Code Generation for Transport Triggered Architectures

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus Prof.ir. K.F. Wakker,
in het openbaar te verdedigen ten overstaan van een commissie,
door het College van Dekanen aangewezen,
op maandag 5 februari 1996 te 16.00 uur
door

Jan HOOGERBRUGGE

informatica ingenieur
geboren te Capelle aan de IJssel

Dit proefschrift is goedgekeurd door de promotor:
prof.dr.ir. A.J. van de Goor

Toegevoegd promotor:
dr. H. Corporaal

De leden van de promotiecommissie zijn:

Rector Magnificus	Technische Universiteit Delft
prof.dr.ir. A.J. van de Goor	Technische Universiteit Delft
dr. H. Corporaal	Technische Universiteit Delft
dr.ir. H.E. Bal	Vrije Universiteit Amsterdam
prof.dr.ir. J. van Katwijk	Technische Universiteit Delft
prof.dr.ir. M.J. Plasmeijer	Katholieke Universiteit Nijmegen
prof.dr.ir. H.J. Sips	Universiteit van Amsterdam
prof.dr. H.A.G. Wijshof	Rijks Universiteit Leiden

Cover design by Jan Hoogerbrugge and Roger van der Laan

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Hoogerbrugge, Jan

Code Generation for Transport Triggered Architectures /
Jan Hoogerbrugge. – [S.l. : s.n.]. – Ill.
Thesis Technische Universiteit Delft. – With ref. – With
summary in Dutch
ISBN 90-9009002-9
Subject headings: code generation / computer architecture

This dissertation is dedicated to
the loving memory of my father
Marinus Hoogerbrugge

Contents

Acknowledgements

First of all I wish to thank prof. Ad van de Goor and Henk Corporaal for their guidance during the four years in which I performed the research described in this dissertation. I appreciate the freedom they gave me to do the research I am interested in.

Furthermore, I would like to thank my fellow AIOs and ex-AIOs within the MOVE project Robert Portier, Andy Verberne, Johan Janssen, Roger Jansen, Jeroen Hordijk, Paul van der Arend, Paul Stravers, and Reinoud Lamberts. Roommates Wilco van Hoogstraeten and Wiebe Cnossen for all the fun we had during our work. Prof. Stamatis Vassiliadis for his rich experience and view on computer architecture. Jaap Hoekstra for explaining me everything about department politics. System administrator Jean-Paul van der Jagt and his successor Tobias Nijweide for providing an excellent working computer environment. Students Theo Baan, Erwin Abrahamse, and Bas van Houte for their contributions to my research. Rogier Wolff for providing the MCCD application that I used in chapter 7.

Finally, I would like to thank my friends and family for supporting me and letting me think about other things than code generation for transport triggered architectures.

Jan Hoogerbrugge

Rotterdam, February 1996

1 Introduction

This dissertation describes the results of research performed within the MOVE Project at Delft University of Technology. The MOVE Project aims at designing *application specific processors* (ASPs) based on a new novel computer architecture paradigm called *transport triggered architectures* (TTAs). ASPs are processors designed especially for one particular application in order to improve their cost/performance. They are often embedded in all kinds of electronic systems. To reduce design costs, an ASP is usually designed according to a *template* architecture. Such a template should be flexible, scalable, and cost efficient. TTAs fulfill these requirements. TTAs are similar to very long instruction word (VLIW) processors in that they provide statically scheduled instruction level parallelism (ILP) in order to improve performance in a scalable and cost efficient way. The difference is that they are not programmed by instructions specifying multiple operations but by instructions specifying data transports. This improves flexibility, scalability, and cost efficiency, but also complicates the already complex compilation process. The main theme of this dissertation is to demonstrate that efficient compilation for TTAs is very well possible. This is done by developing a compiler for TTAs.

This chapter describes ASPs briefly in section 1.1, and TTAs in section 1.2. Section 1.3 gives our motivation for this research, section 1.4 enumerates the major contributions, and section 1.5 gives an overview of the remaining chapters of this thesis.

1.1 Application Specific Processors

One of the first decisions a designer of a processor based electronic system has to make is choosing between a standard off-the-shelf processor and an ASP, specially designed for the application in question. A standard processor is intended for a large class of applications and usually contains hardware that is not effectively used by the given application; and in addition, it misses hardware that could be very useful. For example, a standard processor might contain an expensive multiplier which is a waste of chip area and power consumption when the application seldom performs multiplications (e.g., less than 0.1% of all executed operations). As an example of hardware functionality that might be missing, consider an application that manipulates bit-oriented data. For such an application instructions such as 'find first bit set' and 'count number of bits set' are very welcome since they are easy to implement in hardware and may result in a significant speedup.

If the decision turns out in favor of an ASP the next question becomes: how should it be designed and implemented? A full custom design offers optimal flexibility, performance, chip area, and power consumption at the price of a long and expensive design process. This is only acceptable for high volume production. A method to reduce design costs is to build ASPs according to a *template*. For example, all ASPs have a four stage pipeline, have 32-bit wide words, and are big endian. An ASP is designed by instantiating the *architectural parameters* of the template. Examples of architectural parameters are the number of general purpose registers, cache sizes, the operation set, the amount of instruction level parallelism, and operation latencies. By designing ASPs according to a template, we reduce design costs by giving up some design freedom. This is similar to programming in a high level language instead of in assembly language or designing hardware in a hardware description language instead of designing at gate or layout level. Obviously, the usefulness of a templated ASP design system depends largely on the flexibility of the template and the efficiency of the tools for generating the processor and the code for the processor.

In our opinion a system for templated ASP design should consist of at least the following three components: (1) a processor generator, (2) a code generator, and (3) a design space explorer (see figure 1.1).

The processor generator

The processor generator is responsible for generating VLSI layout for the ASP according to the architectural parameter set. There are several ways to do this. One can use a silicon compiler that generates a layout based on a parameterized processor description. The parameters of the processor description are directly related to the architectural parameters. Another method is to use param-

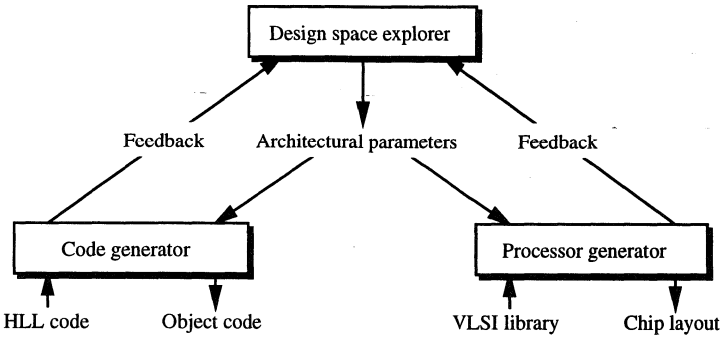


Figure 1.1: The three components of a templated ASP design system

eterized cell generators that generate processor components such as functional units and register files based on the architectural parameter values. After that, the generated cells are placed and routing is performed to connect them.

The code generator

The code generator is responsible for compiling the application written in a high level language into object code for an ASP as described by the architectural parameter set. The back-end of the compiler should be highly parametrized and configurable in order to be able to generate code for all ASPs that can be described by the template and the architectural parameter set.

The design space explorer

The design space explorer is responsible for finding the right architectural parameter set values for a given application. Each set of parameter values corresponds to a particular ASP. The quality of an ASP depends on its implementation costs, its performance for the given application, and design constraints. The design constraints may specify minimum performance (e.g., minimal 10k samples per second), maximum costs (e.g., maximal 80mm^2 chip area), and which cost/performance trade-offs can be made (e.g., 20% more costs should give at least 10% more performance).

It should be clear that it is impossible to calculate, or accurately predict, performance and costs of an ASP as function of the architectural parameters for realistic applications. Therefore the design space explorer invokes the processor generator to generate an ASP for the given architectural parameters and it invokes the code generator to compile the application for an ASP with the given architectural parameters. The processor and code generator report statistics of their produced results to the explorer. With this information the explorer tries

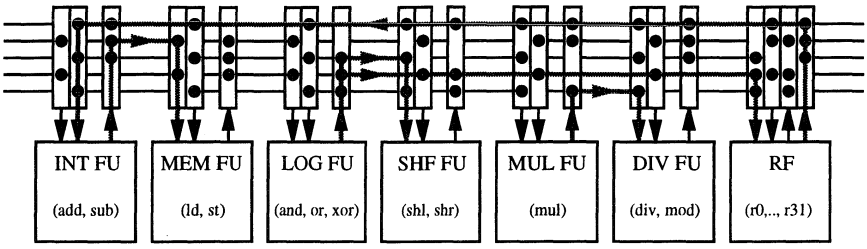


Figure 1.2: General structure of a TTA

to find a better set of architectural parameters. This is repeated until a set of architectural parameters is found that seems to be the best for the given application and design constraints.

1.2 Transport Triggered Architectures

Like a traditional *operation triggered architecture* (OTA), a TTA is a collection of functional units (FUs) and register files (RFs) connected to each other via a set of transport buses. The difference between OTAs and TTAs is that

> *OTAs are programmed by instructions containing operations that trigger operations on FUs explicitly, and trigger data transports between FUs and RFs implicitly (as side-effect),*

while

> *TTAs are programmed by instructions containing data transports that trigger data transports between FUs and RFs explicitly, and trigger operations on FUs implicitly (as side-effect).*

Figure 1.2 shows the general structure of a TTA consisting of six FUs, one RF and an interconnection network consisting of five *move buses*. This TTA is capable of performing five data transports, or *moves*, between FUs and RFs per cycle. It is programmed by instructions containing five *move slots* each containing one move that controls one move bus. The gray lines in figure 1.2 illustrate the operation of a TTA during a particular cycle; data is moved from the RF to the integer FU (INT), from the integer FU to the memory FU (MEM), from the logic FU (LOG) to the shift FU (SHF) and the RF, and from the multiply FU (MUL) to the divide FU (DIV).

Although programming data transports may seem clumsy at first sight, it presents new avenues for compiler optimizations that reduce the number of moves and RF accesses and therefore the required number of move buses and

RF ports. Bypassing, commonly used in pipelined processors to reduce the latency between flow dependent operations, results in many useless data transports and RF accesses in OTAs. A compiler for TTAs is able to suppress these useless transports and RF accesses. Since the freed move buses and RF ports can be used for other operations, the required number of buses and RF ports for TTAs is lower than the required number for OTAs. This reduces die area and improves the achievable cycle time.

Compiling for TTAs, especially code scheduling, the main theme of this dissertation, is quite different from compiling for OTAs. Scheduling the individual moves of an operation instead of a single operation results in scheduling constraints not present in OTAs; e.g., the scheduler should take the first-in-first-out property of FU pipelines into account. Furthermore, the scheduler should detect when bypassing is required and when write backs can be suppressed. A challenging aspect of scheduling for TTAs is scheduling for irregularly, partially connected interconnection networks. The scheduler is responsible for routing the data over the available paths through the interconnection network and making maximal use of the available bandwidth.

1.3 Motivation

The main emphasis of this thesis is on researching compilation techniques for TTAs and design exploration of ASPs. The following motivates this research.

Motivation for studying compilation techniques for TTAs

Instruction level parallelism, in the form of multiple instruction issue and/or superpipelining, is one of the major techniques to improve processor performance. Both multiple instruction issue and superpipelining increase the required transport resources, consisting of buses and RF ports, between the FUs and RFs. This increases die area and may affect the cycle time. By programming data transports instead of operations we have more control over the transport resources which improves their efficiency and lowers their requirements.

Programming processors at the register transfer level instead of at the operation level has a significant impact on the complexity of the compiler and it is not obvious that a compiler can handle this complexity. In many ways TTAs have a lot in common with horizontal microcode system such as the FPS-164 [190]. The main reason that these systems became extinct is that they were considered too hard to program efficiently [74, 103]. Therefore, the success of TTAs will largely be dependent on how well a compiler is able to deal with the extra complexity of programming data transports instead of operations. This motivates research of compilation techniques for TTAs.

Motivation for studying design space exploration of ASPs

The motivation for ASPs is obvious: most processors are not used as central processors for personal computers, workstations, supercomputers, or mainframes, but are embedded in all kinds of electronic equipment such as printers, digital television sets, video games, cars, and industrial process controllers. These processors have in common that they execute a single application. A better cost/performance is possible if such a processor is specially designed for its application, i.e., an application specific processor.

Designing ASPs based on a template makes it possible to reduce design costs and time of ASPs significantly which is a must for low-volume production and a short time-to-market. The design problem is reduced to the problem of finding the right architectural parameter values for a given application, i.e., exploring the design space. Manual design space exploration becomes infeasible for a realistic template with a large number of architectural parameters. It is tedious, error prone, and time consuming work and it is likely that some interesting areas of the design space are not considered. This motivates research of automatic design space exploration.

1.4 Contributions

The major contributions of this dissertation are: (1) the development of a prototype compiler for TTAs, (2) the evaluation of TTAs, and (3) the development of a design space exploration method. These contributions are detailed below.

Development of a compiler for TTAs (chapters 3-5, [57, 106, 108, 109])

Prior to this research, two prototype basic blocks schedulers for TTAs have been developed which were very restricted in their performance and flexibility, e.g., they required full connectivity between FUs and RFs [60, 195]. At that time there was only a vague understanding of the problems that arise when the scheduling scope of a scheduler for TTAs would be enlarged to multiple basic blocks (extended basic block scheduling) or loops (software pipelining) and of the problems that arise with compilation for realistic TTAs (e.g., partial connectivity, limited number of RF ports). The proposed solutions to these problems were even more vague.

This research gave us a much better understanding about the problems and solutions for compilation for TTAs. This has been achieved by developing and implementing a highly parametrized and configurable prototype C/C++/Fortran compiler for TTAs. This compiler is based on the GNU compiler and is extended with advanced techniques such as extended basic block instruction scheduling, software pipelining, memory reference disambiguation, global register allocation, loop unrolling, function inlining, an annotation

system, and a number of TTA specific optimizations.

Most of the used scheduling techniques are derived from existing state-of-the-art techniques and are adapted for TTAs. Most adaptations are far from trivial; TTAs have other resource constraints than OTAs and present unique optimization opportunities.

In addition, this dissertation reports several (minor) contributions that are useful for code generation for OTAs as well. Examples are delay lines (section 5.2.3) to improve the performance of software pipelining and partial loop unrolling (appendix A) to increase parallelism in a code size efficient way.

Evaluation of TTAs (chapter 6, [104, 105])

The goal of evaluating TTAs is two-fold. First, we want to quantify the advantages and disadvantages of TTAs, e.g., how many RF ports are required to sustain a certain number of operations per cycle. The second goal is to evaluate different hardware design options. For example, in section 6.2.8 we will evaluate two different FU pipelining schemes, hybrid pipelining and virtual time latching pipelining. With the outcome of this experiment we can decide which pipelining scheme is preferable.

Design space exploration (chapter 7, [107])

Most prior work related to designing ASPs requires manual exploration of the design space to find proper architectural parameter values for a given application and set of design constraints. This becomes unacceptable when the number of architectural parameters increases, i.e., the dimension of the design space increases. We have developed and implemented a method for exploring the design space for a given application. This method speeds up and improves the quality of the design process.

1.5 Thesis Overview

This thesis is organized as follows. Chapter 2 describes the basics of instruction level parallelism and introduces TTAs. The next three chapters are devoted to code generation for TTAs. Chapter 3 starts with basic block scheduling for TTAs. This is the simplest method to generate instruction level parallel code. In chapter 4 we go one step further: extended basic block scheduling. An extended basic block scheduler moves operations across basic block boundaries in order to find more parallelism. These movements are between basic blocks of the same loop iteration if they belong to a loop body. This restriction is lifted in chapter 5. Code motion along the backward edge of a loop means that the execution of different loop iterations is overlapped. This is known as software

pipelining. In chapter 6 we will describe experiments to evaluate various aspects of TTAs and code generation for TTAs. Chapter 7 describes the design space exploration method and a case study to see how the method works out in practice. Chapter 8 concludes this thesis with a summary and suggestions for future research.

Transport Triggered Architectures

2

The execution time of a program can be expressed as the product of three factors: (1) the number of instructions required to execute the program, (2) the average number of cycles per instruction (CPI), and (3) the clock cycle time [103]. The main objectives of the RISC evolution during the Eighties were to reduce CPI from several cycles to one cycle and to reduce the clock cycle time. This was achieved by hardwired control, large register files, a single chip processor, and a uniform, reduced, and streamlined instruction set that is relatively easy to pipeline.

The challenge of the Nineties is to reduce the CPI from one cycle to a fraction of a cycle and a further reduction of the cycle time. This is achieved by the exploitation of *instruction level parallelism* (ILP) [82, 171]. ILP processors are processors where multiple instructions are simultaneously in the execute stage of the instruction pipeline. Section 2.1 describes the two techniques to realize ILP: superpipelining and multiple instruction issue, and the two ways to control ILP: dynamically at run-time and statically at compile-time.

One of the major arguments for dynamic ILP exploitation is the binary compatibility between ILP processors with the same instruction set architecture (ISA) but with different amounts of ILP. Since binary compatibility is not a real issue for ASPs and static ILP exploitation is easier to implement, better scalable, and more flexible, static ILP exploitation is preferred for ASPs. In section 2.2 we discuss static ILP exploitation which mainly consists of compilation techniques.

Section 2.3 introduces transport triggered architectures (TTAs), a new player in the arena of computer architecture paradigms. TTAs go one step further than traditional static controlled ILP architectures in the sense that they do even less at run-time and more at compile-time. Section 2.3 only discusses the aspects of

interest to the compiler. This excludes issues such as exception handling and instruction pipelining schemes; for these issues the reader is referred to [56, 59].

Section 2.4 describes the advantages and disadvantages of the TTA concept. The advantages fall into two categories, implementation advantages and new compiler optimizations.

2.1 Instruction Level Parallel Processors

Parallelism has always been an important topic in computer science and becomes more important when hardware becomes denser, cheaper, and easier to implement and physical problems begin to limit higher clock frequencies. Parallelism can be exploited between computations of different levels of granularity. *Course grain parallelism* refers to parallelism between computations that correspond to complete programs, procedures, loops, or loop iterations. This type of parallelism is usually exploited on MIMD computers consisting of multiple interconnected processing nodes working in parallel. ILP is very *fine grained parallelism*; parallelism between the instructions executing on a single processor.

Course grain parallelism and ILP have very different characteristics. Course grain parallelism is currently only applicable for a small set of applications that can be parallelized by a compiler or a programmer. However, among these applications are many interesting scientific applications that contain an enormous amount of parallelism that is relatively easy to exploit on MIMD computers. The amount of ILP that can be exploited is limited (2–8 instructions per cycle [122, 133, 198]). However, unlike course grain parallelism, nearly all applications contain some amount of it and it can be found without the assistance of a programmer. Course grain parallelism and ILP are not competitors, they can be applied independently. For example, the Cray T3D [125] and Convex SPP [38] are MIMD computers build out of ILP processor nodes.

2.1.1 Superpipelining and Multiple Instruction Issue

There are two techniques that can be applied independently to realize ILP: superpipelining and multiple instruction issue [122].

Superpipelining

A classic RISC pipeline contains four stages: instruction fetch, decode, execute, and write back. As shown in figure 2.1a, four instructions are executing simultaneously and one of them is in the execute stage. The latency of the instruction pipeline is four cycles and the throughput is one instruction per cycle.

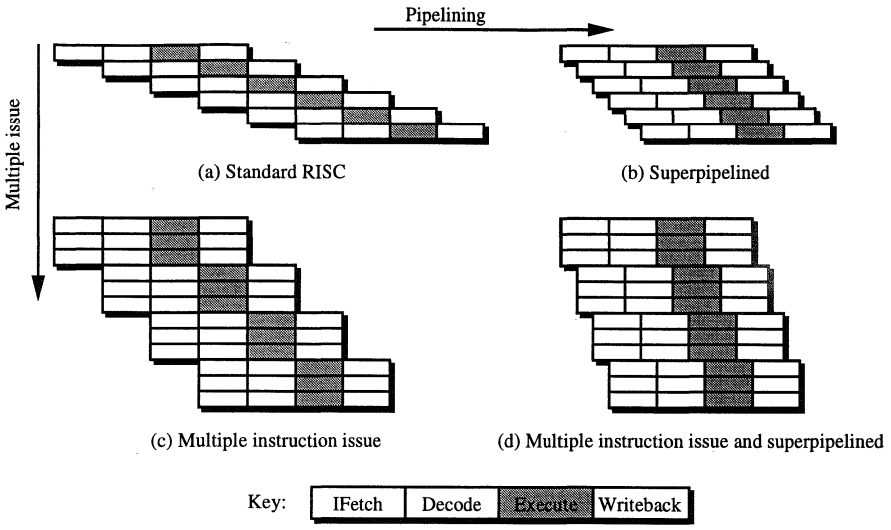


Figure 2.1: Superpipelining and multiple instruction issue

Throughput can be improved by dividing the instruction pipeline in more than four stages. This is known as *superpipelining*. Going from a well balanced (all pipeline stages have roughly the same delay) n-stage pipeline to a well balanced m-stage pipeline (where $m > n$) will potentially:

1. Increase the throughput in terms of instructions per second by m/n.
2. Increase the latency in terms of cycles by m/n.
3. Increase the number of instructions simultaneously in the execute stage (the amount of ILP) by m/n.
4. Increase performance by m/n.

There are, however, factors that cause that the actual performance gain to be less than m/n:

1. Pipelining overhead, such as setup and hold times and clock skew, will increase the pipeline latency.
2. A longer instruction pipeline will cause longer operation latencies, and therefore more interlocks and empty branch delay slots.
3. Superpipelining does not speedup cache and TLB misses (Amdahl's law).

Figure 2.1b illustrates superpipelining. The throughput is three times higher than the original pipeline shown in figure 2.1a. At each moment three instructions are simultaneously in the execute stage.

Superpipelining has been applied in the MIPS R4000 and R4400 [149], and the DEC A21064 [68] processors. The R4000 and R4400 have an 8 stage pipeline, the A21064 has a 7 stage pipeline for integer instructions and a 10 stage pipeline for floating point instructions. All these processors operate at a relative high clock frequency.

Multiple instruction issue

Multiple instruction issue (MII) is the second technique to realize ILP. MII processors increase throughput by issuing (an instruction is issued when it enters the execute stage) multiple instructions per cycle. This requires replication of functional units (FUs), decoding logic, register file (RF) ports, and transport buses. Figure 2.1c illustrates MII, each cycle up to three instructions are fetched, decoded, issued, and up to three results are written to the RF. The number of instructions that an MII processor can issue per cycle is called its *issue-rate*. An MII processor with issue rate of n has a potential speedup of n over a single instruction issue (SII) processor. The actual speedup will be less than n because:

1. The instructions that are simultaneously issued should be independent. When n independent instructions cannot be 'found', less than n instructions are issued in that cycle.

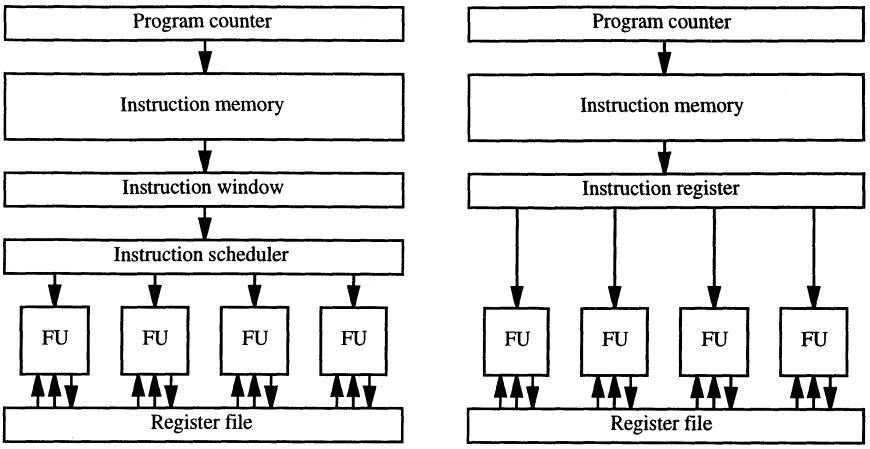
2. In practice not all resources are replicated n times. Examples are expensive load-store and floating point FUs. Therefore not all combinations of n instructions can be issued simultaneously.

3. The extra hardware to control the ILP and the extra connectivity (buses, bypass circuitry, and RF ports) may have a negative impact on the cycle time.

4. MII does not speedup cache and TLB misses.

Nearly all recently introduced high-end processors are MII processors. Their issue rates varies from 2 (e.g., Intel Pentium [11], DEC A21064 [68], and HP PA-7200 [128]) to 4 (e.g., DEC A21164 [23], IBM PowerPC 620 [139], and HP PA-8000 [116]) instructions per cycle.

Superpipelining and MII can be applied independently as shown in figure 2.1d. Nine instructions are simultaneously in the execute stage, which results in a potential speedup of nine times. Three times resulting from superpipelining and another three times from MII. Jouppi and Wall [122] concluded



(a) Superscalar (b) VLIW

Figure 2.2: Global organizations of superscalar and VLIW processors

that superpipelining and MII are roughly equivalent; and when there are no restrictions on the combinations of instructions that can be issued simultaneously and MII does not limit the cycle time, MII has an advantage of about 10% over superpipelining.

2.1.2 Static and Dynamic Scheduling

Instructions that are executed in parallel should be independent and should not use the same hardware resources simultaneously. The question arises who is going to control this, the hardware, the compiler, or both? Rau and Fisher [82, 171] classify architectures according to this question into three categories: sequential architectures, dependence architectures, and independence architectures.

Sequential architectures

Sequential architectures execute programs that do not contain any explicit information regarding ILP. ILP implementations of sequential architectures, called *superscalar* processors [120, 175], are therefore responsible for detecting dependences between instructions in the incoming sequential instruction stream and dispatching every cycle a number of independent instructions to the FUs.

Figure 2.2a shows the global, strongly simplified, organization of a superscalar. A set of FUs is connected to an RF. Like non-ILP implementations, the FUs and the RF are usually partitioned into an integer and a floating point part. When

we assume a three register operand load-store instruction set, each FU requires three RF ports. Each FU is characterized by its latency and the operation set it supports. FUs that support load-store operations will have a connection to the memory system.

A hardware instruction scheduler analyzes incoming instructions stored in a buffer called the *instruction window*. The scheduler selects a number of instructions from the instruction window that are ready to be issued and for which free FUs are available. An instruction is ready for issuing if it does not dependent on an instruction currently being executed or on another instruction in the instruction window. Branch prediction techniques are used to keep the instruction window filled when a control flow instruction is encountered.

