Linköping Studies in Science and Technology
Thesis No. 1393

Integrated Software Pipelining

by

Mattias Eriksson

Submitted to Linköping Institute of Technology at Linköping University in partial fulfilment of the requirements for degree of Licentiate of Engineering

Department of Computer and Information Science
Linköpings universitet
SE-581 83 Linköping, Sweden
Linköping 2009

This document was produced with LaTeX, gnuplot and xfig.
Electronic version available at:
http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-16170
Printed by LiU-Tryck, Linköping 2009.

Integrated Software Pipelining

by

Mattias Eriksson

February 2009
ISBN 978-91-7393-699-6
Linköping Studies in Science and Technology
Thesis No. 1393
ISSN 0280-7971
LIU-TEK-LIC-2009:1

ABSTRACT

In this thesis we address the problem of integrated software pipelining for clustered VLIW architectures. The phases that are integrated and solved as one combined problem are: cluster assignment, instruction selection, scheduling, register allocation and spilling.

As a first step we describe two methods for integrated code generation of basic blocks. The first method is optimal and based on integer linear programming. The second method is a heuristic based on genetic algorithms.

We then extend the integer linear programming model to modulo scheduling. To the best of our knowledge this is the first time anybody has optimally solved the modulo scheduling problem for clustered architectures with instruction selection and cluster assignment integrated.

We also show 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 the memory and have transfer instructions to and from this register file corresponding to stores and loads.

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.

Experiments have been conducted to show the usefulness and limitations of our optimal methods. For the basic block case we compare the optimal method to the heuristic based on genetic algorithms.

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

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

Acknowledgments

I have learned a lot both about how to do and how to present research during the time that I have worked on this thesis. Most thanks for this should go to my supervisor Christoph Kessler who has guided me and given me ideas that I have had the possibility to work on in an independent manner. Thanks also to Andrzej Bednarski, who together with Christoph Kessler started the work on integrated code generation with Optimist.

It has been very rewarding and a great pleasure to co-supervise master's thesis students: Oskar Skoog did great parts of the genetic algorithm in Optimist, Lukas Kemmer implemented a visualizer of schedules, Daniel Johansson and Markus Ålind worked on libraries for the Cell processor (not included in 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. Thanks also to the administrative staff.

Thanks to Vetenskapsrådet and the Swedish graduate school in computer science (CUGS) for funding my work, and to the anonymous reviewers of my papers for constructive comments.

Last, but not least, thanks to my friends and to my families for support and encouragement.

Mattias Eriksson
Linköping, December 2008.

Contents

List of Figures

Chapter 1

Introduction

This chapter gives a brief introduction to the area of integrated code generation and to the Optimist framework. We also give a list of our contributions and describe the thesis outline.

1.1 Motivation

A processor in an embedded device often spends the major part of its life 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 a 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.

1.2 Compilers and code generation for VLIW architectures

This section contains a very brief description of compilers and VLIW architectures. For a more in depth treatment of these topics, please refer to a good text book in this area, such as the "Dragon book" [1].

1.2.1 Compilation

Typically a compiler is a program that translates computer programs from one language to another. In this thesis we consider compilers that translate human readable code, e.g. C, into machine code for processors with *static* instruction level parallelism. For such architectures, it is up to the compiler to generate the parallelism.

The *Front end* of a compiler is the part which reads the input program and does a translation into some *intermediate representation* (IR).

Code generation, which is the part of the compiler that we focus on in this thesis, is performed in the *back end* of a compiler. In essence, it is the process of creating executable code from the previously generated IR. One usual way to do this is to perform three phases in some sequence:

- *Instruction selection phase* — Select target instructions matching the IR.
- *Instruction scheduling phase* — Map the selected instructions to time slots on which to execute them.
- *Register allocation phase* — Select registers in which intermediate values are to be stored.

While doing the phases in sequence is simpler and less computationally heavy, the phases are interdependent. Hence, integrating the phases of the code generator gives more opportunity for optimization. The cost of integrating the phases is that the size of the solution space greatly increases. There is a *combinatorial explosion* when decisions in all phases are considered simultaneously. This is especially the case when we consider complicated processors with clustered register files and functional units where many different target instructions may be applied to a single IR operation, and with both explicit and implicit transfers between the register clusters.

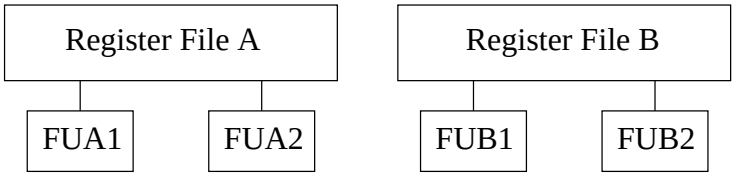

Figure 1.1: An illustration of a clustered VLIW architecture. Here there are two clusters, A and B, and to each of the register files two functional units are connected.

1.2.2 Very long instruction word architectures

In this thesis we are interested in code generation for *very long instruction word* (VLIW) architectures [31]. For VLIW processors the issued instructions contains multiple operations that are executed in parallel. This means that all instruction level parallelism is static, i.e. the compiler (or hand coder) decides which operations are to be executed at the same point in time. Our focus is particularly on *clustered* VLIW architectures in which the functional units of the processor are limited to using a subset of the available registers [27]. 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 interdependencies 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 Figure 1.1 for an illustration of a clustered VLIW architecture.

1.3 Retargetable code generation and the Optimist framework

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

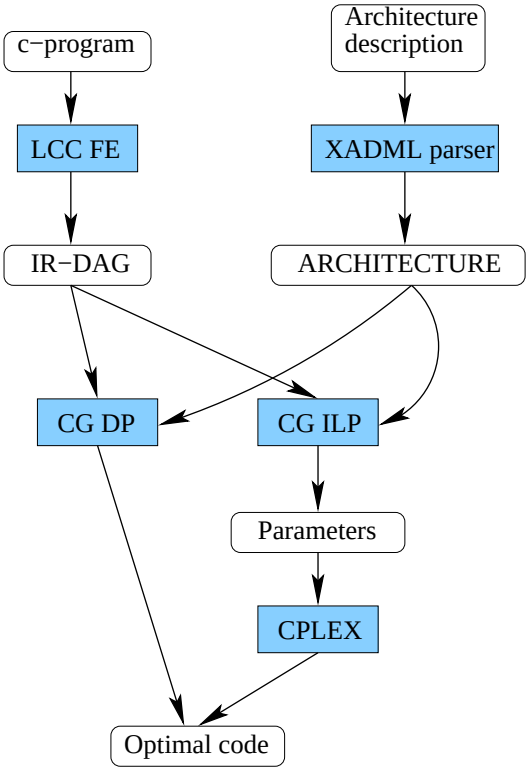

Figure 1.2: Overview of the Optimist compiler.

approach to creating such compilers is called *retargetable compiling* 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. A more thorough treatment on the subject of retargetable compiling can be found in [55].

1.3.1 The Optimist framework

For the implementations in this thesis we use the retargetable Optimist framework [48]. Optimist uses a modified LCC (Little C Compiler) [32] as front end to parse C-code and generate Boost [13] graphs that are used, together with a processor description, as input to a pluggable code generator. 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 [51].
- Integer linear programming (ILP), in which parameters are generated that can be passed on, together with a mathematical model, to an ILP solver such as *CPLEX* [41] or *GLPK* [64]. This method was limited to single cluster architectures [10]. This model is 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 [51]. 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) [8]. This language is versatile enough to describe clustered, pipelined, irregular and asymmetric VLIW architectures.

1.4 Contributions

The contributions of the work presented in this thesis are:

1. The ILP-model [10] is extended to handle clustered VLIW architectures. To our knowledge, no such formulation exists in the literature. 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. A new heuristic based on genetic algorithms is created which solves the integrated code generation problem. This algorithm is also parallelized with the master-slave paradigm, allowing for faster compilation times.

3. We show how to extend the integer linear programming model to also integrate modulo scheduling.

4. And finally 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.

1.5 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 the first integer linear programming model for the acyclic case and a description of the genetic algorithm [25].

- Mattias V. Eriksson, Christoph W. Kessler. Integrated Modulo Scheduling for Clustered VLIW Architectures. To appear in *HiPEAC-2009 High-Performance and Embedded Architecture and Compilers*, Paphos, Cyprus, Jan. 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 [24].

- 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 [49].

1.6 Thesis outline

The remainder of this thesis is organized as follows:

- Chapter 2 contains our integrated code generation methods for the acyclic case: First the integer linear programming model, and then the genetic algorithm. This chapter also contains the experimental comparison of the two methods.
- In Chapter 3 we extend the integer linear programming model to modulo scheduling. Additionally we show the algorithm and prove how the search space can be made finite.
- Chapter 4 shows related work in acyclic and cyclic integrated code generation.
- Chapter 5 lists topics for future work.
- Chapter 6 concludes the thesis.
- In Appendix A we show listings used for the evaluation in Chapter 3.

Chapter 2
Integrated code generation for basic blocks

This chapter describes two methods for integrated code generation for basic blocks[1]. 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.

2.1 Integer linear programming formulation

For optimal code generation we use an integer linear programming formulation. In this section we will first introduce all parameters and variables which are used by the CPLEX solver to generate a schedule with minimal execution time. Then, we introduce the integer linear programming formulation for basic block scheduling. This model integrates instruction selection (including cluster assignment), instruction scheduling and register allocation.

In previous work within the Optimist project an integer linear programming method was compared to the dynamic programming method [10]. This study used a simple hypothetical non-clustered architecture. An advantage that integer linear programing has over dynamic programming is that a mathematical precise description is

[1] 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.

generated as a side effect. Also, the integer linear programming model is easier to extend to modulo scheduling.

2.1.1 Optimization parameters and variables

Data flow graph

The data flow graph of 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 is the set of intermediate representation (IR) nodes, the sets $E_1, E_2 \subset V \times V$ represent edges between operations and their first and second operand respectively. $E_m \subset V \times V$ represent data dependences in memory. The integer parameters Op_i and $Outdg_i$ describe operators and out-degrees of the IR node $i \in V$, respectively.

Instruction set

The instructions of the target machine are modeled by the set P of patterns. P consists of the set P_1 of singletons, which only cover one IR node, the set P_{2^+} of composites, which cover multiple IR nodes, and the set P_0 of patterns for non-issue instructions. The non-issue instructions 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 to be used. The IR is low level enough so that all patterns model exactly one (or zero in the case of constants) 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 paired with more than one instruction. For instance, on the TI-C62x an addition can be done with one of twelve different instructions (not counting the multiply-and-accumulate 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

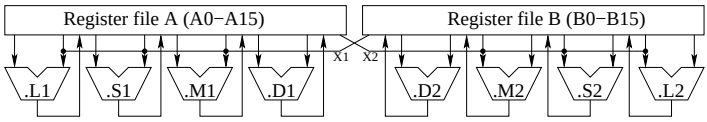

Figure 2.1: The Texas Instruments TI-C62x processor has two register banks and 8 functional units [81]. The crosspaths X1 and X2 are modeled as resources, too.

$EP_p \subset B_p \times B_p$, 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 $t + L_p$.

Resources and register sets

For the integer linear programming model we assume that the functional units are fully pipelined. We model the resources of the target machine with the set $\mathcal{F}$ and the register banks by 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 [49] and irregular reservation tables [77]. $\mathcal{R}_r$ 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 use the integer parameter $LX_{r,s}$ to model 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 2.1 for an illustration of a clustered architecture.

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 $T = \{0, 1, 2, \ldots, t_{\max}\}$, 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.

2.1.2 Removing impossible schedule slots

We can significantly reduce the number of variables in the model by performing soonest-latest analysis [60] on the nodes of the graph[2]. 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:

$$soonest'(i) = \begin{cases} 0 & \text{, if } |\text{pre}(i)| = 0 \\ \max_{j \in \text{pre}(i)}\{soonest'(j) + L_{\min}(j)\} & \text{, otherwise} \end{cases} \tag{2.1}$$

$$latest'(i) = \begin{cases} t_{\max} & \text{, if } |\text{succ}(i)| = 0 \\ \max_{j \in \text{succ}(i)}\{latest'(j) - L_{\min}(i)\} & \text{, otherwise} \end{cases} \tag{2.2}$$

$$T_i = \{soonest'(i), \ldots, latest'(i)\} \tag{2.3}$$

We can also remove all the variables in c where no node in the pattern $p \in P$ has an operator number matching i. Mathematically we can say that the matrix c of variables is sparse; 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.

2.1.3 Optimization constraints

Optimization objective

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

$$\min \tau \tag{2.4}$$

The execution time is the latest time slot where any instruction terminates. For efficiency we only need to check for execution times

[2]The measurements in Section 2.3 of this chapter do not include this optimization with soonest-latest analysis. In Chapter 3 this optimization is extended to loop-carried dependences and is used for all the experiments.

