both INT and SEP are successful, INT is on average 3.70 better than SEP for SOA, and when we have 2 address registers, INT is on average 0.63 better than SEP. When we have 4 address registers there is not a single instance where INT produces a better result than the one found by SEP.

	10-14	**15-19**	**20-29**	**30+**
SOA-TB - SEP	.78	.99	1.11	2.35
SOA-TB - INT	1.70	2.53	3.47	4.69

Table 5.2: Average cost reduction of our DP algorithm compared to a cheap heuristic SOA-TB [LM96] as implemented in [Add10].

INT versus SOA-TB

We have compared our optimal methods to a very cheap heuristic to find the answer to two questions:

- Is there room for improvement in SOA-heuristics?
- How many of our test instances are trivial?

The cheap heuristic algorithm that we have chosen is SOA-TB by Leupers and Marwedel [LM96]. This algorithm is based on a previous algorithm by Liao et al. [LDK+96] in which SOA is modelled by a weighted access graph $G_{\text{access}} = (V, E)$ where each node corresponds to a variable and the weight of edge $(u, v) \in E$ is the total number of transitions from u to v or from v to u in the access sequence. With this model an optimal solution to SOA can be found by finding a Hamiltonian path in G with maximum weight. In SOA-TB this is done by a heuristic that orders the edges by weight, using a tie-break function that prioritizes edges that has low weight neighbors. For our experiments we have used the SOA-TB implementation available in the benchmark-suite from [Add10]. Table 5.2 shows the average value of the cost reduction compared to SOA-TB for the the cases where the DP algorithm finds an optimal solution. The most striking result is that the average cost reduction for SEP for the largest instances is 2.35, while for INT the average cost reduction is 4.69. Furthermore, we found that in 92% of the cases where INT completes, it finds a better result than SOA-TB, i.e. only 8% of our instances can be considered to be trivial.

5.5. Conclusions

The question of how to optimally use an address generation unit when compiling for DSP-machines is important. In this chapter we have studied the importance of scheduling and address register assignment with regards to the outcome of the offset assignment. We have compared optimal algorithms for integrated offset assignment to both optimal and heuristic algorithms that are not fully integrated.

First we studied the accumulator based architecture. We have run experiments where we compare the solutions of our fully integrated method (DP-RS) with the solutions of our separated methods (DP-S, DP-R and DP). While it is true that finding optimal results requires a lot of time and computer memory, we think that the results are important because they show exactly how much is gained, on average, by integrating the different parts of the code generation. The results of our experiments suggest that integrating the phases of the code generation has much potential for improving code when the number of address registers is 1 or 2, and if the number of address registers is more than that, only small improvements can be expected by the integration of phases.

Second, we ran experiments with a register based architecture. For this case we implemented both an integer linear programming method and a dynamic programming algorithm. The experiments show that our integer linear programming method is not as successful as the dynamic programming method. When we compare the integrated method to the non-integrated method the conclusion is about the same as for the accumulator-based case; integration is good when there are few address registers.

Chapter 6.

Related work

In this chapter we list some of the related work in the area of code generation both for basic blocks and for loops. Related work in the area of offset assignment is presented in-place in Chapter 5.

6.1. Integrated code generation for basic blocks

6.1.1. Optimal methods

Kästner [Käs00, Käs01] has developed a retargetable phase coupled code generator which can produce optimal schedules by solving a generated integer linear program in a postpass optimizer. Two integer linear programming models are given. The first one is time based, like ours, and assigns events to points in time. The second formulation is order based where the order of events is optimized, and the assignment to points in time is implicit. The advantage of the second model is that it can be flow-based such that the resources flow from one instruction to the next, and this allows for efficient integer linear program models in some cases.

Wilken et al. [WLH00] have presented an integer linear programming formulation for instruction scheduling for basic blocks. They also discuss how DAG transformations can be used to make the problem easier to solve without affecting the optimality of the solution. The machine model which they use is rather simple.

Wilson et al. [WGB94] created an integer linear programming model for the integrated code generation problem with included instruction selection. This formulation is limited to non-pipelined, single issue architectures.

A constraint programming approach to optimal instruction scheduling of superblocks[1] for realistic architecture was given by Malik et al. [MCRvB08]. The method was shown to be useful also for very large superblocks after a preprocessing phase which prunes the search space in a safe way.

An integer linear programming method by Chang et al. [CmCtK97] performs integrated scheduling, register allocation and spill code generation. Their model targets non-pipelined, multi-issue, non-clustered architectures. Spill code is integrated by preprocessing the DAG in order to insert nodes for spilling where appropriate.

Winkel has presented an optimal method based on integer linear programming formulation for global scheduling for the IA-64 architecture [Win04] and shown that it can be used in a production compiler [Win07]. Much attention is given to how the optimization constraints should be formulated to make up a tight solution space that can be solved efficiently.

The integer linear programming model presented in Chapter 3 for integrated code generation is an extension of the model by Bednarski and Kessler [BK06b]. Several aspects of our model are improved compared to it: Our model works with clustered architectures which have multiple register banks and data paths between them. Our model handles transfer instructions, which copy a value from one register bank to another (transfers do not cover an IR node of the DAG). Another improvement over this model is that we handle memory data dependences. We also allow nodes in the IR DAG that do not need to be covered by an instruction, e.g. IR nodes representing small constants that can be inlined in other target instructions using them. This is achieved by using non-issue instructions which use no resources. The new model better handles operations to which the order of the operands matters (e.g. the order of `a` and `b` in the shift-left operation `shl a,b`). Previously the solution would have to be analyzed

[1] A superblock is a block of code that has multiple exit points but only one entry point [HMC$^+$93].

afterwards to get the order of the arguments correct. We also remodeled the data flow dependences to work with the r variable (see Section 3.2.1), thus removing explicit live ranges. By removing explicit live ranges and adding cluster support with explicit transfers we make it possible to have spill code generation and scheduling integrated with the other phases of the code generation.

Within the Optimist project an integrated approach to code generation for clustered VLIW processors has been investigated by Kessler and Bednarski [KB06]. Their method is based on dynamic programming and includes safe pruning of the solution space by removing comparable partial schedules. The dynamic programming algorithm for solving the integrated offset assignment problem in Chapter 5 is inspired by the algorithm by Kessler and Bednarski; in particular the two-dimensional solution space that we use and the way that scheduling is done is similar to how it is done in [KB06].

6.1.2. Heuristic methods

Hanono and Devadas present an integrated approach to code generation for clustered VLIW architectures in the AVIV framework [HD98]. Their method builds an extended data flow graph representation of a basic block which explicitly represents all alternatives for implementation, and then uses a branch-and-bound heuristic for selecting one alternative.

Barany and Krall have presented a register allocator that integrates rescheduling [BK10]. It works together with any other scheduler and will reschedule instructions to avoid spilling. This is a compromise between having a scheduler that minimizes register need and a scheduler that optimizes pipeline usage.

Lorenz et al. implemented a genetic algorithm for integrated code generation for low energy use [LLM$^+$01] and for low execution time [LM04]. This genetic algorithm includes instruction selection, scheduling and register allocation in a single optimization problem. It also takes the subsequent address code generation, with address generation units, into account. In a preprocessing step additional IR nodes are inserted in the DAG which represent possible explicit transfers between register files. Each gene corresponds to a node in the DAG and an individual translates directly to a schedule. This is different from the algorithm that we propose in Chapter 3 where the genes are seen as preferences when creating the schedule.

Beaty did early work with genetic algorithms for the instruction scheduling problem [Bea91]. Other notable heuristic methods that integrate several phases of code generation for clustered VLIW have been proposed by Kailas et al. [KEA01], Özer et al. [OBC98], Leupers [Leu00b] and Nagpal and Srikant [NS04].

6.2. Integrated software pipelining

6.2.1. Optimal methods

An enumeration approach to software pipelining, based on dynamic programming, was given by Vegdahl [Veg92]. In this algorithm the dependence graph of the original loop body is replicated by a given factor, with extra dependences to the new nodes inserted accordingly. The algorithm then creates a compacted loop body in which each node is represented once, thus the unroll factor determines how many iterations a node may be moved. This method does not include instruction selection and register allocation.

Blachot et al. [BdDH06] have given an integer linear programming formulation for integrated modulo scheduling and register assignment. Their method, named Scan, is a heuristic which searches the solution space by solving integer linear programming instances for varying initiation intervals and numbers of schedule slots in a way that resembles our algorithm in Section 4.2. Their presentation also includes an experimental characterization of the search space, e.g. how the number of schedule slots and initiation intervals affects tractability and feasibility of the integer linear programming instance.