Superscalars differ in the complexity of their instruction scheduler. Simple superscalars, such as the DEC A21064, issue instructions in the same order as they appeared in the instruction steam. Instructions are not issued when a preceding instruction in the instruction stream cannot be issued even though the instruction itself can be issued. More advanced superscalars, such as the PowerPC 620, do not have this limitation. However, the more complex instruction scheduler may limit scalability (issue rate) and cycle time. Furthermore, a complex instruction scheduler requires extra instruction pipeline stages which results in a larger branch (misprediction) latency.

Dependence architectures

Dependence architectures corresponds to the class of dataflow machines [194]. All (flow) dependence information between instructions is provided by the programmer/compiler in the program. The hardware is responsible for detecting the ready for issuing instructions and dispatching them to FUs. Tokens are used to detect when instructions are ready to be issued. Unlike sequential and independence architectures, dependence architectures have not been used in commercial products.

Independence architectures

Independence architectures are programmed by programs that contain information that specifies sets of independent instructions. The programmer/compiler is responsible for detecting dependences. The Horizon [188] and Tera [12] architectures encode a number n into each instruction that tells the hardware that the next n instructions are independent and can therefore be issued in parallel provided that sufficient free FUs are available. The hardware is thus still responsible for dispatching the operations to FUs. The organization of the Horizon is similar to the organization of a superscalar shown in figure 2.2a. The difference is that the hardware for detecting dependences between instructions in the instruction scheduler is much simpler.

	Sequential architectures	Dependence architectures	Independence architectures
Additional information required in program	None	Specification of dependences between operations	Specification of independent sets of operations
Typical kind of ILP processor	Superscalar	Dataflow	VLIW
Analysis of dependences	Hardware	Compiler	Compiler
Analysis of independences	Hardware	Hardware	Compiler
Resource allocation	Hardware	Hardware	Compiler
Role of compiler	May remedy 'tunnel view' (section 2.1.3)	Replaces some hardware	Replaces virtually all analysis and scheduling HW

Table 2.1: A comparison of the three types of architectures

VLIW architectures go one step further by shifting the dispatch task to the compiler. A VLIW instruction contains a set of operation slots; each slot can hold one operation and corresponds directly to an FU. The compiler guarantees the hardware that it can execute the VLIW instructions sequentially and can dispatch the operations in the operation slots directly to their corresponding FUs. This makes dispatching trivial; there is no need for any hardware to find out whether a free and suitable FU is available for a particular operation. Figure 2.2b shows the global organization of a VLIW. Unlike superscalars where an operation is more or less the same as an instruction, in VLIW terminology an instruction corresponds to a group of operations packed together in one instruction.

The processes of packing independent operations into VLIW instructions is known as *(static) instruction scheduling*. The compiler has to reorder operations and pack them in a minimal number of instructions subjected to dependence and resource constraints.

Table 2.1 and figure 2.3 summaries the differences between the three architecture types.

2.1.3 Superscalars vs. VLIWs

Superscalar and VLIW processors are the two most successful ILP exploitation methods at the moment. Both methods have their strengths and weaknesses. We shall use the following issues to compare superscalars and VLIWs.

1. **Dynamic vs. static information:** Static information (known at compile-time) is in some ways less powerful than dynamic information (known at

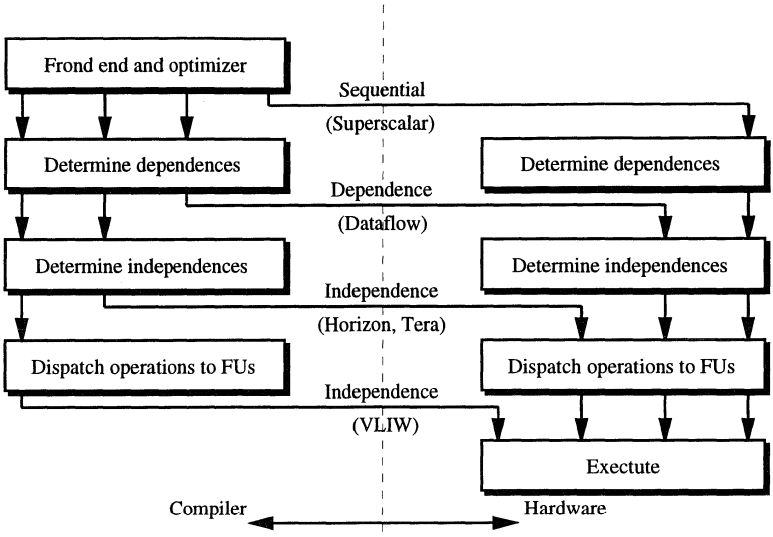


Figure 2.3: Dividing the work between the compiler and the hardware

run-time). A static scheduler for VLIWs is often hindered by ambiguous memory references. A pair of memory references where one of the two references is a store can only be reordered when the scheduler can guarantee that the two references access different memory locations. This is a problem for a static scheduler when it does not have sufficient information about the two references. A superscalar can reorder memory references at run-time by comparing their effective addresses[1]. Hardware support [48, 89] and software techniques [27, 112, 156] have been developed to assist a static scheduler in reordering ambiguous memory references.

Another example is branch prediction. Dynamic branch prediction [148] is usually more accurate than branch prediction based on profiling [81, 197] or static branch prediction [21, 212].

2. **Bird eye vs. tunnel view:** The amount of ILP that can be found depends largely on the number of operations that are considered simultaneously. For a superscalar this corresponds to the size of the instruction window, about 4 to 32 instructions (tunnel view). A static scheduler has a much larger scope, usually several basic blocks containing tens or hundreds of instructions (bird eye view). Although it is against the pure superscalar concept, superscalars can benefit by static scheduling for a particular implementation. This is especially true for superscalars with a small in-

[1] Although it is questionable whether the effective addresses are available in time and whether it is cost-effective to reorder memory references at run-time.

struction window and an in-order-issue instruction scheduler.

3. **Unpredictable external sources:** Operations with a variable latency, such as memory references and I/O operations, are better handled by superscalars than VLIWs. When a VLIW executes an operation that takes longer than the compiler assumed, the hardware should stall to prevent incorrect results. Abraham *et al.* [2] reports that latencies of memory operations can be predicted successfully by profiling them.

4. **Hardware vs. compiler complexity:** VLIW compilers do much of what superscalars do in hardware. This is one of the RISC principles: do what you have to do in hardware and shift the rest to the compiler[2].

5. **Binary compatibility:** For superscalars ILP is an implementation technique to improve performance and is architecturally invisible. This guarantees binary compatibility between superscalars with different ILP characteristics but with the same architecture. Changing ILP characteristics of VLIWs, such as adding an FU, changes the instruction format and therefore compatibility is lost. Compatibility is a very important issue for vendors to keep their costumers base. This is the main reason why most commercial successful ILP processors are superscalars. There are several possibilities to create some form of portability between VLIWs with different ILP characteristics although none of them is completely satisfactory: binary translation [17], dynamically scheduled VLIWs [168], rescheduling during page faults [53], and an architecture neutral distribution format [161].

6. **Code density:** When a static scheduler is not able to fill all operation slots of a VLIW instruction it has to insert no-op operations. This may lower the code density of VLIWs significantly and therefore result in more memory usage and instruction cache misses [181]. Existing VLIWs have tackled this problem by efficient encoding of no-ops in memory and expanding them during cache misses [51], by different instruction formats [50, 170, 179], or by decoupling the direct correspondence between operation slots and FUs [45].

7. **Performance scalability:** The complexity of the instruction scheduler of a superscalar is $O(n^2)$ where n is the number of instructions in the instruction window that are considered for issuing. The quadratic complexity limits the performance scalability of superscalars.

For (semi-)automatically generated ASPs we prefer VLIWs above superscalars. The hardware complexity of superscalars makes (semi-)automatic generation far too difficult. Binary compatibility is not an issue for ASPs. Companies that design ASPs usually also developed or possess the application

[2]Some people claim that RISC stands for 'Relegate the Interesting Stuff to the Compiler.'

source code and have qualified personnel to recompile the source code when needed. A low code density of VLIWs could be a serious problem for ASPs. However, the above mentioned techniques and instruction compression techniques [207] can help to alleviate this problem. Scalability in performance is an important requirement for ASPs since many interesting ASP applications have high performance requirements.

2.1.4 Available ILP

A lot of research has been performed to answer the question of how much ILP is available in applications and how it should be exploited. Most of these studies simulate the operation of a superscalar processor by scheduling an instruction trace subjected to certain conditions. The instruction trace is produced by simulation or execution of an instrumented program on real hardware. The conditions under which the trace is scheduled include: available resources, instruction window size, branch prediction, memory reference disambiguation, and scheduling barriers. The objective of ILP availability studies is to give upper bounds on what can be achieved with ILP and how it depends on hardware and compiler parameters. The value of these studies is limited because:

1. Most ILP upper bounds are not realistic. Most of the conditions under which they are obtained are not likely to be realizable in the near future, e.g., perfect memory reference disambiguation.

2. The studies ignore ILP enhancing techniques already developed or that may be developed in the future. Examples of existing ILP enhancing techniques are combining [154], program restructuring [142,205], tree height reduction [157], and interlock collapsing ALUs [146]. Therefore, the measured upper bounds do not have to be real upper bounds.

Nevertheless trace analysis is valuable to give an idea of what is achievable by ILP and under which conditions it can be achieved. The major conclusions that can be drawn from ILP availability studies are:

1. Jouppi and Wall [122] found that when ILP exploitation is limited to ILP within basic blocks the obtained speedups will not exceed 2-3 due to the limited basic block size.

2. Wall [198] found that even under ambitious conditions, such as good branch prediction, perfect memory reference disambiguation, a large number of registers, single cycle latency operations, a prefect cache, unlimited number of FUs, and a large instruction window, the average speedup will be around 7 and the median around 5.

3. Lam and Wilson [133] analyzed the effect of control flow dependences (dependences between a branch and instructions whose execution depends on the outcome of the branch) and speculative execution (execution of instructions before branches that they are control dependent upon are resolved). Control dependence analysis (at compile-time) and speculative execution result in an average speedup of 13.3. Better speedups can be obtained by executing multiple flows of control in parallel such as in MIMD, XIMD [208], multiscalar [84, 178], multithreading [191], and dataflow machines.

4. Theobald *et al.* [187] analyzed the effect of memory renaming. They found that memory renaming can remove a lot of dependences between memory references and increase ILP significantly. They furthermore introduced the notion of *smoothability*. Ideally, a maximum speedup of S relative to a machine with no ILP requires a machine with $\lceil S \rceil$ ILP. In reality the amount of ILP in an application is not evenly distributed and cannot be spread out evenly. Theobald defined smoothability as the ratio of the speedup of a machine with $\lceil S \rceil$ ILP and S. Most application exhibit a smoothability of more than 75%; this indicates that ILP can be exploited with reasonable hardware utilization.

2.2 Static ILP Exploitation

In the previous section we motivated our choice for VLIWs instead of superscalars. The challenging aspect of VLIWs is scheduling, i.e., reordering operations and packing them into instructions at compile-time.

2.2.1 Scheduling Constraints

Scheduling is subject to constraints that ensure correct semantics and correct hardware usage. Dependences between operations indicate a partial order in which the operations should be executed. Violating this order may change the semantics of the program. Resource constraints describe how operations can be packed into instructions.

Data dependences

Data dependences, or *precedence constraints*, are ordering constraints due to the usage of registers and memory locations. There are three types [127]:

1. There is a *flow dependence* from operation a to b if a defines a register or memory location that may be used by b.

2. There is an *anti dependence* from operation a to b if a uses a register or memory location that may be redefined by b.

3. There is an *output dependence* from operation a to b if a defines a register or memory location that may be redefined by b.

Flow dependences are also known as *true dependences* and anti and output dependences as *false dependences* because the latter can be eliminated by renaming. Figure 2.4b shows a *data dependence graph* (DDG) corresponding to the code fragment in figure 2.4a. In a DDG nodes corresponds to operations and edges to dependences between operations. Some anti and output dependences are not shown in figure 2.4b because they are covered by others. Figure 2.4b shows an anti dependence between operation `d` and `e`. This is due to the reuse of register `r3` in operation `e`. The anti dependence can be eliminated by using another register for `r3` in operation `e`. The resulting DDG is shown in figure 2.4c. In general, fewer dependences means more ILP and better performance.

Data dependences are usually associated with a delay that indicates the minimum number of cycles between the dependent operations to guarantee correct semantics and to prevent interlocks at run-time[3]. For flow dependences the delay is equal to the latency of the operation that produced the value that is used by the other. For anti and output dependences the delays are usually zero and one cycle, respectively. This assumes that when a register is read and written in the same cycle the previous value is read. Furthermore, it assumes that multiple writes to a register in the same cycle are undefined.

The length of a path in a DDG is defined as the sum of the delays of its edges. The longest path in a DDG, called the *critical path*, gives a lower bound on the required number of cycles needed to execute the code corresponding to the DDG. For the DDGs shown in figures 2.4b and 2.4c the lengths of their critical paths are 4 and 2, respectively, when we assume single cycle latency operations.

Data dependences due to memory usage are much harder to detect than dependences due to register usage. Unlike registers, memory is usually accessed by addresses not known at compile-time. A data dependence due to memory usage is present between two memory references if the two references may access the same memory location, i.e., the effective address of the two references might be the same at run-time. Determination of whether two references may access the same memory location is known as *memory reference disambiguation* or *alias analysis*. Memory reference disambiguation is one of the hardest problems for a scheduler. It has been solved reasonably well for numeric code where arrays are accessed regularly [91, 147, 166], but for pointer oriented non-numeric code it remains a hard problem [67, 101, 115]. When a memory refer-

[3]Interlocks will still occur when operations take more cycles than the scheduler expected.

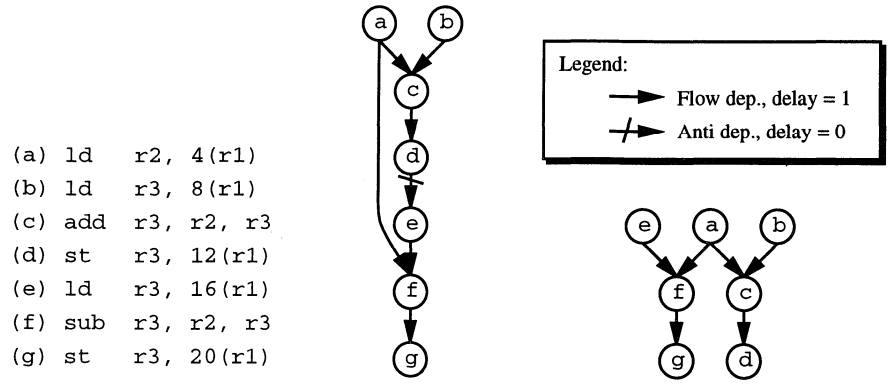


Figure 2.4: Data dependence graphs

ence disambiguator cannot determine independence it has to assume dependence.

The code in figure 2.4a does not contain data dependences due to memory usage. All memory addresses are the sum of the value of `r1` and a different constant. A relatively simple memory reference disambiguator can determine that all references access a different memory location (assuming that the references access four bytes and the memory is byte addressable).

Control dependences

A conditional branch may determine whether other operations are executed or not. This makes these operations control dependent on the branch. Figure 2.5 shows an example *control flow graph* (CFG) to illustrate control dependence. Nodes of the CFG represent basic blocks and edges correspond to possible direct transitions between basic blocks. In figure 2.5 the branch in basic block A controls the execution of all operations of basic blocks B, E, and F. All these operations are therefore control dependent on the branch of basic block A. Similarly, the operations of basic blocks C and D, depend on the branch of basic block B.

Control dependences are less stringent than data dependences; they can sometimes be violated without changing the semantics of the program. For example, figure 2.6 shows how a copy operation can be moved above a conditional branch on which it is control dependent. This is known as *speculative execution*. The scheduler speculates on the outcome of the conditional branch. It should be clear that speculation is allowed when it does not change the semantics of the program. This means that the speculated operation should not cause an ex-

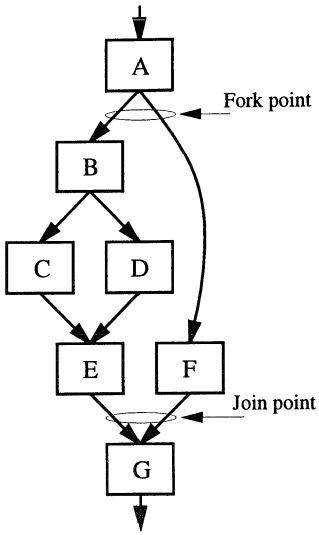


Figure 2.5: A control flow graph

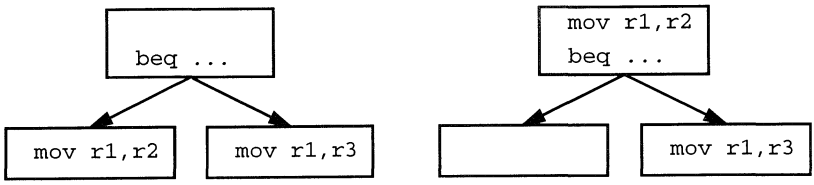


Figure 2.6: Speculative execution

ception (e.g., due to a divide by zero) or change the program state (overwrite live registers or memory locations) in case of misspeculation. The speculation in figure 2.6 is valid; a copy operation between registers cannot cause an exception and the register that it defines (`r1`) is not live below the branch.

As discussed in section 2.1.4, speculative execution is an important technique to improve ILP.

Resource constraints

Resource constraints express the resource limitations of the target machine. Resources of interest for the scheduler include: FUs, buses, RF ports, and operation slots. General purpose registers are usually not managed by the scheduler.

The common way to deal with resource constraints is the usage of a resource vector and reservation tables. The resource vector describes how many instances of each resource are available. For example:

$$\begin{pmatrix} 4 & 1 & 1 & 2 & 2 & 1 \end{pmatrix}$$

indicates the availability of four instances of resource type 1, one instance of resource type 2, and so on. Reservation tables tables describe the resource requirements of an operation. For example:

$$\begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

indicates that the operation needs one resource of type 1 and one of type 3 in its first cycle of execution and one resource of type 6 in its third cycle. The scheduler checks resource constraints by combining the reservation tables of the scheduled operations and comparing the result with the resource vector of the target machine.

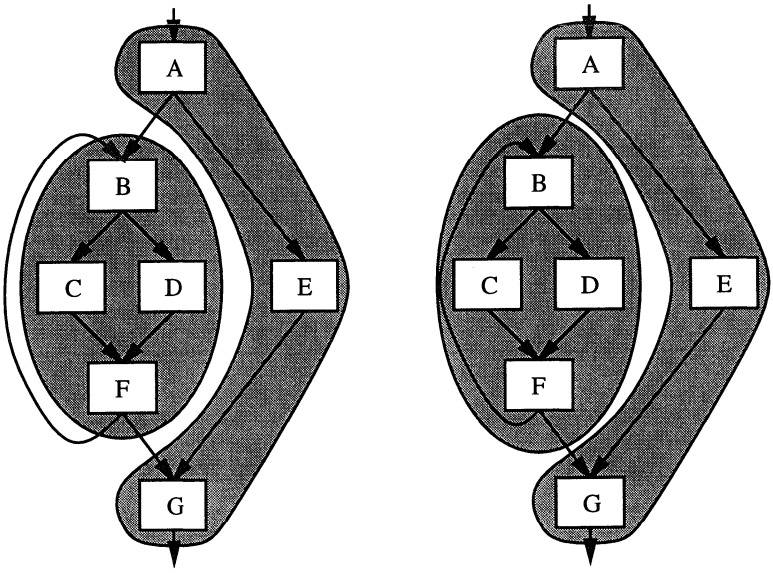
In chapter 3 we shall see that resource vectors and reservation tables are not sufficient to describe all resources of TTAs.

2.2.2 Scheduling Scopes

Compiler optimizations can be performed with different scopes. For example, register allocation can be performed for expressions, basic blocks (local register allocation), procedures (global register allocation), or collections of procedures (interprocedural register allocation). In general, a larger scope means more optimization opportunities and therefore potentially better results. In case of register allocation, a larger scope will contain more live ranges that can be allocated to registers and therefore more registers can be used. This implies that a large scope is required in order to use a large number of registers effectively. The same holds for scheduling; a small scheduling scope is sufficient for processors with a small amount of ILP, a larger scope is required for efficient usage of processors with more ILP.

Schedulers can be classified according to their scheduling scope into three categories:

1. **Basic block scheduling:** The scheduling scope consists of a single basic block. Each basic block is scheduled independently of the others. Since the size of a basic block is typically not more than 4 or 5 operations, which tend to be dependent on each other, the amount of ILP that can be exploited by basic block scheduling is very limited. Basic block scheduling is also known as *local scheduling*.



(a) Extended basic block scheduling (b) Software pipelining

Figure 2.7: Scheduling scopes

2. **Extended basic block scheduling:** The scheduling scope consists of an acyclic CFG. Figure 2.7a shows how an extended basic block scheduler might partition a CFG, corresponding to an if-then-else construct within a loop, into two scheduling scopes. The scheduler exploits inter basic block parallelism by moving operations between basic blocks belonging to the same scheduling scope. Examples of possible code motions are: A to E, E to A, G to E, B to C, and F to C. Inter basic block code motions may require speculative execution, code duplication, or both. For example, moving an operation from C to B requires speculation, and moving an operation from F to C requires code duplication (a copy of the operation needs to be placed in D as well). Extended basic block scheduling is also known as *global scheduling*.

3. **Software pipelining:** The scheduling scope consists of a cyclic CFG corresponding to a loop. By including backward edges (e.g., the control flow edge from F to B in figure 2.7b) and performing code motions over the backward edges, operations from different loop iterations are executed in parallel. The remaining acyclic parts of the CFG, such as the outer scheduling scope in figure 2.7b, are handled by an extended basic block scheduler. Exploitation of inter iteration ILP is very profitable for numeric code and other loop oriented code such as signal processing applications, since these applications spend a lot of their execution time in

loops of independent iterations. Software pipelining is also known as *cyclic scheduling*.

2.3 Transport Triggered Architectures

This section introduces TTAs starting with the principle of TTAs: programming data transports instead of operations. Subsection 2.3.2 gives an example of TTA programming. The next three subsections describe how immediate operands, control flow, and conditional execution can be implemented in TTAs. The last two subsections describe the interconnection network and functional units in more detail.

2.3.1 The Principle

One of the major problems of VLIWs, which is also a problem for other ILP processors, is the central RF and the required connectivity between the RF and the FUs. N FUs require $3N$ ports on the RF, $2N$ read ports and N write ports, and a bypass network between the FUs consisting of $O(N)$ buses and $2N$ multiplexors with $O(N)$ inputs. Both the RF and the bypass network have an $O(N^2)$ area complexity [56]. Furthermore, a large number of RF ports and a large bypass network are likely to limit the cycle time.

This problem is slightly alleviated by replacing the central RF by one RF for integer and one RF for floating point numbers. However, for VLIWs with a large number of FUs, also called wide VLIWs, the data path has to be partitioned into a number of clusters each containing one RF and a few FUs [51, 80] as shown in figure 2.8. The clusters are connected by buses to transfer data between the clusters.

Clustering has several serious problems:

1. Inter cluster communication takes time and resources (buses, RF ports, and operation slots).

2. It is highly questionable whether typical code is well clusterable. With clusterable we mean that the application code can be partitioned in relatively independent partitions that can be executed in lock step on the clusters without a lot of inter cluster communication.

3. It complicates code generation significantly [42, 74]. The compiler has to allocate operations and variables to clusters such that the work load is balanced among the clusters and the inter cluster communication is minimized. This may require duplication of operations and variables in order to reduce the inter cluster communication overhead [74].

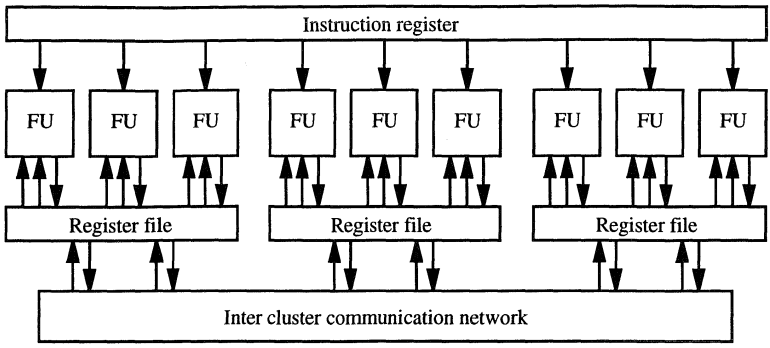


Figure 2.8: A clustered VLIW of three clusters each consisting of three FUs and one RF

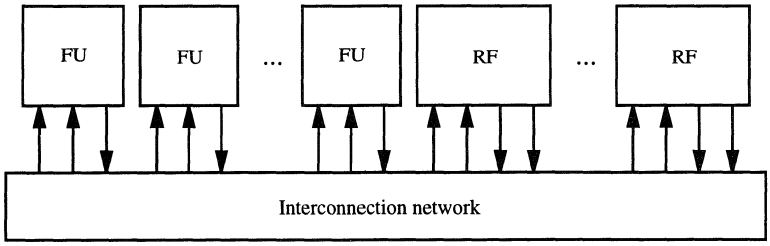
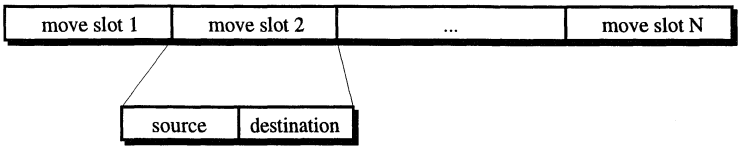


Figure 2.9: General structure of a TTA

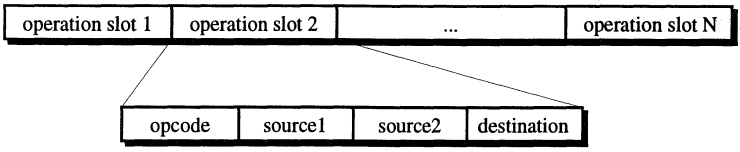
Due to these problems it is highly desirable to avoid clustering as much as possible and, when needed, partitioning the FU set into a few large clusters. This can be achieved by using the transport resources more efficiently. *Transport triggered architectures* (TTAs) do this by letting the compiler control the data transports. The idea is that a compiler is better capable to control the transport resources efficiently. The result is that clustering is not needed anymore or that the FU set can be partitioned into fewer and larger clusters (e.g., 2 clusters of 6 FUs instead of 4 clusters of 3 FUs).

Figure 2.9 shows the general structure of a TTA. A set of FUs and RFs are connected by an interconnection network. The interconnection network consists of a number of *move buses*. Each move bus is directly controlled by a *move slot* of the TTA instruction. Figure 2.10 shows the layout of a TTA instruction together with the instruction layout of a traditional *operation triggered architecture* (OTA). Each data transport, or *move*, is between two internal registers as specified by the source and destination fields of a move slot. From the compiler point of view the internal registers are divided into four categories:

1. **General purpose registers (GPRs):** like GPRs in OTAs, they are used for fast accessible storage of a small set of variables.