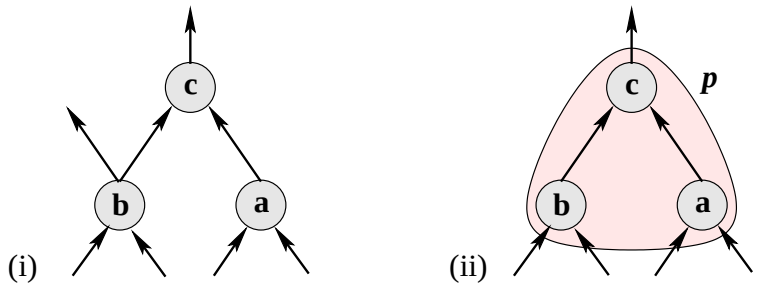

Figure 2.2: (i) Pattern p can not cover the set of nodes since there is another outgoing edge from b, (ii) p covers nodes a, b, c.

for instructions covering an IR node with out-degree 0, let $V_{\text{root}} = \{i \in V : Outdg_i = 0\}$:

$$\forall i \in V_{\text{root}},\ \forall p \in P,\ \forall k \in B_p,\ \forall t \in T, \quad c_{i,p,k,t}(t + L_p) \leq \tau \tag{2.5}$$

Node and edge covering

Exactly one instruction must cover each IR node:

$$\forall i \in V, \quad \sum_{p \in P} \sum_{k \in B_p} \sum_{t \in T} c_{i,p,k,t} = 1 \tag{2.6}$$

Equation 2.7 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:

$$\forall p \in P_{2^+},\ \forall t \in T,\ \forall k \in B_p \quad \sum_{i \in V} c_{i,p,k,t} = s_{p,t} \tag{2.7}$$

An edge within a composite pattern may only be active if there is a corresponding edge (i, j) in the DAG and both i and j are covered by the pattern, see Figure 2.2:

$$\forall (i,j) \in E_1 \cup E_2,\ \forall p \in P_{2^+},\ \forall t \in T,\ \forall (k,l) \in EP_p,$$
$$2w_{i,j,p,t,k,l} \leq c_{i,p,k,t} + c_{j,p,l,t} \tag{2.8}$$

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

$$\forall i \in V,\ \forall p \in P,\ \forall k \in B_p,\ \forall t \in T, \quad c_{i,p,k,t}(Op_i - OP_{p,k}) = 0 \quad (2.9)$$

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:

$$\forall rr \in \mathcal{RS},\ \forall i \in V,\ \forall t \in T,$$

$$r_{rr,i,t} \leq \sum_{\substack{p \in PD_{rr} \cap P \\ k \in B_p}} c_{i,p,k,t-L_p} + \ r_{rr,i,t-1} + \sum_{rs \in \mathcal{RS}} (x_{i,rs,rr,t-LX_{rs,rr}}) \quad (2.10)$$

The operand to an instruction must be available in the correct register bank when we use it. A limitation of this formulation is that composite patterns must have all operands and results in the same register bank:

$$\forall (i,j) \in E_1 \cup E_2,\ \forall t \in T,\ \forall rr \in \mathcal{RS},$$

$$\text{BIG} \cdot r_{rr,i,t} \geq \sum_{\substack{p \in PD_{rr} \cap P_{2^+} \\ k \in B_p}} \left(c_{j,p,k,t} - \text{BIG} \cdot \sum_{(k,l) \in EP_p} w_{i,j,p,t,k,l} \right) \quad (2.11)$$

where BIG is a large integer value.

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

$$\forall p \in P_{2^+},\ \forall (k,l) \in EP_p,\ \forall (i,j) \in E_1 \cup E_2,$$
$$\sum_{rr \in \mathcal{RS}} \sum_{t \in T} r_{rr,i,t} \leq \text{BIG} \cdot \left(1 - \sum_{t \in T} w_{i,j,p,t,k,l} \right) \quad (2.12)$$

If they exist, the first operand (Equation 2.13) and the second operand (Equation 2.14) must be available when they are used:

$$\forall (i,j) \in E_1,\ \forall t \in T,\ \forall rr \in \mathcal{RS}, \quad \text{BIG} \cdot r_{rr,i,t} \geq \sum_{\substack{p \in PS1_{rr} \cap P_1 \\ k \in B_p}} c_{j,p,k,t} \tag{2.13}$$

$$\forall (i,j) \in E_2,\ \forall t \in T,\ \forall rr \in \mathcal{RS}, \quad \text{BIG} \cdot r_{rr,i,t} \geq \sum_{\substack{p \in PS2_{rr} \cap P_1 \\ k \in B_p}} c_{j,p,k,t} \tag{2.14}$$

Transfers may only occur if the source value is available:

$$\forall i \in V,\ \forall t \in T,\ \forall rr \in \mathcal{RS}, \quad r_{rr,i,t} \geq \sum_{rq \in \mathcal{RS}} x_{i,rr,rq,t} \tag{2.15}$$

Memory data dependences

Equation 2.16 ensures that data dependences in memory are not violated, adapted from [34]:

$$\forall (i,j) \in E_m,\ \forall t \in T \quad \sum_{p \in P} \sum_{t_j=0}^{t} c_{j,p,1,t_j} + \sum_{p \in P} \sum_{t_i=t-L_p+1}^{t_{\max}} c_{i,p,1,t_i} \leq 1 \tag{2.16}$$

Resources

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

$$\forall t \in T,\ \forall rr \in \mathcal{RS}, \quad \sum_{i \in V} r_{rr,i,t} \leq R_{rr} \tag{2.17}$$

Condition 2.18 ensures that no resource is used more than once at each time slot:

$$\forall t \in T,\ \forall f \in \mathcal{F},$$

$$\sum_{\substack{p \in P_{2+} \\ o \in \mathbb{N}}} U_{p,f,o} s_{p,t-o} + \sum_{\substack{p \in P_1 \\ i \in V \\ k \in B_p}} U_{p,f,o} c_{i,p,k,t-o} + \sum_{\substack{i \in V \\ (rr,rq) \in (\mathcal{RS} \times \mathcal{RS})}} UX_{rr,rq,f} x_{i,rr,rq,t} \leq 1 \tag{2.18}$$

And, lastly, Condition 2.19 guarantees that we never exceed the issue width:

$$\forall t \in T, \quad \sum_{p \in P_{2+}} s_{p,t} + \sum_{\substack{p \in P_1 \\ i \in V \\ k \in B_p}} c_{i,p,k,t} + \sum_{\substack{i \in V \\ (rr,rq) \in (\mathcal{RS} \times \mathcal{RS})}} x_{i,rr,rq,t} \leq \omega \tag{2.19}$$

2.2 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 [51] in which a search of the solution space is performed 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 very 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 [35] 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 chose 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 [49] 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 1. *For the DAG in Figure 2.3, 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 [81] 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* $\{1 \rightarrow \text{NULL}, 5 \rightarrow \text{NULL}, 3 \rightarrow$ "LDW.D1″$, 4 \rightarrow$ "LDW.D2″$, 2 \rightarrow$ "ADD.D1X″$, 0 \rightarrow$ "MV.L1″$\}$*. And the schedule has no explicit transfers, so the transfer map is empty. (This is a simplification, in reality we need to store more information about the exact instruction in the instruction map, see [51] for details on time profiles and space profiles.)*

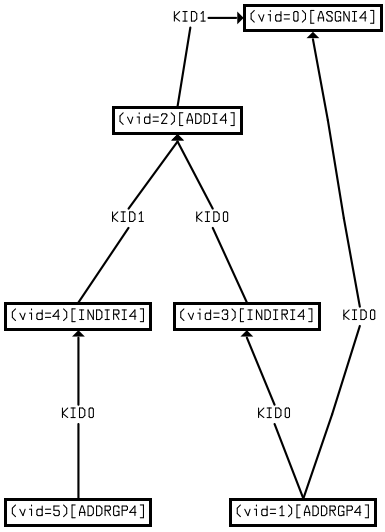

Figure 2.3: A compiler generated DAG of the basic block representing the calculation $a = a + b;$. The vid attribute gives the node index for each IR node.

2.2.1 Evolution operations

All that we now need to perform the evolution of the individuals are: a fitness calculation method for comparing the goodness of 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.

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 is 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.