Yang et al. [YGGT02] presented an integer linear programming formulation for rate- and energy-optimal modulo scheduling on an Itanium-like architecture, where there are fast and slow functional units. The idea is that instructions that are not critical can be assigned to the slow, less energy consuming, functional units thereby optimizing energy use. Hence, this formulation includes a simple kind of instruction selection.

Ning and Gao [NG93] present a method for non clustered architectures where register allocation is done in two steps, the first step assigns temporary values to buffers and the second step does the actual register allocation. Our method is different in that it avoids the intermediate step. This is an advantage when we want to support clustered register banks and integrate spill code generation.

Altman et al. [AGG95] presented an optimal method for simultaneous modulo scheduling and mapping of instructions to functional units. Their method, which is based on integer linear programming, has been compared to a branch and bound heuristic by Ruttenberg et al. [RGSL96].

Fimmel and Müller do optimal modulo scheduling for pipelined non-clustered architectures by optimizing a rational initiation interval [FM00, FM02]. The initiation interval is a variable in the integer linear programming formulation, which means that only a single instance of the problem needs to be solved as opposed to the common method of solving with increasingly large initiation intervals.

Eichenberger et al. have formulated an integer linear programming model for minimizing the register pressure of a modulo schedule where the modulo reservation table is fixed [EDA96].

Cortadella et al. have presented an integer linear programming method for finding optimal modulo schedules [CBS96]. Nagarakatte and Govindarajan [NG07] formulated an optimal method for integrating register allocation and spill code generation. These formulations work only for non-clustered architectures and do not include instruction selection.

Eisenbeis and Sawaya [ES96a] describes an integer linear programming method for integrating modulo scheduling and register allocation. Their method gives optimal results when the number of schedule slots is fixed.

Fan et al. [FKPM05] studied the problem of synthesizing loop accelerators with modulo scheduling. In their problem formulation the initiation interval is given and the optimization problem is to minimize the hardware cost. They present optimal methods based on integer linear programming and branch and bound algorithms. They also show and evaluate several methods for decomposing large problem instances into separate subproblems to increase tractability at the cost of global optimality.

6.2.2. Heuristic methods

Fernandes was the first who described *clustered* VLIW architectures [Fer98], and in [FLT99] Fernandes et al. gave a heuristic method for modulo scheduling for such architectures. The method is called *Distributed modulo scheduling* (DMS) and is shown to be effective for up to 8 clusters. DMS integrates modulo scheduling and cluster partitioning in a single phase. The method first tries to put instructions that are connected by true data dependences on the same cluster. If that is not possible transfer instructions are inserted or the algorithm backtracks by ejecting instructions from the partially constructed schedule.

Huff [Huf93] was the first to create a heuristic modulo scheduling method that schedules instructions in a way that minimizes life times of intermediate values. The instructions are given priorities based on the number of slots on which they may be scheduled and still respect dependences. The algorithm continues to schedule instructions with highest priority either early or late, based on heuristic rules. If an instruction can not be scheduled, backtracking is used, ejecting instructions from the partial schedule.

Another notable heuristic, which is not specifically targeted for clustered architectures, is due to Llosa et al. [LVAG95, LGAV96]. This heuristic, called Swing modulo scheduling, simultaneously tries to minimize the initiation interval and register pressure by scheduling instructions either early or late.

The heuristic by Altemose and Norris [AN01] does register pressure responsive modulo scheduling by inserting instructions in such a way that known live ranges are minimized.

Stotzer and Leiss [SL99] presented a backtracking heuristic for modulo scheduling after instruction selection for Texas Instruments C6x processors. And later [SL09] they have presented heuristic methods for avoiding long lifetimes in modulo schedules. The idea is that if lifetimes of values are shorter than *II* then modulo variable expansion is not necessary. We use the same idea in Section 4.3.4, and our conclusions are similar to the conclusions that Stotzer and Leiss draw; limiting the live range is good since modulo variable expansion can be avoided and the *II* is most of the time not affected.

Nystrom and Eichenberger presented a heuristic method for cluster assignment as a prepass to modulo scheduling [NE98]. Their machine model assumes that all transfer instructions are explicit. The clustering algorithm prioritizes operations in critical cycles of the graph, and tries to minimize the number of transfer instructions while still having high throughput. The result of the clustering prepass is a new graph where operations are assigned to clusters and transfer nodes are inserted. Their experimental evaluation shows that, for architectures with a reasonable number of buses and ports, the achieved initiation interval is most of the time equal to the one achieved with the corresponding fully connected, i.e. non-clustered, architecture, where more data ports are available.

A heuristic method for integrated modulo scheduling for clustered architectures was presented by Codina et al. [CSG01]. The method, which also integrated spill code generation is shown to be useful for architectures with 4 clusters.

Pister and Kästner presented a retargetable method for postpass modulo scheduling implemented in the Propan framework [PK05].

Zalamea et al. have created a backtracking integrated software pipelining heuristic for clustered VLIWs [ZLAV01]. In this heuristic the subproblems that are integrated are: scheduling, spilling, cluster assignment and register allocation. Experiments shows that this heuristic works well for up to as many as 8 clusters.

6.2.3. Discussion of related work

The work presented in Chapters 3 and 4 in this thesis is different from the ones mentioned above in that it aims to produce provably optimal modulo schedules, also when the optimal initiation interval is larger than *MinII*, and in that it integrates also cluster assignment and instruction selection in the formulation.

Creating an integer linear programming formulation for clustered architectures is more difficult than for the non-clustered case since the common method of modeling live ranges simply as the time between definition and use can not be applied. Our formulation does it instead by a method where values are explicitly assigned to register banks for each time slot. This increases the size of the solution space, but we believe that this extra complexity is unavoidable and inherent to the problem of integrating cluster assignment and instruction selection with the other phases.

6.2.4. Theoretical results

Touati [Tou02, Tou07] presented several theoretical results regarding the register need in modulo schedules. One of the results shows that, in the absence of resource conflicts, there exists a finite schedule duration ($t_{\max}$ in our terminology) that can be used to compute the minimal periodic register sufficiency of a loop for all its valid modulo schedules. Theorem 10 in this thesis is related to this result of Touati. We assume unbounded register files and identify an upper bound on schedule duration, in the presence of resource conflicts.

In a technical report Eisenbeis and Sawaya [ES96b] give a theoretical result for an upper bound on the number of schedule slots for integrated modulo scheduling and register allocation (with our notation):

$$t_{\max} \leq (|V| - 1)\left(\hat{L} + II - 1\right) + |V| + 1$$

where $\hat{L}$ is the largest latency of any instruction. Our proof of Theorem 10 in Chapter 4 is similar to their proof. Our theorem provides a tighter bound when the instructions have higher variance of latencies, but their theorem is tighter if the initiation interval is large. However, we believe that their result is incorrect because it does not handle loop carried dependences correctly[2]. A minimal counterexam-

[2]We believe that they need to add the same assumption that we have on unbounded register file sizes.

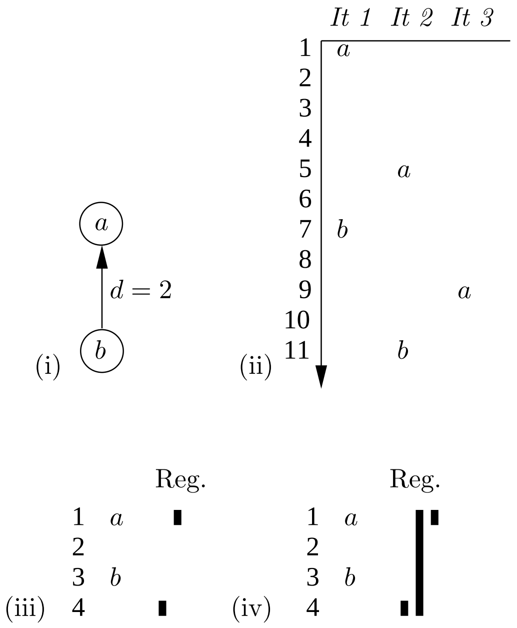


Figure 6.1: The graph in (i) can be modulo scheduled with initiation interval 4 as shown in (ii). If the schedule of an iteration is shortened by 4 cycles the register pressure of the corresponding modulo schedule kernel increases, see (iii) to (iv).

ple is depicted in Figure 6.1. The example consists of a graph with two instructions, *a* and *b*, both with latency 1. The value produced by *b* is consumed by *a* two iterations later. Then, if the initiation interval is 4 the schedule shown in Figure 6.1 can not be shortened by 4 cycles as described in [ES96b], since this would increase the live range of *b* and hence increase the register pressure of the resulting modulo schedule.

Chapter 7.

Possible extensions

In this chapter we will identify a few possible directions for future work.