(a) Instruction format of an N move bus TTA VLIW



(b) Instruction format of an N functional unit OTA VLIW

Figure 2.10: Instruction layouts of OTAs and TTAs

2. **Trigger registers:** each trigger register belongs to an FU. When a value is moved to a trigger register an operation is initiated (triggered) and the value that is used to trigger the operation is used as operand. Usually FUs can perform more than one operation, e.g., an ALU can perform additions, subtractions, logical operations, and shift operations. For this purpose a trigger register is mapped at multiple register address locations. The address that is used to access the trigger register indicates the operation to be initiated, i.e., the operation code is encoded in the address.

3. **Operand registers:** these registers are used to provide operands to FUs that can execute operations that need more operands than the single operand provided by a trigger move.

4. **Result registers:** results of finished operations are placed in the result register of the FU that performed the operation. Although the TTA concept allows for FUs with multiple result registers we shall restrict ourselves to single result register FUs.

With TTAs we have added another layer to the scheme of figure 2.3 as shown in figure 2.11. The responsibility of controlling the data transports has been moved from the hardware to the compiler.

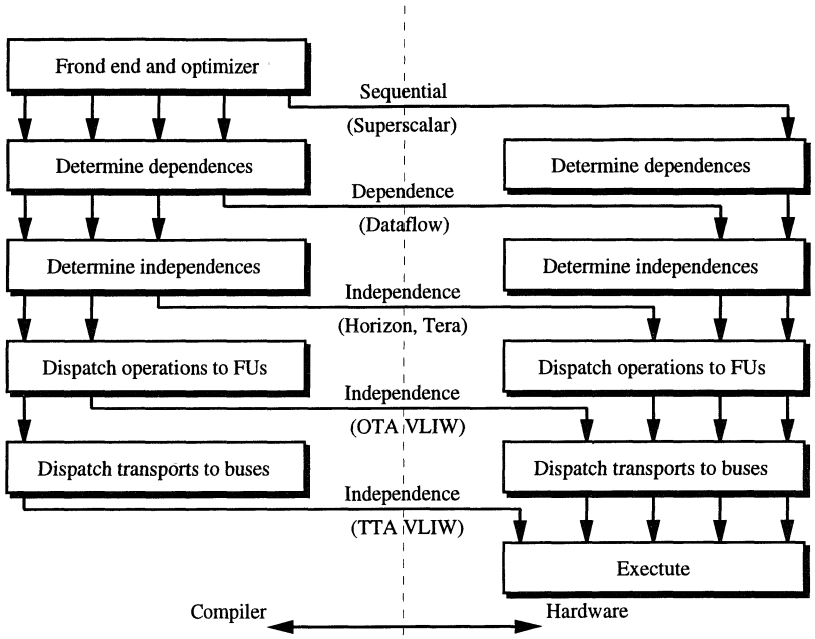


Figure 2.11: Dividing the work between the compiler and the hardware

2.3.2 An Example

To clarify TTA programming, let us consider how to translate the following code into TTA code and how to schedule it.

```
add  r1, r2, r3      /* r1 = r2 + r3 */
sub  r4, r2, r6      /* r4 = r2 - r6 */
st   r4, (r1)        /* store r4 at address r1 */
```

First we translate each n-operand m-result operation into $n - 1$ *operand moves*, one *trigger move*, and m *result moves*:

```
r2 -> add_o; r3 -> add_t; add_r -> r1
r2 -> sub_o; r6 -> sub_t; sub_r -> r4
r1 -> st_o; r4 -> st_t
```

The suffixes `_o`, `_t`, and `_r` denote operand, trigger, and result registers respectively. Registers `r1`–`r6` are GPRs. So far, all moves are to be executed sequentially. Scheduling consists of assigning FUs to operations and assigning cycles and move buses to moves. Figure 2.12 shows the DDG of the example code. The delay of the dependence between operand and trigger moves of the same operation is zero, i.e., the trigger move should be scheduled in the same cycle

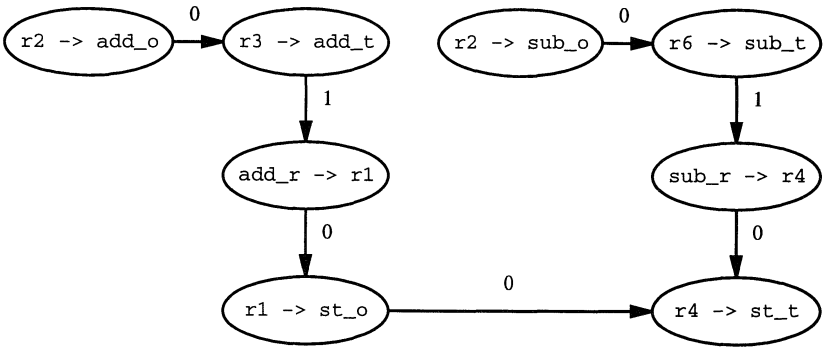


Figure 2.12: Data dependence graph of TTA code

as the operand move or should be scheduled later. The delay between a trigger move and a result move is equal to the latency of the FU that will execute the operation. Scheduling the result move earlier than prescribed by the trigger result delay will result in incorrect results or an interlock when interlocking is implemented. The delay of the two flow dependences is zero. This requires however that the results of the addition and subtraction can be bypassed directly from the FUs that produced the results to the FU that performs the store operation.

Let us assume we have two FUs named `alu1` and `alu2` for ALU operations, and one FU named `lsu` for load-store operations. Furthermore, assume that the three FUs have a single cycle latency and we have an interconnection network of four move buses with sufficient connectivity. Scheduling the example code will give the following two instructions:

```
r2 -> alu1.add_o; r3 -> alu1.add_t; r2 -> alu2.sub_o; r6 -> alu2.sub_t
alu1.add_r -> lsu.st_o; alu2.sub_r -> lsu.st_t
```

The prefixes `alu1`, `alu2`, and `lsu` indicate the FU on which the operation is executed[4]. The operand and trigger moves of the store operation have been bypassed, their source fields have been changed from a GPR into a result register. This is required when a move that uses a GPR (the source field of the move is a GPR) is scheduled in the same cycle as the move that defined the GPR (the destination field of the move is a GPR). Due to bypassing the result moves of the addition and subtraction have become useless if we assume that `r1` and `r4` are not used anymore after the store operation. Therefore the two result moves can be eliminated.

This simple example shows that due to bypassing the required number of moves and RF accesses is reduced. This reduces the required number of move

[4]The notation used has some redundancy. In the actual code the operation code is only specified with the trigger move.

buses and RF ports, i.e., transport triggering reduces the required number of transport resources.

2.3.3 Immediates

Immediates are usually divided into short and long immediates. Short immediates are provided by TTAs by storing them in the source field of a move slot and adding an *immediate bit* to the move slot to indicate whether the source field contains a register specifier or an immediate. This can be done for all move slots or a subset. The size of the immediate is equal to the size of the source specifier; and is therefore usually not more than 8 bits.

There are several ways to provide long immediates:

1. Making the instruction register partly accessible, i.e., part of the instruction register becomes readable from the interconnection network. This means that one or more immediate fields are added to the instruction format.
2. The disadvantage of the first method is the wasted instruction bandwidth when a long immediate field is not used. This can be improved by using multiple instruction formats that differ in number of long immediate fields and move slots. The different instruction formats can be distinguished by a few bits per instruction that indicate the instruction format.
3. Special FUs can be used to compose long immediates out of short immediates. This method is similar to the 'load upper immediate' instruction found in many RISC architectures [103].

In this thesis we will restrict ourselves to the first two methods.

2.3.4 Control Flow

Control flow is realized by making the program counter (PC) accessible. Writing an address to the PC causes a jump to that address. Depending on the instruction pipeline, a jump can be delayed by one or more instructions, i.e., one or more instructions after the instruction containing the jump are executed before the jump takes place. PC-relative and page-relative jumps can be provided by an extra adder and making the lower bits of the PC accessible, respectively.

Reading the PC is useful for obtaining the return address of a procedure call. Depending on the instruction pipelining scheme the read value may need to be corrected by adding a small constant to it.

2.3.5 Conditional Execution

Conditional execution is provided by means of *guarded* or *predicated* execution. Each move can be guarded by a boolean expression, called the *guard expression*. The move takes only place when its guard expression evaluates to *true*. The guard expression is built out of boolean variables stored in an RF of boolean registers. The boolean registers are defined by compare operations. For example, with guarded execution the following code:

```
if(r2 > 0 && r3 > 0)
    r5 = r4;
```

can be scheduled as follows:

```
r2 -> cmp.gt_o; 0 -> cmp.gt_t                    /* b1 = r2 > 0 */
r3 -> cmp.gt_o; 0 -> cmp.gt_t; cmp.gt_r -> b1;   /* b2 = r3 > 0 */
cmp.gt_r -> b2;
b1.b2: r4 -> r5;              /* if(r2 > 0 && r3 > 0) r4 = r5 */
```

Transforming multiple basic blocks into a single basic block by means of guarded execution is known as *if-conversion*. If-conversion makes basic blocks larger and transforms control dependences into data dependences [10].

Since all possible guard expressions of n boolean registers requires 2^n bits to encode, the number of boolean registers or the number of possible guard expressions should be limited. What is preferable depends on how the scheduler uses guarded execution.

Guarded execution is not unique for TTAs, it has been used for OTAs as well. Examples are the IBM VLIW [71], Cydrome Cydra 5 [170], and Philips LIFE [45, 130] VLIWs and sequential architectures that have conditional instructions such as the Acorn ARM [44], the HP PA-RISC [137], SPARC V9 [204], and the DEC Alpha [174]. Motivations for guarded execution are:

1. **Elimination of jumps:** as shown in the example above, jumps around small basic blocks can be eliminated.

2. **Filling jump delay slots:** operations from both the taken and the not taken paths can be moved into the delay slots of a jump by guarding them properly [111].

3. **Facilitating scheduling:** jumps are hard to schedule since they may change the shape of the CFG, and reordering jumps may lead to a significant code expansion. If-conversion prior to scheduling removes jumps and therefore facilitates scheduling [143, 202].

4. **Reducing branch mispredictions:** if-conversion tends to remove jumps with a high misprediction rate [145, 192]. This makes it interesting for superscalars with a high misprediction penalty.

If-conversion should be done with care. Guarded operations/moves that are fetched from memory require resources but do not contribute to the execution of the program when their guard expression evaluates to *false*. Furthermore, the length of the critical path of the DDG of the basic block resulting from if-conversion may be determined by dependences along infrequently executed paths of the if-converted CFG. Therefore, if-conversion has to be controlled by heuristics that take resource usage, dependence chains, and execution profiles into account.

2.3.6 The Interconnection Network

The interconnection network consists of a set of move buses. Each move bus is controlled by a move slot of the TTA instruction. Besides carrying data, the move buses also carry the register ids of the move slot and control signals for interlocking, guarding, and exceptions.

FUs and RFs are interfaced to the interconnection network of move buses by means of *sockets*. Input sockets transfer data from the move buses to the FU and RF inputs; and output sockets vise versa. Figure 2.13 shows the organization of a socket. A decoder compares the ids on the id buses to check whether a register accessible through the socket is selected. In case a register is selected the decoder controls the selector to select the right move bus. Besides controlling the selector the decoder is also capable to provide an opcode in case of accessing a trigger register, or a register index in case of accessing an RF.

RFs have usually one input/output socket per write/read port on the RF. Operand and trigger registers of FUs may share a socket in order to reduce hardware costs. However, since a socket can be used only once per cycle, only one of the operand and trigger registers that share a socket can be accessed per cycle.

The interconnection network may be fully connected, in which case every socket is connected to all move buses, or partially connected. A fully connected interconnection network simplifies the code generation task, but may also result in a high bus load on the move buses which may affect the cycle time. Therefore, in practice, the interconnection network will be partially connected and the compiler is responsible to use the available connections as well as possible.

Full connectivity is not required to be able to perform all possible combinations of data transports per cycle. It is possible to remove some connections while still all data transports combinations are possible. Figure 2.14 shows what we call *quasi fully* connected. Each move bus has $N - 1$ different input sockets to which it is not connected, where N is the number of move buses. The reduction compared to fully connected is thus $N(N - 1)$ connections. It is easy to verify that still all possible data transports are possible. It is also easy to see that removing more than $N - 1$ connections per bus makes some data transport com-

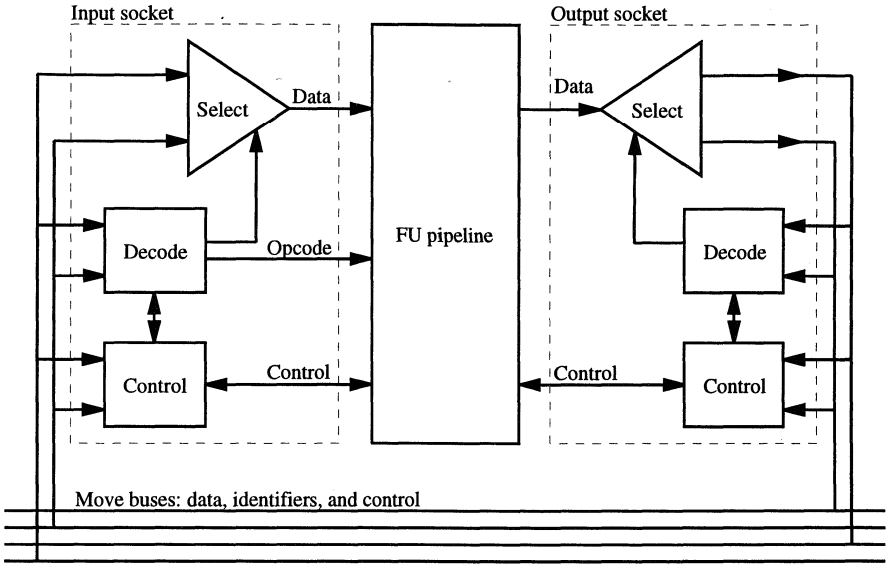


Figure 2.13: Input and output sockets to interface an FU to the move buses. Each socket is connected to two of the four move buses.

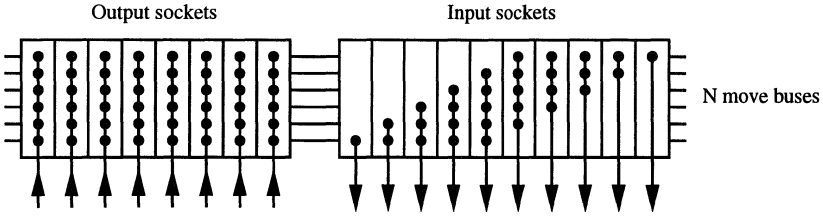


Figure 2.14: A quasi fully connected interconnection network

binations impossible. Although in theory quasi full connectivity should give the same performance as full connectivity it is not evident that a realistic compiler can achieve this.

The interconnection network should be designed based on the communication requirements between the FUs and RFs. This means that the interconnection network should provide the connections that are frequently requested by the compiler. For example, the address of a memory operation is often the result of an addition. This means that the address inputs of load-store FUs should be strongly connected to outputs of adders or ALUs. This can be achieved by dedicated move buses or by connecting them to the interconnection network such that their sockets have many move buses in common.

Besides one-to-one communication it is also possible to support one-to-many

communication or *multicasts*. This is achieved by multicast addresses which correspond to multiple inputs. Multicasts make it possible to combine multiple data transports from the same source. For example, the following two moves:

```
alu.add_r -> r3; alu.add_r -> lsu.ld_t
```

can be combined into a single multicast:

```
alu.add_r -> {r3, lsu.ld_t}
```

In the remainder of this thesis we will not utilize multicasts, although we will perform an experiment in chapter 6 that estimates the possible performance improvement due to multicasts.

2.3.7 Functional Units

A functional unit consists of a pipeline that performs the operations. The first stage of the pipeline corresponds to the operand and trigger registers, and the last stage of the pipeline corresponds to the result register; see figure 2.15. Non-pipelined or sub-pipelined FUs are possible by replacing the pipeline by circuitry that does not accept one operation per cycle. A non-pipelined or sub-pipelined FU is usually more cost-effective if the FU is not frequently used and a fully pipelined FU is expensive, e.g., a divide FU.

Having both operand and trigger registers at the input and a result register at the output means that it takes at least two cycles to perform an operation. If an operation is triggered on an FU with no intermediate pipeline stages in cycle T, the result is clocked into the result register in cycle $T + 1$, and the result can be used in cycle $T + 2$. The latency of the FU can be reduced by one cycle by removing the result register. This makes single cycle operations possible. The consequence is however that the delay of the last pipeline stage is added to the time required to do a data transport over a move bus.

There are several alternatives for pipelined FUs that differ in scheduling freedom, implementation complexity, and exception handling [56]. The two most useful alternatives are hybrid pipelines and virtual time latching pipelines.

Hybrid pipelines

Operations executed on *hybrid pipelined* FUs advance from one stage to the next stage if they do not overwrite results of previous operations. This is accomplished by attaching a valid bit to the trigger register, the result register, and each intermediate pipeline stage. Pipeline stage i of a fully pipelined FU will accept an operation from pipeline stage $i-1$ if: (1) pipeline stage $i-1$ contains

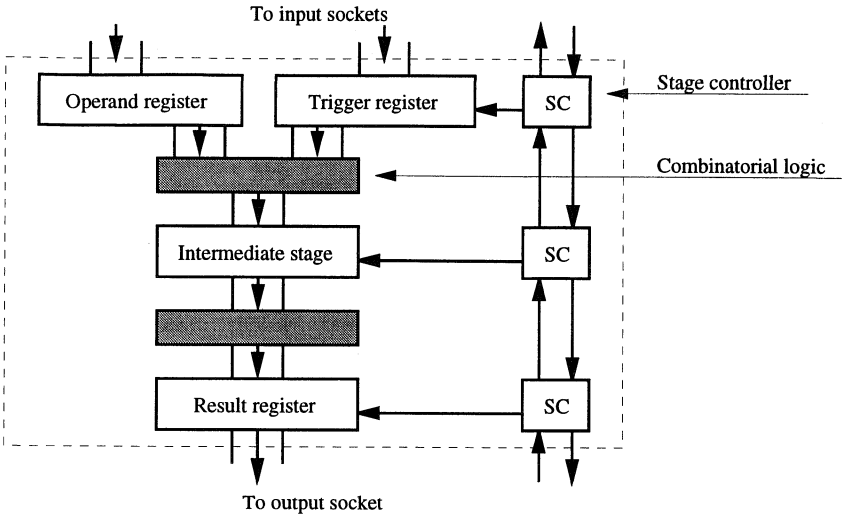


Figure 2.15: Organization of a functional unit

an operation (its valid bit is on), and (2) pipeline stage i does not contain an operation (its valid bit is off) or the operation it contains will be accepted by pipeline stage $i+1$. In other words:

$$\begin{aligned} accept_i &= old\_valid_{i-1} \wedge (\neg old\_valid_i \vee accept_{i+1}) \\ new\_valid_i &= accept_i \vee (old\_valid_i \wedge \neg accept_{i+1}) \end{aligned}$$

The equations for the first and last pipeline stages are slightly different. Writing to a trigger register causes its valid bit to be set. Similarly, reading from a result register causes its valid bit to be cleared, unless the result of another operation is simultaneously entering the result register.

Writing to a trigger register that cannot accept an operation because it contains an operation that cannot proceed to the next pipeline stage will cause a *pipeline full* exception[5]. Reading from a result register that does not contain a result of an operation will cause an interlock[6] until a result enters the result register. If the pipeline does not contain an operation, a *pipeline empty* exception will be raised.

The definition of $accept_i$ causes a ripple from the last pipeline stage to the first stage. This becomes a problem for deeply pipelined FUs. The other pipeline type, virtual time latching pipelines, does not have this problem.

[5]Low cost implementations do not have to support these exceptions.

[6]Low cost implementations do not have to support interlocking. Without interlocking the read value will be undefined and therefore the compiler cannot schedule a result move before the result is available.

Virtual time latching pipelines

Virtual time latching pipelines (VTLP) do not have the notion of stages containing valid operations. Operations continue from one stage to the next stage unconditionally. This means that when an operation is triggered in cycle T_1 on an N stage FU its result is available in cycle $T_1 + N$. How long this result will be available depends on how soon after cycle T_1 the FU is triggered for another operation. If the next operation executed on the same FU is triggered in cycle T_2 ($T_2 > T_1$), the result of the operation triggered in cycle T_1 will be available from cycle T_1+N until T_2+N-1. To ensure the availability of the result during this interval, VTLP FUs are locked during stalls.

A register that needs to be read within a certain time after it has been defined is known as a *hot spot*. Hot spots are present in many microcode programmable machines [134, 190] but also in some VLIWs, e.g., the Intel iWarp [50].

The following example illustrates the difference between hybrid pipelined and VTLP FUs.

```
... -> fu.add_o; ... -> fu.add_t[7]   /* trigger of operation 1 */
... -> fu.add_o; ... -> fu.add_t     /* trigger of operation 2 */
... -> fu.add_o; ... -> fu.add_t     /* trigger of operation 3 */
...
fu.add_r -> ...                      /* result of operation 1 */
fu.add_r -> ...                      /* result of operation 2 */
fu.add_r -> ...                      /* result of operation 3 */
```

On a 3 cycle latency hybrid pipelined FU this code will give correct results. After cycle 3 and during cycle 4 the pipeline is full; triggering another operation would cause a pipeline full exception. Figure 2.16b shows the contents of the three pipeline stages during the execution of the example code on a hybrid pipelined FU.

Executing the example code on a 3 cycle latency VTLP FU would result in incorrect results. In cycle 4 the result of operation 1 is overwritten by the result of operation 2, which causes that the result of operation 2 is read by the result move in cycle 5; see figure 2.16c. This is known as a *VTLP collision;* the result of an operation collides with the result of another operation that has not been read in time.

This example illustrates that operations scheduled on VTLP FUs have less scheduling freedom than operations scheduled on hybrid pipelined FUs. On the other hand, VTLP FUs have a scheduling advantage as well: results of operations can be discard without a result move. This is very useful for speculative execution of long latency operations. Operations speculatively executed on hybrid pipelined FUs may need a so called *junk move* on the mispredicted path to discard their results. Junk moves are result moves that move the result of a mispredicted operation to a non-existing address in order to discard it.

[7] Unrelevant details are replaced by three dots.

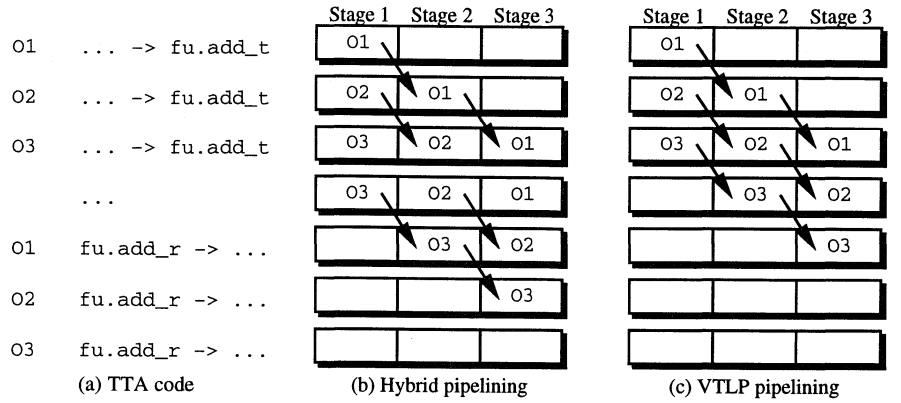


Figure 2.16: Pipeline contents of hybrid pipelined and VTLP FUs

2.4 Advantages and Disadvantages of TTAs

This section gives a qualitative description of the advantages and disadvantages of TTAs for ASPs. Quantitative evaluations of several aspects will be described in chapter 6.

2.4.1 Implementation Advantages

Regularity

TTAs are built out of a limited number of building blocks: move buses, sockets, RFs, FUs, and an instruction unit containing the PC and the instruction register. The RFs, FUs, and instruction unit are interfaced to the move buses by means of sockets. When the socket interface specification has been established, all components can be designed and implemented relatively independent of each other. This makes it possible to use layout generators and libraries of pre-designed components.

Flexibility

FUs can be added easily by 'plugging' them into the interconnection network. Global changes are only needed when there is insufficient addressing space on move buses to address the added FUs or when more connectivity is required to use the added FUs effectively.

There are virtually no constraints on the FUs. They can have many inputs and outputs. The only constraint that has to be obeyed is the socket interface specification.

Processor cycle time

The cycle time of a TTA implementation is lower bounded by the time required to do a data transport over a move bus which can be very short. This requires result registers at FU outputs and pipelining of all other processor components as long as they limit the cycle time.

Hardware utilization and scalability

Programming data transports leads to a better control over the transport resources and therefore a better utilization. A better utilization of the register ports allows for more FUs per RF. This reduces the need for clustering and if clustering is still necessary allows for fewer and larger clusters. This simplifies the code generation task and results in lower cycle counts if the application code is not well clusterable.

Decoupling of RF ports, buses, and FUs

On an OTA, each FU has three dedicated buses and RF ports. These ports can not be used for other purposes when the FU does not need them all in a cycle; for example, when the FU is not used or the FU executes a unary operation. Decoupling of RF ports, buses, and FUs allows for more efficient usage of RF ports and buses.

2.4.2 Compiler Optimizations

Bypassing

Bypassing allows two flow dependent moves to be scheduled in the same cycle. For example, if `r3 -> X` is flow dependent on `Y -> r3`, they can be scheduled in the same cycle provided that the first move is changed into `Y -> X`, i.e., the value of `r3` is bypassed.

Bypassing a value saves an RF read access. This, together with the fact that TTAs allow that freed RF read ports can be used for other purposes, reduces the RF port requirement of TTAs.

Dead result move elimination

When all moves that use a particular GPR are bypassed, the move that defines the GPR can be eliminated. These are usually result moves but it is also possible for copy moves/operations (moves from a GPR to a GPR or moves with an

```
ld r1, (r2)++; ld r3, (r4)++
ld r1, (r2)++; ld r3, (r4)++; add r5, r1, r3
ld r1, (r2)++; ld r3, (r4)++; add r5, r1, r3; st r5, (r6)++
                              add r5, r1, r3; st r5, (r6)++
                                              st r5, (r6)++
```

Figure 2.17: A software pipeline that adds two integer vectors. The loop control code is omitted for simplicity. The steady state of the software pipeline is shown between horizontal lines. This loop illustrates how large the difference between actual and worst case RF port requirement can be.

immediate source to a GPR). Dead result move elimination saves data transports, RF write accesses, and GPR usages. Again, due to the fact that the freed move buses and RF ports can be used for other purposes, the move bus and RF port requirements are reduced.