2.2.2 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 2.4 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 2.1. 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.

```
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 2.4: 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	644.78	349.1	1.85

Table 2.1: 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.

2.3 Results

All tests were performed on an Athlon X2 6000+, 64-bit, dual core, 3 GHz processor with 4 GB RAM, 1 MB L2 cache, and $2 \times 2 \times 64$ kB L1 cache. The genetic algorithm implementation was compiled with gcc at optimization level `-O2`. The version of CPLEX is 10.2. All execution times are measured in wall clock time.

The target architecture that we use is 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 2.1 for an illustration.

2.3.1 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.

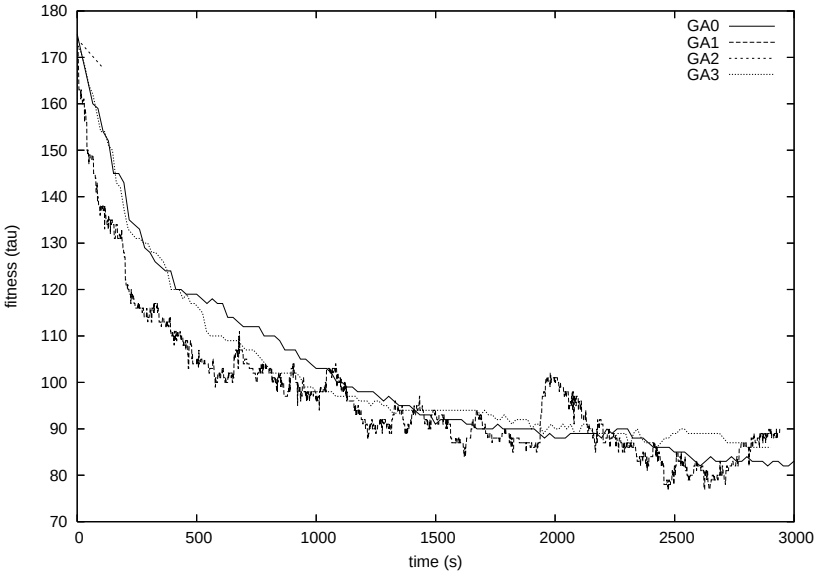

Figure 2.5: A plot of the progress of the best individual at each generation for 4 different parameter configurations. The best result, $\tau = 77$, is found with GA1 — 10 individuals, 75% mutation probability with the parents survive strategy. For comparison, running a random search 34'000 times (corresponding to the same time budget as for the GA algorithms) only gives a schedule with fitness 168.

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

The basic block that we use for evaluating the parameter configurations is a part of the inverse discrete cosine transform calculation in Mediabench's [54] 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 2.5 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 we run 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.

2.3.2 Comparing ILP and GA performance

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 [54] 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 size of the largest basic block is 191 IR nodes.

The result of the evaluation is found in Tables 2.3, 2.4 and 2.5. The first column, $|G|$, 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 8 columns show the results of: the HS1 heuristic (see Section 2.2), the GA heuristic with the parameter sets GA0 and GA1

Δ_{opt}	GA0	GA1
0	44	40
1	7	8
2	1	3
3	0	2
4	2	1
5	1	0
6	1	1
7	0	1

Table 2.2: 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.

(see Section 2.3.1) and the integer linear program execution (see Section 2.1). The results of the integer linear programming does not include the soonest-latest analysis described in Section 2.1.2. For the integer linear programming case the results differ from previously published results in [25]. The reason is that the constraints dealing with data dependences in memory in the integer linear program formulation have been improved.

The execution time ($t(s)$) is measured in seconds, and as expected the HS1 heuristic is very fast for most cases. The execution times for GA0 and GA1 are approximately between 650 and 800 seconds 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 both cores of the host processor, or after the population had evolved for 20000 generations. The time limit for the integer linear programming tests was set to 900 seconds, cases where no solution was found are marked with a '-'.

Table 2.2 summarizes the results for the cases where the optimum is known. We see that out of the 81 basic blocks the integer linear programming method finds an optimal solution to 56 of them. We also note that for these 56 cases where we know the optimal solution GA0 finds a worse solution only in 12 cases, on average GA0 finds a

result 0.5 cycles from the optimum when optimum is known. GA1 is worse in 16 cases, on average GA1 finds a result 0.66 cycles from the optimum when optimum is known. The comparison between GA and integer linear programming is not completely fair since GA is parallelized and can utilize both cores of the host machine, however the fact that it is simple to parallelize is one of the strengths of the genetic algorithm method (CPLEX also has a parallel version, but have not had an opportunity to test it since we do not have a license for it).

The largest basic block that is solved to optimality by the integer linear programming method is Block 115 of jpeg jcsample which contains 84 IR nodes (Table 2.4). After presolve it consists of 33627 variables and 16337 constraints and the solution time is 860 seconds. We also see that there are examples of basic blocks that are smaller in size (e.g. 30 IR nodes) that are not solved to optimality, hence 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).

When comparing the results of GA0 and GA1 we see that GA1 is better than GA0 in 9 cases and that GA0 is better than GA1 in 14 cases. They produce schedules with the same τ for the other 58 basic blocks. I.e. we can not conclude that one of the parameter sets is better than the other for all cases, but GA0 seems to perform slightly better on average.

		HS1		GA0		GA1		ILP	
$\|G\|$	BB	t(s)	τ	t(s)	τ	t(s)	τ	t(s)	τ
idct — inverse discrete cosine transform									
27	01	1	19	793	15	690	15	17	15
34	14	1	31	780	23	740	23	39	23
47	00	165	56	765	27	741	27	53	27
120	04	631	146	755	84	706	77	-	-
138	17	533	166	746	107	702	106	-	-
spatscal — spatial prediction									
35	33	1	60	780	24	741	25	48	24
44	51	2	57	777	33	735	33	82	29
46	71	2	57	775	40	740	40	146	39
predict — motion compensated prediction									
25	228	2	26	787	19	519	19	24	19
27	189	3	31	784	20	569	19	26	19
30	136	1	58	799	53	731	53	-	-
30	283	5	41	782	18	747	18	22	18
30	50	1	43	793	42	673	42	-	-
34	106	1	50	804	48	754	48	-	-
35	93	1	51	805	40	756	40	-	-
35	94	1	51	800	40	750	40	-	-
36	107	1	50	797	44	759	44	-	-
36	265	18	45	771	21	725	22	32	21
45	141	4	50	773	27	744	27	53	25
quantize — quantization									
26	120	1	38	808	20	587	20	24	19
26	145	1	37	807	19	611	19	23	18
30	11	1	38	813	25	754	25	37	25
31	29	1	42	802	27	750	27	43	26
transfrm — forward/inverse transform									
25	122	1	35	800	21	494	21	26	21
26	160	1	36	805	15	565	15	18	15
36	103	1	47	797	26	747	26	-	-
36	43	1	47	796	26	745	26	-	-
37	170	1	48	772	31	736	31	58	31

Table 2.3: 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	t(s)	τ	t(s)	τ	t(s)	τ
jcsample — downsampling									
26	94	1	41	791	19	613	19	24	19
31	91	1	42	810	29	761	29	48	29
34	143	1	40	798	29	745	29	51	29
58	137	1	89	761	38	724	41	325	38
84	115	55	135	755	44	713	47	860	40
85	133	2	142	782	72	713	72	-	-
109	111	53	186	737	74	697	74	-	-
jdcolor — colorspace conversion									
30	34	2	46	826	23	747	22	30	22
32	83	2	47	803	23	760	23	35	23
36	33	1	62	779	39	742	39	85	39
38	30	1	59	786	32	741	32	66	32
38	82	1	63	775	40	737	40	102	40
45	79	2	71	774	32	738	32	74	32
jdmerge — colorspace conversion									
39	63	1	69	788	30	739	30	52	30
53	89	1	89	767	37	733	37	118	37
55	110	1	100	773	41	724	41	135	41
55	78	1	100	767	41	729	41	135	41
62	66	1	103	760	41	729	42	333	41
62	92	1	103	762	41	729	41	339	41
jdsample — upsampling									
26	100	1	34	813	22	577	22	34	22
28	39	1	45	819	28	658	28	38	28
28	79	1	45	814	28	683	28	39	28
30	95	1	50	802	30	753	30	51	30
33	130	1	43	810	20	755	20	31	20
47	162	1	68	777	34	735	34	95	34
52	125	1	78	768	23	732	25	79	23
jfdctfst — forward discrete cosine transform									
60	06	2	70	760	39	732	39	172	39
60	20	2	70	765	39	730	39	173	39
76	02	3	87	754	42	718	43	-	-
76	16	3	87	772	45	714	42	-	-

Table 2.4: 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	t(s)	τ	t(s)	τ	t(s)	τ
jfdctint — forward discrete cosine transform									
74	06	23	81	761	33	728	31	536	28
74	20	23	81	750	34	725	34	533	28
78	02	4	88	755	44	713	46	-	-
80	16	4	89	764	43	717	45	-	-
jidctflt — inverse discrete cosine transform									
31	03	1	44	793	19	744	19	23	19
142	30	9	171	785	78	701	84	-	-
164	16	47	189	768	95	696	101	-	-
jidctfst — inverse discrete cosine transform									
31	03	1	44	792	19	735	19	24	19
44	30	1	61	781	20	736	21	29	19
132	43	7	160	801	78	703	71	-	-
162	16	46	196	786	102	704	99	-	-
jidctint — inverse discrete cosine transform									
31	03	1	44	790	19	737	19	24	19
44	30	1	61	777	19	736	20	29	19
166	43	275	204	773	134	696	127	-	-
191	16	576	226	758	165	690	161	-	-
jidctred — discrete cosine transform reduced output									
27	06	1	38	801	18	727	18	21	18
32	63	1	43	806	16	755	16	20	16
35	78	1	63	796	38	740	38	96	38
40	23	1	55	785	18	741	19	24	18
47	70	1	64	778	40	736	40	129	40
79	56	4	101	777	36	719	39	-	-
92	32	8	116	781	48	722	50	-	-
118	14	30	145	782	58	712	69	-	-

Table 2.5: 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 3

Integrated modulo scheduling

In this chapter we extend the integer linear programming model in Chapter 2 to modulo scheduling. We also show theoretical results on an upper bound for the number of schedule slots.

3.1 Introduction

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

The *back end* of a compiler transforms an intermediate representation into executable code. This transformation is usually performed in three phases: *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.

It is beneficial to integrate the phases of the code generation since this gives more opportunity for optimizations. However, this integration of phases comes at the cost of a greatly increased size of the solution space. In Chapter 2 we gave an integer linear program formulation for integrating instruction selection, instruction scheduling and register allocation. In this chapter we will show how to extend

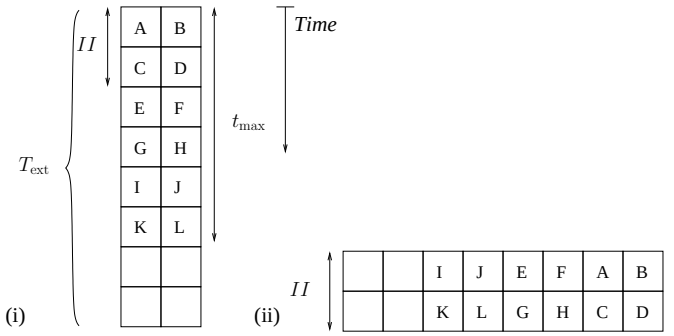

Figure 3.1: An example showing how an acyclic schedule (i) can be rearranged into a modulo schedule (ii), A-L are target instructions in this example.

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.

3.2 Extending the model to modulo scheduling

Software pipelining [16] is an optimization for loops where the following iterations of the loop are initiated before the current iteration is finished. One well known kind of software pipelining is *modulo scheduling* [76] where new iterations of the loop are issued at a fixed rate determined by the *initiation interval*. For every loop the initiation interval has a lower bound $MinII = \max(ResMII, RecMII)$, 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. Methods for calculating *RecMII* and *ResMII* are well documented in e.g. [52].

With some work the model in Section 2.1 can be extended to also integrate modulo scheduling. We note that a kernel can be formed

from the schedule of a basic block by scheduling each operation modulo the initiation interval, see Figure 3.1. The modulo schedules that we create have a corresponding acyclic schedule, and by the length of a modulo schedule we mean $t_{\max}$ of the acyclic schedule. We also note that 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: $E_1, E_2, E_m \subset V \times V \times \mathbb{N}$. The element (i, j, d) 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 2.11, 2.13 and 2.14, and modify Equation 2.16 to

$$\forall (i,j,d) \in E_m,\ \forall t \in T_{\text{ext}} \quad \sum_{p \in P} \sum_{t_j=0}^{t-II\cdot d} c_{j,p,1,t_j} + \sum_{p \in P} \sum_{t_i=t-L_p+1}^{t_{\max}+II\cdot d_{\max}} c_{i,p,1,t_i} \leq 1 \tag{3.1}$$

For this to work the initiation interval II must be a parameter to the solver. To find the best 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 3.3.

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 3.2 for an illustration.

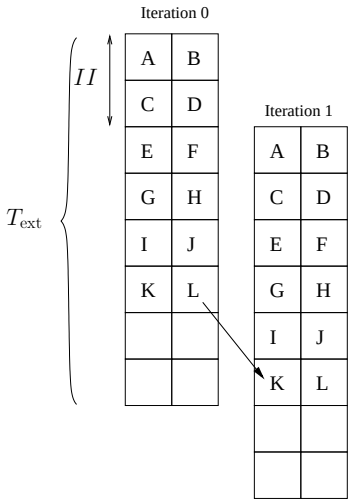

Figure 3.2: An example showing why T_{ext} has enough time slots to model the extended live ranges. Here $d_{\max} = 1$ and $II = 2$ so any live value from from Iteration 0 can not live after time slot $t_{\max} + II \cdot d_{\max}$ in the acyclic schedule.

3.2.1 Resource constraints

The inequalities in the previous section now only need a few further modifications to also do modulo scheduling. The resource constraints of the kind $\forall t \in T,\ \text{expr} \leq \text{bound}$ is 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, Inequality 2.17 becomes

$$\forall t_o \in \{0, 1, \ldots, II - 1\},\ \forall rr \in \mathcal{RS}, \quad \sum_{i \in V} \sum_{\substack{t \in T_{\text{ext}} \\ t \equiv t_o (\text{mod } II)}} r_{rr,i,t} \leq R_{rr} \tag{3.2}$$

Inequalities 2.18 and 2.19 are modified in the same way.

Inequality 3.2 guarantees that the number of live values in each register bank does not exceed the number of available registers. However 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 positive integer $k_i > 1$ the values in consecutive iterations can not use the same register for this value. We may solve this by doing *variable modulo expansion* [52].

3.2.2 Removing more variables

As we saw in Section 2.1.2 it is possible to improve the solution time for the integer linear programming model by removing variables whose values can be inferred.

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

$$soonest(i) = \max \left\{ \begin{array}{l} soonest'(i), \\ \max_{\substack{(j,i,d)\in E \\ d\neq 0}} \left(soonest'(j) + L_{\min}(j) - II \cdot d\right)\} \end{array} \right\} \tag{3.3}$$

$$latest(i) = \max \left\{ \begin{array}{l} latest'(i), \\ \max_{\substack{(i,j,d)\in E \\ d\neq 0}} \left(latest'(j) - L_{\min}(i) + II \cdot d\right)\} \end{array} \right\} \tag{3.4}$$

With these new derived parameters we create

$$T_i = \{soonest(i), \ldots, latest(i)\} \tag{3.5}$$

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.

Equations 3.3 and 3.4 differ from Equations 2.1 and 2.2 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, while *soonest* and *latest* can be derived parameters.

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 3.3: Pseudocode for the integrated modulo scheduling algorithm.

3.3 The algorithm

Figure 3.3 shows the algorithm for finding a modulo schedule. The algorithm explores a two-dimensional solution space as depicted in Figure 3.4. The dimensions in this solution space are number of schedule slots ($t_{\max}$) and throughput (II). Note that if there is no 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.

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

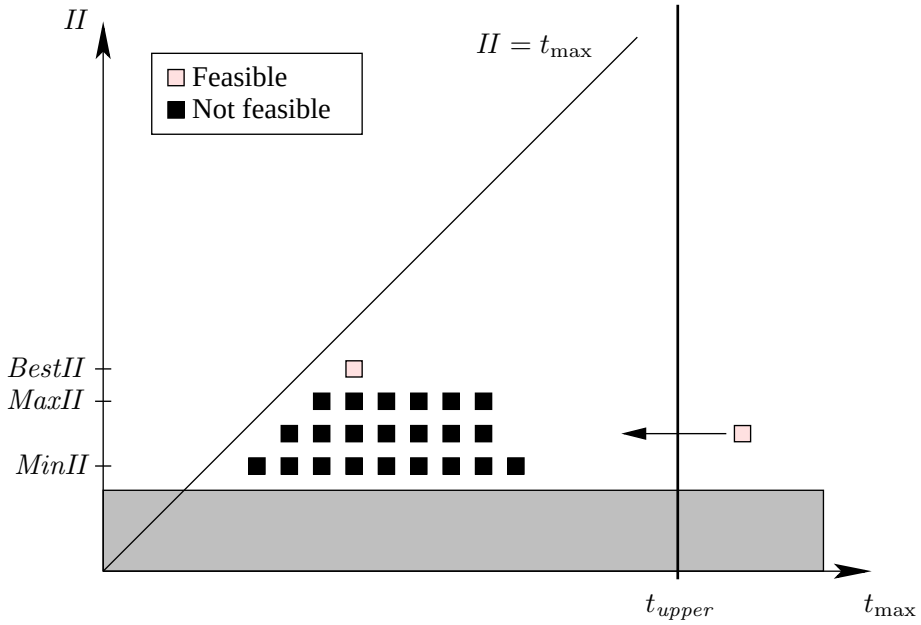

Figure 3.4: This figure shows the solution space of the algorithm. *BestII* is the best initiation interval found so far. For some architectures we can derive a bound, t_{upper}, on the number of schedule slots, $t_{\max}$, such that any solution to the right of t_{upper} can be moved to the left by a simple transformation.

problem with this approach is that the solution time of the integer linear program increases superlinearly with $t_{\max}$. Therefore we find that beginning with a low value of $t_{\max}$ and iteratively increase it 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.

3.3.1 Theoretical properties

In this section we will have a look at the theoretical properties of Algorithm 3.3 and show how the algorithm can be modified so that it finds optimal modulo schedules in finite time for a certain class of architectures.

Definition 2. *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 [49].*