7.1. Integration with a compiler framework

The IR-graphs used for the evaluation of the modulo scheduling algorithm in this thesis have two sources: some were created by hand, and some are translated from a set of graphs from the st200cc compiler. If our methods were integrated in a proper, state of the art, compiler it would allow more systematic testing of complete programs. We are currently limited to basic blocks and inner loops.

Much of our work is based on Optimist which uses the LCC [FH95] front-end. But LCC is quite old and simplistic; for instance, it does not build graphs with loop carried data dependences and it includes very few machine independent optimizations. We would like to integrate our algorithms with another, more modern, compiler such as GCC [Fre], LLVM [LA04], Trimaran [Tri] or Open64 [Ope].

An alternative, that is easier to implement, is to make a post-pass tool that is run after the compiler is finished. One advantage of such an approach is that the results are very easy to compare. And the big disadvantage is that the information that is collected throughout the analysis phases of the compiler is lost; the only information left is the final code. I.e. much of the analysis effort is doubled, and with incomplete information we may constrain the post-pass tool too much by introducing false constraints.

7.2. Benders decomposition

As a topic for future work we will consider applying Benders decomposition [Ben62] to the separated integer linear programming algorithm of Chapter 4. Benders decomposition is a method of decomposing an optimization problem into smaller parts. These smaller parts can then be solved to optimality, and for each solution that is found part of the search space is removed from the subproblems. When all of the search space has been cut away optimality is guaranteed. The key to a good Benders decomposition is to find good cuts. We have already divided the code generation problems into subproblems, so the only thing that remains is to create good Benders cuts and we can solve the integrated problem by solving many small problems.

A general decomposable optimization problem can be expressed like this:

$$\begin{aligned} &\min f(\mathbf{x},\mathbf{y}) \\ &s.t.\ (\mathbf{x},\mathbf{y}) \in S \end{aligned} \tag{7.1}$$

Benders decomposition uses the fact that this problem can be decomposed by solving for only a subset $\mathbf{y}$ of the variables:

$$\begin{aligned} &\min g(\mathbf{y}) \\ &s.t.\ \mathbf{y} \in S' \end{aligned} \tag{7.2}$$

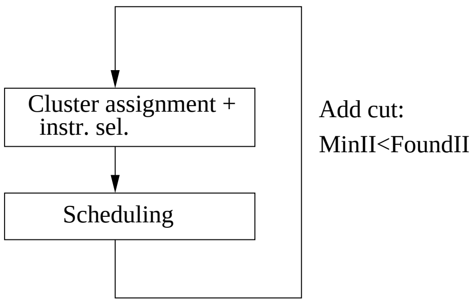


Figure 7.1: Example of a cut for software pipelining.

where g is a new optimization goal to the subproblem where the $\mathbf{x}$ variables are not included. An optimal solution to (7.2) is not necessarily part of an optimal solution to (7.1).

Now if the problem defined by (7.2) is solved to optimality for $\mathbf{y} = \overline{\mathbf{y}}$, we can use this partial solution in the full problem and solve for the remaining variables ($\mathbf{x}$):

$$\begin{aligned} &\min f(\mathbf{x}, \overline{\mathbf{y}}) \\ &s.t.\ (\mathbf{x}, \overline{\mathbf{y}}) \in S \end{aligned} \tag{7.3}$$

Now, if (7.3) has a solution we store it. But we can still not guarantee that this solution is optimal. So, what we do is to generate *Benders cuts* where we restrict the search space of (7.1) and (7.2) so that this particular solution is removed. Also for Benders decomposition to be efficient we must make it so that the Benders cuts remove parts of the search space where we know there can not exist better solutions than the one we have found, for instance using symmetries or domain knowledge. This process is iterated until there are no feasible solutions to (7.2). At that point we know that the best solution that we have found so far is the optimal solution to the full problem.

If we apply Benders decomposition to our software pipelining problem in Chapter 4 the first subproblem can be instruction selection, where we solve for *MinII*, and the second subproblem can be the rest of the code generation. One obvious Benders cut that can be generated once a solution is found is to remove all instruction selections that have a lower bound on *II* that is not lower than the *II* that we have already found, see Figure 7.1.

One big advantage of using the Benders decomposition technique is that we can bring in domain knowledge into general optimization methods. Another advantage is that not all subproblems need to be solved with the same optimization technique. It would be possible to do, for instance, scheduling with constraint logic programming and the rest of the code generation problem can be solved with integer linear programming.

7.3. Improving the theoretical results

We also want to find ways to relax the constraints on the machine model in the theoretical part of Chapter 4. One limitation is that we require the generated code to be transfer free, i.e., no instructions are generated that do not cover a node in the intermediate representation graph. One possible way to get around this limitation is to add transfer-nodes in the intermediate representation graph. The obvious question then is how many transfer nodes we must add? How do we limit the number of times a value may be transferred? A reasonable guess is that we could add transfer nodes before each use of a value, but then there is the question if this affects the generality of the model. This transformation might exclude optimal solutions where a value is "juggled" back and forth between register files.

The other limitation is in the number of available registers. In the theoretical parts we assume that register pressure is never the limiting factor. This assumption is necessary because in some cases reducing the number of schedule slots of a modulo schedule leads to an increase in the register pressure. It would be interesting to investigate how often this situation occurs in *real* examples. Also, under which circumstances does it occur and can we find a bound on how much the register pressure can increase. Touati has done much work in the area of register pressure in modulo schedules [Tou02], we should investigate if his methods can be applied to refine our results on the upper bound on the number of schedule slots.

7.4. Genetic algorithms for modulo scheduling

One natural extension to the work presented here is to modify the genetic algorithm of Section 3.3 to modulo scheduling. There are several ways in which this could be done. One way would be to keep much of the existing genetic algorithm from Section 3.3 and use a fitness function that penalizes infeasible solutions. For instance, if a produced acyclic schedule has a resource conflict when converted into a modulo schedule it has a negative fitness value. Then the goal of the genetic algorithm is to produce an individual with no conflicts, i.e., with fitness 0. But since the genetic algorithm uses a sort of list scheduling for producing the acyclic schedule, and we know that locally compacted schedules are not necessarily good when producing modulo schedules we would need a way to add slack into the genes describing an individual.

7.5. Improve the solvability of the integer linear programming model

When solving integer linear program instances the integrality constraint on solution variables is usually dropped and the relaxed problem is solved using a simplex algorithm. If the solution to the relaxed problem happens to be integer we are done, but if the relaxed solution is not integral we need to add constraints that remove this solution from the solution space, e.g. by branching on a fractional binary variable. If we can make sure that the constraints of the model yield a solution space where as many corners as possible are integral, the time required to solve the problem becomes smaller.

Finding ways to tighten our solution space is a topic for future work. One possible starting point could be to investigate if the methods used by Winkel [Win04] for global scheduling can be applied also to modulo scheduling.

7.5.1. Reformulate the model to kernel population

In our experiments we have found that one problem of our integer linear programming model for modulo scheduling is that it does not scale well when the the number of schedule slots is increased. In many cases increasing the number of schedule slots by just 1 can increase the solution time by several orders of magnitude. Our model, like many others, creates a modulo schedule by adding constraints to an acyclic schedule. We should try methods where instructions from different iterations are chosen to populate the kernel, similar to Vegdahl's method with dynamic programming [Veg92]. This would remove the problem with the number of schedule slots for the corresponding acyclic schedule. Instead we would have new problems, such as how many iterations to consider when creating the kernel. It is not clear if such a method would lead to better solution times compared to our current method.

Chapter 8.

Conclusions

It is very common that the problem of code generation is divided into subproblems that are solved one a a time in a compiler. This division into subproblems can lead to missed chances for optimization because a solution to the first subtask may impose constraints on the subsequent tasks. It is not known in which order the subtasks should be solved to give the best results. In this thesis we have presented implementations for optimal integrated, as well as non-integrated, code generation problems. The purpose of these implementations is to study the effects of the integration. In particular, we want to find how much can be gained by integrating the subtasks of code generation.

We have created integrated algorithms for 3 different code generation problems: First for basic blocks with clustered VLIW as the target; the phases that are integrated are: cluster assignment, instruction selection, scheduling, register allocation and spilling. Second, we have extended this method to software pipelining of loops. And the third code generation problem that we have considered is offset assignment where we simultaneously solve scheduling, stack layout and address register usage.

Our algorithm for modulo scheduling iteratively considers schedules with increasing number of schedule slots. A problem with such an iterative method is that, if the initiation interval is not equal to the lower bound, there is no way to determine whether the found solution is optimal or not. We have proven that, for a class of architectures that we call transfer free, we can set an upper bound on the schedule length. I.e. we can prove optimality also when the *II* is larger than the calculated value of the lower bound.