Due to the greedy nature of most schedulers, usages of a GPR are often scheduled as close as possible after the define of the GPR. Furthermore, most computed values are used a few times shortly after they have been defined [85]. Both facts promote bypassing and dead result move elimination.

Bypassing, dead result move elimination, and the fact that many operations do not need two GPR operands and produce a GPR result are the main reasons that the actual RF port requirement of TTAs is much lower than the three ports per FU worst case requirement of OTAs. An extreme example is shown in figure 2.17. During the steady state of the software pipeline shown in this example, the loop uses 6 GPRs and defines 6 GPRs per cycle. However, all usages are bypassed and all writes to the RF are dead, thus the actual RF port requirement is zero! This information is known to the compiler. OTAs cannot make use of this information because there is simply no way to pass it to the hardware.

Operand sharing

Operand moves can be shared by multiple operations if (1) they have a common operand whose value is the same for all operations, (2) the operations are scheduled on the same FU, (3) the common operand is provided to the FU via the same operand register, and (4) this operand register is not changed by other intervening operations. The following example illustrates operand sharing.

```
sp -> alu.add_o; 4 -> alu.add_t
sp -> alu.add_o; 8 -> alu.add_t; alu.add_r -> lsu.ld_t;
alu.add_r -> lsu.ld_t; lsu.ld_r -> alu.sub_o
lsu.ld_r -> alu.sub_t
alu.sub_r -> r4
```

This code computes the difference of two values stored in the current stack frame (`sp` is an alias for the GPR containing the stack pointer). The second move to the operand register of `alu` can be eliminated because the value of `sp` is already present in the operand register of `alu`.

Most of the shared operand moves have as source small immediates (such as 1, 0, -1, 4, etc.) or the stack pointer. In the first case a move bus is saved. In the second case an RF read access is saved as well.

Socket sharing

An output socket can be shared by multiple moves if they access the same register via the socket. For example, if two ALUs are available, the two additions in the previous example can be performed in parallel and the two stack pointer accesses can share an RF port, so only one RF read port is required.

```
sp -> alu1.add_o; 4 -> alu1.add_t sp -> alu2.add_o; 8 -> alu2.add_t
...
```

Socket sharing among mutually exclusive guarded moves[8] that access different registers is possible if the selection logic of the sockets takes guard expressions into account. This is usually not implemented since it increases the delay before a defined boolean register can be used.

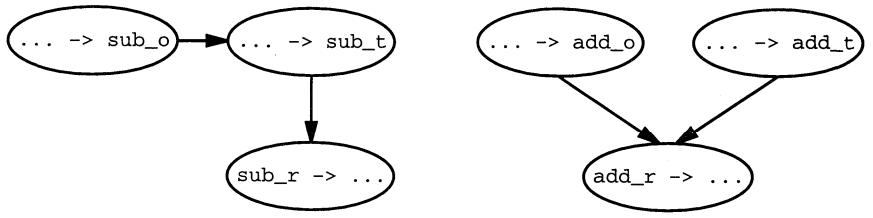
Scheduling freedom

In an operation triggered VLIW all operands of an operation are transported to an FU in the same cycle and results are transported as soon as they are available. A scheduler for TTAs has the freedom to schedule operand moves earlier than their corresponding trigger moves and to leave results longer in FUs. Scheduling an operand move earlier than its corresponding trigger move is useful when insufficient resources (e.g, a move bus or RF port) for the operand move are available in the cycle of the trigger move. The same holds for scheduling a result later than strictly necessary. Furthermore, the extra scheduling freedom might increase opportunities for bypassing, dead result move elimination, operand sharing, and socket sharing.

Operand swapping

The scheduler is free to swap operands of commutative operations when this leads to better results. Figure 2.18 illustrates this freedom. Unlike non-commutative dyadic operations, commutative operations do not have a dependence between their operand and trigger moves. If the operand move is

[8] At most one of the guard expressions will evaluate to *true*. For example, guard expressions `b1.b2` and `!b1` are mutual exclusive.



(a) DDG of a non–commutatitve operation

(b) DDG of a commutatitve operation

Figure 2.18: Operand swapping increases scheduling freedom of commutative operations

scheduled after the trigger move, the operand move becomes the trigger move and vice versa.

2.4.3 Disadvantages

Code density

TTA programs are less densely coded than OTA programs. This increases the code size and the instruction cache miss rate. The MOVE32INT [55, 61] and Phoenix [58] TTA implementations both have 16 bit move slots. When we assume that a typical OTA operation is equivalent to 2.5 TTA moves and an OTA operation requires 32 bits, TTA programs require roughly 25% more bits to encode.

Compiler complexity

Shifting responsibilities from the hardware to software increases the complexity of the compiler. This increases the development time of the compiler and may therefore increase the time-to-market of a TTA based product. By developing a prototype compiler we have demonstrated that compiling for TTAs is feasible.

Static bypassing

The bypass logic of an OTA is not completely predictable at compile-time even when we assume no disturbances of the control flow due to exceptions. Consider for example the following OTA code:

```
b1: add r1, r2, r3; !b1: sub r1, r2, r3
st r1, (r4)
```

Operand `r1` of the store operation is bypassed from the FU that performed the addition or from the FU that performed the subtraction depending on the value of boolean register `b1`. The only way to bypass `r1` is to generate two operand moves of `r1` for the store operation. One guarded with `b1` and the other guarded with `!b1`.

```
...
b1: alu1.add_r -> lsu.st_o; !b1: alu2.sub_r -> lsu.st_o; ...
```

Generating this code requires extra resources and complicates the scheduling process.

3 Basic Block Scheduling

This chapter and the following two chapters describe the developed compiler for TTAs. The main difference between a compiler for TTAs and a compiler for OTAs is the instruction scheduler. Other components such as the scalar optimizer, the register allocator, and the memory reference disambiguator are more or less the same.

The scheduler has been developed in three stages. We started with a basic block scheduler which will be described in this chapter. Subsequently, we upgraded this scheduler to an extended basic block scheduler and a software pipelining scheduler which will be described in the next two chapters.

This chapter consists of two sections. Section 3.1 gives an overview of the compiler and section 3.2 describes the basic block scheduler.

3.1 Overview of the Compiler

The compiler consists of two parts, (1) a front-end that transforms an application written in a HLL into sequential TTA code for a generic TTA, and (2) a back-end that transforms the generic sequential TTA code into parallel TTA code for a specific TTA. The generic sequential TTA code acts as intermediate code between the front-end and back-end. Simulators for both the sequential and the parallel TTA code are provided to verify the compilation process, to obtain application statistics, and to obtain profiling data for the back-end. Figure 3.1 shows the front-end, the back-end, the simulators, and the relations between them.

The back-end does not rewrite the sequential code if it is not possible to map it directly on the target hardware. This means that the front-end should only

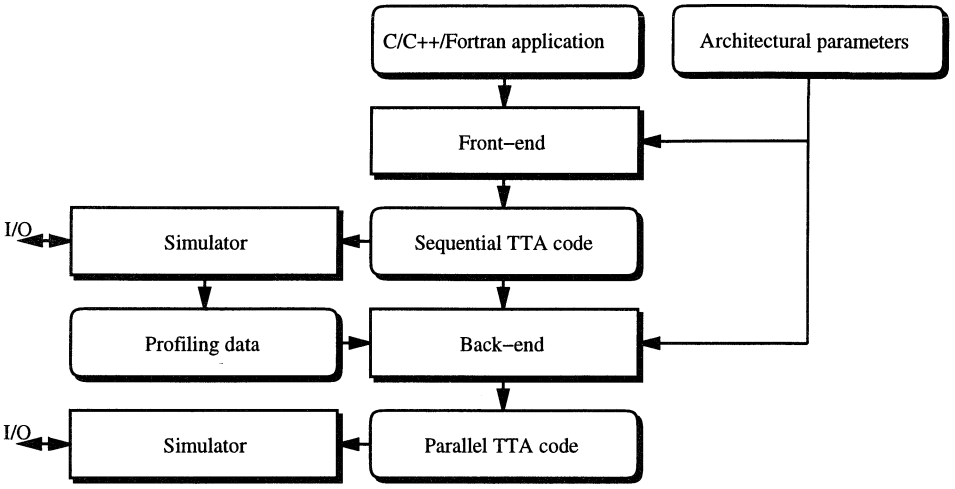


Figure 3.1: The code generation trajectory

generate operations that are supported by the target hardware. It also means that there is a minimum connectivity for the interconnection network. In practice this means that there should be (1) a path from an integer/FP RF read port to each integer/FP FU operand or trigger register, and (2) a path from each integer/FP FU result register to an integer/FP RF write port, where a path consists of an output socket, a move bus, and an input socket.

3.1.1 The Front-End

We use the GNU compiler (version 2.6.3) [180] of the Free Software Foundation (FSF) as front-end to transform an application coded in ANSI C, C++, or Fortran 77 into generic sequential TTA code. Our port of the GNU compiler, called *gcc-move*, together with other tools from the FSF such as the GNU assembler and linker, and the BSD4.3 libc and libm libraries give a stable production quality compilation platform which accepts the most popular HLLs.

The output of gcc-move is a binary executable which includes application code as well as library code in BSD a.out format. The executable is augmented with annotations for passing information from gcc-move to the back-end which is not directly available in a regular executable. This includes annotations for indicating the location of jump tables in the data segment, annotations for specifying the arguments and result registers of procedure calls, and annotations about memory references for the memory reference disambiguator of the back-end. An example of the latter is an annotation for volatile memory references that should not be reordered by the scheduler according to the ANSI C standard [124].

Operation type	Mnemonic	Optional
Integer add and subtract	`add, sub`	No
Integer multiply and divide	`mul, div, divu, mod, modu`	Yes
Word load/store	`ld, st`	No
Sub-word load/store	`ldb, ldh, stb, sth`	Yes
Integer compare	`eq, gt, gtu`	No
Shift	`shl, shr, shru`	No
Logical	`and, ior, xor`	No
Sign-extend	`sxbh, sxbw, sxhw`	Yes
Sub-word insert/extract	`insb, insh, extb, exth`	Yes
FP/integer conversion	`f2i, f2u, i2f, u2f`	Yes
FP load/store	`ldd, lds, std, sts`	Yes
FP operations	`addf, subf, negf, mulf, divf`	Yes
FP compare	`eqf, gtf`	Yes
User defined operations	user defined (see chapter 7)	Yes

Table 3.1: The operation repertoire of gcc-move

The operation set of gcc-move is shown in table 3.1. Most operations can be disabled by means of a compiler switch if the operation is not supported by the target TTA. In that case the operation is replaced by other operations or a call to a library function. The current version of gcc-move supports only register indirect and absolute addressing modes [103], i.e., load-store FUs do not perform address arithmetic. Experiments have shown that these two simple addressing modes are sufficient [1, 14, 182].

Gcc-move compiles for 128 32-bit integer and 128 64-bit FP registers. The FP register file can be disabled by means of a compiler switch. In that case the integer register file is used for storing FP values. Furthermore, a single boolean register is available for conditional execution. The motivation for a large number of GPRs is that the back-end re-performs register allocation. A large number of GPRs prevents that gcc-move will generate spill code that needs to be removed by the register allocator of the back-end.

The sequential code produced by gcc-move has several restrictions that simplify the scheduling task: (1) sources of operand and trigger moves are always GPRs or immediates, (2) destinations of result moves are always GPRs, (3) only moves to the PC (jumps) are guarded, and (4) moves belonging to the same operation are placed after each other. Writing parts of an application in sequential assembly code is therefore only possible when these restrictions are taken into account.

read sequential program	(section 3.1.3)
read machine description file	(section 3.1.3)
for each procedure do	
perform function inlining	(section 3.1.6)
for each procedure do	
transform an irreducible CFG into a reducible CFG	(section 3.1.4)
perform control flow analysis	(section 3.1.5)
perform loop unrolling	(section 3.1.6)
perform data flow analysis	(section 3.1.7)
perform memory reference disambiguation	(section 3.1.8)
perform register allocation	(section 3.1.9)
for each scheduling scope do	
perform instruction scheduling	(section 3.2)
write parallel program	

Figure 3.2: Transforming a sequential program into a parallel program

3.1.2 The Back-End

Figure 3.2 shows how a sequential program is transformed into a parallel program. The back-end starts with reading the sequential program and a machine description file that describes the target TTA. Next, function inlining is performed to improve ILP and to reduce procedure call overhead. Next, each procedure is individually transformed into a parallel procedure. This consists of various analysis phases and the actual scheduling itself. After all procedures have been processed, the parallel program is written to an output file.

The following sections describe the components of the back-end in more detail.

3.1.3 Reading the Sequential Program and the Machine Description File

The back-end starts with reading the sequential program. The program is stored in a collection of C++ classes. The hierarchy between the C++ classes is fairly straightforward; a program consists of a list of procedures, a procedure consists of a list of basic blocks with control flow edges between them, a basic block consists of a list of instructions, and finally, an instruction consists of a list of moves.

If available, the back-end reads a file containing profiling data produced by the sequential code simulator. The profiling data contains execution counts for each basic block and each control flow edge.

The machine description file contains all information that the back-end needs to know about the target TTA. It describes the available move buses, sockets,

FUs, and RFs. Furthermore, it provides information such as the jump latency and information about the guarding system. Figure 3.3 shows an example machine description file.

3.1.4 Transforming Irreducible CFGs into Reducible CFGs

The scheduler, and also the memory reference disambiguator, operate on reducible CFGs. These are CFGs consisting of loops with a single entry basic block. Reducible CFGs have the property that they can be reduced (hence the name reducible) to a CFG consisting of a single node by means of two simple transformations, T1, which reduces the number of edges, and T2, which reduces the number of nodes [3]. When this fails a node, consisting of a number of basic blocks in the original CFG, with n incoming edges is split into n copies, one for each incoming edge. This should restart the blocked reduction process. The node selected for splitting is based on the required amount of code duplication.

3.1.5 Control Flow Analysis

Control flow analysis computes the dominator and post-dominator relations between basic blocks, identifies loops, and computes the relations between loops.

Basic block A *dominates* basic block B if basic block A is on every path in the CFG between the entry node and basic block B. Similarly, basic block B *post-dominates* basic block A if basic block B is on every path in the CFG between basic block A and the exit basic block. Both relationships are computed by solving control flow equations [3]. After computing the dominate relationship the edges of the CFG are partitioned into backward edges and forward edges. Backward edges are edges whose head dominates their tail. All other edges are forward edges which form an acyclic graph.

A basic block is a loop header when it has an incoming backward control flow edge. The loop body of the loop belonging to the loop header H consists of all basic blocks from which a tail T of a backward control flow edge (T, H) can be reached without going through H. The found loops, called natural loops, have one header or entry node, and may have multiple backward edges and exit edges.

3.1.6 Function Inlining and Loop Unrolling

Since procedures are processed by the back-end individually, optimizations such as scheduling are not performed among operations belonging to different procedures. Function inlining removes these barriers by replacing a pro-

```
MoveBuses {
    m1      32, 8, signed;[a]
    m2      32, 8, signed;
    m3      32, 8, signed;
    m4      32, 8, signed;
    ...
}

Sockets {
    fu1_o   input,  {m1, m2, m3, m4        };[b]
    fu1_t   input,  {m1, m2,         m5, m6};
    fu1_r   output, {        m3, m4, m5, m6};
    fu2_o   input,  {m1, m2, m3, m4        };
    fu2_t   input,  {m1, m2,         m5, m6};
    fu2_r   output, {        m3, m4, m5, m6};
    ...
}

FunctionalUnits {
    fu1     hybrid, 1, fu1_o, fu1_t, fu1_r, {add, sub, eq, gt, gtu};[c]
    fu2     hybrid, 3, fu2_o, fu2_t, fu2_r, {ld, st};
    fu3     hybrid, 1, fu3_o, fu3_t, fu3_r, {and, ior, xor};
    ...
}

RegisterFiles {
    Integer         32, {r_i1, r_i2}, {r_o1, r_o2};[d]
    FloatingPoint   16, {f_i1, f_i2}, {f_o1, f_o2};
    Boolean         4,  {b_i1};
}

InstructionUnit {
    JumpLatency     2;
    BoolExprSize    2;
    ...
}
```

[a] Per move bus, its name, its width, the size of its immediate, and the type of its immediate.
[b] Per socket, its name, its type, and a set of move buses that it is connected to.
[c] Per FU, its name, its type, its latency, the socket that it is connected to, and its operation set.
[d] Per RF, its type, its size, and the sockets connected to its read and write ports.

Figure 3.3: Excerpt of a machine description file

cedure call with the body of the callee (the called procedure). This increases scheduling scopes and therefore exploitable ILP. It also improves the scope of other optimizations such as register allocation and it eliminates procedure call overhead. The only reason to omit function inlining is code expansion which increases the required amount of instruction memory, the number of memory system stalls (cache misses, TLB misses, and page faults), and the compilation time. Therefore, inlining should be controlled by a heuristic that takes the possible performance improvement and the required code duplication into account. The back-end uses the following heuristic for inlining: a function is inlined if

1. $100 \times a \times d_1/d_2 \times c_1/c_2 \geq s$ and
2. $d_1/c_2 \leq 1000$

where a is a user specified parameter that controls the aggressiveness (default value 10), d_1 the dynamic operation count of the callee, d_2 the total dynamic operation count of the whole application, c_1 the execution count of the call site, c_2 the invocation count of the callee, and s the static operation count of the callee. The first condition prevents excessive code expansion, while the second condition prevents inlining that is not likely to result in a better performance.

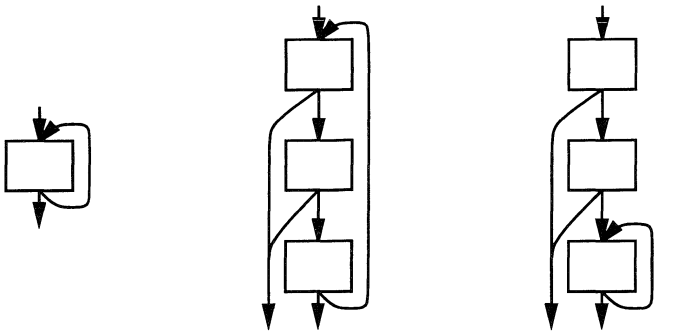
Since the back-end operates on complete programs, the function inliner can remove procedures that are not referenced anymore, directly or indirectly, after inlining. Procedures that are referenced once can therefore be inlined without code size expansion.

Another important scheduling barrier are backward control flow edges. Software pipelining schedules operations across backward control flow edges but software pipelining schedulers are usually limited in the structure of the loops that they can handle. Loop unrolling replaces the body of a loop with multiple copies of the loop body. This effectively reduces the number of executed backward control flow edges by N, where N is the number of copies or the *unrolling factor*. The back-end uses the following expression to compute N:

$$N = \min\{\lfloor \frac{250 \times a \times d_1}{d_2 \times s} \rfloor, 5\}$$

where a is a user specified parameter that controls the aggressiveness (default value 10), d_1 the dynamic operation count of the loop, d_2 the total dynamic operation count of the whole application, and s the static operation count of the loop body.

For loops with low trip counts it is usually better to perform loop peeling instead of loop unrolling [15]. Loop peeling places a few copies of the loop body before the loop header so that their operations can be scheduled with other operations of the scheduling scope enclosing the loop (see figure 3.4). The back-



(a) The original loop (b) Loop unrolling (c) Loop peeling

Figure 3.4: Loop unrolling and loop peeling

end performs peeling instead of unrolling if the average trip count of a loop is less than N.

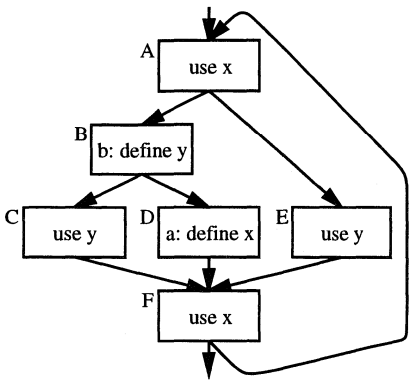
3.1.7 Data Flow Analysis

Data flow analysis consists of computation of live variables and DU (definition-use) chains [3]. The former is used for checking the validity of speculative execution during scheduling and for the computation of the interference graph used by the register allocator. DU chains are used by the memory reference disambiguator for building symbolic expressions and by the register allocator for renumbering live-ranges [39]. Both computations are performed by means of standard iterative data flow algorithms [3].

For dead result move elimination the scheduler needs to known whether a defined value in a loop may be used by an operation of the same loop in a future iteration. Similarly, for bypassing the scheduler needs to know whether a used value in a loop may be defined by an operation of the same loop in a previous iteration. This information, which we call *loop-carried DU chains*, is computed by solving a system of equations similar to the equations used for the computation of DU chains:

$$\begin{aligned} in'_B &= \bigcup_{P \in pred(B)} out'_P \\ out'_B &= (in'_B \cup insert_B) - kill_B \end{aligned}$$

The sets in'_B and out'_B are similar to the in_B and out_B sets used for the computation of DU chains. They contain the definitions that are reachable via the loop header of the loop to which basic block B belongs. The sets in'_B and out'_B are



BB	*insert*	*kill*	*in'*	*out'*
A	$\{a,b\}$	$\emptyset$	$\{a,b\}$	$\{a,b\}$
B	$\emptyset$	$\{b\}$	$\{a,b\}$	$\{a\}$
C	$\emptyset$	$\emptyset$	$\{a\}$	$\{a\}$
D	$\emptyset$	$\{a\}$	$\{a\}$	$\emptyset$
E	$\emptyset$	$\emptyset$	$\{a,b\}$	$\{a,b\}$
F	$\emptyset$	$\emptyset$	$\{a,b\}$	$\{a,b\}$

Figure 3.5: A CFG to illustrate loop-carried DU chains

computed out of $kill_B$, which is the same set used for computing DU chains, and a set called $insert_B$. The set $insert_B$ contains all definitions of a loop that are reachable at the entry of B if B is a loop header and is empty otherwise.

$$insert_B = \begin{cases} in_B \cap \bigcup_{M \in loop(B)} gen_M & \text{if } B \text{ is a loop header} \\ \emptyset & \text{otherwise} \end{cases}$$

Solving the equations by means of an iterative data flow algorithm gives the required information. Figure 3.5 shows a CFG to illustrate loop-carried DU chains.

3.1.8 Memory Reference Disambiguation

Memory reference disambiguation determines whether two memory references are dependent. Two memory references are dependent if at least one of them is a store operation and they may refer to a same memory location. Memory reference disambiguation consists of two steps: (1) deriving symbolic expressions that describe the address of each memory reference, and (2) testing whether the symbolic expressions of two memory references can become equal to each other.

A symbolic expression is a linear expression that describes the address of a memory reference in terms of (1) incoming procedure arguments, (2) induction variables of enclosing loops, and (3) definition points. The following example illustrates symbolic expressions:

```
void foo(int *a, int n)
{
    int *b = bar(...);

    for(int i = 0; i < n; i++) {
       a[2 * i] = ...;          /* ia(0) + 8iv(0) */
       ... = a[2 * i + 1];      /* ia(0) + 8iv(0) + 4 */
       ... = b[i];              /* dp(_bar -> call) + 4iv(0) */
    }
}
```

The first expression (written as comments) states that the address of `a[2*i]` is equal to the contents of the first incoming procedure argument ($ia(0)$) plus eight times the iteration counter of the enclosing loop ($8iv(0)$, the size of an integer is four bytes). Similarly, the last expression states that the address of `b[i]` is equal to the result of the procedure call to `bar` (dp(`_bar -> call`)) plus four times the iteration counter of the enclosing loop.

The method to find symbolic expressions is similar to methods described in [74, 151]. An expression is derived by chasing DU chains produced by data flow analysis. If the reaching definition of an address of a memory reference consists of an integer addition, the expression is obtained by summing the expressions of the operands of the integer addition. Similar actions are applied for copy, subtract, multiply, and shift operations. Chasing stops when an immediate, an incoming procedure argument, an induction variable update, or the stack pointer is encountered, or the expression becomes non-linear (e.g., due to a procedure call or a logic operation). In the latter case, the expression becomes a definition point.

If the symbolic expressions of two memory references are unequal for all integer values of induction variables, incoming procedure arguments, and values produced at definition points the two memory references are independent[1]. This is a sufficient but not a necessary condition. This means that memory reference disambiguation is not exact; the disambiguator may report dependences that do not exist in reality. The standard method to find out whether a linear expression has an integer solution is the gcd-test [22, 209, 213]. The gcd-test is based on the fact that a linear equation with integer coefficients $c = \sum_{i=1}^{n} c_i x_i$ has an integer solution if and only if $\gcd(c_1, \ldots, c_n)$ divides c.

Our memory reference disambiguator is limited due to the limited information that gcc can provide about memory references. For example, in the following code the two references to arrays `a` and `b` are clearly independent because the two arrays are different local variables.

[1] If the two memory references with symbolic expressions e_1 and e_2 access elements with different sizes, s_1 and s_2, where $s_1 < s_2$, the memory reference disambiguator has to test for $e_1 - ks_1 \neq e_2$ for $0 \leq k < s_2/s_1$. This assumes aligned memory references.

```
void foo()
{
    int a[100], b[100], ia[100], ib[100];

    for(int i = 0; i < 100; i++) {
        int ia_i = ia[i];
        int ib_i = ib[i];
        a[ia_i] = ...;        /* ia(sp) + ... */
        ... = b[ib_i];        /* ia(sp) + ... */
    }
}
```

The information about the references to `a` and `b` is lost in an early stage of gcc. The two references become references to locations somewhere within the current stack frame. This is a major problem for compilers for ILP processors; scheduling is a low-level optimization that requires source-level information for effective memory reference disambiguation. There are two ways to deal with this problem [88]: (1) perform memory reference disambiguation at source level and pass the results down to the scheduler, and (2) perform memory reference disambiguation just before scheduling and pass the required information down from source level to the disambiguator. Both approaches require a significant amount of engineering effort to maintain the transferred information during the numerous intermediate compiler phases.

To alleviate the memory reference disambiguation problem, our compiler supports annotations that allow the user to specify the absence of dependencies. For example, in the following code a `__not_aliased__` annotation specifies that the two incoming arguments `a` and `b` of `foo` are not aliased. This allows the scheduler to reorder the two assignments.

```
void foo(int *a, int *b)
{
    __not_aliased__(a, b);

    a[...] = ...;
    ... = b[...];
}
```

The `__not_aliased__` annotation and other annotations that are provided are implemented by means of pseudo operations. The memory reference disambiguator recognizes these operations, removes them, and stores them in a table. This table is referenced whenever the disambiguator has to disambiguate two memory references. The pseudo operations are treated as normal operations during previous phases such as function inlining and loop unrolling.

Annotations for memory reference disambiguation are very useful in an ASP design environment where a designer is usually willing to add annotations for a better performance. Annotations are also useful for standard library functions such as `memcpy` and `strcpy`.

3.1.9 Register Allocation

Register allocation is the most profitable optimization for processors with a modest amount of ILP [103]. For processors with more ILP, scheduling will become the most important optimization followed by register allocation.