Definition 3. *The* slack window *of an instruction i in a schedule s is a sequence of 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 4. *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.*

Lemma 5. *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 are 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 4.1 for such a case).

□

Corollary 6. *An n-dawdling schedule for the data flow graph (V, E) has length*

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

Lemma 7. *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. □

Theorem 8. *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 5, there exists an instruction i in s with a slack window larger than $2\tilde{II}-1$ and hence, by Lemma 7, 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 $t_{\max} \leq \sum_{i \in V}(\hat{L}(i) + 2\tilde{II} - 1)$. □

Corollary 9. *We can guarantee optimality in the algorithm in Section 3.3 for transfer free architectures if, every time we find an improved II, we set* $t_{\text{upper}} = \sum_{i \in V}(\hat{L}(i) + 2(II - 1) - 1)$.

3.4 Experiments

The experiments were run on a computer with an Athlon X2 6000+ processor with 4 GB RAM. The version of CPLEX is 10.2.

3.4.1 A contrived example

First let us consider an example that demonstrates how Corollary 9 can be used. Figure 3.5 shows a graph of an example program with 4 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 may 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 if it is the optimal solution. Now we can use Corollary 9 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 18 seconds by our algorithm.

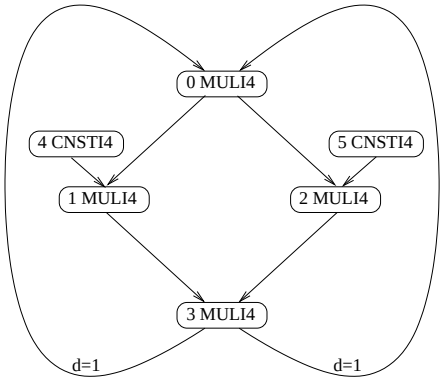

Figure 3.5: A loop body with 4 multiplications. The edges between Node 3 and Node 0 are loop carried dependences with distance 1.

3.4.2 DSPSTONE kernels

Table 3.1 shows the results of our experiments with the algorithm from Section 3.3. We used 5 target architectures, all variations of the Texas Instruments TI-C62x DSP processor [81]:

- single: single cluster, no MAC, no transfers and no spill,
- 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 taken from the DSPSTONE benchmark suite [85] and the dependence graphs were generated by hand. The columns marked *II* shows the found initiation interval and the columns marked $t_{\max}$ shows 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 3.3 *theoretically* produces optimal results for transfer free architectures with the t_{upper} bound of Corollary 9, 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, the algorithm finds a schedule with initiation interval within 3 of the optimum.

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 3.1: Experimental results with 5 DSPSTONE kernels on 5 different architectures.

Chapter 4

Related work

Much work has been done in the area of code generation and software pipelining. In this chapter we list some of the related work both for the basic blocks and for modulo scheduling.

4.1 Integrated code generation for basic blocks

4.1.1 Optimal methods

Kästner [43, 44] 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. [87] 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. [88] 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. [65]. 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.

Another integer linear programming method by Chang et al. [15] 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 [89] and shown that it can be used in a production compiler [90]. 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 here for integrated code generation is an extension of the model by Bednarski and Kessler [10]. 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 constants. 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. shl a,b). Previously the solution would have to

[1] A superblock is a block of code that has multiple exit points but only one entry point [40].

be analyzed afterwards to get the order of the arguments correct. We also remodeled the data flow dependences to work with the r variable, 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 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 [51]. Their method is based on dynamic programming and includes safe pruning of the solution space by removing comparable partial schedules.

4.1.2 Heuristic methods

Hanono and Devadas present an integrated approach to code generation for clustered VLIW architectures in the AVIV framework [37]. 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.

Lorenz et al. implemented a genetic algorithm for integrated code generation for low energy use [62] and for low execution time [63]. This genetic algorithm includes instruction selection, scheduling and register allocation in a single optimization problem. It is 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 2 where the genes are seen as preferences when creating the schedule.

Beaty did early work with genetic algorithms for the instruction scheduling problem [7]. The genetic algorithm presented in this thesis is largely based on work done by Skoog as a part of his Master's thesis project [79]. Other notable heuristic methods that integrate several phases of code generation for clustered VLIW have been proposed

by Kailas et al. [42], Özer et al. [73], Leupers [57] and Nagpal and Srikant [69].

4.2 Integrated software pipelining

4.2.1 Optimal methods

An enumeration approach to software pipelining, based on dynamic programming, was given by Vegdahl [86]. 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. [12] 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 number of schedule slots in a way that resembles our algorithm in Section 3.3. 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. [91] 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 [70] 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. [5] 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. [78].

Fimmel and Müller do optimal modulo scheduling for pipelined non-clustered architectures by optimizing a rational initiation interval [29, 30]. 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 [21].

Cortadella et al. have presented an integer linear programming method for finding optimal modulo schedules [19]. Nagarakatte and Govindarajan [68] 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.

4.2.2 Heuristic methods

Fernandes was the one who first described *clustered* VLIW architectures [27], and in [28] 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 [38] 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. [60, 61]. 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 [4] does register pressure responsive modulo scheduling by inserting instructions in such a way that known live ranges are minimized.

Stotzer and Leiss [80] presented a backtracking heuristic for modulo scheduling after instruction selection for Texas Instruments C6x processors.

Nystrom and Eichenberger presented a heuristic method for cluster assignment as a prepass to modulo scheduling [71]. 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. [18]. 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 [75].

Eisenbeis and Sawaya [22] 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.

4.2.3 Discussion of related work

The work presented 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 novel 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.

4.2.4 Theoretical results

Touati [82, 83] 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 8 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 [23] 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 8 in Chapter 3 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 counter example is depicted in Figure 4.1. 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 4.1 can not be shortened by 4 cycles as described in [23], since this would increase the live range of b and hence increase the register pressure of the resulting modulo schedule.

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

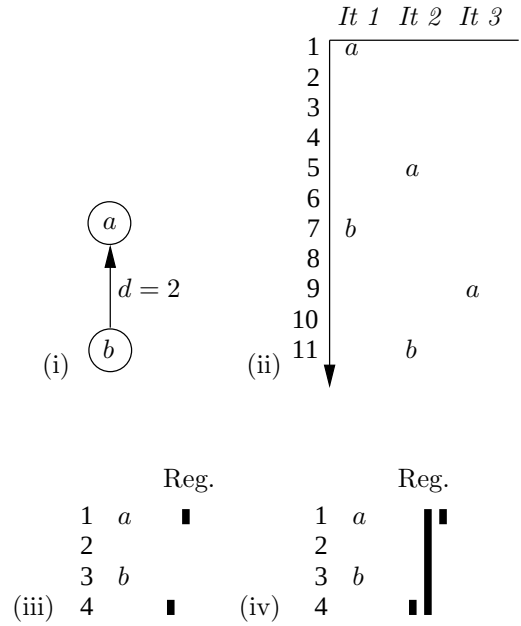

Figure 4.1: The graph in (i) can be modulo scheduled with initiation interval 4 as shown in (ii). If the schedule of an iteration is shortened by 4 cycles the register pressure of the corresponding modulo schedule kernel increases, see (iii) to (iv).

Chapter 5

Possible extensions

This thesis is, in many ways, a small step on the path to a dissertation (in all humility). Hence, the presented material in this thesis is a work in progress. In this chapter we identify a few possible directions for future work.

5.1 Quantify the improvement due to the integration of phases

A topic for future work is to 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 in which this could be achieved is to first use a state-of-the-art algorithm for cluster assignment and instruction selection. In this way we could define values for some of the solution variables in the integer linear programming model and then re-solve the instance, see Figure 5.1. This would allow for a comparison of the achieved initiation interval for the less integrated method to the more integrated one. With this method we could experimentally classify architectures into ones that are likely to benefit from the integration and ones that are not likely to benefit from integration.

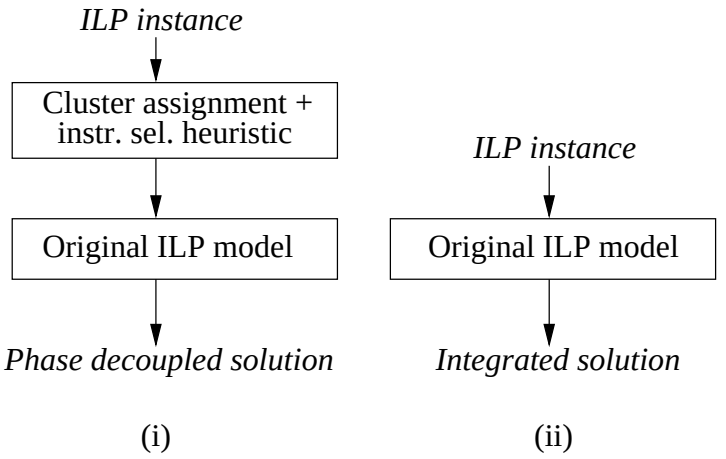

Figure 5.1: By applying a heuristic for instruction selection and cluster assignment as a preprocessing step to the ILP solver (i) we could quantify the improvement due to fully integrating the code generation phases (ii).

5.2 Improving the theoretical results

We also want to find ways to relax the constraints on the machine model in the theoretical part of Chapter 3. 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 [82], we should investigate if his methods can be applied to refine our results on the upper bound on the number of schedule slots.

5.3 Integrating other phases in code generation

In this thesis we have described how to integrate instruction selection, modulo scheduling, register allocation, cluster assignment and spill code generation. Another situation where it could be beneficial to integrate the phases of code generation is the *general offset assignment* problem (GOA) [6, 58, 59]. In the offset assignment problem the target architecture has an *address generation unit* (AGU) and one or several address registers that can be incremented or decremented automatically when a variable is accessed. Hence, for code generation, we have to optimize the use of the automatic increment and decrement, as well as memory layout of variables on the stack, as is done in e.g. [39, 74]. Taking it one step further we can also integrate scheduling of operations to maximize the utilization of automatic increment and decrement. A heuristic method for solving this integrated scheduling and general offset assignment problem was presented by Choi and Kim [17]. Solving the integrated scheduling and GOA problem optimally is an interesting challenge.

5.4 Genetic algorithms for modulo scheduling

One natural extension to the work presented here is to modify the genetic algorithm 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 Chapter 2.2 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.

5.5 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 removes 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 in [89] for global scheduling can be applied also to modulo scheduling.

5.5.1 Parallelization of code generators

The current trend in processor development points to increasing the number of cores on a chip. Since code optimization is computationally heavy we believe that it will become increasingly important to parallelize compilers in the future. We are interested in finding good ways to parallelize both the acyclic and the cyclic integrated code generation problems. The simplest way would be to use the parallel

version of CPLEX, another way would be to make our own parallelized branch and bound algorithm.

5.5.2 Reformulate the model to kernel population

A fundamental problem of our integer linear programming model is that it does not scale well with regard to the number of schedule slots. 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, create 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 [86]. 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.

5.6 Integration with a compiler framework

The graphs used for the evaluation of the modulo scheduling algorithm in Chapter 3 were created by hand. It would greatly simplify our experiments if we could generate such graphs automatically from an existing compiler. Optimist is built as a modification of LCC [32], which 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 algorithm with another, more modern, compiler such as GCC [33], LLVM [53], Trimaran [84] or Open64 [72]. A fully automated process would make it possible to perform extensive evaluations of our algorithms.

Chapter 6

Conclusions

In this thesis we have studied the problem of integrated code generation for clustered architectures. The phases that are integrated are: cluster assignment, instruction selection, scheduling, register allocation and spilling.

First, we addressed the basic block case, i.e. code generation for a block of code that contains no jump instructions, and no other jump targets than the beginning of the block. Two methods were developed. The first method is optimal and based on integer linear programming. The second method is a heuristic based on genetic algorithms.

Second, we extended the optimal method for basic blocks to modulo scheduling. To the best of our knowledge this is the first time anybody has optimally solved the modulo scheduling problem for clustered VLIW architectures with instruction selection and cluster assignment integrated.

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.

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.

In our experimental evaluation we have shown that for the basic block case we can optimally generate code for DAGs with up to 84 IR nodes in less than 900 seconds. But we also saw that in some cases with only 30 IR nodes the integer linear programming model was not successful. In our experiments we also saw that for the basic blocks where we have an optimal solution the genetic algorithm produces a schedule that is on average only 0.5 cycles worse. The experiments with modulo scheduling showed that our algorithm produces valid results for all the graphs that we tested, the largest one consisted of 38 IR nodes. For the larger tests the algorithm failed to prove optimality within the time limit, but all initiation intervals are within 3 of the theoretical lower bound.

We have also described topics for future work in the area of integrated code generation that we plan to investigate further.

Appendix A
ILP model and problem specifications

In this appendix we give listings of the integer linear program and problem instances formulation used in Chapter 3. The listings are shown in AMPL syntax. The architecture parameters are too verbose to be listed here.

A.1 Integrated software pipelining in AMPL code