Creating an integer linear programming formulation for clustered architectures is more difficult than for the non-clustered case since the common method of modeling live ranges simply as the time between definition and use of the value cannot be applied. Our formulation handles live ranges by explicitly assigning values to register banks for each time slot. This increases the size of the solution space, but we believe that this extra complexity is unavoidable and inherent to the problem of integrating cluster assignment and instruction selection with the other phases.

We have also shown that optimal spilling is closely related to optimal register allocation when the register files are clustered. In fact, optimal spilling is as simple as adding an additional virtual register file representing memory and have transfer instructions to and from this register file corresponding to stores and loads.

In our experimental evaluation we have shown that for the basic block case we can optimally generate code for DAGs with up to 191 IR nodes in less than 30 seconds. For modulo scheduling we compare the integrated method to one in which instruction selection and cluster assignment is done in a separate phase. Our experiments show that the integrated method rarely results in better results than the separated one for the single cluster architecture, but for the double cluster architecture the integrated method beats the separated one in 39% of the cases, and in these cases, assuming a large number of iterations, the average speedup is 20%.

For the offset assignment problem we have developed a dynamic programming algorithm for optimal integrated offset assignment including scheduling. In the experiments we found that the integration of the subtasks in this case is beneficial when there are 1 or 2 address registers. We also developed an integer linear programming formulation, but the results of this method were not encouraging. In none of our tests did we find that the integer linear programming method was better than the dynamic programming approach. Hence, we find that integer linear programming is better than dynamic programming in our code generation for clustered VLIW architectures, but dynamic programming is better than integer linear programming in the offset assignment problem.

The results of these experiments are important because we try to find out if the integration *can* be beneficial: all steps in the separated methods produce locally optimal results and the results of integrated method are globally optimal. Showing how often the integrated method is better than the separated one is interesting also from a theoretical perspective.

Appendix A.

Complete integer linear programming formulations

In this appendix we give verbatim listings of the integer linear program model that we have used for the experiments. We also show the dot product problem specification as an example. The listings are shown in AMPL syntax. The architecture parameters are much too verbose to be listed here.

A.1. Integrated software pipelining in AMPL

This section contains the complete software pipelining model that we have used in the experiments in Chapter 4.

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

# DFG nodes
set G;

# DFG edges
## Edges for src1 (KID0)
set EG0 within (G cross G cross Integers);
## Edges for src2 (KID1)
set EG1 within (G cross G cross Integers);
## Edges for data dependences (DDEP)
set EGDEP within (G cross G cross Integers);
set EG := EG0 union EG1;
```

```
# 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_nonissue; # patterns that does not use resources
set P := P_prime union P_second union P_nonissue;

set PN; # 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} default 0;
# Latencies
param L {P} default 0; #nonissue have 0, the rest are listed

set match{i in G} within P
  default setof{p in P, k in PN : OPP[p,k] = OPG[i]} p;

set dagop := setof{i in G} OPG[i];

set P_pruned :=
  setof {p in P, pn in PN: OPP[p,pn] in dagop} (p);

param min_lat {i in G} :=
  min {p in P, k in PN: k in B[p] && OPP[p,k] == OPG[i]}
          if p in P_prime then 0 else L[p];

param max_lat {i in G} :=
  max {p in P, k in PN: k in B[p] && OPP[p,k] == OPG[i]} 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;

#Register banks
set RS;

# Resources for transfers. Is 1 if transfer RS-RS requires F.
param UX{RS,RS,F} binary default 0;
# Latencies for transfers
param LX{RS,RS} integer > 0 default 10000000;

# Register banks of pattern variables
## Destinations
set PRD{RS} within P;
## First argument source
set PRS1{RS} within P;
## Second argument source
set PRS2{RS} within P;

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

# Issue width (omega)
param W integer > 0;
# Number of registers
param R{RS} integer > 0 default 1000000;

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

# Maximum time
param max_t integer > 0;
set T := 0 .. max_t;
# Increase II until a feasible solution exists.
param II integer >= 1;
param soonest {i in G} in T default 0;
param latest  {i in G} in T default 0;
param derived_soonest {i in G} :=
  max(soonest[i],
      max {(a,delta) in
       setof { (a,ii,ddelta) in (EG union EGDEP):
                                 ii = i && ddelta != 0
                                }
                                 (a,ddelta)}
          (soonest[a] + min_lat[a] - II*delta));

param derived_latest {i in G} :=
  min(latest[i],
      min {(a,delta) in
       setof { (ii,a,ddelta) in (EG union EGDEP):
                                 ii = i && ddelta != 0
                                }
                                 (a, ddelta)}
          (latest[a] - min_lat[i] + II*delta));


# Slots on which i in G may be scheduled, override
# derived_latest if the node can be covered by a
# nonissue pattern.
set slots {v in G} :=
  derived_soonest[v]
    ..
  min(derived_latest[v],
      prod{p in P_nonissue}
        prod{k in B[p]} abs(OPG[v]-OPP[p,k]));

# max_d is largest distance of a dependence in EG0/1
param max_d integer >= 0;

set TREG := 0 .. (max_t+II*max_d);

var c {i in G, match[i], PN, slots[i]} binary default 0;

var w {(i,j,d) in EG,
       p in P_prime inter match[i] inter match[j],
       slots[i] inter slots[j],
       EP[p]
      } binary default 0;
```

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

# Transfer
var x {i in G,RS,RS,TREG} binary default 0;
# Availability
var r {RS, v in G, t in TREG: t >= derived_soonest[v]}
  binary default 0;

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

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

# Used for basic block scheduling only.

#var exec_time integer;

#minimize Test:
#  exec_time;

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

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

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

# Record which patterns have been selected at time t
subject to Selected {p in P_prime inter P_pruned,
                     t in T,
                     k in B[p]}:
```

```
  (sum{i in G:t in slots[i] && p in match[i]} c[i,p,k,t]) = s[p,t];

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

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

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

#limit on availability
subject to AvailabilityLimit {rr in RS,
                              i in G,
                              t in TREG
                             :t>=derived_soonest[i]
                             }:
               #just ready
  r[rr,i,t]
    <=
  sum{p in (PRD[rr] inter match[i]): (t-L[p]) in slots[i]}
    sum{k in B[p]}
      c[i,p,k,t-L[p]] +
      #available in prev. time step
      sum{tt in t..t : tt-1>=derived_soonest[i]}
        r[rr,i,tt-1] +
      #just transfered
      sum{rs in RS:(t-LX[rs,rr])>=0} x[i,rs,rr,t-LX[rs,rr]];

# Make sure inner nodes of a pattern are never visible
```

```
subject to AvailabilityLimitPattern
  {rr in RS,
   (i,j,d) in EG,
   p in match[i] inter match[j] inter P_prime,
   tp in slots[i] inter slots[j],
   tr in TREG,
   (k,l) in EP[p]
   : tr>=derived_soonest[i]
  }:
  r[rr,i,tr] <= (1-w[i,j,d,p,tp,k,l]);


# Data must be available when we use it. For now, we only
# handle patterns that take all operands from the same
# register bank as the destination (PRD).
subject to ForcedAvailability {(i,j,d) in EG,
                               t in slots[j],
                               rr in RS
                              }:
  r[rr,i,t+d*II]
    >=
  sum{p in PRD[rr] inter match[j] inter P_prime}
    sum{k in B[p]}
      (c[j,p,k,t]
#but not if (i,j) is an edge in p
       -
       sum{(k,l) in EP[p] : p in match[j] inter match[i]
                            && t in slots[i]
                            && t in slots[j]}
          w[i,j,d,p,t,k,l]);

# Singletons
subject to ForcedAvailability2
  {(i,j,d) in EG0,
   t in slots[j],
   rr in RS}:
  r[rr,i,t+d*II]
    >=
  sum{p in PRS1[rr] inter match[j] inter P_second}
    sum{k in B[p]}
      c[j,p,k,t];

subject to ForcedAvailability3
  {(i,j,d) in EG1,
   t in slots[j],
   rr in RS}:
  r[rr,i,t+d*II]
    >=
  sum{p in PRS2[rr] inter match[j] inter P_second}
    sum{k in B[p]}
      c[j,p,k,t];

# Must also be available when we transfer
subject to ForcedAvailabilityX
  {i in G,
   t in TREG,
   rr in RS}:
  sum{tt in t..t: tt >= derived_soonest[i]}
    r[rr,i,tt]
    >=
  sum{rq in RS} x[i,rr,rq,t];

subject to TightDataDependences
  {(i,j,d) in EGDEP,
   t in TREG}:
  sum{p in match[j]}
    sum{tt in 0..t-II*d: tt in slots[j]}
      c[j,p,0,tt]
  +
  sum{p in match[i]}
    sum{ttt in t-L[p]+1..max_t + II*max_d:
        ttt>0
        && ttt in slots[i]}
      c[i,p,0,ttt]
    <=
  1;

# Check that the number of registers is not exceeded at any
# time
subject to RegPressure {t_offs in 0..(II-1), rr in RS}:
  sum{i in G}
    sum{t in t_offs..(max_t+II*max_d) by II:
        t>=derived_soonest[i]}
      r[rr,i,t]
    <=
  R[rr];

subject to LiveRange
  {rr in RS,
   v in G,
   t in TREG:
   t >= derived_soonest[v]
  }:
  sum{tt in t..t+II:
       tt in TREG
       && tt >= derived_soonest[v]} r[rr,v,tt]
    <=
  II
  +
  sum{tt in t..t+II-1:
       tt in TREG && tt >= derived_soonest[v]}
    x[v,rr,rr,tt];

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

# At each scheduling step we should not exceed the
# number of resources
subject to Resources {t_offs in 0..(II-1), f in F}:
  sum{t in t_offs..max_t by II}(
    sum{p in P_prime inter P_pruned : U[p,f] = 1} s[p,t]
  +
  sum{p in P_second inter P_pruned: U[p,f] = 1}
    sum{i in G:t in slots[i]
               && p in match[i]}
      sum{k in B[p]} c[i,p,k,t])
  +
  sum{t in t_offs..(max_t+II*max_d) by II}(
    sum{i in G} sum{(rr,rq) in (RS cross RS):
         UX[rr,rq,f] = 1 }
      x[i,rr,rq,t])
    <=
  M[f];

# At each time slot, we should not exced the issue width w
subject to IssueWidth {t_offs in 0..(II-1)}:
  sum{t in t_offs..max_t by II}(
    sum{p in P_prime inter P_pruned} s[p,t]
  +
  sum{p in P_second inter P_pruned}
    sum{i in G:t in slots[i] && p in match[i]}
```

```
      sum{k in B[p]} c[i,p,k,t])
  +
  sum{t in t_offs..(max_t+II*max_d) by II}(
   sum{i in G} sum{(rr,rq) in (RS cross RS)} x[i,rr,rq,t])
    <=
  W;

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

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

# Each pattern should be associated with some functional unit
check {p in P_prime union P_second}:
  sum{f in F} U[p,f] > 0;

end;
```