One of the major problems in designing a scheduling compiler is the ordering of the scheduling and register allocation phases. There are three possibilities: (1) scheduling after register allocation, called *postpass scheduling*, (2) scheduling before register allocation, called *prepass scheduling*, and (3) combined scheduling and register allocation. Each of the three alternatives has its problems:

1. Postpass scheduling [102] suffers from false dependences introduced by the register allocator that restricts the scheduling freedom of the scheduler. For example, consider the following sequential OTA code:

```
add  pr1, pr2, pr3          /* pr1 = pr2 + pr3 */
st   pr1, 4(sp)             /* store pr1 at address sp + 4 */
add  pr4, pr5, pr6          /* pr4 = pr5 + pr6 */
st   pr4, 8(sp)             /* store pr4 at address sp + 8 */
```

   where `pr1` – `pr6` are pseudo registers created by the code generator and all six pseudo registers are dead after the code fragment. A register allocator will assign `pr1` and `pr4` to the same physical register since the live ranges corresponding to these pseudo registers are not overlapping. The consequence of this assignment is a false dependence between the first store and the second add operation which serializes the final schedule.

2. Prepass scheduling [47] also has its problems. First, scheduling before register allocation may needlessly increase lifetimes of pseudo registers which may increase register pressure significantly and may lead to generation of spill code. This typically occurs when operations that define pseudo registers can be scheduled early but the operations that use these pseudo registers cannot be moved with them because of dependences. Most schedulers exhibit this behavior; they try to schedule operations as early as possible.

   A second problem of prepass scheduling is that the register allocator inserts sequential code in the schedule for spilling, reloading, saving and restoring caller saved registers around call sites, and saving and restoring callee saved registers in procedure entries and exits. This extra code needs to be integrated with the already scheduled code by a postpass scheduling phase.

   A third problem of prepass scheduling occurs in combination with guarded execution. Accurate live-variable analysis of guarded code required for register allocation is complicated. Consider the following OTA code:

```
b1:  add  pr1, pr2, pr3; ...   /* if(b1) pr1 = pr2 + pr3 */
!b1: sub  pr1, pr2, pr3; ...   /* else   pr1 = pr2 - pr3 */
```

Both guarded operations individually cannot kill `pr1` but both operations together can (`pr1` is therefore dead above the first instruction). Special data structures, such as the predicate hierarchy graph [143], or time consuming symbolic analysis [73] are required to perform accurate live-variable analysis.

3. Combined scheduling and register allocation performs the two optimizations in one phase [74, 86]. A physical register is assigned to a pseudo register when the first operation that uses or defines that pseudo register is scheduled. Also this method has its problems. First, solving two complex problems in one phase leads to a more complex problem than the 'sum of the complexities' of the individual problems. Second, a stand alone register allocator wants to assign machine registers to pseudo registers in another order than pseudo registers are encountered by the scheduler. And third, overlapping of live ranges is not well defined for partially scheduled code.

The solution to the problems of postpass and prepass scheduling is to make the first phase aware of how its decisions affect the second phase and the overall performance. This is realized for prepass scheduling by letting the scheduler keeping track on the number of pseudo register simultaneously alive and preventing that this number approaches or exceeds the number physical registers [33, 92]. When this limit is approached the scheduler tries to delay operations that increase the number of live pseudo registers.

Postpass scheduling can be improved by making the register allocator aware of the code motions the scheduler wants to perform and preventing register assignments that prevent these code motions [13, 159, 162]. This is realized by adding extra interference edges to the interference graph used by a register allocator based on graph coloring [46]. In the example above the anti dependence between the second and third operation can be prevented by adding an extra interference edge between `pr1` and `pr4`. This prevents that the two pseudo registers will be assigned to the same physical register.

For our compiler we have chosen for postpass scheduling with the above mentioned solution to prevent the introduction of false dependences. Our motivation for this decision is that postpass scheduling is more suitable for our situation than the other two alternatives. Combined scheduling and register allocation makes the already complicated scheduler for TTAs even more complicated. Prepass scheduling was rejected mainly because the insertion of code by the register allocator in scheduled code is problematic for TTAs. Inserting load and store operations is likely to interfere with the already scheduled code. Consider for example the following scheduled TTA code:

```
... -> ls.ld_t; ...    /* trigger move of a load operation */
pr1 -> ...; ...        /* pr1 is spilled */
ls.ld_r -> ...; ...    /* result move of the load operation */
```

If `pr1` is spilled it is not possible to simply insert a load operation between the first and second instruction. The load operation and also the the addition to compute the address of memory location that is used to spill `pr1` needs to be interleaved with the already scheduled operations where all scheduling constraints have to be taken into account. There are many situations where this is not possible. In that case some of the scheduled code needs to be rescheduled; a complicated task.

Our compiler performs register allocation by the following steps:

1. A true data dependence graph (TDDG) is built. This is a DDG without false dependences.

2. A false dependence prevention graph (FDPG) is built. Nodes of the FDPG correspond to pseudo registers, and edges indicate that assignment of the incident pseudo registers to the same physical register will result in a false dependence. An FDPG is constructed by scanning the sequential code and adding an edge between pseudo registers A and B if (1) there is a move M_A that uses or defines A, (2) there is a move M_B that defines B, and (3) the scheduler might want to schedule M_B before M_A. The last condition is true when there is no path in the TDDG from M_A to M_B.

   Each FDPG edge has a priority reflecting the possible negative effect on the performance when the two incident pseudo registers are assigned to the same physical register. This priority value is proportional to the execution count of M_A, the probability that M_B will be executed after M_A, and inversely proportional to the number of moves between M_A and M_B in the sequential code.

   $$priority(A,B) = \frac{execution\_count(M_A) \times probability(M_A, M_B)}{distance(M_A, M_B)}$$

   The rationale for the distance factor is that there is less need for reordering independent moves located far apart from each other in the sequential code.

   As will be described in chapter 5, DDGs of loops that will be software pipelined are cyclic. This means that no edges will be added to the FDPG in case the DDGs of these loops are strongly connected. Therefore, we have to handle these loops differently. We do this by pre-software pipelining them before register allocation without considering resource constraints and false dependences. The resulting schedule is analyzed to see which operations have been reordered. Edges are added to the FDPG to avoid register assignments that prevent these code motions.

3. Register allocation is performed by Briggs' optimistic register allocation algorithm [39]. Briggs' register allocator is an improved version of Chatin's register allocator [46]. False dependences are prevented by adding the edges of the FDPG to the interference graph used by the register allocator. When the register allocator runs out of physical registers (in the select phase of Briggs' register allocator [39]), it has to choose between spilling and introducing a false dependence. Our compiler always chooses for the latter. It does that by ignoring low priority FDPG edges.

The above described register allocator works for a single RF, or one RF for integer and one RF for FP numbers. Code generation for multiple RFs of the same type can be done by distributing the pseudo registers across the RFs. The simplest method to do this is by numbering the physical registers such that register ri is located in RF $i \bmod N$, where N is the number of RFs that are numbered from 0 to $N-1$ (low-order interleaving). This requires that all RFs have the same number of registers and ports. This method is similar to banked data caches [177, 206] used for example in the Intel Pentium [11] and MIPS R8000/TFP [110] processors. The difference is that access conflicts of banked caches are handled dynamically. In order to reduce RF port resource conflicts during scheduling, a smart register allocator should distribute the pseudo registers such that RF accesses are distributed evenly, spatial and temporal, among the RFs [119].

3.2 The Basic Block Scheduler

The task of a basic block scheduler is to reorder operations of a basic block and pack them into a minimum number of instructions subjected to dependence and resource constraints. This problem is known to be NP-complete [31, 90], so a both efficient and optimal algorithm is very unlikely. Several techniques have been developed that generate optimal or high quality near-optimal code at the expense of long scheduling times. Examples are [49] which uses branch and bound techniques, [93] which transforms the scheduling problem to an integer linear programming formulation, [24] which solves the scheduling problem by means of genetic algorithms, and [62] which uses neural networks. Due to their time complexity, these techniques are only applicable to small code fragments and situations where long scheduling times are acceptable.

Virtually all efficient scheduling techniques are based on list scheduling [34, 77, 95, 102, 199] which had its origins in local microcode compaction [134]. List scheduling works very well in practice. It gives most of the time optimal results especially when basic blocks are not very large and resource constraints are not very tight. This is the reason why most existing schedulers, including ours, are based on list scheduling.

```
proc Schedule(DDG = (V, E))
beginproc
    ready = {v | ¬∃(u, v) ∈ E}
    sched = ∅
    while sched ≠ V do
        v = SelectOperation(ready)
        cycle(v) = max{cycle(u) + delay(u, v) | (u, v) ∈ E}
        while ResourceConflicts(v, cycle, sched) do
            cycle(v) = cycle(v) + 1
        endwhile
        sched = sched ∪ {v}
        ready = {v | v ∉ sched ∧ ∀(u, v) ∈ E, u ∈ sched}
    endwhile
endproc
```

Figure 3.6: Operation based list scheduling

3.2.1 List Scheduling for OTAs

List scheduling works by repeatedly assigning a cycle to an operation. This is done without any form of backtracking and lookahead. In order to prevent that the scheduler will get stuck because an operation cannot be placed within an interval determined by its scheduled predecessors and successors in the DDG, operations are scheduled in a topological order, i.e., an operation is scheduled after all its predecessors have been scheduled.

There are two variants of list scheduling: *operation based* and *instruction based* list scheduling. Figure 3.6 shows the operation based list scheduling algorithm. The input of the scheduler is a DDG (V, E), the result is an assignment to $cycle(v)$ for every node $v \in V$. The scheduler maintains a *ready list* of operations that are ready to be scheduled. An operation is ready when it is not scheduled yet and all its predecessors in the DDG are scheduled. The scheduler repeatedly selects an operation from the ready list and places it in the first cycle where dependence and resource constraints are satisfied. This is repeated until all nodes are scheduled.

Instruction based list scheduling, shown in figure 3.7, fills one cycle (instruction) at a time. It proceeds to the next cycle if it is not possible to place more operations in the current cycle. An operation can be placed in the current cycle if it is ready and its incoming edges allow it to be placed in the current cycle. This is repeated until all nodes are scheduled.

Besides scheduling the DDG top-town, where an operation is scheduled after all its predecessors have been scheduled, it is also possible to schedule the DDG bottom-up. This means that an operation becomes ready when all its suc-

proc Schedule($DDG = (V, E)$)
beginproc
  $ready = \{v \mid \neg\exists(u, v) \in E\}$
  $ready' = ready$
  $sched = \emptyset$
  $current\_cycle = 0$
  while $sched \neq V$ **do**
    for each $v \in ready'$ **do**
      if $\neg$ResourceConflicts($v, current\_cycle, sched$) **then**
        $cycle(v) = current\_cycle$
        $sched = sched \cup \{v\}$
      endif
    endfor
    $current\_cycle = current\_cycle + 1$
    $ready = \{v \mid v \notin sched \wedge \forall(u, v) \in E, u \in sched\}$
    $ready' = \{v \mid v \in ready \wedge \forall(u, v) \in E, cycle(u) + delay(u, v) \leq current\_cycle\}$
  endwhile
endproc

Figure 3.7: Instruction based list scheduling

cessors are scheduled. Furthermore, operations are moved to an earlier cycle, instead of a later cycle, if there are resource conflicts with already scheduled operations. It has been claimed that bottom-up leads to shorter live-ranges and therefore a lower register pressure [132]. Nevertheless, most schedulers use top-town scheduling. The main reason for this is that bottom-up scheduling is hard to combine with extended basic block scheduling. This will become clear in the next chapter.

The performance of list scheduling is highly dependent on the order in which the operations are scheduled. This order is determined by the SelectOperation function in figure 3.6 and the *for each* loop in figure 3.7. A lower bound on the schedule length is given by the critical path length L_{max}. In order to reach this bound every node should be scheduled between its as-soon-as-possible (ASAP) and its as-late-as-possible (ALAP) limits.

$$asap(v) = \begin{cases} \max\{asap(u) + delay(u, v) \mid (u, v) \in E\} & \text{if } pred(v) \neq \emptyset \\ 0 & \text{otherwise} \end{cases}$$

$$alap(v) = \begin{cases} \min\{alap(u) - delay(v, u) \mid (v, u) \in E\} & \text{if } succ(v) \neq \emptyset \\ L_{max} & \text{otherwise} \end{cases}$$

The distance between these limits corresponds to the number of cycles in which the operation can be placed. This value is known as the *slack* or *mobility* of an operation.

$$slack(v) = alap(v) - asap(v)$$

For operation based list scheduling priority should be given to operations with a small slack above operations with a larger slack. Operations with more slack have more chance that they can be placed between their *asap* and *alap* limits. For instruction based list scheduling the actual slack of a ready operation corresponds to the difference between its *alap* value and the current cycle instead of the difference between *alap* and *asap*. Therefore, priority should be given to operations with a low *alap* value.

3.2.2 List Scheduling for TTAs

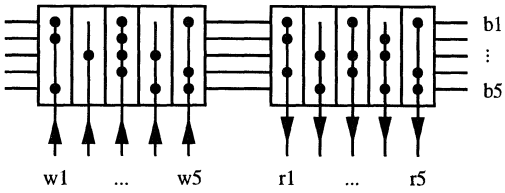
Scheduling an operation on a TTA consists of scheduling the moves that perform the operation. We schedule the moves of an operation in one indivisible action. Scheduling moves individually may lead to ineffective hardware usage or even scheduling deadlocks. For example, operand and trigger moves of an operations should be scheduled close together to prevent that an operand register is needlessly long occupied. At the moment an operand move is scheduled it is hard to determine in which cycle the corresponding trigger move will be scheduled. If it is far away from the operand move, the operand register is occupied for a long time and other operations cannot use the FU. Scheduling deadlocks may occur due to the fact that an operand register is occupied from the moment of the operand move to the corresponding trigger move. There are scenarios possible where the trigger move of operation A cannot be scheduled because it depends on an operation B that needs the same FU, and B cannot be scheduled because operation A has occupied the operand register.

Another motivation for not scheduling moves individually are the problems that arise because of VTLP FUs. This type of FU requires that results are read in time before they are overwritten by successive operations scheduled on the same FU. If moves are scheduled individually and a result move cannot be scheduled in time, because of resource or dependence constraints, the scheduler has to unschedule the operation, and the operations that depend on it, and schedule the operation on another FU or in a later cycle. This increases compilation time and engineering complexity of the scheduler.

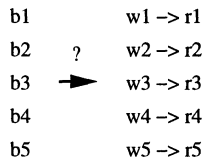
Due to the decision to schedule the moves of an operation in one indivisible step the choice for operation based list scheduling is the most natural one. Instruction based list scheduling fills instruction by instruction which is impossible due to the positive delay between trigger and result moves.

3.2.3 Resource Assignment

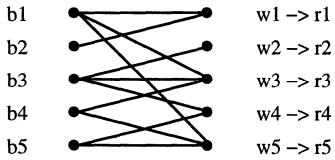
Another important design decision is when to do resource assignments. The scheduler has to assign FUs to operations, immediate fields to (long) immediate operands, and move buses and sockets to moves. Ideally, resource assignments should be postponed as long as possible, and during scheduling only the possibility of an assignment should be checked. In reality it is both hard to implement and computational expensive to do this for all resources of a TTA. We decided to perform FU, immediate field, and socket assignment during scheduling, and to perform move bus assignment after scheduling. The motivation for this is that there are usually only a few candidates for FU, immediate field, and socket assignments, i.e., there are usually only one or two FUs per FU type, one or two immediate fields, and a low number of read/write ports per RF especially when there are multiple RFs.



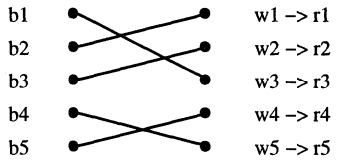
(a) An interconnection network



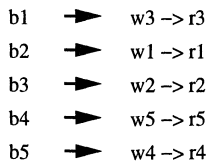
(b) An assignment problem



(c) A bipartite matching problem



(d) A matching



(e) An assignment

Figure 3.8: Move bus assignment via bipartite matching

Move bus assignment is performed after scheduling. This is necessary for efficient code generation for irregularly connected interconnection networks and if we want that quasi full connectivity performs as well as full connectiv-

ity. Figure 3.8 shows how move bus assignment is checked during scheduling and how the actual assignment is performed after scheduling. The problem to be solved is: given a set of moves between sockets that have to be performed ($w1 \rightarrow r1$,..., $w5 \rightarrow r5$ in figure 3.8) and an interconnection network of move buses ($b1$,..., $b5$), can each move be assigned to a different move bus that is capable to do the data transport? This problem is solved by transforming it to a bipartite matching problem [5, 69, 108]. An undirected bipartite graph $G = (V_d \cup V_m, E)$ is built were V_d corresponds to the data transports, V_m to the move buses, and E contains an edge between $d \in V_d$ and $m \in V_m$ if move bus m is capable to perform data transport d. An assignment can be made if there is a matching M with $|M| = |V_d|$. A matching M is a subset of E such that for all nodes in $v \in V_d \cup V_m$, at most one edge of M is incident on v. Clearly, a matching M of size $|V_d|$ gives us directly an assignment we are looking for. The bipartite matching problem can be solved exactly by an $O(m\sqrt{n})$ time complexity algorithm, where n is $|V_d \cup V_m|$ and m is $|E|$ [5, 69]. Due to the large number of times that the assignment question needs to be answered, it is computationally to expensive to solve it by means of a bipartite matching. Therefore, the bipartite matching algorithm is preceded by a simple, fast, but inexact first-fit type assignment algorithm that can handle most cases successfully ($\geq 95\%$ for irregular interconnection networks). Only if the first-fit type assignment algorithm cannot find an assignment the bipartite matching algorithm is used.

Assigning move buses during scheduling may or may not succeed for the example of figure 3.8 depending on the order in which the moves are scheduled and the used assignment policy. If $b1$ is assigned to $w1 \rightarrow r1$ first, then $b3$ to $w2 \rightarrow r2$, $b4$ to $w3 \rightarrow r3$, and $b5$ to $w4 \rightarrow r4$, the assignment fails because $b2$, the only free move bus left, cannot be assigned to $w5 \rightarrow r5$.

Incorporating multicasts in the move bus assignment algorithm is not directly possible. A possible solution might be to first try to assign move buses without using multicasts. If this fails, sets of moves that can be combined by a multicast can be represented by one element in V_d. The bipartite matching problem has to be solved for each combination until a matching is found. This works as long as the number of possible combinations is not too large.

Functional unit, immediate field, and socket assignments are performed by a first-fit type assignment algorithm. The first free resource that fits is selected. If there are resources with a different but overlapping functionality, it is preferable that the most specialized resource is selected, and more universal resources are kept for other operations. For example, if there are two FUs, f_1 and f_2 where the operation set of f_1 consists of load and store operations, the operation set of f_2 consists of only load operations, and both are available for a load operation, f_2 is preferred. The order in which resources are examined for assignment corresponds to the order in which they are listed in the machine description file. The user should therefore specify the resources in increasing generality, e.g., f_2 before f_1.

```
proc ScheduleOperation(o, t, r)
beginproc
    for cycle(t) = FirstCycle(t) to ∞ do
        for each f ∈ FUset do
            if • Operation(t) ∈ OperationSet(f) ∧
               • IsNotTriggered(f, cycle(t)) ∧
               • TransportResourcesAvailable(t, cycle(t)) ∧
               • ScheduleOperandMove(o, t, f) ∧
               • ScheduleResultMove(r, t, f)
            then
                return
            endif
        endfor
    endfor
endproc
```

Figure 3.9: Scheduling an operation

3.2.4 Scheduling an Operation

Scheduling an operation on a TTA differs significantly from scheduling an operation on an OTA. On an OTA scheduling an operation consists of (1) computing the first cycle where the operation can be placed and (2) combining reservation tables of already scheduled operations with the reservation table of the operation to be scheduled and comparing the result with the resource vector of the target machine to check whether the operation can be place in a certain cycle. Scheduling an operation on a TTA is much more complex; checking for resource conflicts is much more complex and most of the TTA specific optimizations mentioned in section 2.4.2 have to be performed during scheduling.

The scheduler distinguishes the following operations types: jumps, (procedure or operating system) calls, copies, and data operations. Scheduling jumps, calls, and copies is not much different from scheduling them for OTAs. Scheduling a data operation starts with scheduling the trigger move of the operation followed by the operand and result moves of the operation.

Scheduling a data operation consists of finding a cycle c and an FU f such that (1) the operation is member of the operation set of f, (2) f is not triggered by another operation in cycle c, (3) transport resources are available for the trigger move in cycle c, and (4) operand and result moves of the operation can be scheduled; see figure 3.9. The transport resources include input and output sockets, a move bus, and if required a long immediate field. Sockets and immediate fields are assigned immediately to the move. Assignment of a move bus takes place after scheduling; only the possibility of an assignment is checked during scheduling.

```
proc ScheduleOperandMove(o, t, f)
beginproc
    for cycle(o) = cycle(t) downto FirstCycle(o) do
        if IsOperandRegisterOccupied(f, cycle(o)) then
            return false
        endif
        if TransportResourcesAvailable(o, cycle(o)) then
            return true
        endif
    endfor
    return false
endproc
```

Figure 3.10: Scheduling an operand move

Scheduling an operand move starts by trying to schedule the operand move in the same cycle as the trigger move. It proceeds to an earlier cycle if not all required transport resources are available. Scheduling of an operand move fails if the cycle becomes earlier than is permitted by data dependences or if the operand move overwrites an operand register occupied by another operation; see figure 3.10. If scheduling fails, the operation has to be scheduled on another FU or the trigger move has to be delayed.

Scheduling a result move is slightly more complex. The following checks have to be made: (1) trigger and result moves have to be scheduled in FIFO order, (2) on hybrid pipelined FUs the number of operations in the pipeline may not exceed the capacity of the pipeline in order to prevent pipeline overflows, and (3) on VTLP FUs collisions have to be prevented. All these tests are performed by keeping track on the state of the FU. The FIFO ordering is checked by tracking the number of result moves of other operations that have to take place before the result move can be scheduled, and by marking the operations that are triggered after the operation being scheduled is triggered. The capacity check is made by means of information that contains the number of operations per cycle for each FU and cycle. The collision check is made by administrating the interval $[T + L, R]$ for each operation scheduled on a VTLP FU, where T is the trigger move cycle, R the result move cycle, and L the latency of the operation. A collision occurs when these intervals overlap.

Procedure calls on TTAs give extra scheduling constraints (not checked in figures 3.9–3.11). Operations should not be be scheduled across call sites, i.e., if a call is scheduled in cycle C, and the latency of a call is L cycles, all moves belonging to the same operations should be scheduled either before cycle $C + L$ or after cycle $C + L - 1$. The algorithms in figures 3.10 and 3.11 should therefore fail (return false) if an operand/result move is scheduled at the other side of a call site than their corresponding trigger move.

```
proc ScheduleResultMove(r, t, f)
beginproc
    count = OperationsInPipeline(f, cycle(t))
    for cycle(r) = cycle(t) + 1 to ∞ do
        if IsHybridPipelinedFU(f) ∧ IsFull(f, cycle(r) − 1) then
            return false
        endif
        if IsVTLPPipelinedFU(f) ∧ Collision(f, cycle(t), cycle(r)) then
            return false
        endif
        if IsAnOperationEntered(f, cycle(r)) then
            MarkEnteredOperation(f, cycle(r))
        endif
        if IsAnOperationLeaving(f, cycle(r)) then
            if IsOperationLeavingMarked(f, cycle(r)) then
                return false
            endif
            count = count − 1
        elseif count > 0 then
            continue
        elseif cycle(r) < FirstCycle(r) then
            continue
        elseif TransportResourcesAvailable(r, cycle(r)) then
            return true
        endif
    endfor
endproc
```

Figure 3.11: Scheduling a result move

3.2.5 TTA Specific Optimizations

All TTA specific optimizations except operand swapping are performed during scheduling. There are many heuristics possible that stimulate these optimizations. The problem is how to combine these heuristics, which may be contradictory, with the critical path heuristic of list scheduling. We have made several attempts to add TTA specific heuristics to the critical path heuristic but without any significant success. For example, we experimented with a heuristic to improve dead result move elimination by increasing the priority of usages of a value that is used once or twice in order to try to schedule them in the same cycle as the result move that defined the value. However the effect was not significant. Therefore, we just detect and apply TTA specific optimizations without stimulating them.

Bypassing

Bypassing is required for zero delay between flow dependent operations. Before a move is scheduled it is checked whether it is flow dependent on a move scheduled in the same cycle. If this is the case the source field of the move being scheduled is changed into the source field of the move it depends on. Notice that a move cannot be flow dependent on more than one move per basic block. When it turns out later that the move could not be scheduled, its source field is restored.

Dead result move elimination

If a move is bypassed, the move that it is flow dependent on (usually a result move) can become dead, and if so, it can be eliminated. This check should be made before scheduling the move that *killed* the move so that the freed transport resources can be used by the move being scheduled. A move that defines a GPR is dead if (1) all moves that are flow dependent on it are scheduled in the same cycle and (2) the GPR is not used outside the basic block or in a future loop iteration if move belongs to a loop body. The latter condition is checked by means of loop-carried DU chains provided by the data flow analysis phase (see section 3.1.7).

When a move is being killed, its outgoing false dependences are killed as well. These dependences exist if the register allocator was not able to make a false dependence free allocation.

Operand sharing

Operand sharing is implemented by checking whether the value that is moved by an operand move to an operand register of an FU is already present. An

```
proc IsSocketFree(s, r, c)
beginproc
    if IsInputSocket(s) then
        return share_count(s, c) = 0
    else
        return share_count(s, c) = 0 ∨ register(s, c) = r
    endif
endproc

proc ClaimSocket(s, r, c)
beginproc
    share_count(s, c) = share_count(s, c) + 1
    register(s, c) = r
endproc

proc ReleaseSocket(s, c)
beginproc
    share_count(s, c) = share_count(s, c) − 1
endproc
```

Figure 3.12: Functions for socket state administration

operand move $s_1 \rightarrow d_1$ belonging to a trigger move scheduled in cycle c_1 is dead due to operand sharing if there is another operand move $s_2 \rightarrow d_2$ belonging to the same basic block scheduled in cycle c_2, such that (1) $s_1 = s_2$, (2) $d_1 = d_2$, (3) $c_1 > c_2$, (4) and s_1 and d_1 are not redefined by moves scheduled between c_2 and c_1. Since the operand register will be claimed between c_2 and c_1 and cannot be used by other operations (unless operand sharing is applied again), the distance between c_2 and c_1 should be restricted. Therefore, the scheduler uses as heuristic that the number of free move buses between c_2 and c_1 should be less than a certain number, called the *maximum operand sharing distance*. The idea is that when the instructions between c_2 and c_1 are reasonably filled the chance that another operation can be scheduled within this range becomes smaller and therefore the distance can become larger.