```
# ---------------
# 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 match{G} within P;

set PN; # Generic pattern nodes

# Patterns
set B {P} within PN;
# Pattern edges (These need to be annotated with operands,
# eg. mul_arg1, mul_arg2 and add_arg1)
set EP {P_prime} within (PN cross PN);
# Operator of patterns
param OPP {P,PN} default 0;
# Out-degree of patterns
param ODP {P,PN} default 0;
# Latencies
param L {P} default 0; #nonissue have 0, the rest are listed

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: OPP[p,0] == OPG[i]} L[p];

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

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

#Residence classes
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;

# Residence class 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;
param latest  {i in G} in T;
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
set slots {i in G} := derived_soonest[i] .. derived_latest[i];

# 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 {EG,P_prime inter P_pruned, T, (PN cross PN)} 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 {G,RS,RS,TREG} binary default 0;
# availability
var r {RS,G,TREG} binary default 0;

# ---------------------
# Cuts
# ---------------------
subject to Cutr0 {i in G, t in T: t in slots[i]}:
  sum {tt in 0..t: tt not in slots[i]}
       (sum {rr in RS} r[rr,i,tt]
        +sum{rr in RS} sum {rs in RS} x[i,rr,rs,tt]) = 0;

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

# (7.1) 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;

# (7.2) 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];
```

```
# (7.5) 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 T, (k,l) in EP[p]
                          : t in slots[i] && t in slots[j] &&
                          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]);

# (7.5') For each pattern edge, assure that DFG nodes are matched
# to pattern nodes
subject to WithinPattern_0 {(i,j,d) in EG, p in P_prime inter
                          P_pruned, t in T, (k,l) in EP[p]
                          : t not in slots[i] || t not in slots[j] ||
                          p not in match[i] || p not in match[j]
                          }:
  w[i,j,d,p,t,k,l] = 0;

# (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}:
              #just ready
  r[rr,i,t] <= sum{p in (PRD[rr] inter match[i]): t >= L[p]
                  && t <= max_t && 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>0} 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 {p in P_prime inter P_pruned,
                                     (k,l) in EP[p], (i,j,d) in EG}:
  (sum{rr in RS} sum{t in TREG} r[rr,i,t])
  <= 1000*(1-sum{t in T}w[i,j,d,p,t,k,l]);
```

```
#data must be available when we use it
### For now, we only handle patterns that take all operands from the
### same residence class as the destination (PRD). To fix this, we
### need to annote edges within patterns.
subject to ForcedAvailability {(i,j,d) in EG,
                               t in slots[j], rr in RS}:
  r[rr,i,t+d*II]*5 >= 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
                          -
                          5 * sum{(k,l) in EP[p]}
                              w[i,j,d,p,t,k,l]);

##(10)&(11) singletons
subject to ForcedAvailability2 {(i,j,d) in EG0, t in slots[j],
                                rr in RS}:
  r[rr,i,t+d*II]*5
  >=
  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]*5
  >=
  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}:
  r[rr,i,t] >= 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;
```

```
# (~7.11) 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}
         r[rr,i,t] <= R[rr];

# ----------------------
# Instruction scheduling
# ----------------------
# Conditions for scheduling included is included in the other
# constraints.