A.2. Dot product ddg

```
# 5 Loads *5
# 1 MPY *2
# 5 Ops *1
# sum 32.
# MinII = single 5, double 3

param max_d := 1;

param: G : OPG :=
  2  4405  # ADDI4
  4  4565  # MULI4
  5  4165  # INDIRI4
  6  4407  # ADDP4
  7  4437  # LSHI4
  8  4165  # INDIRI4
  9  4391  # ADDRLP4
  10 4117  # CNSTI4
  11 4167  # INDIRP4
  12 4359  # ADDRGP4
  13 4165  # INDIRI4
  14 4407  # ADDP4
  15 4167  # INDIRP4
  16 4359  # ADDRGP4
;

set EG0 :=
  (2,2,1) (9,8,0) (8,7,0) (7,6,0) (12,11,0) (6,5,0) (5,4,0)
  (7,14,0) (16,15,0) (14,13,0);

set EG1 :=
  (10,7,0) (11,6,0) (15,14,0) (13,4,0) (4,2,0);

set EGDEP :=
;
```

A.3. Integrated offset assignment model

This section contains the complete integrated offset assignment model that we have used in the experiments in Section 5.4.

```
param k default 1;
param n;
param nrvar;
param reg default 2;
param aguco default 1;
param aluco default 1;

set Vert := 0..n-1;
set T := 0..n-1;
set AGU := 0..k-1;
set Var := 1..nrvar;
set Loc := 0..nrvar-1;

set E within Vert cross Vert;
set Dep within Vert cross Vert;
param n2v {Vert} within Var default 0;

param soonest {Vert} default 0;
param latest {Vert} default n-1;
set Slots {x in Vert} = soonest[x]..latest[x];

var S{t in T, x in Vert, AGU: t in Slots[x]} binary;
var r{T, AGU, Var} binary;
var L{Loc, Var} binary;
var C{T, AGU} binary;
var Calu{T} binary;

# helper parameter, isload[x] > 0 iff x is a load.
param isload {x in Vert} := sum{(s,d) in E:s==x} 1;
var vl{t in T,
       x in Vert:
         t in soonest[x]+1..n-2 &&
         isload[x] > 0
       } binary; #vertex live
var A{t in T,
      d in Vert:
         isload[d] == 0 &&
         t in 0..latest[d]
      } binary;

minimize Cost:
  aguco*sum{i in AGU, t in T} C[t, i] +
  aluco*sum{t in T} Calu[t];

#All stores must have an alu operation.
subject to AluOp {d in Vert: isload[d] == 0}:
  sum{t in 0..latest[d]} A[t,d] == 1;

#On each time slot we can only do one alu operation.
subject to OneAluAtATime {t in T}:
  sum{d in Vert:
        isload[d] == 0 &&
        t in 0..latest[d]
      } A[t,d] <= 1;

subject to Scheduling {(x,y) in E union Dep, t in T}:
  sum{tp in T:
        tp <= t &&
        tp in Slots[y]
      } sum {i in AGU} S[tp, y, i]
  +
  sum{tp in T:
        tp >= t &&
        tp in Slots[x]
      } sum {i in AGU} S[tp, x, i]
  <=
  1;
```

```
#Aluop must not come before load
subject to SchedulingAlu1 {t in T, (s,d) in E}:
  sum{tp in T: tp <= t && tp in 0..latest[d]} A[tp, d]
  +
  sum{tp in T: tp >= t && tp in Slots[s]}
    sum{i in AGU} S[tp, s, i]
  <=
  1;

#Aluop must not come after store
subject to SchedulingAlu2 {t in T, (s,d) in E}:
  sum{tp in T:
        tp <= t &&
        tp in Slots[d]
      } sum{i in AGU} S[tp, d, i]
  +
  sum{tp in T: tp > t && tp in 0..latest[d]} A[tp, d]
  <=
  1;


subject to AddressRegister {t in T, v in Var, i in AGU}:
  r[t, i, v]
  >=
  sum {x in Vert: n2v[x] = v && t in Slots[x]} S[t, x, i];

subject to CostTimeAGU {t in T,
                        p in Loc,
                        v in Var,
                        w in Var,
                        i in AGU :
                          t!=n-1 && v!=w}:
  C[t, i]
  >=
  L[p, v] + r[t, i, w] + r[t+1, i, v] - 2
  - sum{1..1: p-1 in Loc} L[p-1,w]
  - sum{1..1: p+1 in Loc} L[p+1,w];

subject to ChangePossible {i in AGU, t in T, v in Var:
                           t != n-1}:
  r[t, i, v]
  +
  sum {w in Var : w!=v} r[t+1, i, w]
  -
  sum {x in Vert: t in Slots[x]} S[t, x, i]
  <=
  1;

subject to AllNodes{x in Vert}:
  sum {i in AGU, t in Slots[x]} S[t, x, i] = 1;

subject to OneNodeAtATime{t in T}:
  sum {x in Vert, i in AGU : t in Slots[x]} S[t, x, i] = 1;

subject to RegUse {t in T, i in AGU}:
  sum {v in Var} r[t, i, v] = 1;

subject to LocUse {p in Loc}:
  sum {v in Var} L[p, v] = 1;

subject to VarLoc {v in Var}:
  sum {p in Loc} L[p, v] = 1;

# alucost

subject to CostTimeALU {t in T, (s,d) in E:
                        t in Slots[d] && t-1 in Slots[s] }:
  Calu[t]
  >=
  sum {i in AGU} (S[t-1, s, i] + S[t, d, i])
  + A[t-1, d] + A[t, d] - 2;

subject to VertLive {t in T,
                     (s1, d1) in E:
                       t in soonest[s1]+1..n-2}:
  vl[t,s1]
  >=
  sum{i in AGU}
    sum{tp in T: tp<t && tp in Slots[s1]} S[tp, s1, i]
  +
  sum{tp in T: tp>=t && tp in 0..latest[d1]} A[tp, d1]
  -
  1;

subject to NrLive {t in T}:
  sum {s in Vert:
         isload[s]>0 &&
         t in soonest[s]+1..n-2
       } vl[t,s]
  <=
  reg;

subject to StoreOrder {t in T, (s1,d1) in E, (s2,d2) in E
        : s1!=s2 && d1!=d2 && t in Slots[d2]}:
  sum{tp in T: tp < t && tp in 0..latest[d1]} A[tp,d1]
  +
  sum{i in AGU} S[t, d2, i]
  +
  sum{i in AGU}
    sum{tp in T: tp > t && tp in Slots[d1]} S[tp, d1, i]
  <= 2;

subject to SymCut:
  sum {p in Loc : p <= (nrvar-1.0)/2} L[p,1] = 1;
```

Bibliography

[Add10] Address-code-optimization.org. Address code optimization webpage. http://www.address-code-optimization.org/, 2010.