Socket sharing

Socket sharing is implemented by a counter $share\_count(s, c)$ and a register identifier $register(s, c)$ for each socket s and cycle c. The counter contains the number of moves that share the socket; the register identifier contains the register that is accessed through the socket. A socket s is free in cycle c if $share\_count(s, c)$ is zero; an output socket is free as well if the register that is read through s is equal to $register(s, c)$; see figure 3.12. $share\_count(s, c)$ is incremented when the socket is claimed, and is decremented when it released.

Operand swapping

Operand swapping is performed after the DDG has been built and before the actual scheduling begins. Operand swapping exchanges the sources of operand and trigger moves of communicative operations if it is expected that the operand provided via the operand move will be later available. The sources of operand and trigger moves are exchanged if the trigger move depends on another move within the basic block currently being scheduled and the operand move not. Operand swapping is not performed if the source of the operand move is the stack pointer. This exception is made because the stack pointer tends to be a good candidate for operand sharing.

Operand swapping could be more effective if it would be performed during scheduling since at that moment the cycles where the two operands are available are exactly known. However, experiments have shown us that the improvement is not worth the increased scheduling complexity. This is mainly due to the fact that most communicative operations have an immediate and a non-immediate operand (60–90% of all communicative operations; most of them are integer additions and equal compares) for which the operand swapping decision is trivial.

References [4, 50] also describe operand swapping of commutative operations in order to improve performance. They do this to reduce the performance loss due to an incomplete bypassing network, i.e., not all bypass paths are present. Similar to our approach, [50] performs operand swapping prior to scheduling. The other reference, [4], does not consider ILP.

Extended Basic Block Scheduling

4

It is well known that the size of a typical basic block is limited; 4.57 operations for the benchmarks and architecture used in chapter 6. Furthermore, these operations tend to depend on each other. The amount of ILP that can be exploited by a basic block scheduler is therefore limited. Extended basic block scheduling, or global scheduling, exploits ILP that exists between operations of different basic blocks. This means that operations have to be moved across basic block boundaries.

The main difference between extended basic block schedulers is their scheduling scope. In section 4.1 we will describe scheduling scopes known from the literature. Section 4.2 describes possible code motions between basic blocks and their consequences. The developed extended block scheduler will be described in the next two sections. Section 4.3 describes how the scheduling algorithm works for OTAs, and section 4.4 discusses the TTA specific issues of the scheduling algorithm. Section 4.5 concludes this chapter with a discussion.

4.1 Scheduling Scopes

Extended basic block schedulers partition a CFG of a procedure into acyclic sub-CFGs and schedule each sub-CFG individually. An important characteristic of an extended basic block scheduler is the shape of the sub-CFGs, or scheduling scopes, they operate on. Five extended basic block scheduling scopes are known from the literature: *traces*, *superblocks*, *hyperblocks*, *decision trees*, and *regions*. All these scheduling scopes are acyclic and have a single entry basic block from which all other basic blocks are reachable. They differ in the number of execution paths through the scheduling scope, whether they al-

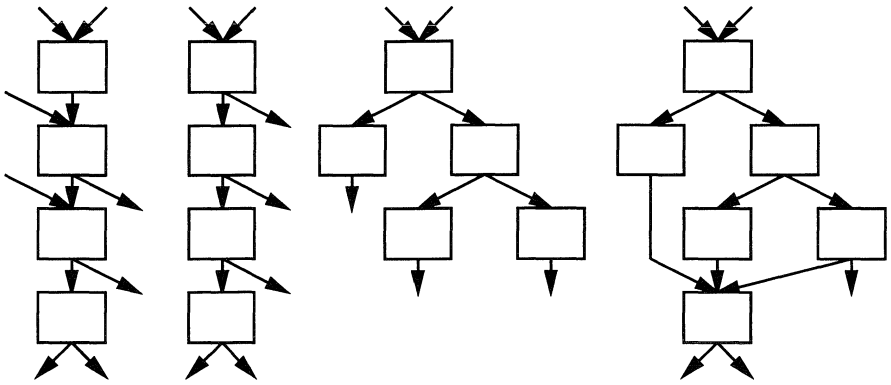


Figure 4.1: Scheduling scopes for extended basic block scheduling

low side-entries, and whether they allow join points.

Trace scheduling was the first extended basic block scheduling technique [79]. A *trace* is a likely acyclic execution path of basic blocks with possible side-entries from other traces; see figure 4.1a. Trace scheduling consists of three actions: (1) trace selection, (2) scheduling, and (3) bookkeeping. Trace selection starts with the basic block with the highest execution count that has not been scheduled yet. From this *seed* basic block the trace is extended forwards/backwards along the outgoing/incoming edges with the highest execution counts. A trace is scheduled as a single basic block with constraints that prevent reordering of branches and incorrect results due to speculative execution. After scheduling *bookkeeping* is performed to correct code motions past fork and join points. These three actions are repeated until all basic blocks are scheduled.

The difficulty of trace scheduling is the bookkeeping due to the upward code motion past join points and downward code motion past fork points. This has led to the development of superblock scheduling [117]. A *superblock* is a trace without join points; what remains is a linear execution path with only outgoing edges; see figure 4.1b. Superblock scheduling starts with trace selection followed by *tail duplication* which transforms traces into superblocks. Tail duplication duplicates basic blocks after a join point and redirects the side-entry to the duplicates. By preventing code motion down fork points and the absence of join points bookkeeping is no longer necessary.

Another attempt to simplify the scheduling task is *percolation scheduling* [155]. Instead of scheduling operations directly, percolation scheduling schedules operations by means of semantics-preserving transformations that move operations between adjacent instructions. Repeatedly application of these transformations leads to scheduled code. Drawbacks of this approach are long com-

pilation times and the possibility of code explosion and overspeculation. Furthermore, it is not very clear how heuristics should drive the transformations.

The major criticism of trace and superblock scheduling is that they only exploit parallelism within one execution path. It only works when the *completion ratio* is close to 100%. The completion ratio is the probability that all basic blocks of a trace/superblock are executed once it is entered. A high completion ratio requires biased branches and accurate static branch prediction. The other three scheduling scopes exploit parallelism along multiple execution paths.

A *hyperblock* corresponds to a single-entry CFG with possibly multiple execution paths that has been if-converted to a superblock [143]. In order to if-convert it the single-entry CFG should not contain procedure calls and it should be 'wise' to do it (see section 2.3.5). After if-conversion the hyperblock is scheduled similar to a superblock. *Promotion* is applied to remove some data dependences between defines of boolean registers and guarded operations that use those booleans. This makes it possible to move up operations past compares on which they were control dependent (comparable to speculative execution). After scheduling, *reverse if-conversion* [201] can be applied if the target machine does not support guarded execution. The code duplication for reverse if-conversion is exponential in the number of booleans that are simultaneously live.

Decision tree scheduling operates on tree shaped scheduling scopes without join points [111]. Similar to superblock scheduling, the absence of join points and code motion down past fork point makes bookkeeping unnecessary. Since each basic block with multiple predecessors becomes a root of a decision tree, decision trees tend to be very small in practice. Therefore, a technique similar to tail duplication is used to duplicate basic blocks with multiple predecessors, one copy for each predecessor. This results in larger decision trees and therefore more parallelism. Decision tree scheduling and superblock scheduling have in common that all code duplication is performed before scheduling and that no code duplication is required during scheduling.

Regions[1] correspond to loop bodies of natural loops [29, 141, 152]. Similar to natural loops, regions can be nested within each other, have one entry basic block, and have no side-entries. Unlike the other scheduling scopes, regions are not selected based on profiling data but on the structure of the code. This means that no profiling data is required for region selection although profiling data is useful for selecting between code motions. A region is the most general scheduling scope; each superblock, hyperblock, and decision tree can be scheduled by a region based scheduler but not the other way around. A trace could be scheduled by a region based scheduler if side-entries would be allowed. Regions have no side-entries simply because of the fact that natural

[1] Extended basic block scheduling with region scheduling scope should not be confused with the work described in [96]. This paper describes transformations to distribute parallelism by moving operations from regions with excessive parallelism to regions with insufficient parallelism.

	Trace	Sup. blk.	Hyp. blk	Dec. tree	Region
Multiple execution paths	No	No	Yes	Yes	Yes
Side-entries are allowed	Yes	No	No	No	No
Join points are allowed	Yes	No	Yes	No	Yes
Code motion down past joins	Yes	No	No	No	No
Must be if-convertible	No	No	Yes	No	No
Code dupl. before scheduling	No	Yes	No	Yes	No

Table 4.1: A comparison of scheduling scopes

loops have a single entry basic block. Due to this generality, regions should potentially give the best performance. This is the major reason that we have chosen a region scheduling scope for our extended basic block scheduler.

Table 4.1 summarizes the most important features of the five scheduling scopes.

Figure 4.2 shows how a CFG is partitioned into scheduling scopes. Assume that the most frequently executed path through the two hammocks is $A \rightarrow B \rightarrow D \rightarrow E \rightarrow G$. Trace scheduling partitions the CFG into three scheduling scopes. The first one corresponds to the most likely path; the other two are the two remaining basic blocks. Superblock scheduling selects the same traces and performs tail duplication to convert them into superblocks. The duplicated tail is $D \rightarrow E \rightarrow G$. Decision tree scheduling duplicates the lower hammock in order to enlarge its scheduling scope. Notice that there can be multiple copies of the same operation in a decision tree, e.g., operations from basic block D. It is therefore possible that, without some form of unification, multiple copies of the same operation are placed on an execution path. Region and hyperblock scheduling are able to schedule this example CFG without partitioning. For hyperblock scheduling the CFG needs to be if-convertible and it needs to be wise to do this. This means that resource and dependence constraints due to operations in C and F should not dominate the resource and dependence constraints due to operations on the most likely path.

4.2 Inter Basic Block Code Motion

Code motion between basic blocks can be upward or downward. Code motion from basic block A to B is upward if A is reachable from B (there is a path in the CFG of the scheduling scope from B to A) and is downward if B is reachable from A. Code motion between basic blocks that are not reachable from each other makes no sense.

Most schedulers perform only upward scheduling. The reason for this is shown in figure 4.3. Possible downward code motions are: operation a to basic block B, a to C, a to D, c to D, and d to D. The first three code motions are

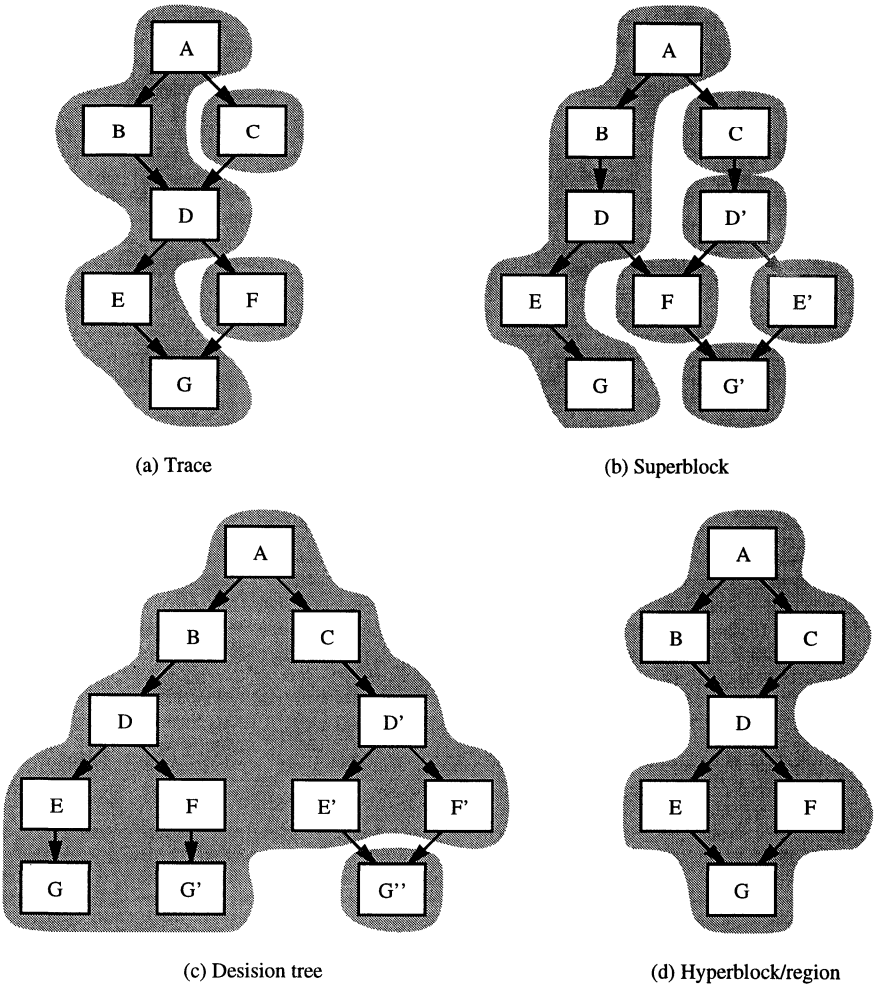


Figure 4.2: Partitioning a CFG into scheduling scopes

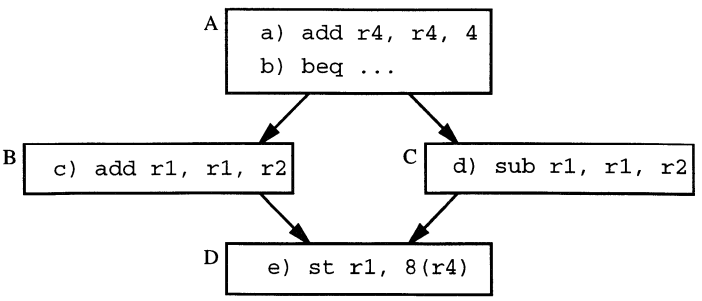


Figure 4.3: A CFG to illustrate possible code motions

often not possible because branches tend to be dependent on the operations in the same basic block. The last two code motions are not possible because they produce incorrect results. This is typical for moving operations down past join points. The register that is defined is usually live on the other incoming control flow edges and will be overwritten by the downward moved operation.

Upward code motion is less problematic. Possible upward code motions are: c to A, d to A, e to B, e to C, and e to A. The first two are speculative; the operations are executed before the branch on which they are control dependent is resolved. The next two code motions require duplication. Moving an operation from D to B requires a duplicate in C and vise versa. The last possible code motion, e to A, is not speculative and does not require duplication. However, since e depends on c and d, it requires that these two operations are also moved to A.

Speculative execution is gambling; it is profitable if control flows to the predicted direction and useless if control flows to another direction. Static branch prediction should be used to select between code motions with a different degree of speculation, e.g., code motion from a basic block that has a probability of 90% to be executed is more likely to be profitable than speculating from a 10% probability basic block.

Speculative execution is valid if no state is changed that should not change in case the branch is mispredicted[2]. This means that mispredicted speculatively executed operations should not (1) produce exceptions, (2) overwrite live registers, and (3) overwrite live memory locations.

The first requirement is realized by providing non-trapping versions of operations that may cause an exception and use these operations for speculative execution. The consequence is that exceptions produced by correctly speculated speculatively executed operations will not be signaled anymore. This is often acceptable for application specific processors, where applications are well debugged and where exceptions should not occur, but it is unacceptable in many other situations. For these situations there are techniques that postpone exceptions produced by speculatively executed operations until the branch on which they are control dependent is resolved. Examples are: shadow register files [16, 176], exception tag bits [71, 78, 144], and speculative tagging [152]. There are also software techniques available that prove that operations will not cause exceptions [30, 141].

The second requirement means that the register that is defined by the speculatively executed operation should not overwrite an off-live register. A register is *off-live* if it is live on a mispredicted path. If the defined register is off-live the scheduler can omit the code motion or it can rename the defined register and generate a copy operation that commits the result of the speculative oper-

[2] An operation may be moved past several branches on which it is control dependent. For simplicity, we will discuss speculative execution for moving an operation past a single branch on which it is control dependent.

ation [71]. Shadow register files [16, 176] can be used to support the last option.

The third requirement is usually realized by forbidding speculative execution of store operations. Live-variable analysis of memory locations is often inexact and hardware support for speculative stores is complicated to implement. Speculative execution of other operations with side-effects that cannot be undone is also forbidden. Examples are procedure calls and I/O operations.

Guarded execution can facilitate speculative execution. Operations can be speculated safely if they are guarded with a guard expression that evaluates to false in case of a misprediction [111]. If an operation is guarded, it becomes flow dependent on the compare operations that define the booleans of the guard expression.

4.3 Region Scheduling for OTAs

Our extended basic block scheduler is inspired by the global instruction scheduler of David Bernstein *et al.* [28, 29] and has extensions for multi-way branching, predicated execution, usage of profiling data, and TTA specific scheduling constraints and optimizations.

The scheduler schedules all basic blocks of a region in a topological order, i.e., a basic block is scheduled after all its predecessors are scheduled. Scheduling a basic block b consists of two steps: (1) basic block scheduling of b, and (2) importing operations from basic blocks that are reachable from b into b; see figure 4.4. The first step is described in chapter 3.

4.3.1 Importing Operations

An operation is ready for importing to the currently being scheduled basic block b if it does not depend on an operation located in a basic block reachable from b. This corresponds with the M-ready concept of [28]. The scheduler repeatedly tries to import ready operations from reachable basic blocks until all ready operations have been tried. The order in which the operations are tried is determined by the priority function described in the next subsection.

To import an operation from basic block b' to the currently being scheduled basic block b the scheduler first checks whether code duplication is necessary. It does this by means of the dominate relation. Code duplication is needed if b does not dominate b'. To determine where duplicates need to be placed the scheduler computes all intermediate basic blocks between b and b':

$$I(b, b') = \{v \in V \mid b \rightsquigarrow v \wedge v \rightsquigarrow b'\}$$

where $v \rightsquigarrow u$ means that there is a control flow path within the region from

```
proc Schedule(CFG = (V, E))
beginproc
    for each b ∈ V in topological order do
        while not all operations in b have been scheduled do
            v = SelectOperation(b)
            Schedule(b, v)
        endwhile
        is_scheduled(b) = true
        reachable = ReachableBasicBlocks(V, E, b)
        while not all operations in reachable have been tried do
            b' = SelectBasicBlock(reachable)
            v = SelectOperation(b')
            TryToImport(b, b', v)
        endwhile
        RemoveEmptyBasicBlocks(reachable)
    endfor
endproc

proc TryToImport(b, b', v)
beginproc
    dupl_set = DuplicationSet(b, b')
    if ¬IsSCP(dupl_set) then return
    for each b'' ∈ dupl_set do
        if ¬PostDominate(b'', b') then
            CheckSpeculativeExecution(b'', b', v)
            if failure then return
        endif
        if is_scheduled(b'') then
            TryToSchedule(b'', v)
            if failure then return
        else
            AddOperation(b'', v)
        endif
    endfor
endproc
```

Figure 4.4: Extended basic block scheduling

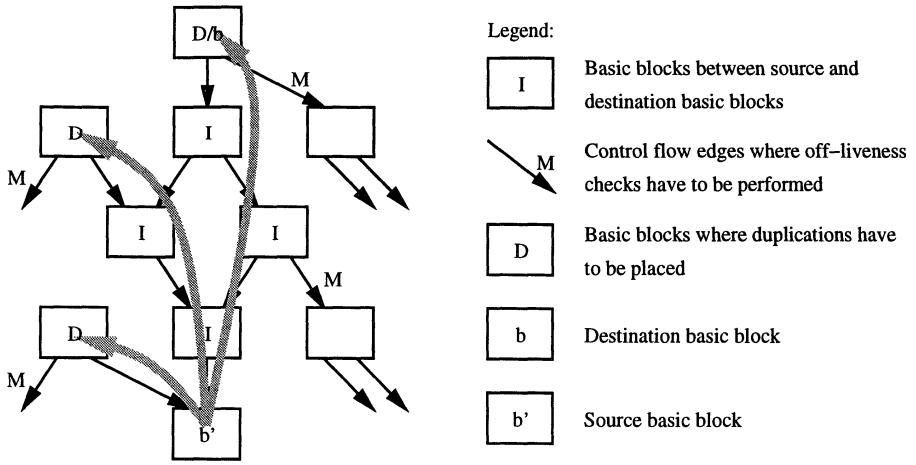


Figure 4.5: Inter basic block code motion from b' to b

v to u. A duplicate is needed in each predecessor basic block of $I(b, b')$ and b' that is not in $I(b, b')$; see figure 4.5.

$$D(b, b') = \{v \in V - I(b, b') \mid \exists u \in I(b, b') \cup \{b'\}, v \rightarrow u\}$$

where $v \rightarrow u$ means that there is a control flow edge between v to u. Note that $D(b, b')$ includes b. We use the single copy on a path rule (SCP) of [28] which means that no paths are allowed between elements of $D(b, b')$.

$$\forall u, v \in D(b, b'), u \neq v \Rightarrow u \not\rightarrow v$$

The SCP rule simplifies scheduling. Importing fails if $D(b, b')$ does not satisfy the SCP rule. After the set of duplication points $D(b, b')$ have been determined a copy of the imported operation has to be placed in each basic block $b'' \in D(b, b')$. If b'' is not scheduled yet the operation is added to the operations of b'', otherwise it has to be scheduled in b''. In the last case it is not permitted to enlarge b'' by adding instructions to it. Importing fails if the operation could not be scheduled between the first and last instruction of b''.

To move an operation from basic block b' to b'' the scheduler has to check whether the code motion is speculative, and if so, whether it is possible to do it correctly. The code motion is speculative if b' does not post-dominate b''. In that case the scheduler examines all control flow edges between basic blocks in $I(b'', b') \cup \{b''\}$ and basic blocks not in $I(b'', b')$:

$$M(b'', b') = \{(v, u) \in E \mid v \in I(b'', b') \cup \{b''\} \wedge u \notin I(b'', b')\}$$

The speculatively executed operation needs to be guarded if it defines a register that is live on an control flow edge in $M(b'', b')$ or if it is a store operation. The guard expression is computed by combining the guard expressions corresponding to the control flow edges in $M(b'', b')$ for which the operation needs to be guarded. Importing fails if the compare operations that define the booleans of the guard expressions have not been scheduled yet, or if the guard expression is not supported by the target machine. When an operation is guarded the operation becomes flow dependent on the compare operations that define the booleans of the guard expression.

Notice that code motion between a basic block b' and $b'' \in D(b, b')$ can be speculative even if code motion between b' and b is not speculative. This may be a deterioration if b'' is executed more frequently than b. To prevent these problems, the basic blocks are scheduled in a topological order such that basic blocks with a higher execution count are prioritized and it is not allowed to enlarge basic blocks once they have been scheduled.

After an operation has been imported successfully, live-variable information has to be updated. This is done by recomputing *use* and *def* for all basic blocks that have been changed and recomputing *in* and *out* for all basic blocks in $I(b, b') \cup D(b, b') \cup \{b'\}$ in a bottom-up fashion.

4.3.2 The Operation Selection Heuristic

The order in which ready operations from reachable basic blocks are tried determines the performance of extended basic block scheduling. The priority to import operation v of basic block b' in basic block b should depend on the probability that control flow will reach b' after b and the criticality of v. We use the following priority function:

$$priority(b, b', v) = probability(b, b') \times (1 - \frac{slack(v)}{L_{max}(b')})$$

where $probability(b, b')$ is the probability that b' will be executed after b, $L_{max}(b')$ the critical path length of b', and $slack(v)$ the slack of operation v whereby only intra basic block dependences are used.

The critical path component is normalized by means of L_{max} to prevent that operations from small basic blocks are prioritized over operations from larger basic blocks. This priority function does not take inter basic block dependences into account. We have experimented with various other functions that do this, such as the priority function described in [152], but without success. Furthermore, we have experimented with a priority function that prioritized code motions that do not require code duplication over code motions that do. This was done by adding a negative constant to the priority function if b does not dominate b'. The effect on the code size was negligible.

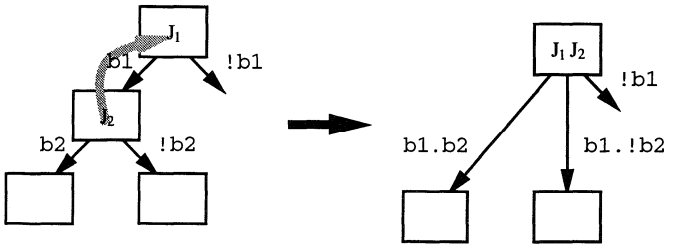


Figure 4.6: Importing jump operations

4.3.3 Importing a Compare Operation

Initially all compare operations define the same boolean register (b0). When a compare operation is imported from b' to b, a check is made to verify whether the boolean that it defines is live in a basic block of $I(b, b') \cup \{b\}$. If so, the destination boolean register is renamed to a free boolean register. Importing fails if this is not possible. This on-the-fly renaming is similar to the register renaming technique of [158]. The current version of our back-end does not move compares above calls. This requires operations for saving and restoring boolean registers at calls and returns.

4.3.4 Importing a Jump Operation

Inter basic block code motion of jumps past several basic blocks is complicated, may result in a large amount of code duplication, and it is often questionable whether it is useful [163, 164]. Therefore, our scheduler performs only code motion of jumps from basic block b' to b if b' is an immediate successor of b. Figure 4.6 illustrates how a jump is imported. The control flow edge between b and b' is replaced by the the outgoing control flow edges of b'. This results in multi-way branching if both b and b' have multiple successors. The jump operation of b' named J_2 will be scheduled in the same cycle as the jump operation of b named J_1. If b' contains other operations besides J_2 they are moved downward to the successor basic blocks of b'[3]. This requires that these successor basic blocks have no other predecessors than b', i.e., they are not join points.

The guard expressions associated to the control flow edges that describe under which condition control flow takes place are updated as shown in figure 4.6. These expressions are used by the scheduler to compute guard expressions for speculatively executed operations.

Since the current version of our scheduler does not support guard expressions larger than two booleans, multi-way branching is limited to three-way branch-

[3]This is the only situation where our scheduler moves operations downward.

ing. The guard expressions associated with the outgoing control flow edges are: b_x, $!b_x \,.\, b_y$, and $!b_x \,.\, !b_y$. Four-way branching with guard expressions $!b_x \,.\, !b_y$, $!b_x \,.\, b_y$, $b_x \,.\, !b_y$, and $b_x \,.\, b_y$ is possible but requires complex CFG transformations and requires usually more code duplication.

4.4 TTA Specific Issues

Importing an operation on a TTA is done similar to scheduling an operation as described in chapter 3. However, some extensions have to be made in order to schedule operations over basic block boundaries and bypassing is slightly complicated due to guarded execution.