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

# (7.14) 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])
  # and extra for transfers
  +
  sum{t in t_offs..(max_t+II*max_d)}(
    sum{i in G} sum{(rr,rq) in (RS cross RS): UX[rr,rq,f] = 1 }
      x[i,rr,rq,t])
 <= 1;

# (7.15) 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])
# and extra for transfers
  +
  sum{t in t_offs..(max_t+II*max_d)}
```

```
    (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 som functional unit
check {p in P_prime union P_second}:
  sum{f in F} U[p,f] > 0;
```

A.2 Data flow graphs used in experiments

A.2.1 Dot product

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

param max_d := 1;

param: G : OPG ODG :=
  2  4405 1 # ADDI4
  4  4565 1 # MULI4
  5  4165 1 # INDIRI4
  6  4407 1 # ADDP4
  7  4437 2 # LSHI4
  8  4165 1 # INDIRI4
  9  4391 1 # ADDRLP4
  10 4117 1 # CNSTI4
  11 4167 1 # INDIRP4
  12 4359 1 # ADDRGP4
  13 4165 1 # INDIRI4
  14 4407 1 # ADDP4
  15 4167 1 # INDIRP4
  16 4359 1 # 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.2.2 FIR filter

```
# RecMII = 1
# ResMII = 3 (3 add 2 sub 2 shi 6 loads)
# sum_l = 39
# Critical path 15

param max_d := 1;

param: G : OPG ODG :=
  2  4405 1 # ADDI4
  4  4565 1 # MULI4
  5  2117 1 # INDIRI2
  6  4407 1 # ADDP4
  7  4421 1 # SUBI4
  8  4437 1 # LSHI4
  9  4165 2 # INDIRI4
  10 4391 1 # ADDRLP4
  11 4117 2 # CNSTI4
  12 4117 1 # CNSTI4
  13 4167 1 # INDIRP4
  14 4375 1 # ADDRFP4
  15 2117 1 # INDIRI2
  16 4407 1 # ADDP4
  17 4437 1 # LSHI4
  18 4421 1 # SUBI4
  19 4165 1 # INDIRI4
  20 4391 1 # ADDRLP4
  21 4167 1 # INDIRP4
  22 4375 1 # ADDRFP4
;

set EG0 :=
  (2,2,1) (10,9,0) (9,8,0) (8,7,0) (7,6,0) (14,13,0) (6,5,0) (5,4,0)
```

```
  (20,19,0) (19,18,0) (18,17,0) (17,16,0) (22,21,0) (16,15,0);

set EG1 :=
  (11,8,0) (12,7,0) (13,6,0) (9,18,0) (11,17,0) (21,16,0) (15,4,0)
  (4,2,0);

set EGDEP :=
;
```

A.2.3 N complex updates

```
param max_d = 1;

#RecMII = 2
# 11 add 8 load 2 store 4 mul 1 sub
#ResMII = 4
#sum_l = 56

param: G : OPG ODG :=
  0  4117 3 # CNSTI4 (4)
  1  4407 1 # ADDP4 1-8 i.v.
  2  4407 1 # ADDP4
  3  4407 1 # ADDP4
  4  4407 1 # ADDP4
  5  4407 1 # ADDP4
  6  4407 1 # ADDP4
  7  4407 1 # ADDP4
  8  4407 1 # ADDP4
  11 4167 3 # INDIRP4 a0
  12 4167 3 # INDIRP4 a1
  13 4167 3 # INDIRP4 b0
  14 4167 3 # INDIRP4 b1
  15 4167 3 # INDIRP4 c0
  16 4167 3 # INDIRP4 c1
  17 4167 3 # INDIRP4 d0
  18 4167 3 # INDIRP4 d1
  19 4149 0 # ASGNI4 set *d0 G'
  20 4149 0 # ASGNI4 set *d1 G''
###G'
  21 4565 1 # MULI4
  22 4565 1 # MULI4
  23 4405 1 # ADDI4
  24 4421 1 # SUBI4
```

```
###G''
  25 4565 1 # MULI4
  26 4565 1 # MULI4
  27 4405 1 # ADDI4
  28 4405 1 # ADDI4
;

set EG0 :=
### G'
  (11,21,0) #a0 (15,23,0) #c0 (12,22,0) #a1 (23,24,0) (17,19,0)
### G''
  (12,25,0) #a1 (11,26,0) #a0 (16,27,0) #c1 (27,28,0) (18,20,0)
### Main graph
  (0,1,0) (0,2,0) (0,3,0) (0,4,0) (0,5,0) (0,6,0) (0,7,0) (0,8,0)
#### indir
  (1,18,0) (3,16,0) (5,14,0) (7,12,0) (2,17,1) (4,15,1) (6,13,1)
  (8,11,1);

set EG1 :=
### G'
  (13,21,0) # b0 (14,22,0) # b1 (21,23,0) (22,24,0)
  (24,19,0) # connector
### G''
  (13,25,0) # b0 (14,26,0) # b1 (25,27,0) (26,28,0)
  (28,20,0) # connector
### Main graph
#### Add connectors
  (1,2,0) (3,4,0) (5,6,0) (7,8,0) (2,1,1) (4,3,1) (6,5,1) (8,7,1);

set EGDEP :=
;
```

A.2.4 Biquad n

```
#9 loads, gives ResMII=9 for single cluster archi and
# ResMII=5 for 2 cluster.
#11 adds, 2 subs
#5 muls
#2 stores
#sum_l=70
#critical cycle 11 13 17 18 gives RecMII=5

param max_d := 1;
```

```
param: G : OPG ODG :=
  0  4117 10 # CNSTI4
  1  4407 1  # ADDP4
  2  4407 1  # ADDP4
  3  4407 1  # ADDP4
  4  4407 1  # ADDP4
  5  4407 1  # ADDP4
  6  4407 1  # ADDP4
  7  4407 1  # ADDP4
  8  4407 1  # ADDP4
  9  4407 1  # ADDP4
  10 4565 1  # MULI4
  12 4565 1  # MULI4
  14 4565 1  # MULI4
  15 4565 1  # MULI4
  17 4565 1  # MULI4
  11 4421 1  # SUBI4
  13 4421 1  # SUBI4
  16 4405 1  # ADDI4
  18 4405 1  # ADDI4
  20 4149 1  # ASGNI4
  21 4149 1  # ASGNI4
  31 4167 1  # INDIRP4
  32 4167 1  # INDIRP4
  33 4167 1  # INDIRP4
  34 4167 1  # INDIRP4
  35 4167 1  # INDIRP4
  36 4167 1  # INDIRP4
  37 4167 1  # INDIRP4
  38 4167 1  # INDIRP4
  39 4167 1  # INDIRP4
;

set EG0 :=
# pointer increments
  (0,1,0) (0,2,0) (0,3,0) (0,4,0) (0,5,0) (0,6,0) (0,7,0) (0,8,0)
  (0,9,0)
# arithmetic
  (35,10,1) (18,11,1) (11,13,0) (32,17,0) (17,18,0) (31,12,0)
  (33,14,0) (14,16,0) (34,15,0)
# stores
  (8,20,0) (6,21,0)
# indirs
```

```
  (1,31,0) (2,32,0) (3,33,0) (4,34,0) (5,35,1) (6,36,0) (7,37,1)
  (8,38,0) (9,39,1);

set EG1 :=
# pointer increments chain
  (1,2,0) (2,3,0) (3,4,0) (4,5,0) (5,1,1) (6,7,0) (7,6,1) (8,9,0)
  (9,8,1)
# arithmetic
  (37,10,1) (10,11,0) (39,12,0) (12,13,0) (13,17,0) (36,14,0)
  (38,15,0) (15,16,0) (16,18,0) (36,20,0) (13,21,0)
;

set EGDEP :=
  (20,21,1)
;
```

A.2.5 IIR filter

```
param max_d := 1;

# RecMII = 18 (no patterns) (17 with mac)
# sum_l 3*1 store + 12*1 add + 4*2 mul + 2*1 shi + 13*5 = 90
# Critical cycle 3,2,0,34,33,32,28,27,25,24,3

param: G : OPG ODG :=
  0  2101 9 # ASGNI2
  1  4391 2 # ADDRLP4
  2  4405 1 # ADDI4
  3  2117 1 # INDIRI2
  4  4391 2 # ADDRLP4
  5  4469 1 # RSHI4
  6  4405 1 # ADDI4
  7  4565 1 # MULI4
  8  2117 1 # INDIRI2
  9  4407 1 # ADDP4
  10 4167 4 # INDIRP4
  11 4391 1 # ADDRLP4
  12 4117 1 # CNSTI4
  13 2117 1 # INDIRI2
  14 4167 4 # INDIRP4
  15 4391 1 # ADDRLP4
  16 4565 1 # MULI4
  17 2117 1 # INDIRI2
```

```
  18 4407 1 # ADDP4
  19 4117 1 # CNSTI4
  20 2117 1 # INDIRI2
  21 4407 3 # ADDP4
  22 4117 2 # CNSTI4
  23 4117 2 # CNSTI4
  24 2101 0 # ASGNI2
  25 4405 1 # ADDI4
  26 2117 1 # INDIRI2
  27 4469 1 # RSHI4
  28 4405 1 # ADDI4
  29 4565 1 # MULI4
  30 2117 1 # INDIRI2
  31 2117 1 # INDIRI2
  32 4565 1 # MULI4
  33 2117 1 # INDIRI2
  34 4407 1 # ADDP4
  35 2117 1 # INDIRI2
  36 2101 0 # ASGNI2
  37 2117 1 # INDIRI2
;

set EG0 :=
  (1,0,0) (4,3,0) (3,2,0) (11,10,0) (10,9,0) (9,8,0) (8,7,0)
  (15,14,0) (14,13,0) (7,6,0) (10,18,0) (18,17,0) (17,16,0)
  (14,21,0) (21,20,0) (6,5,0) (4,24,0) (1,26,0) (26,25,0)
  (10,30,0) (30,29,0) (14,31,0) (29,28,0) (10,34,0) (34,33,0)
  (33,32,0) (21,35,0) (28,27,0) (21,36,0) (14,37,0);

set EG1 :=
  (12,9,0) (13,7,0) (19,18,0) (22,21,0) (20,16,0) (16,6,0)
  (23,5,0) (5,2,0) (2,0,0) (31,29,0) (22,34,0) (35,32,0)
  (32,28,0) (23,27,0) (27,25,0) (25,24,0) (37,36,0);