[AEBS08] Hesham S. Ali, Hatem M. El-Boghdadi, and Samir I. Shaheen. A new heuristic for SOA problem based on effective tie break function. In *SCOPES '08: Proc. of the 11th int. workshop on Software & compilers for embedded systems*, pages 53–59, 2008.

[AGG95] Erik R. Altman, R. Govindarajan, and Guang R. Gao. Scheduling and mapping: Software pipelining in the presence of structural hazards. In *Proc. SIGPLAN '95 Conf. on Programming Language Design and Implementation*, pages 139–150, 1995.

[AJLA95] Vicki H. Allan, Reese B. Jones, Randall M. Lee, and Stephen J. Allan. Software pipelining. *ACM Comput. Surv.*, 27(3):367–432, 1995.

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison Wesley, August 2006.

[AN88] A. Aiken and A. Nicolau. Optimal loop parallelization. *SIGPLAN Not.*, 23(7):308–317, 1988.

[AN01] Glenn Altemose and Cindy Norris. Register pressure responsive software pipelining. In *SAC '01: Proceedings of the 2001 ACM symposium on Applied computing*, pages 626–631, New York, NY, USA, 2001. ACM.

[Ana90] Analog Devices. *ADSP-2100 Family User's Manual*, 1990.

[ARK00] Sunil Atri, J. Ramanujam, and Mahmut T. Kandemir. Improving offset assignment on embedded processors using transformations. In *HiPC '00: Proceedings of the 7th International Conference on High Performance Computing*, pages 367–374, London, UK, 2000. Springer-Verlag.

[ATJ09] Samir Ammenouche, Sid-Ahmed-Ali Touati, and William Jalby. On instruction-level method for reducing cache penalties in embedded VLIW processors. In *HPCC '09: Proceedings of the 2009 11th IEEE International Conference on High Performance Computing and Communications*, pages 196–205, Washington, DC, USA, 2009. IEEE Computer Society.

[Bar92] David H. Bartley. Optimizing stack frame accesses for processors with restricted addressing modes. *Softw. Pract. Exp.*, 22(2):101–110, 1992.

[BdDH06] F. Blachot, Benoît Dupont de Dinechin, and Guillaume Huard. SCAN: A heuristic for near-optimal software pipelining. In *European conference on Parallel Computing (EuroPar) Proceedings*, pages 289–298, 2006.

[Bea91] Steven J. Beaty. Genetic algorithms and instruction scheduling. In *MICRO 24: Proceedings of the 24th annual international symposium on Microarchitecture*, pages 206–211, New York, NY, USA, 1991. ACM.

[Bed06] Andrzej Bednarski. *Integrated Optimal Code Generation for Digital Signal Processors*. PhD thesis, Linkoping University, 2006.

[Ben62] J. F. Benders. Partitioning procedures for solving mixed-variables programming problems. *Numerische Mathematik*, 4:238–252, 1962. 10.1007/BF01386316.

[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, 1989.

[BK06a] Andrzej Bednarski and Christoph Kessler. Integer linear programming versus dynamic programming for optimal integrated VLIW code generation. In *12th Int. Workshop on Compilers for Parallel Computers (CPC'06)*, January 2006.

[BK06b] Andrzej Bednarski and Christoph W. Kessler. Optimal integrated VLIW code generation with integer linear programming. In *Proc. Euro-Par 2006*, pages 461–472. Springer LNCS 4128, Aug. 2006.

[BK10] Gergo Barany and Andreas Krall. Optimistic integrated instruction scheduling and register allocation. In *CPC '10: Proceedings of the 15th Workshop on Compilers for Parallel Computing*, 2010.

[Boo] Boost C++ Libraries. Boost project homepage. http://www.boost.org.

[CBS96] Jordi Cortadella, Rosa M. Badia, and Fermín Sánchez. A mathematical formulation of the loop pipelining problem. In *XI Conference on Design of Integrated Circuits and Systems*, pages 355–360, Barcelona, 1996.

[CCK88] David Callahan, John Cocke, and Ken Kennedy. Estimating interlock and improving balance for pipelined machines. *Journal of Parallel and Distributed Computing*, 5:334–358, 1988.

[Cha81] A.E. Charlesworth. An approach to scientific array processing: The architectural design of the AP-120b/FPS-164 family. *Computer*, 14(9):18–27, Sept. 1981.

[CK02] Yoonseo Choi and Taewhan Kim. Address assignment combined with scheduling in DSP code generation. In *DAC '02: Proceedings of the 39th conference on Design automation*, pages 225–230, New York, NY, USA, 2002. ACM.

[CK03] Yoonseo Choi and Taewhan Kim. Address assignment in DSP code generation - an integrated approach. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 22(8):976–984, 2003.

[CmCtK97] Chia-ming Chang, Chien ming Chen, and Chung ta King. Using integer linear programming for instruction scheduling and register allocation. 34(9):1–14, November 1997.

[CSG01] Josep M. Codina, Jesús Sánchez, and Antonio González. A unified modulo scheduling and register allocation technique for clustered processors. In *PACT '01: Proc. 2001 Int. Conf. on Parallel Architectures and Compilation Techniques*, pages 175–184. IEEE Computer Society, 2001.

[DJG98] Amod K. Dani, V. Janaki, and Ramanan R. Govindarajan. Register-sensitive software pipelining. In *Proceedings of the Merged 12th International Parallel Processing Symposium and 9th International Symposium on Parallel and Distributed Systems*, 1998.

[EDA96] Alexandre E. Eichenberger, Edward S. Davidson, and Santosh G. Abraham. Minimizing register requirements of a modulo schedule via optimum stage scheduling. *Int. J. Parallel Program.*, 24(2):103–132, 1996.

[EK09] Mattias V. Eriksson and Christoph W. Kessler. Integrated modulo scheduling for clustered VLIW architectures. In *International Conference on High Performance Embedded Architectures and Compilers (HiPEAC2009)*, pages 65–79, Paphos, Cyprus, January 2009. Springer Berlin / Heidelberg.

[ELM95] Christine Eisenbeis, Sylvain Lelait, and Bruno Marmol. The meeting graph: a new model for loop cyclic register allocation. In *Proceedings of the IFIP WG10.3 working conference on Parallel architectures and compilation techniques*, PACT '95, pages 264–267, Manchester, UK, UK, 1995. IFIP Working Group on Algol.

[Ert99] M. Anton Ertl. Optimal code selection in DAGs. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 242–249, New York, NY, USA, 1999. ACM.

[ES96a] Christine Eisenbeis and Antoine Sawaya. Optimal loop parallelization under register constraints. In *Sixth Workshop on Compilers for Parallel Computers CPC'96*, pages 245 – 259, Aachen, Germany, December 1996.

[ES96b] Christine Eisenbeis and Antoine Sawaya. Optimal loop parallelization under register constraints. Rapport de Recherche INRIA 2781, INRIA, January 1996.

[ESK08] Mattias V. Eriksson, Oskar Skoog, and Christoph W. Kessler. Optimal vs. heuristic integrated code generation for clustered VLIW architectures. In *SCOPES '08: Proceedings of the 11th international workshop on Software & compilers for embedded systems*, pages 11–20, 2008.

[Fer98] Marcio Merino Fernandes. *A clustered VLIW architecture based on queue register files.* PhD thesis, University of Edinburgh, 1998.

[FFY05] Joseph A. Fisher, Paolo Faraboschi, and Cliff Young. *Embedded computing : a VLIW approach to architec-*

ture, compilers and tools. Amsterdam: Elsevier; San Francisco, Calif.: Morgan Kaufmann, 2005.

[FH95] Christopher W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[Fis83] Joseph A. Fisher. Very long instruction word architectures and the ELI-512. In *ISCA '83: Proceedings of the 10th annual international symposium on Computer architecture*, pages 140–150, Los Alamitos, CA, USA, 1983. IEEE Computer Society Press.

[FKPM05] Kevin Fan, Manjunath Kudlur, Hyunchul Park, and Scott Mahlke. Cost sensitive modulo scheduling in a loop accelerator synthesis system. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 219–232, Washington, DC, USA, 2005. IEEE Computer Society.