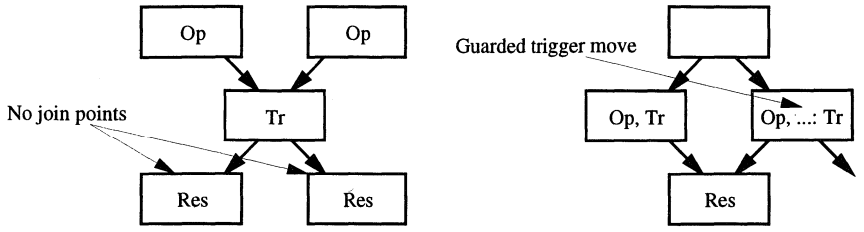
Scheduling operations over basic block boundaries

As described in chapter 3, scheduling a data operation starts with scheduling its trigger move followed by scheduling its operand and result moves. During scheduling an operand move it may happen that the scheduler 'hits' the 'top' of basic block b, i.e., scheduling the operand move in the first cycle failed. Similarly, during scheduling of a result move the scheduler may hit the 'bottom' of b, i.e., it was not possible to schedule the result move in the last cycle of b (remember that enlarging b is not allowed while importing). In the first case the scheduler will try to schedule the operand move in the predecessor basic blocks of b; and in the second case the scheduler will try to schedule the result move in the successors of b. The result is that the operand, trigger, and result moves of an operation are scheduled in different basic blocks, although the distance between an operand and a trigger move and between a trigger and a result move is limited to one basic block[4]; see figure 4.7a.

Pushing a result move downward if it hits the bottom is only needed in the directions it came from. Thus, if an operation is imported from b' to b its result move does not have to be scheduled in a successor s of b if b' is not reachable from s. However, if the operation is scheduled on a hybrid pipelined FU and its trigger move cannot be guarded with an expression that evaluates to false if control flows from b to s (because the booleans of the guard expressions are not ready at that moment) a junk move has to be scheduled in s. This usually occurs if an operation has a latency longer than the latency of a jump. Scheduling a junk move is similar to scheduling a result move. The only difference is that a junk move writes data to a non-existing location instead of a register.

Pushing a result move downward is not possible if the successor basic block is a join point. One way to tackle this problem is to guard the result move such that it is only executed if control flows from the basic block where its trigger

[4]This limit has to be removed for long latency operations ($\geq$ 8 cycles) for which this becomes a serious problem.



(a) Scheduling over basic block boundaries (b) Scheduling over join points

Figure 4.7: Scheduling operations over basic block boundaries

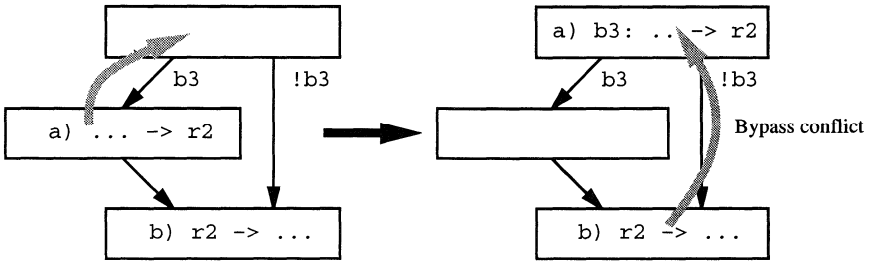


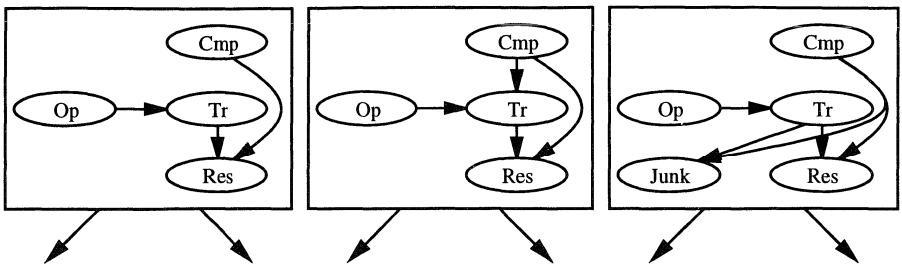
Figure 4.8: A bypass conflict

move is located. This approach has been examined but was rejected because it is complicated and increases the required number of booleans. The latter is because booleans have to stay live until join points instead of fork points, and insertion of boolean assignment operations may be necessary.

Our solution to make scheduling of trigger and result moves over join points possible is to employ an alternative *result-trigger-operand* scheduling scheme, next to the *trigger-operand-result* scheduling scheme described in chapter 3. The result-trigger-operand scheduling scheme schedules the result move first followed by scheduling the trigger and operand moves in the predecessor basic blocks. This scheduling scheme is tried before the trigger-operand-result scheme and results in scheduling operations over join points as shown in figure 4.7b. If trigger and operand moves are scheduled above a fork point, the trigger move needs to be guarded such that it only executes if control flows in the direction of the result move.

Bypassing

As described in section 2.4.3, the bypass logic of an OTA is dynamic and not always predictable at compile-time. This manifests itself during scheduling for TTAs in *bypass conflicts*. Figure 4.8 illustrates a bypass conflict. First, a move



(a) VTLP FUs

(b) Hybrid FUs, guarding trigger and result move

(c) Hybrid FUs, guarding result move

Figure 4.9: Guarding a speculatively executed operation

named a that defines GPR `r2` is speculated. This move needs to be guarded with guard expression `b3` because `r2` is off-live. If another move b is both flow-dependent on a and another move c[5] it cannot be scheduled in the same cycle as a. This is because it needs to be bypassed due to the flow-dependence on a, and it should not be bypassed due to the flow-dependence on c. There are two options when the scheduler signals a bypass conflict, (1) it can delay a by one cycle which will resolve the bypass conflict, or (2) it can duplicate b and guard one copy with `b3` and the other with `!b3`. In general, the last option needs one copy for each move from which it needs to be bypassed, and a copy if it is flow-dependent on a move from which it should not be bypassed. This option was rejected because it is very hard for a scheduler to determine whether it is worthwhile to generate extra copies (which use extra resources) instead of delaying the move that is being scheduled. Furthermore, the second option needs multiple input sockets on FU registers if input socket sharing is not possible due to the used instruction pipelining scheme (see section 2.4.2). In section 6.2.12 we will measure the negative effect of bypass conflicts on the performance.

Guarding

Figure 4.9 shows how speculatively executed operations are guarded. Operations scheduled on VTLP pipelined FUs are guarded by guarding their result move. The operand and trigger moves are allowed to be scheduled before the compare operation. For operations scheduled on hybrid pipelined FU we have two options: (1) guarding the trigger and result moves (figure 4.9b), and (2) guarding only the result move and adding a junk move to remove the operation from the pipeline in case the result move is not executed (figure 4.9c). The

[5]Or `r2` is defined by another move outside the scheduling scope or in a previous iteration if a loop body is being scheduled. The latter check is made by means of loop-carried DU chains described in section 3.1.7.

latter option gives the same scheduling freedom as scheduling the operation on a VTLP pipelined FU at the expense of an extra junk move. Which option is selected is determined during scheduling. If the trigger move can be guarded, the first option is chosen, otherwise the second option. Notice that we do not guard operand moves.

Dead result move elimination

Dead result move elimination and guarded execution is also slightly more complicated for extended basic block scheduling if operations are scheduled on hybrid pipelined FUs. The scheduler has to make sure that every operation that is triggered on a hybrid pipelined FUs is also removed from the FU. Consider a speculatively executed operation a that is scheduled on a hybrid pipelined FU without being guarded, and another flow-dependent speculatively executed operation b that needs to be guarded. Operation b alone cannot kill the result move of a because the result of a has to be read if the guard expression of b evaluates to false. The scheduler should therefore, examine the guard expressions of the involved operations to determine whether the result move of an operation scheduled on a hybrid pipelined FU can be removed safely.

Operand sharing

Operand sharing is also performed over basic block boundaries. If the 'top' of a basic block is 'hit', operand sharing proceeds in the predecessor basic blocks. In predecessor basic blocks the maximum operand sharing distance becomes infinity. The rationale for this is that predecessor basic blocks are already scheduled and it is not likely that many other moves will be scheduled in them. Therefore, operand registers may be occupied for a longer time.

4.5 Discussion

Developing an extended basic block scheduler requires many design decisions, especially if the target architecture is not yet completely specified and can be influenced by the compiler writer. The effects of many of these decisions are usually hard to predict and can only become clear by prototyping. The most debatable design decisions we had to make are the used scheduling scope and the guarding mechanism. Furthermore, designing a scheduler leads to many engineering design decisions such as, what data structures to use, when to compute which information, how long is information valid, and how to update information?

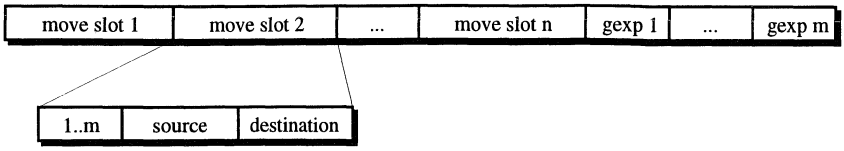


Figure 4.10: An alternative instruction format for denser code

The region scheduling scope was mainly chosen for its generality. There are no experimental results available that compare the performance and implementation complexity of the five mentioned scheduling scopes. After experiencing several problems with scheduling for TTAs, we came to the conclusion that superblock scheduling could be very appealing alternative for TTAs. Superblocks (1) have no join points, which avoids the scheduling problems described in the previous section, and (2) each move is flow-dependent on at most one preceding move in the same superblock which means that bypass conflicts cannot occur. The first advantage comes at the price of tail duplication, i.e., code expansion. The second advantage is a consequence of single execution path parallelization, which is a disadvantage. Hyperblock scheduling is also appealing; it is relatively easy to implement and seems to work very well in practice [143]. However, due to multiple path parallelization, bypass conflicts are re-introduced. Furthermore, we experienced if-conversion as 'risky' (see next chapter). Applying if-conversion too aggressively leads to performance degradation.

The used guarding mechanism is also debatable. We have chosen for a small number of booleans and the following expressions:

1. Simple expressions: $\mathtt{b}_x$, $\mathtt{!b}_x$
2. And expressions: $\mathtt{b}_x\,\mathtt{.}\,\mathtt{b}_y$, $\mathtt{b}_x\,\mathtt{.}\,\mathtt{!b}_y$, $\mathtt{!b}_x\,\mathtt{.}\,\mathtt{b}_y$, $\mathtt{!b}_x\,\mathtt{.}\,\mathtt{!b}_y$
3. Or expressions: $\mathtt{b}_x\mathtt{+}\mathtt{b}_y$, $\mathtt{b}_x\mathtt{+!b}_y$, $\mathtt{!b}_x\mathtt{+b}_y$, $\mathtt{!b}_x\mathtt{+!b}_y$

Where and/or expressions can be disabled via a switch in the machine description file. Other researches have chosen for a large set of booleans without expressions (guarding is only possible with $\mathtt{b}_x$) [123, 143, 160, 170]. The advantage of expressions is that operations can be moved past several branches for which they need to be guarded. To do this without expressions, compare results have to be combined by means of guarding compare operations (such as done by the RK-algorithm [160]), or by means of hardware support (such as provided by the HPL PlayDoy architecture [123]). There are two disadvantages to guard expressions, (1) evaluating the guard expression may affect the cycle time, and (2) supporting all and/or expressions of two out of n booleans requires $\lceil 3 + 2 \log n \rceil$ bits per move slot. The latter problem can be alleviated

by limiting the number of guard expressions per instruction as shown in figure 4.10. Each instruction contains besides n move slots also m guard expression slots; and each move slot contains a guard expression index field that refers to one of the m guard expression slots.

5 Software Pipelining

Backward control flow edges are barriers for an extended basic block scheduler; no operations are moved over backward control flow edges. This limits parallelism and thus performance, especially if much of the execution time of an application is spent in small loops with high trip counts. The easiest way to improve this situation is to apply loop unrolling, e.g., the Multiflow Trace compiler unrolls loops up to 96 times [140]. Unrolling a loop N times roughly reduces the number of 'executed' backward control edges of that loop by N[1] at the expense of N times code expansion. *Software pipelining* refers to scheduling techniques that move operations over backward control flow edges. It achieves potentially the same performance as infinite loop unrolling with a modest code expansion. Figure 5.1 illustrates software pipelining. The loop shown in figure 5.1a is translated into the OTA code shown in figure 5.1b. The loop control code has been omitted for simplicity. Figure 5.1c shows a software pipeline for this code. A software pipeline consists of a *kernel*, a *prologue*, and an *epilogue*. The kernel is the loop body of the software pipeline. During the execution of the kernel, called the *steady state*, operations from four adjacent iterations of the original loop are executed. The prologue fills the pipeline with iterations, and the epilogue drains the pipeline. Similar to hardware pipelines, a software pipeline is characterized by its throughput and latency. The throughput of the example software pipeline is one iteration per cycle and its latency is four cycles. A high throughput is important for loops with high trip counts, and a low latency is important for loops with low trip counts. One usually optimizes for throughput.

This chapter discusses our software pipelining scheduler. Section 5.1 describes modulo scheduling, the class of software pipelining algorithms to which the

[1] Assuming high trip counts.

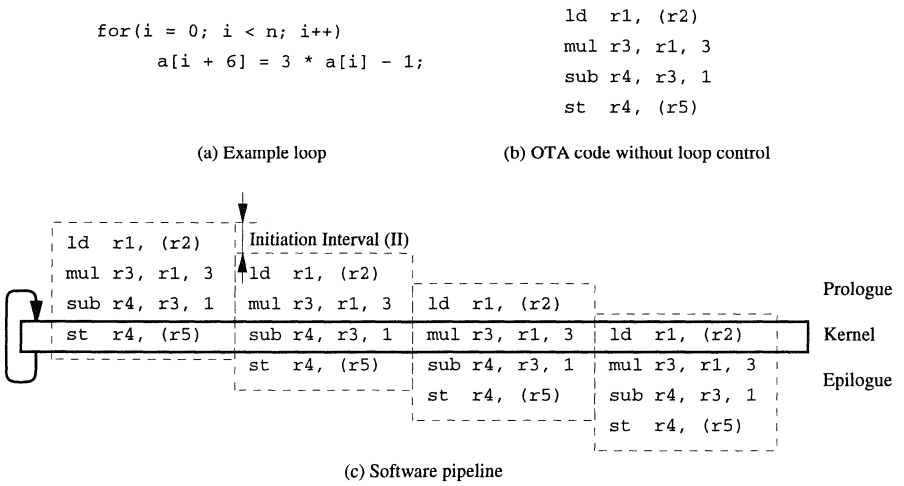


Figure 5.1: Software pipelining a loop

used software pipelining algorithm belongs. Section 5.2 describes what is needed to prepare loops for software pipelining. Section 5.3 describes the TTA specific issues of our software pipelining scheduler.

5.1 Modulo Scheduling

Allan *et al.* [8] divide software pipelining algorithms into three categories: modulo scheduling, kernel recognition techniques, and enhanced pipeline scheduling.

1. **Modulo scheduling** [172] determines a schedule of operations and an *initiation interval II* such that the schedule can be initiated every *II* cycles without resource and dependence conflicts. The goal is usually to minimize *II*, which corresponds to the number of instructions of the kernel of the software pipeline. Algorithms belonging to this category are Lam's software pipelining algorithm [131, 132], slack-scheduling [114], and iterative modulo scheduling [169].

2. **Kernel recognition techniques** unroll the loop a number of times, schedule the iterations, and try to identify a repeating pattern in the schedule which becomes the kernel of the software pipeline. Algorithms belonging to this category are perfect pipelining [6, 7], URPR [184, 185], circular scheduling [118], and petri net pipelining [9].

3. **Enhanced pipeline scheduling** [70, 72] creates software pipelines by alternating filling an instruction with operations and wrapping the opera-

tions of the filled instruction around the backward edge of the loop. Repeated application of these two steps leads to a software pipeline. Variations on the original enhanced pipeline scheduling algorithm are described in [150, 163].

The software pipelining algorithm used by our scheduler belongs to the modulo scheduling category and is based on Rau's iterative modulo scheduling [169]. Motivations for choosing for modulo scheduling are: (1) it has been worked out very well and has been implemented in several production compilers [66, 167], (2) previous positive experience with modulo scheduling [104, 109], and (3) its performance in comparison with other software pipelining algorithms [121].

The weak point of modulo scheduling is its inability to generate software pipelines for multi basic block loops with multiple initiation intervals. Iterations are initiated in a constant rate independent of the paths taken through the loop body. This means that multi basic block loops with long infrequently taken paths through their loop body cannot be software pipelined efficiently.

As alternative for modulo scheduling we considered enhanced pipeline scheduling. Enhanced pipeline scheduling has been rejected because it requires instruction based scheduling instead of operation based scheduling which is preferable for TTA code scheduling (see section 3.2.2)[2].

5.1.1 Cyclic Data Dependency Graphs

The schedule produced by a software pipelining scheduler contains operations of one loop iteration. This schedule will be initiated every II cycles. Data dependences between operations within the same iteration, called *intra-iteration* dependences, and between operations of different iterations, called *inter-iteration dependences*, have to be satisfied. Adding inter-iteration dependences to a DDG may lead to cyclic DDGs. Figure 5.2 shows the DDG corresponding to the example code of figure 5.1b. Each edge of the DDG is labeled with two values, its delay and its iteration distance. An edge from node u to node v with delay $delay(u, v)$ and iteration distance $distance(u, v)$ describes that the operation corresponding to v in iteration i should be executed at least $delay(u, v)$ cycles after the operation corresponding to u in iteration $i - distance(u, v)$. For example, the edge labeled $(1, 6)$ indicates that the load of `a[i+6]` should be executed at least one cycle after the store to `a[i]` of 6 iterations before.

[2] Allan introduces in [8] the notion of persistent resource requirements, which means that reservation tables of operations have more than one row. Furthermore, Allan remarks that enhanced pipeline scheduling is incompatible with architectures with operations with persistent resource requirements. TTAs belong to these architectures since trigger and result moves of an operation require resources in different cycles which implies multi row reservation tables.

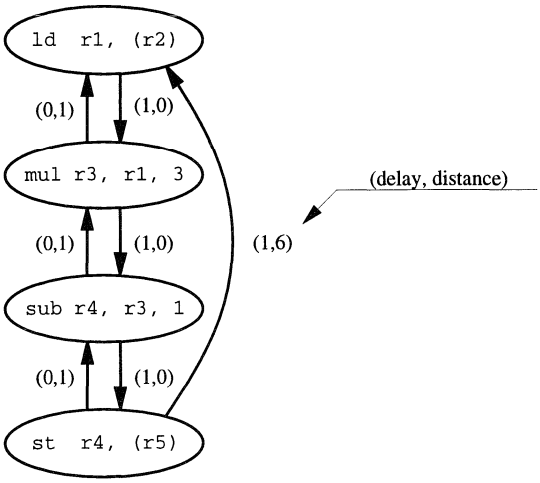


Figure 5.2: The DDG corresponding to the code shown in figure 5.1.

5.1.2 Modulo Scheduling Constraints

Similar to basic block scheduling, modulo scheduling has to consider two scheduling constraints: precedence constraints and resource constraints. A dependence edge from u to v states that v should be executed at least $delay(u, v)$ cycles after u in the $distance(u, v)^{th}$ previous iteration which is executed $II \cdot distance(u, v)$ cycles before u of the same iteration:

$$cycle(v) \geq cycle(u) + delay(u, v) - II \cdot distance(u, v) \tag{5.1}$$

This formula shows that a lower delay and a higher iteration distance leads to more scheduling freedom.

If iterations are initiated every II cycles, a resource that is used in cycle c is also used in cycle $c + k \cdot II$, where $k \in \mathbb{N}$. Therefore, resource conflicts should not only be checked between operations scheduled within the same cycle, but between all operations scheduled within the same cycle *modulo II*; hence the name modulo scheduling.

5.1.3 Modulo Scheduling

The software pipelining algorithm has to determine a schedule and an initiation interval II. The scheduling constraints depend on II and II depends on the schedule, i.e., a chicken-and-egg problem. To break this cycle, modulo scheduling computes a lower bound on II called the minimum initiation interval MII. Next, modulo scheduling tries to schedule the loop with this initiation

interval. If this fails, *II* is incremented by one cycle and another scheduling attempt is made. This is repeated until a schedule is found.

Computing *MII*

Modulo scheduling uses resource and precedence constraints to compute *MII*. The lower bound on *II* due to resource constraints *ResMII* is computed by computing the number of cycles a resource r is used and dividing it by the number of available instances of r. The most heavily used resource determines *ResMII*.

$$ResMII = \max_{r \in resources} \left\lceil \frac{used(r)}{available(r)} \right\rceil \tag{5.2}$$

Computation of *ResMII* becomes slightly more complex if operations can be performed by multiple FU types, e.g,. if an add operation can be performed by an ALU FU and a more specialized adder FU. For these situations, the scheduler makes a temporary assignment of FUs to operations, in order to compute $used(r)$ for each FU. The order in which FUs are assigned to operations is such that operations with more alternatives are handled after operations with fewer alternatives.

Recurrences, which show up as cycles in the DDG, impose another lower bound on *II* called *RecMII*. A cycle c through node v leads to the following precedence constraint:

$$cycle(v) \geq cycle(v) + \sum_{e \in c} \{delay(e) - II \cdot distance(e)\} \tag{5.3}$$

Therefore:

$$RecMII = \min\{II \in \mathbb{N} \mid \forall_{c \in cycles}, 0 \geq \sum_{e \in c} \{delay(e) - II \cdot distance(e)\}\} \tag{5.4}$$

or:

$$RecMII = \max_{c \in cycles} \left\lceil \frac{\sum_{e \in c} delay(e)}{\sum_{e \in c} distance(e)} \right\rceil \tag{5.5}$$

Straightforward computation of *RecMII* by enumerating all cycles in DDG is not tractable since the number of cycles can be exponential in the number of nodes of the DDG. Lam solved this by means of a symbolic version of Floyd-Warshall's all-pairs shortest path algorithm [54, 132]. Huff calculates *RecMII* by transforming the problem to a minimum cost-to-time ratio cycle problem

[114, 136]. We calculate *RecMII* by means of the Bellman-Ford algorithm for computing the longest path in a graph [26, 83]. We define the length of an edge e of the DDG as $delay(e) - II \cdot distance(e)$. The Bellman-Ford algorithm is able to report the existence of cycles with a positive length which lead to infinitely long paths. In that case the used II for calculating the edge length is smaller than *RecMII*. We use binary search to find the smallest II such that there are no cycles with a positive length. This II value corresponds to *RecMII*. All three mentioned methods to calculate *RecMII*, Lam's, Huff's, and our method, have in common that their time complexity equals $O(n^3)$, where n is the number of nodes in the DDG. This becomes a problem for large loops. One way to reduce the computation time of *RecMII* is to partition the DDG in strongly connected components (SCCs) and to compute *RecMII* for each SCC individually. *RecMII* of the loop is the maximum of the *RecMII*s of the SCCs. Combining *ResMII* and *RecMII* gives *MII*:

$$MII = \max\{ResMII, RecMII\} \tag{5.6}$$

Both equations 5.2 and 5.5 contain ceiling functions to make *ResMII* and *RecMII*, respectively, integral. If the performance degradation due rounding up is relatively high, the loop could be unrolled a few times [43, 135]. For example, if the loop contains 3 FP operations, 2 FP FUs are available, and the FP FUs are the critical resources, *ResMII* becomes $\lceil 3/2 \rceil = 2$ cycles. By unrolling this loop two times *ResMII* becomes $\lceil 6/2 \rceil = 3$ cycles (for two iterations). This optimization is not performed by the current version of the compiler. The reason is that the compiler should either ensure that the trip count of the loop is a multiple of the unrolling factor, or it should generate loops with multiple exit control flow edges. The current version of our compiler has no information about trip counts and is not capable to software pipeline loops with multiple exits (see section 5.2.1).

Scheduling the loop

Modulo scheduling a loop for an initiation interval II is similar to operation based list scheduling of a basic block. Operations are repeatedly selected and scheduled until all operations have been scheduled. The differences are:

1. In order to implement modulo scheduling constraints, reservation tables for recording the state of resources need to be accessed modulo II, i.e., instead of accessing element (c, r) for the state of resource r in cycle c, the scheduler should access element $(c \bmod II, r)$.

2. Due to cycles in the DDG, it is no longer possible to schedule the operations of a loop in a topological order as done by list scheduling. This means that when an operation is scheduled it is possible that some of its

predecessors have not been scheduled yet. The first cycle in which an operation can be scheduled is therefore computed as follows:

$$cycle\_min(v) = \max_{u \in sched\_pred(v)} cycle(u) + delay(u, v) - II \cdot distance(u, v)$$

where $sched\_pred(v)$ is the set of scheduled predecessors of v and $\max(\emptyset) = 0$. To schedule an operation v, the scheduler tries to find a resource conflict free cycle between $cycle\_min(v)$ and $cycle\_min(v) + II - 1$ starting in $cycle\_min(v)$. Searching beyond $cycle\_min(v) + II - 1$ is useless because of the modulo scheduling constraints; if there is a resource conflict in cycle c, then there is also a resource conflict in cycle $c + k \cdot II$. Scheduling fails if no cycle can be found without resource conflicts; the schedule is discarded, II is incremented, and another attempt is made[3].

Another consequence of not being able to schedule operations in a topological order is that when an operation v is scheduled some of its successors in the DDG may have already been scheduled. If a precedence constraint between v and a scheduled successor u of v has been violated, i.e.,

$$cycle(u) < cycle(v) + delay(v, u) - II \cdot distance(v, u)$$

iterative modulo scheduling unschedules u to correct the partial schedule. In order to prevent an infinite loop of scheduling and unscheduling, iterative modulo scheduling uses a *scheduling budget*. Each time an operation is scheduled, the scheduling budget is decremented by one. Scheduling fails when the budget becomes negative. The scheduling budget is initialized with $\alpha \cdot n$, where n is the number of operations and α is a parameter, called the *budget ratio*, with a value between 1.5 and 4.5 that determines how hard iterative modulo scheduling tries to schedule a loop for a given II. A larger α may result in better schedules at the cost of a longer compilation time.

3. The last difference between modulo scheduling and operation based list scheduling is the scheduling priority. Huff [114] modified the slack based priority described in section 3.2.1, whereby the delay of an edge has been replaced by the length of an edge ($delay(e) - II \cdot distance(e)$). Slacks of not yet scheduled operations are updated when operations are scheduled. Rau [169] uses a DDG height based priority. The DDG height of an operation is the length of the longest path from that operation to a stop

[3]The original iterative modulo scheduling algorithm [169] does not give up in this situation. It unschedules operations which use resources that are required by v. This is not implemented in our scheduler since it is not trivial for TTAs to determine which operations should be unscheduled.

pseudo operation that is dependent on all operations in the loop. Priority is given to the operations with largest DDG height. We have experimented with both priority functions. Neither appeared to be clearly better than the other. Therefore we decided to use both. First we use Huff's priority function to find a schedule for a given II. If this fails use Rau's priority function for a second attempt. If this fails again we increment II. By using both priority functions, results are always better than one of them individually. The drawback is a slightly longer compilation time.