set EGDEP :=
  (24,3,1) (0,24,0) (0,26,0) (0,30,0) (0,31,0) (0,34,0) (0,35,0)
  (0,27,0) (0,36,0) (0,37,0);
```

Bibliography

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

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

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

[4] 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.

[5] 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.

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

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] Boost C++ Libraries. Homepage. http://www.boost.org.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

[21] 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.

[22] 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.

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

[24] 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.

[25] 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.

[26] 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.

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

[28] 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.

[29] 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.

[30] 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.

[31] 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.

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

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

[34] 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.

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

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

[37] 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.

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

[39] Johnny Huynh, José Amaral, Paul Berube, and Sid Touati. Evaluation of offset assignment heuristics. In *International Conference on High Performance Embedded Architectures and Compilers (HiPEAC2007)*, pages 261–275, Ghent, January 2007.

[40] 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.

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

[42] 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.

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

[44] 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.

[45] 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.

[46] 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.

[47] 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.

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

[49] 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.

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

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

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

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

[54] 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.

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

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

[57] 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.

[58] Stan Liao, Srinivas Devadas, Kurt Keutzer, Steve Tjiang, and Albert Wang. Storage assignment to decrease code size. In

ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pages 186–195, 1995.

[59] 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.

[60] J. Llosa, A. Gonzalez, E. Ayguade, and M. Valero. Swing modulo scheduling: a lifetime-sensitive approach. *Proceedings of the Conference on Parallel Architectures and Compilation Techniques*, pages 80–86, Oct 1996.

[61] 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.

[62] 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.

[63] 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.

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

[65] 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.

[66] 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.

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

[68] 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.

[69] 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.

[70] 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.

[71] 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.

[72] Open research compiler. Homepage. http://www.open64.net.

[73] 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.

[74] O. Ozturk, M. Kandemir, and S. Tosun. An ILP based approach to address code generation for digital signal processors. In *GLSVLSI '06: Proceedings of the 16th ACM Great Lakes symposium on VLSI*, pages 37–42, New York, NY, USA, 2006. ACM.

[75] 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.

[76] 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.

[77] 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.

[78] 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.

[79] Oskar Skoog. Evaluation of heuristic algorithms for code generation. Master's thesis, Linköping university, forthcoming.

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

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

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

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

[84] Trimaran compiler. Homepage. http://www.trimaran.org.

[85] 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.

[86] 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.

[87] 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.

[88] 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.

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

[90] 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.

[91] 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.

LINKÖPINGS UNIVERSITET

Avdelning, Institution
Division, department

Institutionen för datavetenskap

Department of Computer and Information Science

Datum
Date

2009-02-16

Språk
Language

☐ Svenska/Swedish
☒ Engelska/English
☐ _____

Rapporttyp
Report: category

☒ Licentiatavhandling
☐ Examensarbete
☐ C-uppsats
☐ D-uppsats
☐ Övrig rapport
☐ _____

ISBN 978-91-7393-699-6

ISRN LiU-Tek-Lic-2009-1

Serietitel och serienummer
Title of series, numbering

ISSN 0280-7971

Linköping Studies in Science and Technology

Thesis No. 1393

URL för elektronisk version

http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-16170

Titel
Title

Integrated Software Pipelining

Författare
Author

Mattias Eriksson

Sammandrag
Abstract

In this thesis we address the problem of integrated software pipelining for clustered VLIW architectures. The phases that are integrated and solved as one combined problem are: cluster assignment, instruction selection, scheduling, register allocation and spilling.

As a first step we describe two methods for integrated code generation of basic blocks. The first method is optimal and based on integer linear programming. The second method is a heuristic based on genetic algorithms.

We then extend the integer linear programming model to modulo scheduling. To the best of our knowledge this is the first time anybody has optimally solved the modulo scheduling problem for clustered architectures with instruction selection and cluster assignment integrated.

We also show 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 the memory and have transfer instructions to and from this register file corresponding to stores and loads.

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.

Experiments have been conducted to show the usefulness and limitations of our optimal methods. For the basic block case we compare the optimal method to the heuristic based on genetic algorithms.

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

Nyckelord
Keywords

Code generation, compilers, instruction scheduling, register allocation, spill code generation, modulo scheduling, integer linear programming, genetic programming.

Department of Computer and Information Science
Linköpings universitet

Linköping Studies in Science and Technology
Faculty of Arts and Sciences - Licentiate Theses

No 17 **Vojin Plavsic:** Interleaved Processing of Non-Numerical Data Stored on a Cyclic Memory. (Available at: FOA, Box 1165, S-581 11 Linköping, Sweden. FOA Report B30062E)

No 28 **Arne Jönsson, Mikael Patel:** An Interactive Flowcharting Technique for Communicating and Realizing Algorithms, 1984.

No 29 **Johnny Eckerland:** Retargeting of an Incremental Code Generator, 1984.

No 48 **Henrik Nordin:** On the Use of Typical Cases for Knowledge-Based Consultation and Teaching, 1985.

No 52 **Zebo Peng:** Steps Towards the Formalization of Designing VLSI Systems, 1985.

No 60 **Johan Fagerström:** Simulation and Evaluation of Architecture based on Asynchronous Processes, 1985.

No 71 **Jalal Maleki:** ICONStraint, A Dependency Directed Constraint Maintenance System, 1987.

No 72 **Tony Larsson:** On the Specification and Verification of VLSI Systems, 1986.

No 73 **Ola Strömfors:** A Structure Editor for Documents and Programs, 1986.

No 74 **Christos Levcopoulos:** New Results about the Approximation Behavior of the Greedy Triangulation, 1986.

No 104 **Shamsul I. Chowdhury: S**tatistical Expert Systems - a Special Application Area for Knowledge-Based Computer Methodology, 1987.

No 108 **Rober Bilos:** Incremental Scanning and Token-Based Editing, 1987.

No 111 **Hans Block:** SPORT-SORT Sorting Algorithms and Sport Tournaments, 1987.

No 113 **Ralph Rönnquist:** Network and Lattice Based Approaches to the Representation of Knowledge, 1987.

No 118 **Mariam Kamkar, Nahid Shahmehri:** Affect-Chaining in Program Flow Analysis Applied to Queries of Programs, 1987.

No 126 **Dan Strömberg:** Transfer and Distribution of Application Programs, 1987.

No 127 **Kristian Sandahl:** Case Studies in Knowledge Acquisition, Migration and User Acceptance of Expert Systems, 1987.

No 139 **Christer Bäckström:** Reasoning about Interdependent Actions, 1988.

No 140 **Mats Wirén:** On Control Strategies and Incrementality in Unification-Based Chart Parsing, 1988.

No 146 **Johan Hultman:** A Software System for Defining and Controlling Actions in a Mechanical System, 1988.

No 150 **Tim Hansen:** Diagnosing Faults using Knowledge about Malfunctioning Behavior, 1988.

No 165 **Jonas Löwgren:** Supporting Design and Management of Expert System User Interfaces, 1989.

No 166 **Ola Petersson:** On Adaptive Sorting in Sequential and Parallel Models, 1989.

No 174 **Yngve Larsson:** Dynamic Configuration in a Distributed Environment, 1989.

No 177 **Peter Åberg:** Design of a Multiple View Presentation and Interaction Manager, 1989.

No 181 **Henrik Eriksson:** A Study in Domain-Oriented Tool Support for Knowledge Acquisition, 1989.

No 184 **Ivan Rankin:** The Deep Generation of Text in Expert Critiquing Systems, 1989.

No 187 **Simin Nadjm-Tehrani:** Contributions to the Declarative Approach to Debugging Prolog Programs, 1989.

No 189 **Magnus Merkel:** Temporal Information in Natural Language, 1989.

No 196 **Ulf Nilsson:** A Systematic Approach to Abstract Interpretation of Logic Programs, 1989.

No 197 **Staffan Bonnier:** Horn Clause Logic with External Procedures: Towards a Theoretical Framework, 1989.

No 203 **Christer Hansson:** A Prototype System for Logical Reasoning about Time and Action, 1990.

No 212 **Björn Fjellborg:** An Approach to Extraction of Pipeline Structures for VLSI High-Level Synthesis, 1990.

No 230 **Patrick Doherty:** A Three-Valued Approach to Non-Monotonic Reasoning, 1990.

No 237 **Tomas Sokolnicki:** Coaching Partial Plans: An Approach to Knowledge-Based Tutoring, 1990.

No 250 **Lars Strömberg:** Postmortem Debugging of Distributed Systems, 1990.

No 253 **Torbjörn Näslund:** SLDFA-Resolution - Computing Answers for Negative Queries, 1990.

No 260 **Peter D. Holmes:** Using Connectivity Graphs to Support Map-Related Reasoning, 1991.

No 283 **Olof Johansson:** Improving Implementation of Graphical User Interfaces for Object-Oriented Knowledge-Bases, 1991.

No 298 **Rolf G Larsson:** Aktivitetsbaserad kalkylering i ett nytt ekonomisystem, 1991.

No 318 **Lena Strömbäck:** Studies in Extended Unification-Based Formalism for Linguistic Description: An Algorithm for Feature Structures with Disjunction and a Proposal for Flexible Systems, 1992.

No 319 **Mikael Pettersson:** DML-A Language and System for the Generation of Efficient Compilers from Denotational Specification, 1992.

No 326 **Andreas Kågedal:** Logic Programming with External Procedures: an Implementation, 1992.

No 328 **Patrick Lambrix:** Aspects of Version Management of Composite Objects, 1992.

No 333 **Xinli Gu:** Testability Analysis and Improvement in High-Level Synthesis Systems, 1992.

No 335 **Torbjörn Näslund:** On the Role of Evaluations in Iterative Development of Managerial Support Sytems, 1992.

No 348 **Ulf Cederling:** Industrial Software Development - a Case Study, 1992.

No 352 **Magnus Morin:** Predictable Cyclic Computations in Autonomous Systems: A Computational Model and Implementation, 1992.

No 371 **Mehran Noghabai:** Evaluation of Strategic Investments in Information Technology, 1993.

No 378 **Mats Larsson:** A Transformational Approach to Formal Digital System Design, 1993.

No 380 **Johan Ringström:** Compiler Generation for Parallel Languages from Denotational Specifications, 1993.

No 381 **Michael Jansson:** Propagation of Change in an Intelligent Information System, 1993.

No 383 **Jonni Harrius:** An Architecture and a Knowledge Representation Model for Expert Critiquing Systems, 1993.

No 386 **Per Österling:** Symbolic Modelling of the Dynamic Environments of Autonomous Agents, 1993.

No 398 **Johan Boye:** Dependency-based Groudness Analysis of Functional Logic Programs, 1993.

No 402	**Lars Degerstedt:** Tabulated Resolution for Well Founded Semantics, 1993.
No 406	**Anna Moberg:** Satellitkontor - en studie av kommunikationsmönster vid arbete på distans, 1993.
No 414	**Peter Carlsson:** Separation av företagsledning och finansiering - fallstudier av företagsledarutköp ur ett agentteoretiskt perspektiv, 1994.
No 417	**Camilla Sjöström:** Revision och lagreglering - ett historiskt perspektiv, 1994.
No 436	**Cecilia Sjöberg:** Voices in Design: Argumentation in Participatory Development, 1994.
No 437	**Lars Viklund:** Contributions to a High-level Programming Environment for a Scientific Computing, 1994.
No 440	**Peter Loborg:** Error Recovery Support in Manufacturing Control Systems, 1994.
FHS 3/94	**Owen Eriksson:** Informationssystem med verksamhetskvalitet - utvärdering baserat på ett verksamhetsinriktat och samskapande perspektiv, 1994.
FHS 4/94	**Karin Pettersson:** Informationssystemstrukturering, ansvarsfördelning och användarinflytande - En komparativ studie med utgångspunkt i två informationssystemstrategier, 1994.
No 441	**Lars Poignant:** Informationsteknologi och företagsetablering - Effekter på produktivitet och region, 1994.
No 446	**Gustav Fahl:** Object Views of Relational Data in Multidatabase Systems, 1994.
No 450	**Henrik Nilsson:** A Declarative Approach to Debugging for Lazy Functional Languages, 1994.
No 451	**Jonas Lind:** Creditor - Firm Relations: an Interdisciplinary Analysis, 1994.
No 452	**Martin Sköld:** Active Rules based on Object Relational Queries - Efficient Change Monitoring Techniques, 1994.
No 455	**Pär Carlshamre:** A Collaborative Approach to Usability Engineering: Technical Communicators and System Developers in Usability-Oriented Systems Development, 1994.
FHS 5/94	**Stefan Cronholm:** Varför CASE-verktyg i systemutveckling? - En motiv- och konsekvensstudie avseende arbetssätt och arbetsformer, 1994.
No 462	**Mikael Lindvall:** A Study of Traceability in Object-Oriented Systems Development, 1994.
No 463	**Fredrik Nilsson:** Strategi och ekonomisk styrning - En studie av Sandviks förvärv av Bahco Verktyg, 1994.
No 464	**Hans Olsén:** Collage Induction: Proving Properties of Logic Programs by Program Synthesis, 1994.
No 469	**Lars Karlsson:** Specification and Synthesis of Plans Using the Features and Fluents Framework, 1995.
No 473	**Ulf Söderman:** On Conceptual Modelling of Mode Switching Systems, 1995.
No 475	**Choong-ho Yi:** Reasoning about Concurrent Actions in the Trajectory Semantics, 1995.
No 476	**Bo Lagerström:** Successiv resultatavräkning av pågående arbeten. - Fallstudier i tre byggföretag, 1995.
No 478	**Peter Jonsson:** Complexity of State-Variable Planning under Structural Restrictions, 1995.
FHS 7/95	**Anders Avdic:** Arbetsintegrerad systemutveckling med kalkylkprogram, 1995.
No 482	**Eva L Ragnemalm:** Towards Student Modelling through Collaborative Dialogue with a Learning Companion, 1995.
No 488	**Eva Toller:** Contributions to Parallel Multiparadigm Languages: Combining Object-Oriented and Rule-Based Programming, 1995.
No 489	**Erik Stoy:** A Petri Net Based Unified Representation for Hardware/Software Co-Design, 1995.
No 497	**Johan Herber:** Environment Support for Building Structured Mathematical Models, 1995.
No 498	**Stefan Svenberg:** Structure-Driven Derivation of Inter-Lingual Functor-Argument Trees for Multi-Lingual Generation, 1995.
No 503	**Hee-Cheol Kim:** Prediction and Postdiction under Uncertainty, 1995.
FHS 8/95	**Dan Fristedt:** Metoder i användning - mot förbättring av systemutveckling genom situationell metodkunskap och metodanalys, 1995.
FHS 9/95	**Malin Bergvall:** Systemförvaltning i praktiken - en kvalitativ studie avseende centrala begrepp, aktiviteter och ansvarsroller, 1995.
No 513	**Joachim Karlsson:** Towards a Strategy for Software Requirements Selection, 1995.
No 517	**Jakob Axelsson:** Schedulability-Driven Partitioning of Heterogeneous Real-Time Systems, 1995.
No 518	**Göran Forslund:** Toward Cooperative Advice-Giving Systems: The Expert Systems Experience, 1995.
No 522	**Jörgen Andersson:** Bilder av småföretagares ekonomistyrning, 1995.
No 538	**Staffan Flodin:** Efficient Management of Object-Oriented Queries with Late Binding, 1996.
No 545	**Vadim Engelson:** An Approach to Automatic Construction of Graphical User Interfaces for Applications in Scientific Computing, 1996.
No 546	**Magnus Werner :** Multidatabase Integration using Polymorphic Queries and Views, 1996.
FiF-a 1/96	**Mikael Lind:** Affärsprocessinriktad förändringsanalys - utveckling och tillämpning av synsätt och metod, 1996.
No 549	**Jonas Hallberg:** High-Level Synthesis under Local Timing Constraints, 1996.
No 550	**Kristina Larsen:** Förutsättningar och begränsningar för arbete på distans - erfarenheter från fyra svenska företag. 1996.
No 557	**Mikael Johansson:** Quality Functions for Requirements Engineering Methods, 1996.
No 558	**Patrik Nordling:** The Simulation of Rolling Bearing Dynamics on Parallel Computers, 1996.
No 561	**Anders Ekman:** Exploration of Polygonal Environments, 1996.
No 563	**Niclas Andersson:** Compilation of Mathematical Models to Parallel Code, 1996.
No 567	**Johan Jenvald:** Simulation and Data Collection in Battle Training, 1996.
No 575	**Niclas Ohlsson:** Software Quality Engineering by Early Identification of Fault-Prone Modules, 1996.
No 576	**Mikael Ericsson:** Commenting Systems as Design Support—A Wizard-of-Oz Study, 1996.
No 587	**Jörgen Lindström:** Chefers användning av kommunikationsteknik, 1996.
No 589	**Esa Falkenroth:** Data Management in Control Applications - A Proposal Based on Active Database Systems, 1996.
No 591	**Niclas Wahllöf:** A Default Extension to Description Logics and its Applications, 1996.
No 595	**Annika Larsson:** Ekonomisk Styrning och Organisatorisk Passion - ett interaktivt perspektiv, 1997.
No 597	**Ling Lin:** A Value-based Indexing Technique for Time Sequences, 1997.

No 598 **Rego Granlund:** C^3Fire - A Microworld Supporting Emergency Management Training, 1997.
No 599 **Peter Ingels:** A Robust Text Processing Technique Applied to Lexical Error Recovery, 1997.
No 607 **Per-Arne Persson:** Toward a Grounded Theory for Support of Command and Control in Military Coalitions, 1997.
No 609 **Jonas S Karlsson:** A Scalable Data Structure for a Parallel Data Server, 1997.
FiF-a 4 **Carita Åbom:** Videomötesteknik i olika affärssituationer - möjligheter och hinder, 1997.
FiF-a 6 **Tommy Wedlund**: Att skapa en företagsanpassad systemutvecklingsmodell - genom rekonstruktion, värdering och vidareutveckling i T50-bolag inom ABB, 1997.
No 615 **Silvia Coradeschi**: A Decision-Mechanism for Reactive and Coordinated Agents, 1997.
No 623 **Jan Ollinen:** Det flexibla kontorets utveckling på Digital - Ett stöd för multiflex? 1997.
No 626 **David Byers:** Towards Estimating Software Testability Using Static Analysis, 1997.
No 627 **Fredrik Eklund:** Declarative Error Diagnosis of GAPLog Programs, 1997.
No 629 **Gunilla Ivefors:** Krigsspel coh Informationsteknik inför en oförutsägbar framtid, 1997**.**