[FLT99] Marcio Merino Fernandes, Josep Llosa, and Nigel Topham. Distributed modulo scheduling. In *HPCA '99: Proceedings of the 5th International Symposium on High Performance Computer Architecture*, page 130, Washington, DC, USA, 1999. IEEE Computer Society.

[FM00] Dirk Fimmel and Jan Müller. Optimal software pipelining under register constraints. In *PDPTA '00: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 2000.

[FM02] Dirk Fimmel and Jan Müller. Optimal software pipelining with rational initiation interval. In *PDPTA '02: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 638–643. CSREA Press, 2002.

[Fre] Free Software Foundation. GCC. http://gcc.gnu.org.

[GE93] C.H. Gebotys and M.I. Elmasry. Global optimization approach for architectural synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(9):1266–1278, Sep 1993.

[Geb97] Catherine Gebotys. DSP address optimization using a minimum cost circulation technique. In *ICCAD '97: Proceedings of the 1997 IEEE/ACM international conference on Computer-aided design*, pages 100–103, Washington, DC, USA, 1997. IEEE Comp. Society.

[Gol89] D. E. Goldberg, editor. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison–Wesley, Reading, Mass., 1989.

[GSZ01] Elana Granston, Eric Stotzer, and Joe Zbiciak. Software pipelining irregular loops on the TMS320C6000 VLIW DSP architecture. *SIGPLAN Not.*, 36(8):138–144, 2001.

[Gur10] Gurobi optimization, inc. *Gurobi 4.0 Optimizer Reference Manual*, 2010.

[HABT07] Johnny Huynh, José Amaral, Paul Berube, and Sid Touati. Evaluation of offset assignment heuristics. In *HiPEAC '07: Int. Conf. on High Performance Embedded Architectures and Compilers*, pages 261–275, Ghent, January 2007.

[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.

[HMC$^+$93] Wen-Mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. The superblock: an effective technique for VLIW and superscalar compilation. *J. Supercomput.*, 7(1-2):229–248, 1993.

[Huf93] Richard A. Huff. Lifetime-sensitive modulo scheduling. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 258–267, 1993.

[ILO06] ILOG. *ILOG AMPL CPLEX System Manual*, 2006.

[Joh75] Donald B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM J. Comput.*, 4(1):77–84, 1975.

[Käs00] Daniel Kästner. *Retargetable Postpass Optimisations by Integer Linear Programming.* PhD thesis, 2000.

[Käs01] Daniel Kästner. Propan: A retargetable system for postpass optimisations and analyses. In *LCTES '00: Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 63–80, London, UK, 2001. Springer-Verlag.

[KB01] Christoph Kessler and Andrzej Bednarski. A dynamic programming approach to optimal integrated code generation. In *LCTES '01: Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems*, pages 165–174, New York, NY, USA, June 2001. ACM.

[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*, June 2002.

[KB04] Christoph Kessler and Andrzej 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. Shaker-Verlag Aachen, Germany, July 2004.

[KB05] Christoph Kessler and Andrzej Bednarski. OPTIMIST. www.ida.liu.se/∼chrke/optimist, 2005.

[KB06] Christoph W. Keßler and Andrzej Bednarski. Optimal integrated code generation for VLIW architectures. *Concurrency and Computation: Practice and Experience*, 18(11):1353–1390, 2006.

[KBE07] Christoph Kessler, Andrzej Bednarski, and Mattias Eriksson. Classification and generation of schedules for VLIW processors. *Concurrency and Computation: Practice and Experience*, 19(18):2369–2389, 2007.

[KEA01] Krishnan Kailas, Kemal Ebcioglu, and Ashok Agrawala. CARS: A new code generation framework for clustered ILP processors. In *7th Int. Symp. on High-Performance Computer Architecture (HPCA'01)*, pages 133–143. IEEE Computer Society, June 2001.

[Keß98] Christoph W. Keßler. Scheduling Expression DAGs for Minimal Register Need. *Computer Languages*, 24(1):33–53, September 1998.

[LA04] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO 04)*, 2004.

[Lam88] Monica Lam. Software pipelining: an effective scheduling technique for VLIW machines. *SIGPLAN Notices*, 23(7):318–328, 1988.

[LDK$^+$96] Stan Liao, Srinivas Devadas, Kurt Keutzer, Steven Tjiang, and Albert Wang. Storage assignment to decrease code size. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(3):235–253, 1996.

[Leu97] Rainer Leupers. *Retargetable Code Generation for Digital Signal Processors.* Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[Leu00a] Rainer Leupers. *Code Optimization Techniques for Embedded Processors: Methods, Algorithms, and Tools.* Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[Leu00b] Rainer Leupers. Instruction scheduling for clustered VLIW DSPs. In *PACT '00: Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*, page 291, Washington, DC, USA, 2000. IEEE Computer Society.

[Leu03] Rainer Leupers. Offset assignment showdown: Evaluation of DSP address code optimization algorithms. In *proc. of the 12th int. conf. on compiler construction*, pages 290–302, 2003.

[LF02] Josep Llosa and Stefan M. Freudenberger. Reduced code size modulo scheduling in the absence of hardware support. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 35, pages 99–110, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[LGAV96] Josep Llosa, Antonio Gonzalez, Eduard Ayguade, and Mateo Valero. Swing modulo scheduling: a lifetime-sensitive approach. *Proceedings of the Conference on Parallel Architectures and Compilation Techniques*, pages 80–86, Oct 1996.

[LLM+01] Markus Lorenz, Rainer Leupers, Peter Marwedel, Thorsten Dräger, and Gerhard Fettweis. Low-energy dsp code generation using a genetic algorithm. In *ICCD '01: Proceedings of the International Conference on Computer Design: VLSI in Computers & Processors*, pages 431–437, Washington, DC, USA, 2001. IEEE Computer Society.

[LM96] Rainer Leupers and Peter Marwedel. Algorithms for address assignment in DSP code generation. In *ICCAD '96: Proc. of the 1996 IEEE/ACM int. conf. on Computer-aided design*, pages 109–112, Washington, DC, USA, 1996. IEEE Computer Society.

[LM04] Markus Lorenz and Peter Marwedel. Phase coupled code generation for DSPs using a genetic algorithm. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, pages 1270–1275, Washington, DC, USA, 2004. IEEE Computer Society.

[LPMS97] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *International Symposium on Microarchitecture*, pages 330–335, 1997.

[LVAG95] Josep Llosa, Mateo Valero, Eduard Ayguadé, and Antonio González. Hypernode reduction modulo scheduling. In *MICRO 28: Proceedings of the 28th annual international symposium on Microarchitecture*, pages 350–360, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press.

[Mak] Andrew Makhorin. GNU Linear Programming Kit. http://www.gnu.org/software/glpk/.

[Mas87] Henry Massalin. Superoptimizer: a look at the smallest program. *SIGPLAN Not.*, 22:122–126, October 1987.

[MCRvB08] Abid M. Malik, Michael Chase, Tyrel Russell, and Peter van Beek. An application of constraint programming to superblock instruction scheduling. In *Proceedings of the 14th International Conference on Principles and Practice of Constraint Programming*, pages 97–111, September 2008.

[MMvB06] Abid M. Malik, Jim McInnes, and Peter van Beek. Optimal basic block instruction scheduling for multiple-issue processors using constraing programming. *Tools with Artificial Intelligence, 2006. ICTAI '06. 18th IEEE International Conference on*, pages 279–287, Nov. 2006.

[Muc97] Steven S. Muchnick. *Advanced compiler design and implementation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[NE98] Erik Nystrom and Alexandre E. Eichenberger. Effective cluster assignment for modulo scheduling. In *MICRO 31: Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 103–114, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[NG93] Qi Ning and Guang R. Gao. A novel framework of register allocation for software pipelining. In *POPL '93: Proceedings of the 20th ACM symp. on Principles of programming languages*, pages 29–42. ACM, 1993.

[NG07] Santosh G. Nagarakatte and R. Govindarajan. Register allocation and optimal spill code scheduling in software pipelined loops using 0-1 integer linear programming formulation. In Shriram Krishnamurthi and Martin Odersky, editors, *CC*, volume 4420 of *Lecture Notes in Computer Science*, pages 126–140. Springer, 2007.

[NS04] Rahul Nagpal and Y. N. Srikant. Integrated temporal and spatial scheduling for extended operand clustered

VLIW processors. In *First conf. on Computing frontiers*, pages 457–470. ACM Press, 2004.

[OBC98] Emre Ozer, Sanjeev Banerjia, and Thomas M. Conte. Unified assign and schedule: a new approach to scheduling for clustered register file microarchitectures. In *31st annual ACM/IEEE Int. Symposium on Microarchitecture*, pages 308–315, 1998.

[OKT06] O. Ozturk, M. Kandemir, and S. Tosun. An ILP based approach to address code generation for digital signal processors. In *Proc. 16th ACM Great Lakes symposium on VLSI*, pages 37–42, New York, USA, 2006. ACM.

[Ope] Open research compiler. Project homepage. http://www.open64.net.

[PK05] Markus Pister and Daniel Kästner. Generic software pipelining at the assembly level. In *SCOPES '05: Proc. workshop on Software and compilers for embedded systems*, pages 50–61. ACM, 2005.

[Rau94] B. Ramakrishna Rau. Iterative modulo scheduling: an algorithm for software pipelining loops. In *MICRO 27: Proceedings of the 27th annual international symposium on Microarchitecture*, pages 63–74, New York, NY, USA, november 1994. ACM.

[RG81] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. *SIGMICRO Newsl.*, 12(4):183–198, 1981.

[RGSL96] John Ruttenberg, G. R. Gao, A. Stoutchinin, and W. Lichtenstein. Software pipelining showdown: optimal vs. heuristic methods in a production compiler. In *PLDI '96: Proceedings of the ACM SIGPLAN 1996* *conference on Programming language design and implementation*, pages 1–11, New York, NY, USA, 1996. ACM Press.

[RLTS92] B. R. Rau, M. Lee, P. P. Tirumalai, and M. S. Schlansker. Register allocation for software pipelined loops. *SIGPLAN Not.*, 27:283–299, July 1992.

[RP99] Amit Rao and Santosh Pande. Storage assignment optimizations to generate compact and efficient code on embedded DSPs. In *PLDI*, pages 128–138, 1999.

[SL99] Eric Stotzer and Ernst Leiss. Modulo scheduling for the TMS320C6x VLIW DSP architecture. *SIGPLAN Not.*, 34(7):28–34, 1999.

[SL09] Eric J. Stotzer and Ernst L. Leiss. Modulo scheduling without overlapped lifetimes. *SIGPLAN Not.*, 44:1–10, June 2009.

[Tar73] Robert Endre Tarjan. Enumeration of the elementary circuits of a directed graph. *SIAM J. Comput.*, 2(3):211–216, 1973.

[Tex88] Texas Instruments Incorporated, Upper Saddle River, NJ, USA. *Second-generation TMS320 user's guide*, 1988.

[Tex00] Texas Instruments Incorporated. *TMS320C6000 CPU and Instruction Set Reference Guide*, 2000.

[Tex10] Texas Instruments Incorporated. *TMS320C6672 CPU and Instruction Set Reference Guide*, 2010.

[Tou02] Sid Touati. *Register Pressure in Instruction Level Parallelism.* PhD thesis, Université de Versailles, France, June 2002.

[Tou07] Sid-Ahmed-Ali Touati. On periodic register need in software pipelining. *IEEE Trans. Comput.*, 56(11):1493–1504, 2007.

[Tou09] Sid Touati. Data dependence graphs from Spec, Mediabench and Ffmpeg benchmark suites. Personal communication, 2009.

[Tri] Trimaran. Homepage. http://www.trimaran.org.

[UC01] Sathishkumar Udayanarayanan and Chaitali Chakrabarti. Address code generation for digital signal processors. In *DAC '01: Proc. of the 38th annual Design Automation Conference*, pages 353–358, New York, USA, 2001. ACM.

[uVSM94] Vojin živojnović, Juan M. Velarde, Christian Schläger, and Heinrich Meyr. DSPSTONE: A DSP-oriented benchmarking methodology. In *Proc. Int. Conf. on Signal Processing and Technology (ICSPAT'94)*, 1994.

[Veg92] Stephen R. Vegdahl. A dynamic-programming technique for compacting loops. In *MICRO 25: Proceedings of the 25th annual international symposium on Microarchitecture*, pages 180–188, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[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. Int. Conf. on Computer Design (ICCD)*, pages 581–586, 1994.

[Win04] Sebastian Winkel. *Optimal Global Instruction Scheduling for the Itanium Processor Architecture.* PhD thesis, Universität des Saarlandes, September 2004.

[Win07] Sebastian Winkel. Optimal versus heuristic global code scheduling. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 43–55, Washington, DC, USA, 2007. IEEE Computer Society.

[WLH00] Kent Wilken, Jack Liu, and Mark Heffernan. Optimal instruction scheduling using integer programming. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 121–133, New York, NY, USA, 2000. ACM.

[YGGT02] Hongbo Yang, R. Govindarajan, Guang R. Gao, and Kevin B. Theobald. Power-performance trade-offs for energy-efficient architectures: A quantitative study. In *ICCD '02: Proceedings of the 2002 IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD'02)*, page 174, Washington, DC, USA, 2002. IEEE Computer Society.

[ZLAV01] Javier Zalamea, Josep Llosa, Eduard Ayguadé, and Mateo Valero. Modulo scheduling with integrated register spilling for clustered VLIW architectures. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 34, pages 160–169, Washington, DC, USA, 2001. IEEE Computer Society.

Index

Department of Computer and Information Science
Linköpings universitet

Dissertations

Linköping Studies in Science and Technology
Linköping Studies in Arts and Science
Linköping Studies in Statistics
Linköpings Studies in Informatics

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 Languages from Two-Level Semantics Specifications, 1997, ISBN 91-7219-045-0.

No 512 **Anna Moberg:** Närhet och distans - Studier av kommunikationsmönster 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-supported Interorganisational 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 informationssystem, 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 Empirical 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 Linguistic 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 - Modelling 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 Tešanovic:** 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.

No 1022 **Peter Aronsson:** Automatic Parallelization of Equation-Based Simulation Programs, 2006, ISBN 91-85523-68-2.

No 1030 **Robert Nilsson:** A Mutation-based Framework for Automated Testing of Timeliness, 2006, ISBN 91-85523-35-6.

No 1034 **Jon Edvardsson:** Techniques for Automatic Generation of Tests from Programs and Specifications, 2006, ISBN 91-85523-31-3.

No 1035 **Vaida Jakoniene:** Integration of Biological Data, 2006, ISBN 91-85523-28-3.

No 1045 **Genevieve Gorrell:** Generalized Hebbian Algorithms for Dimensionality Reduction in Natural Language Processing, 2006, ISBN 91-85643-88-2.

No 1051 **Yu-Hsing Huang:** Having a New Pair of Glasses - Applying Systemic Accident Models on Road Safety, 2006, ISBN 91-85643-64-5.

No 1054 **Åsa Hedenskog:** Perceive those things which cannot be seen - A Cognitive Systems Engineering perspective on requirements management, 2006, ISBN 91-85643-57-2.

No 1061 **Cécile Åberg:** An Evaluation Platform for Semantic Web Technology, 2007, ISBN 91-85643-31-9.

No 1073 **Mats Grindal:** Handling Combinatorial Explosion in Software Testing, 2007, ISBN 978-91-85715-74-9.

No 1075 **Almut Herzog:** Usable Security Policies for Runtime Environments, 2007, ISBN 978-91-85715-65-7.

No 1079 **Magnus Wahlström:** Algorithms, measures, and upper bounds for Satisfiability and related problems, 2007, ISBN 978-91-85715-55-8.

No 1083 **Jesper Andersson:** Dynamic Software Architectures, 2007, ISBN 978-91-85715-46-6.

No 1086 **Ulf Johansson:** Obtaining Accurate and Comprehensible Data Mining Models - An Evolutionary Approach, 2007, ISBN 978-91-85715-34-3.

No 1089 **Traian Pop:** Analysis and Optimisation of Distributed Embedded Systems with Heterogeneous Scheduling Policies, 2007, ISBN 978-91-85715-27-5.

No 1091 **Gustav Nordh:** Complexity Dichotomies for CSP-related Problems, 2007, ISBN 978-91-85715-20-6.

No 1106 **Per Ola Kristensson:** Discrete and Continuous Shape Writing for Text Entry and Control, 2007, ISBN 978-91-85831-77-7.

No 1110 **He Tan:** Aligning Biomedical Ontologies, 2007, ISBN 978-91-85831-56-2.

No 1112 **Jessica Lindblom:** Minding the body - Interacting socially through embodied action, 2007, ISBN 978-91-85831-48-7.

No 1113 **Pontus Wärnestål:** Dialogue Behavior Management in Conversational Recommender Systems, 2007, ISBN 978-91-85831-47-0.

No 1120 **Thomas Gustafsson:** Management of Real-Time Data Consistency and Transient Overloads in Embedded Systems, 2007, ISBN 978-91-85831-33-3.