No 631 **Jens-Olof Lindh:** Analysing Traffic Safety from a Case-Based Reasoning Perspective, 1997
No 639 **Jukka Mäki-Turja:**. Smalltalk - a suitable Real-Time Language, 1997.
No 640 **Juha Takkinen**: CAFE: Towards a Conceptual Model for Information Management in Electronic Mail, 1997.
No 643 **Man Lin**: Formal Analysis of Reactive Rule-based Programs, 1997.
No 653 **Mats Gustafsson**: Bringing Role-Based Access Control to Distributed Systems, 1997.
FiF-a 13 **Boris Karlsson:** Metodanalys för förståelse och utveckling av systemutvecklingsverksamhet. Analys och värdering av systemutvecklingsmodeller och dess användning, 1997.
No 674 **Marcus Bjäreland:** Two Aspects of Automating Logics of Action and Change - Regression and Tractability, 1998.
No 676 **Jan Håkegård**: Hiera rchical Test Architecture and Board-Level Test Controller Synthesis, 1998.
No 668 **Per-Ove Zetterlund**: Normering av svensk redovisning - En studie av tillkomsten av Redovisningsrådets rekommendation om koncernredovisning (RR01:91), 1998.
No 675 **Jimmy Tjäder**: Projektledaren & planen - en studie av projektledning i tre installations- och systemutvecklingsprojekt, 1998.
FiF-a 14 **Ulf Melin**: Informationssystem vid ökad affärs- och processorientering - egenskaper, strategier och utveckling, 1998.
No 695 **Tim Heyer**: COMPASS: Introduction of Formal Methods in Code Development and Inspection, 1998.
No 700 **Patrik Hägglund:** Programming Languages for Computer Algebra, 1998.
FiF-a 16 **Marie-Therese Christiansson:** Inter-organistorisk verksamhetsutveckling - metoder som stöd vid utveckling av partnerskap och informationssystem, 1998.
No 712 **Christina Wennestam:** Information om immateriella resurser. Investeringar i forskning och utveckling samt i personal inom skogsindustrin, 1998.
No 719 **Joakim Gustafsson:** Extending Temporal Action Logic for Ramification and Concurrency, 1998.
No 723 **Henrik André-Jönsson:** Indexing time-series data using text indexing methods, 1999.
No 725 **Erik Larsson:** High-Level Testability Analysis and Enhancement Techniques, 1998.
No 730 **Carl-Johan Westin:** Informationsförsörjning: en fråga om ansvar - aktiviteter och uppdrag i fem stora svenska organisationers operativa informationsförsörjning, 1998.
No 731 **Åse Jansson:** Miljöhänsyn - en del i företags styrning, 1998.
No 733 **Thomas Padron-McCarthy:** Performance-Polymorphic Declarative Queries, 1998.
No 734 **Anders Bäckström:** Värdeskapande kreditgivning - Kreditriskhantering ur ett agentteoretiskt perspektiv, 1998.
FiF-a 21 **Ulf Seigerroth:** Integration av förändringsmetoder - en modell för välgrundad metodintegration, 1999.
FiF-a 22 **Fredrik Öberg:** Object-Oriented Frameworks - A New Strategy for Case Tool Development, 1998.
No 737 **Jonas Mellin:** Predictable Event Monitoring, 1998.
No 738 **Joakim Eriksson:** Specifying and Managing Rules in an Active Real-Time Database System, 1998.
FiF-a 25 **Bengt E W Andersson:** Samverkande informationssystem mellan aktörer i offentliga åtaganden - En teori om aktörsarenor i samverkan om utbyte av information, 1998.
No 742 **Pawel Pietrzak:** Static Incorrectness Diagnosis of CLP (FD), 1999.
No 748 **Tobias Ritzau:** Real-Time Reference Counting in RT-Java, 1999.
No 751 **Anders Ferntoft:** Elektronisk affärskommunikation - kontaktkostnader och kontaktprocesser mellan kunder och leverantörer på producentmarknader,1999.
No 752 **Jo Skåmedal:** Arbete på distans och arbetsformens påverkan på resor och resmönster, 1999.
No 753 **Johan Alvehus:** Mötets metaforer. En studie av berättelser om möten, 1999.
No 754 **Magnus Lindahl:** Bankens villkor i låneavtal vid kreditgivning till högt belånade företagsförvärv: En studie ur ett agentteoretiskt perspektiv, 2000.
No 766 **Martin V. Howard:** Designing dynamic visualizations of temporal data, 1999.
No 769 **Jesper Andersson:** Towards Reactive Software Architectures, 1999.
No 775 **Anders Henriksson:** Unique kernel diagnosis, 1999.
FiF-a 30 **Pär J. Ågerfalk:** Pragmatization of Information Systems - A Theoretical and Methodological Outline, 1999.
No 787 **Charlotte Björkegren:** Learning for the next project - Bearers and barriers in knowledge transfer within an organisation, 1999.
No 788 **Håkan Nilsson:** Informationsteknik som drivkraft i granskningsprocessen - En studie av fyra revisionsbyråer, 2000.
No 790 **Erik Berglund:** Use-Oriented Documentation in Software Development, 1999.
No 791 **Klas Gäre:** Verksamhetsförändringar i samband med IS-införande, 1999.
No 800 **Anders Subotic:** Software Quality Inspection, 1999.
No 807 **Svein Bergum**: Managerial communication in telework, 2000.

No 809 **Flavius Gruian:** Energy-Aware Design of Digital Systems, 2000.

FiF-a 32 **Karin Hedström:** Kunskapsanvändning och kunskapsutveckling hos verksamhetskonsulter - Erfarenheter från ett FOU-samarbete, 2000.

No 808 **Linda Askenäs:** Affärssystemet - En studie om teknikens aktiva och passiva roll i en organisation, 2000.

No 820 **Jean Paul Meynard:** Control of industrial robots through high-level task programming, 2000.

No 823 **Lars Hult:** Publika Gränsytor - ett designexempel, 2000.

No 832 **Paul Pop:** Scheduling and Communication Synthesis for Distributed Real-Time Systems, 2000.

FiF-a 34 **Göran Hultgren:** Nätverksinriktad Förändringsanalys - perspektiv och metoder som stöd för förståelse och utveckling av affärsrelationer och informationssystem, 2000.

No 842 **Magnus Kald:** The role of management control systems in strategic business units, 2000.

No 844 **Mikael Cäker:** Vad kostar kunden? Modeller för intern redovisning, 2000.

FiF-a 37 **Ewa Braf**: Organisationers kunskapsverksamheter - en kritisk studie av ”knowledge management”, 2000.

FiF-a 40 **Henrik Lindberg:** Webbaserade affärsprocesser - Möjligheter och begränsningar, 2000.

FiF-a 41 **Benneth Christiansson:** Att komponentbasera informationssystem - Vad säger teori och praktik?, 2000.

No. 854 **Ola Pettersson:** Deliberation in a Mobile Robot, 2000.

No 863 **Dan Lawesson**: Towards Behavioral Model Fault Isolation for Object Oriented Control Systems, 2000.

No 881 **Johan Moe:** Execution Tracing of Large Distributed Systems, 2001.

No 882 **Yuxiao Zhao:** XML-based Frameworks for Internet Commerce and an Implementation of B2B e-procurement, 2001.

No 890 **Annika Flycht-Eriksson:** Domain Knowledge Management inInformation-providing Dialogue systems, 2001.

FiF-a 47 **Per-Arne Segerkvis**t: Webbaserade imaginära organisationers samverkansformer: Informationssystemarkitektur och aktörssamverkan som förutsättningar för affärsprocesser, 2001.

No 894 **Stefan Svarén:** Styrning av investeringar i divisionaliserade företag - Ett koncernperspektiv, 2001.

No 906 **Lin Han**: Secure and Scalable E-Service Software Delivery, 2001.

No 917 **Emma Hansson:** Optionsprogram för anställda - en studie av svenska börsföretag, 2001.

No 916 **Susanne Odar:** IT som stöd för strategiska beslut, en studie av datorimplementerade modeller av verksamhet som stöd för beslut om anskaffning av JAS 1982, 2002.

FiF-a-49 **Stefan Holgersson:** IT-system och filtrering av verksamhetskunskap - kvalitetsproblem vid analyser och beslutsfattande som bygger på uppgifter hämtade från polisens IT-system, 2001.

FiF-a-51 **Per Oscarsson:**Informationssäkerhet i verksamheter - begrepp och modeller som stöd för förståelse av informationssäkerhet och dess hantering, 2001.

No 919 **Luis Alejandro Cortes:** A Petri Net Based Modeling and Verification Technique for Real-Time Embedded Systems, 2001.

No 915 **Niklas Sandell:** Redovisning i skuggan av en bankkris - Värdering av fastigheter. 2001.

No 931 **Fredrik Elg:** Ett dynamiskt perspektiv på individuella skillnader av heuristisk kompetens, intelligens, mentala modeller, mål och konfidens i kontroll av mikrovärlden Moro, 2002.

No 933 **Peter Aronsson:** Automatic Parallelization of Simulation Code from Equation Based Simulation Languages, 2002.

No 938 **Bourhane Kadmiry**: Fuzzy Control of Unmanned Helicopter, 2002.

No 942 **Patrik Haslum**: Prediction as a Knowledge Representation Problem: A Case Study in Model Design, 2002.

No 956 **Robert Sevenius:** On the instruments of governance - A law & economics study of capital instruments in limited liability companies, 2002.

FiF-a 58 **Johan Petersson:** Lokala elektroniska marknadsplatser - informationssystem för platsbundna affärer, 2002.

No 964 **Peter Bunus:** Debugging and Structural Analysis of Declarative Equation-Based Languages, 2002.

No 973 **Gert Jervan:** High-Level Test Generation and Built-In Self-Test Techniques for Digital Systems, 2002.

No 958 **Fredrika Berglund:** Management Control and Strategy - a Case Study of Pharmaceutical Drug Development, 2002.

FiF-a 61 **Fredrik Karlsson:** Meta-Method for Method Configuration - A Rational Unified Process Case, 2002.

No 985 **Sorin Manolache:** Schedulability Analysis of Real-Time Systems with Stochastic Task Execution Times, 2002.

No 982 **Diana Szentiványi:** Performance and Availability Trade-offs in Fault-Tolerant Middleware, 2002.

No 989 **Iakov Nakhimovski:** Modeling and Simulation of Contacting Flexible Bodies in Multibody Systems, 2002.

No 990 **Levon Saldamli:** PDEModelica - Towards a High-Level Language for Modeling with Partial Differential Equations, 2002.

No 991 **Almut Herzog:** Secure Execution Environment for Java Electronic Services, 2002.

No 999 **Jon Edvardsson:** Contributions to Program- and Specification-based Test Data Generation, 2002

No 1000 **Anders Arpteg:** Adaptive Semi-structured Information Extraction, 2002.

No 1001 **Andrzej Bednarski:** A Dynamic Programming Approach to Optimal Retargetable Code Generation for Irregular Architectures, 2002.

No 988 **Mattias Arvola:** Good to use! : Use quality of multi-user applications in the home, 2003.

FiF-a 62 **Lennart Ljung:** Utveckling av en projektivitetsmodell - om organisationers förmåga att tillämpa projektarbetsformen, 2003.

No 1003 **Pernilla Qvarfordt:** User experience of spoken feedback in multimodal interaction, 2003.

No 1005 **Alexander Siemers:** Visualization of Dynamic Multibody Simulation With Special Reference to Contacts, 2003.

No 1008 **Jens Gustavsson:** Towards Unanticipated Runtime Software Evolution, 2003.

No 1010 **Calin Curescu:** Adaptive QoS-aware Resource Allocation for Wireless Networks, 2003.

No 1015 **Anna Andersson:** Management Information Systems in Process-oriented Healthcare Organisations, 2003.

No 1018 **Björn Johansson:** Feedforward Control in Dynamic Situations, 2003.

No 1022 **Traian Pop:** Scheduling and Optimisation of Heterogeneous Time/Event-Triggered Distributed Embedded Systems, 2003.

FiF-a 65 **Britt-Marie Johansson:** Kundkommunikation på distans - en studie om kommunikationsmediets betydelse i affärstransaktioner, 2003.

No 1024 **Aleksandra Tešanovic:** Towards Aspectual Component-Based Real-Time System Development, 2003.
No 1034 **Arja Vainio-Larsson**: Designing for Use in a Future Context - Five Case Studies in Retrospect, 2003.
No 1033 **Peter Nilsson:** Svenska bankers redovisningsval vid reservering för befarade kreditförluster - En studie vid införandet av nya redovisningsregler, 2003.
FiF-a 69 **Fredrik Ericsson:** Information Technology for Learning and Acquiring of Work Knowledge, 2003.
No 1049 **Marcus Comstedt:** Towards Fine-Grained Binary Composition through Link Time Weaving, 2003.
No 1052 **Åsa Hedenskog:** Increasing the Automation of Radio Network Control, 2003.
No 1054 **Claudiu Duma:** Security and Efficiency Tradeoffs in Multicast Group Key Management, 2003.
FiF-a 71 **Emma Eliason:** Effektanalys av IT-systems handlingsutrymme, 2003.
No 1055 **Carl Cederberg:** Experiments in Indirect Fault Injection with Open Source and Industrial Software, 2003.
No 1058 **Daniel Karlsson:** Towards Formal Verification in a Component-based Reuse Methodology, 2003.
FiF-a 73 **Anders Hjalmarsson:** Att etablera och vidmakthålla förbättringsverksamhet - behovet av koordination och interaktion vid förändring av systemutvecklingsverksamheter, 2004.
No 1079 **Pontus Johansson:** Design and Development of Recommender Dialogue Systems, 2004.
No 1084 **Charlotte Stoltz:** Calling for Call Centres - A Study of Call Centre Locations in a Swedish Rural Region, 2004.
FiF-a 74 **Björn Johansson:** Deciding on Using Application Service Provision in SMEs, 2004.
No 1094 **Genevieve Gorrell:** Language Modelling and Error Handling in Spoken Dialogue Systems, 2004.
No 1095 **Ulf Johansson:** Rule Extraction - the Key to Accurate and Comprehensible Data Mining Models, 2004.
No 1099 **Sonia Sangari:** Computational Models of Some Communicative Head Movements, 2004.
No 1110 **Hans Nässla:** Intra-Family Information Flow and Prospects for Communication Systems, 2004.
No 1116 **Henrik Sällberg:** On the value of customer loyalty programs - A study of point programs and switching costs, 2004.
FiF-a 77 **Ulf Larsson:** Designarbete i dialog - karaktärisering av interaktionen mellan användare och utvecklare i en systemutvecklingsprocess, 2004.
No 1126 **Andreas Borg:** Contribution to Management and Validation of Non-Functional Requirements, 2004.
No 1127 **Per-Ola Kristensson:** Large Vocabulary Shorthand Writing on Stylus Keyboard, 2004.
No 1132 **Pär-Anders Albinsson:** Interacting with Command and Control Systems: Tools for Operators and Designers, 2004.
No 1130 **Ioan Chisalita:** Safety-Oriented Communication in Mobile Networks for Vehicles, 2004.
No 1138 **Thomas Gustafsson:** Maintaining Data Consistency im Embedded Databases for Vehicular Systems, 2004.
No 1149 **Vaida Jakoniené:** A Study in Integrating Multiple Biological Data Sources, 2005.
No 1156 **Abdil Rashid Mohamed**: High-Level Techniques for Built-In Self-Test Resources Optimization, 2005.
No 1162 **Adrian Pop:** Contributions to Meta-Modeling Tools and Methods, 2005.
No 1165 **Fidel Vascós Palacios:** On the information exchange between physicians and social insurance officers in the sick leave process: an Activity Theoretical perspective, 2005.
FiF-a 84 **Jenny Lagsten:** Verksamhetsutvecklande utvärdering i informationssystemprojekt, 2005.
No 1166 **Emma Larsdotter Nilsson:** Modeling, Simulation, and Visualization of Metabolic Pathways Using Modelica, 2005.
No 1167 **Christina Keller:** Virtual Learning Environments in higher education. A study of students' acceptance of educational technology, 2005.
No 1168 **Cécile Åberg:** Integration of organizational workflows and the Semantic Web, 2005.
FiF-a 85 **Anders Forsman:** Standardisering som grund för informationssamverkan och IT-tjänster - En fallstudie baserad på trafikinformationstjänsten RDS-TMC, 2005.
No 1171 **Yu-Hsing Huang:** A systemic traffic accident model, 2005.
FiF-a 86 **Jan Olausson:** Att modellera uppdrag - grunder för förståelse av processinriktade informationssystem i transaktionsintensiva verksamheter, 2005.
No 1172 **Petter Ahlström**: Affärsstrategier för seniorbostadsmarknaden, 2005.
No 1183 **Mathias Cöster**: Beyond IT and Productivity - How Digitization Transformed the Graphic Industry, 2005.
No 1184 **Åsa Horzella**: Beyond IT and Productivity - Effects of Digitized Information Flows in Grocery Distribution, 2005.
No 1185 **Maria Kollberg**: Beyond IT and Productivity - Effects of Digitized Information Flows in the Logging Industry, 2005.
No 1190 **David Dinka**: Role and Identity - Experience of technology in professional settings, 2005.
No 1191 **Andreas Hansson**: Increasing the Storage Capacity of Recursive Auto-associative Memory by Segmenting Data, 2005.
No 1192 **Nicklas Bergfeldt:** Towards Detached Communication for Robot Cooperation, 2005.
No 1194 **Dennis Maciuszek:** Towards Dependable Virtual Companions for Later Life, 2005.
No 1204 **Beatrice Alenljung:** Decision-making in the Requirements Engineering Process: A Human-centered Approach, 2005
No 1206 **Anders Larsson:** System-on-Chip Test Scheduling and Test Infrastructure Design, 2005.
No 1207 **John Wilander:** Policy and Implementation Assurance for Software Security, 2005.
No 1209 **Andreas Käll:** Översättningar av en managementmodell - En studie av införandet av Balanced Scorecard i ett landsting, 2005.
No 1225 **He Tan:** Aligning and Merging Biomedical Ontologies, 2006.
No 1228 **Artur Wilk:** Descriptive Types for XML Query Language Xcerpt, 2006.
No 1229 **Per Olof Pettersson:** Sampling-based Path Planning for an Autonomous Helicopter, 2006.
No 1231 **Kalle Burbeck:** Adaptive Real-time Anomaly Detection for Safeguarding Critical Networks, 2006.
No 1233 **Daniela Mihailescu:** Implementation Methodology in Action: A Study of an Enterprise Systems Implementation Methodology, 2006.
No 1244 **Jörgen Skågeby:** Public and Non-public gifting on the Internet, 2006.
No 1248 **Karolina Eliasson:** The Use of Case-Based Reasoning in a Human-Robot Dialog System, 2006.
No 1263 **Misook Park-Westman:** Managing Competence Development Programs in a Cross-Cultural Organisation-What are the Barriers and Enablers, 2006.
FiF-a 90 **Amra Halilovic:** Ett praktikperspektiv på hantering av mjukvarukomponenter, 2006.
No 1272 **Raquel Flodström**: A Framework for the Strategic Management of Information Technology, 2006.

No 1277 **Viacheslav Izosimov:** Scheduling and Optimization of Fault-Tolerant Embedded Systems, 2006.
No 1283 **Håkan Hasewinkel:** A Blueprint for Using Commercial Games off the Shelf in Defence Training, Education and Research Simulations, 2006.
FiF-a 91 **Hanna Broberg:** Verksamhetsanpassade IT-stöd - Designteori och metod, 2006.
No 1286 **Robert Kaminski:** Towards an XML Document Restructuring Framework, 2006
No 1293 **Jiri Trnka:** Prerequisites for data sharing in emergency management, 2007.
No 1302 **Björn Hägglund:** A Framework for Designing Constraint Stores, 2007.
No 1303 **Daniel Andreasson**: Slack-Time Aware Dynamic Routing Schemes for On-Chip Networks, 2007.
No 1305 **Magnus Ingmarsson:** Modelling User Tasks and Intentions for Service Discovery in Ubiquitous Computing, 2007.
No 1306 **Gustaf Svedjemo**: Ontology as Conceptual Schema when Modelling Historical Maps for Database Storage, 2007.
No 1307 **Gianpaolo Conte**: Navigation Functionalities for an Autonomous UAV Helicopter, 2007.
No 1309 **Ola Leifler**: User-Centric Critiquing in Command and Control: The DKExpert and ComPlan Approaches, 2007.
No 1312 **Henrik Svensson**: Embodied simulation as off-line representation, 2007.
No 1313 **Zhiyuan He:** System-on-Chip Test Scheduling with Defect-Probability and Temperature Considerations, 2007.
No 1317 **Jonas Elmqvist:** Components, Safety Interfaces and Compositional Analysis, 2007.
No 1320 **Håkan Sundblad:** Question Classification in Question Answering Systems, 2007.
No 1323 **Magnus Lundqvist**: Information Demand and Use: Improving Information Flow within Small-scale Business Contexts, 2007.
No 1329 **Martin Magnusson:** Deductive Planning and Composite Actions in Temporal Action Logic, 2007.
No 1331 **Mikael Asplund:** Restoring Consistency after Network Partitions, 2007.
No 1332 **Martin Fransson:** Towards Individualized Drug Dosage - General Methods and Case Studies, 2007.
No 1333 **Karin Camara:** A Visual Query Language Served by a Multi-sensor Environment, 2007.
No 1337 **David Broman:** Safety, Security, and Semantic Aspects of Equation-Based Object-Oriented Languages and Environments, 2007.
No 1339 **Mikhail Chalabine:** Invasive Interactive Parallelization, 2007.
No 1351 **Susanna Nilsson:** A Holistic Approach to Usability Evaluations of Mixed Reality Systems, 2008.
No 1353 **Shanai Ardi:** A Model and Implementation of a Security Plug-in for the Software Life Cycle, 2008.
No 1356 **Erik Kuiper:** Mobility and Routing in a Delay-tolerant Network of Unmanned Aerial Vehicles, 2008.
No 1359 **Jana Rambusch**: Situated Play, 2008.
No 1361 **Martin Karresand:** Completing the Picture - Fragments and Back Again, 2008.
No 1363 **Per Nyblom:** Dynamic Abstraction for Interleaved Task Planning and Execution, 2008.
No 1371 **Fredrik Lantz:**Terrain Object Recognition and Context Fusion for Decision Support, 2008.
No 1373 **Martin Östlund:** Assistance Plus: 3D-mediated Advice-giving on Pharmaceutical Products, 2008.
No 1381 **Håkan Lundvall:** Automatic Parallelization using Pipelining for Equation-Based Simulation Languages, 2008.
No 1386 **Mirko Thorstensson:** Using Observers for Model Based Data Collection in Distributed Tactical Operations, 2008.
No 1387 **Bahlol Rahimi:** Implementation of Health Information Systems, 2008.
No 1392 **Maria Holmqvist:** Word Alignment by Re-using Parallel Phrases, 2008.
No 1393 **Mattias Eriksson**: Integrated Software Pipelining, 2009.