Figure 5.3 summarizes the iterative modulo scheduling algorithm. ModuloScheduleIteration is repeatedly invoked to schedule the loop for successive values of II starting with MII until a schedule is found. ModuloScheduleIteration repeatedly selects an operation v based on a priority function, determines the interval in which v should be scheduled, and searches for a resource conflict free cycle within this interval. ModuloScheduleIteration fails if no such cycle can be found. If a resource conflict free cycle for v can be found, a check is made to determine whether precedence constraints between v and scheduled successors of v have been violated. If so, these scheduled successors are unscheduled. These steps are repeated until all nodes have been scheduled or ModuloScheduleIteration runs out of the scheduling budget.

5.2 Preprocessing Loops

Many loops have to be preprocessed to make them suitable for modulo scheduling or to make modulo scheduling more effective. If-conversion transforms multiple basic block loops into single basic block loops. Promotion removes some of the data dependences introduced by if-conversion in order to reduce $RecMII$. Delay lines are inserted to avoid false dependences that limit $RecMII$. Finally, some dependences have to be added to the DDG to ensure correct software pipelining of *while* loops.

5.2.1 If-conversion

Modulo scheduling is not directly capable to software pipeline loops consisting of multiple basic blocks. There are two methods known from the literature to make modulo scheduling applicable to multiple basic block loops: hierarchical reduction and if-conversion.

A loop consisting of an 'if-then-else' construct can be modulo scheduled by means of hierarchical reduction [132, 211] by the following steps illustrated in figure 5.4:

1. List scheduling of basic blocks B and C (the 'then' and 'else' parts) by a basic block scheduler.

```
proc ModuloSchedule(V, E, budget_ratio)
beginproc
    budget = budget_ratio · |V|
    II = MinimumInitiationInterval(V, E)
    while ¬ModuloScheduleIteration(V, E, II, budget, Huff) ∧
          ¬ModuloScheduleIteration(V, E, II, budget, Rau) do
        II = II + 1
    endwhile
endproc

proc ModuloScheduleIteration(V, E, II, budget, priority_function)
beginproc
    sched = ∅
    while sched ≠ V ∧ budget > 0 do
        v = SelectOperation(V − sched, priority_function)
        cycle_min = max{cycle(u)+Length(u, v, II) | (u, v) ∈ E, u ∈ sched}
        cycle_max = cycle_min + II − 1
        cycle(v) = cycle_min
        while ModuloResourceConflicts(v, cycle, sched, II) do
            cycle(v) = cycle(v) + 1
            if cycle(v) > cycle_max then
                return false
            endif
        endwhile
        sched = sched ∪ {v}
        budget = budget − 1
        for each u ∈ succ(v) ∩ sched do
            if cycle(u) < cycle(v) + Length(v, u, II) then
                sched = sched − {u}
            endif
        endfor
    endwhile
    return sched = V
endproc

proc Length(u, v, II)
beginproc
    return delay(u, v) − II · distance(u, v)
endproc
```

Figure 5.3: Iterative modulo scheduling

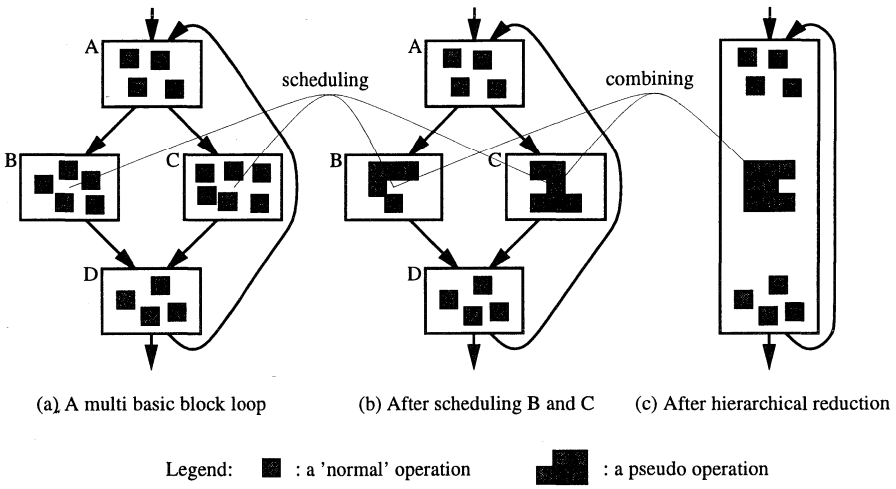


Figure 5.4: Hierarchical reduction.

2. Combining the schedules of B and C and replacing the result by a pseudo operation whose resource requirement is the union of the resource requirements of B and C, i.e., the combined schedule uses a resource r in cycle c if the B schedule *or* the C schedule uses r in cycle c.

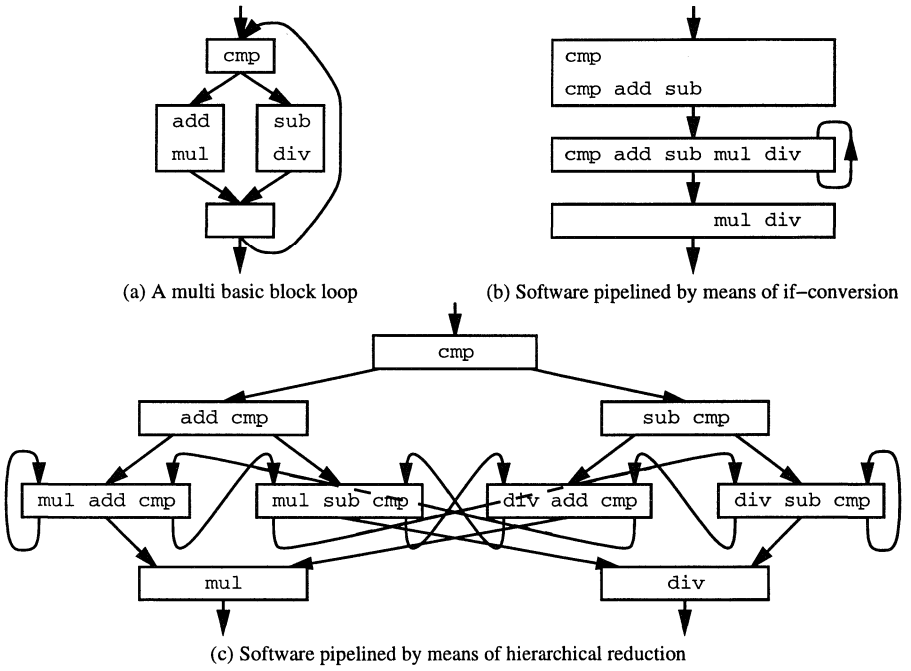
3. Modulo scheduling the pseudo operation with the other operations of the loop, i.e,. the operations of A and D.

Loops consisting of a more complex structure than a single 'if-then-else' are handled in a similar way. A limitation of hierarchical reduction is that the code should be structured [19], which is not always true for C code. After modulo scheduling the 'if-then-else' is regenerated by expanding pseudo operations in 'then' and 'else' parts. Operations, including pseudo operations, that are overlapping with pseudo operations are duplicated and placed in both parts.

If-conversion is the other method to enable modulo scheduling for multi basic block loops [10, 66, 160, 202]. If-conversion converts a multi basic block loop to a single basic block loop which can be software pipelined by means of modulo scheduling.

Both hierarchical reduction and if-conversion have their problems. The main problem of hierarchical reduction is that pseudo operations resulting from hierarchical reduction are hard to schedule due to their complex resource requirements. A pseudo operation usually requires many resources which makes it hard to find a resource conflict free cycle. Furthermore, the regeneration may require a significant amount of code duplication [202]. The main disadvantage of if-conversion is that more resources are required which may increase $ResMII$. The resource requirements of an 'if-then-else' construct is the

sum of the requirements of the 'then' and 'else' parts instead of the *union* as is required by hierarchical reduction. Figure 5.5 shows an example that illustrates the differences between if-conversion and hierarchical reduction. Hierarchical reduction requires fewer resources than if-conversion (3 vs. 5 FUs) but requires more code (9 vs. 4 instructions and 11 vs. 1 conditional jump). The main disadvantage of hierarchical reduction, the hard to schedule pseudo operations, is not illustrated in this example.



(a) A multi basic block loop

(b) Software pipelined by means of if–conversion

(c) Software pipelined by means of hierarchical reduction

Figure 5.5: If-conversion vs. hierarchical reduction. Assume that the `mul` operation depends on the `add` operation and the `div` operation depends on the `sub` operation. Furthermore, assume single cycle latencies and ample resources. The loop control code has been omitted for simplicity.

In our compiler we have chosen for if-conversion, mainly because of the hard to schedule pseudo operations. For if-conversion we use a simple scheme where each compare operation in the loop defines a unique boolean and each operation is guarded with an expression that is directly derived from its control conditions relative to the loop header. If the loop requires more booleans than available or more complex expressions than are supported by the hardware, the loop is rejected for software pipelining. If-conversion of such loops requires a more advanced if-conversion algorithm such as the RK-algorithm described in [160].

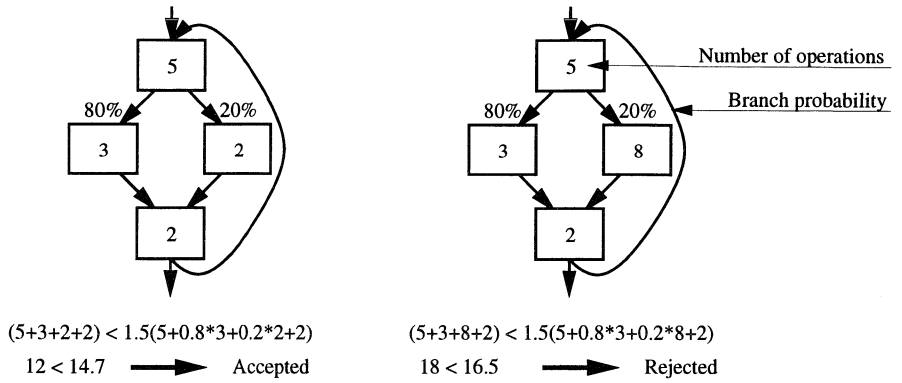


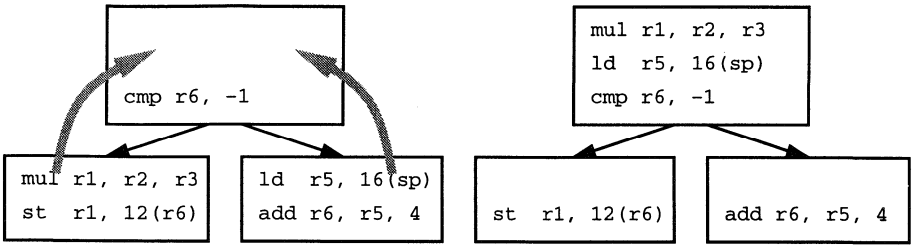
Figure 5.6: Two loops to illustrate the if-conversion selection heuristic

Furthermore, it should be wise to apply if-conversion. Loops where infrequently taken paths through the loop body use, in comparison to the frequently taken paths, many resources and are involved in long recurrences should not be if-converted, since *ResMII* and *RecMII* of these loops will be determined by the infrequently taken paths. The following selection heuristic is used to select loops for if-conversion:

$$\sum_{b \in loop} size(b) < \alpha \sum_{b \in loop} probability(header, b) \cdot size(b)$$

where $probability(b, b')$ is the probability that b' will be executed after b, $size(b)$ the number of operations of b, *header* the loop header, and α a parameter that specifies the aggressiveness of if-conversion. The default value for α is 1.5. If-conversion can be more aggressive if more resources are available. Figure 5.6 illustrates the selection heuristic. The selection heuristic takes only resource requirements into account. Dependence information is not used since it is not available at the moment when loops are selected, e.g., memory reference disambiguation has not been performed yet. Other compilers that use if-conversion seem to have the same problem, e.g., [143].

Other reasons for rejecting loops for software pipelining are: (1) The size of the loop; large loops are not software pipelined ($\geq$ 100 operations). (2) The loop should not contain inner loops or function calls. (3) The loop should not have multiple backward or exit control flow edges. The first restriction is required to prevent excessive compilation times; the other two restrictions reduce the complexity of the scheduler. Other software pipelining schedulers have similar restrictions [114, 169, 200]. Rejected loops are scheduled by the extended basic block scheduler after loop unrolling has been performed to improve intra-iteration parallelism.



(a) Before promotion

(b) After promotion

Figure 5.7: Promotion

5.2.2 Promotion

If-conversion as described in the previous subsection has two problems that limit parallelism: (1) Many data dependences are introduced between compare operations that define booleans and operations that are guarded with expressions of these booleans. (2) Booleans are often long live which increases *RecMII*. This happens when there is a long dependence path between two operations guarded with the same boolean. Such a path with a total delay of N cycles gives a lower bound on *RecMII*, i.e., $MII \geq RecMII \geq N$.

Promotion or *predicate lifting* [143,189] is a technique to alleviate both problems. Promotion removes some of the data dependences introduced by if-conversion. This corresponds to moving an operation above one or more compare operations that it is control dependent on. This has some similarities with speculative execution. This code motion has to be done under the same restriction as speculative code motions, i.e., no off-live registers and memory locations should be overwritten. Figure 5.7 illustrates promotion. The load and multiply operations are moved above the compare operation. This requires that `r1` and `r5` are dead at the beginning of the right and left branch respectively. The store operation cannot be promoted because it updates a memory location that might be live at the beginning of the right branch. The add operation cannot be promoted because `r6` is live at the beginning of the left branch.

To exemplify the benefit of promotion, assume that the load has a long latency of L_{ld} cycles. This has two consequences: (1) Since the load and add operation are guarded with the same expression and there is a dependence path between them of L_{ld} cycles, the booleans of their expressions have to be live for at least $L_{ld} + 1$ cycles. Remember that this is a lower bound on *RecMII*. (2) There is a recurrence cycle through the compare, load, and add operations. This recurrence has delay $L_{ld} + L_{cmp} + L_{add}$ cycles and an iteration distance of one iteration. This causes a lower bound on *RecMII* of $L_{ld} + L_{cmp} + L_{add}$ cycles (see equation 5.5). Both problems disappear after promotion of the load operation.

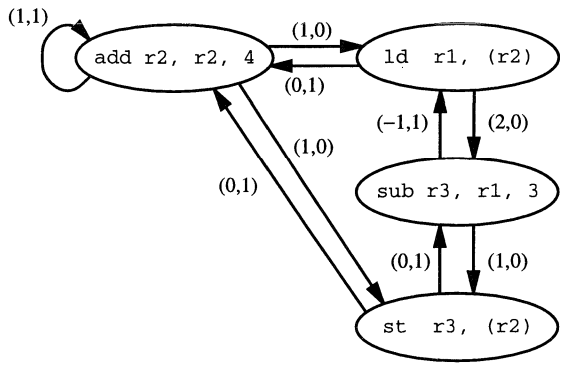


Figure 5.8: The long living register `r2` limits *RecMII*

5.2.3 Delay Lines

The problem of long living booleans, that was alleviated by promotion, also occurs for general purpose registers. A value that is live for at least N cycles results in a *RecMII* of at least N cycles. Figure 5.8 shows a DDG that illustrates the problem. Register `r2` is live for 3 cycles since the load and store operations both have `r2` as operand and there is a dependence path between them with a delay of 3 cycles. This causes a cycle through the DDG with a total delay of 4 cycles and a total iteration distance of 1 iteration, i.e., $RecMII = 4$. Two methods are known from the literature to deal with this problem: modulo variable expansion and rotating register files.

Modulo variable expansion ignores inter-iteration anti dependences during scheduling and performs register renaming and loop unrolling afterwards to correct the schedule [132]. Ignoring the two anti dependences between the load and add operations and between the store and add operations breaks the cycle described above and reduces *RecMII* to one cycle. If we assume one FU for memory and one FU for ALU operations, the loop can be scheduled with an initiation interval of 2 cycles (II=2, $RecMII$=1, $ResMII$=2). Two times unrolling and a copy of `r2` named `r2'` are required to correct the ignorance of the inter-iteration anti dependences. The kernel of the software pipeline becomes:

```
st  r3, (r2);  add r2, r2', 4
sub r3, r1, 3; ld  r1, (r2)
st  r3, (r2'); add r2', r2', 4
sub r3, r1, 3; ld r1, (r2')
```

If `r2` would be post-incremented instead of pre-incremented an intra-iteration anti dependence will show up in the critical dependence cycle. Such a cycle cannot be broken by means of modulo variable expansion. A technique called *induction variable reversal* can be used to transform pre-increments into

post-increments and intra-iteration anti dependences into inter-iteration anti dependences in order to make modulo variable expansion effective [200].

A *rotating register file* is a hardware method to deal with long living registers defined in loops. It is supported by the Cydra 5 architecture [25,65,170]. Rotating register files are rotated every loop iteration, i.e., r_i becomes r_{i+1} and r_{n-1} becomes r_0. The kernel for an architecture with a rotating register file becomes:

```
st  r3, (r2");  add r2, r2, 4
sub r3, r1, 3;  ld  r1, (r2)
```

Registers r2 and r2" have to be allocated to a rotating register file where r2" has to be allocated two positions before r2 such that r2" has the value of r2 of two iterations before.

Our compiler uses another method to deal with long living registers defined in loops. Modulo variable expansion was not applicable to our situation since it requires prepass scheduling. Rotating register files were rejected because they may affect the cycle time and we did not want to introduce hardware support for loop scheduling. Our method for the long living register problem are *delay lines*. Delay lines are the software counter part of rotating register files implemented as a sequence of copy operations. The kernel of the software pipeline with a delay line looks like:

```
st  r3, (r2'); add r2, r2, 4; mov r2', r2
sub r3, r1, 3; ld  r1, (r2)
```

In this case the delay line consist of one copy operation (mov r2',r2). Figure 5.9 illustrates the operation of a delay line. The software pipeline has three stages of two cycles. Register r2 is defined in stage one and used in stages one and three. A copy operation in stage two allows that r2 can be redefined in the next iteration before it is used for the last time in stage three. The compiler generates delay lines before register allocation. It builds a DDG for the loops without false dependences and determines *MII*. Next, it determines for each register r defined within the loop the length L_r of the longest path from the define of r to a consumer of r. Next, it generates a delay line of $n = \lfloor L_r / MII \rfloor$ copy operations:

```
mov r', r
mov r", r'
 ...
mov r^n, r^n-1
```

The delay line is inserted after the define of r, at the loop entry, or at both locations depending on where the consumers of r are located. Finally, usages of r are renamed to a proper copy of r which depends on the length of the path between the define of r and the usage of r.

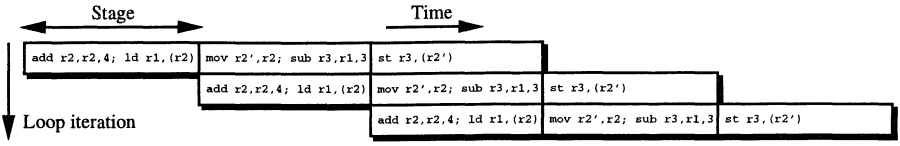


Figure 5.9: A software pipeline with a delay line of one copy operation

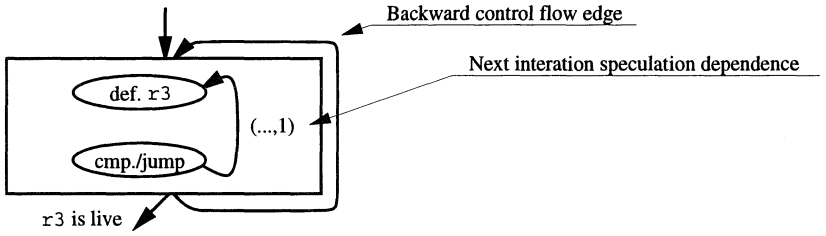


Figure 5.10: Next iteration speculation dependences

5.2.4 Software Pipelining *While* Loops

When the termination condition of a loop becomes true in iteration I_{last}, the kernel of the software pipeline terminates and the epilogue is entered to execute the remaining operations of iteration I_{last} and iterations before I_{last} that have not completed yet. At this moment, several iterations after I_{last} may have been initiated. Operations of these iterations should not change state, i.e., memory locations and registers, that is required after execution of the loop. This can be prevented by stopping the kernel earlier or by dependences that prevent operations that change state required after execution of the loop will be executed before the loop termination condition is known. The first option corresponds usually to changing upper bounds of *for* loops, e.g., instead of testing for `i<n` the loop condition becomes `i<n-1` or `i<n-2`. We have chosen for the second option for two reasons: (1) it is complicated to change loop conditions when the program representation does not contain any explicit information about loop bounds, and (2) we want to be able to software pipeline loops where the trip count cannot be determined before entering the loop. Such loops arise from *while* and *do/while* loops, but also from *for* loops (e.g., the loop shown in figure 5.11a).

Next iteration speculation (NIS) dependences are inter-iteration dependences to prevent that state used after execution of the loop will be overwritten by operations of iterations after I_{last}. These are store operations and operations that define registers that are live at the exit of the loop. Figure 5.10 shows how a NIS dependence is used to prevent that `r3`, which is live after execution of the loop, will be overwritten by operations of iterations after I_{last}. The tail of a NIS dependence with as head operation O is either the compare operation O_{cmp} that

evaluates the loop condition or the backward jump O_{jump}.

1. **A NIS dependence from O_{cmp} to O:** O will be guarded with the boolean defined by O_{cmp} such that O will not be executed if the loop condition evaluated in the previous iteration becomes false. The delay of such a NIS dependence is equal to the latency of the compare operation. If O was already guarded, because of if-conversion, it will be guarded with the conjunction of its original guard expression and the boolean defined by O_{cmp}. If this is not possible, because the target architecture does not support the required guard expression, the compiler has to choose for the second option, a NIS dependence from O_{jump} to O.

2. **A NIS dependence from O_{jump} to O:** the delay of the NIS dependence will be equal to the jump latency. This archives the same as the previous option without using guarded execution.

The first option is preferable since it allows O to be started L_{cmp} cycles after O_{cmp} of the previous iteration instead of $L_{cmp} + L_{jump}$ cycles, where L_{cmp} and L_{jump} are the latencies of compare and jump operations, respectively. This allows for a lower $RecMII$. The second option is only used if the first option is not possible due to the above mentioned reason.

Tirumalai *et al.* describe in [189] a similar method to deal with loops whose trip count cannot be determined before the loop is entered. The main difference between this method and our method is that their method relies on loop support of the Cydra 5 architecture.

5.3 TTA Specific Issues

Software pipelining for TTAs has several TTA specific issues besides the issues discussed in sections 3.2 and 4.4.

Resource constraints checking

First, one should be aware that iterations are initiated every II cycles. This has for example consequences for checking the FIFO ordering constraint of trigger and result moves of operations scheduled on the same FU. For example, an operation O_1 whose trigger and result moves are scheduled in cycles 2 and 6, respectively, cannot be scheduled on an FU with another operation O_2 whose trigger and result moves are scheduled in cycles 8 and 10 if II is 5 cycles. In this situation there would be a FIFO ordering conflict between the O_1 of iteration i and O_2 of iteration $i + 1$. Similar situations occur with other TTA specific resource checks and optimizations.

```
for(p = list; p; p = p->next)
        if(p->data > 0)
                sum += p->data;
```

(a) The C code

```
L1: r4 -> add_o; 4 -> add_t; add_r -> r3          /* r3 = &p->data */
    r3 -> ld_t; ld_r -> r5                       /* r5 = p->data */
    r5 -> gt_o; 0 -> gt_t; gt_r -> b0            /* p->data > 0? */
    !b0:L2 -> jump                               /* skip addition */
    r6 -> add_o; r5 -> add_t; add_r -> r6        /* sum += p->data */
L2: r4 -> ld_t; ld_r -> r4                       /* p = p->next */
    r4 -> eq_o; 0 -> eq_t; eq_r -> b0            /* p == NULL? */
    !b0:L1 -> jump                               /* next iteration */
```

(b) The sequential code (after register allocation, before if-conversion)

```
L1: r4 -> f3.add; 4 -> f3.add; r4 -> f1.ld
    f3.add -> f1.ld
    0 -> f4.eq
    f1.ld -> f4.eq; 4 -> f3.add; f1.ld -> f3.add; f1.ld -> f1.ld
    f1.ld -> r5; 0 -> f3.gt; f1.ld -> f3.gt; f4.eq -> b0; f3.add -> f1.ld
    f3.gt -> b1; r6 -> f5.add; r5 -> f5.add; b0:L3 -> jump; 0 -> f4.eq
    f1.ld -> f4.eq; 4 -> f3.add; f1.ld -> f3.add; b1:f5.add -> r6; f1.ld -> f1.ld
L2: f3.add -> f1.ld; f1.ld -> r5; 0 -> f3.gt; f1.ld -> f3.gt; f4.eq -> b0
    0 -> f4.eq; f3.gt -> b1; r6 -> f5.add; r5 -> f5.add; !b0:L2 -> jump
    f1.ld -> f4.eq; 4 -> f3.add; f1.ld -> f3.add; b1:f5.add -> r6; f1.ld -> f1.ld
L3:
```

(c) The software pipeline

Figure 5.11: A software pipeline to illustrate bypassing between iterations. The loop computes the sum of positive integers stored in a linked list. The '_o/_t/_r' suffixes in the scheduled code (figure c) have been omitted.

Bypassing

Software pipelining makes bypassing significantly more complex. The complications arise from not being able to schedule operations in a topological order and bypassing between inter-iteration flow dependent operations. Since operations are not scheduled in a topological order, moves which are flow dependent on a move being scheduled may already have been scheduled. If a move m_1 is being scheduled in cycle $cycle(m_1)$, and a flow dependent successor m_2 of m_1 has already been scheduled, and

$$cycle(m_2) = cycle(m_1) - II \cdot distance(m_1, m_2),$$

the value defined by m_1 and used by m_2 needs to be bypassed. This means that the source field of m_2 has to be modified. To do this, the transport resources of m_2 are released, the source field of m_2 is modified, and new transport re-

sources are assigned[4]. The whole situation is restored if no transport resources are available for m_2 or m_1 could not be scheduled for another reason.

Bypassing between moves from adjacent iterations is another source of complications. Consider the loop shown in figure 5.11. Assume a three cycle latency for load operations, single cycle latencies for all other operations, and VTLP FUs. The critical recurrence in this loop is caused by the `p = p->next` operation, which corresponds to the `r4 -> ld_t; ld_r -> r4` operation in the sequential code. This critical recurrence results in a *RecMII* of three cycles. In order to schedule this loop in three cycles, `r4` should be bypassed from `ld_r -> r4` to `r4 -> ld_t` of the next iteration. However, `r4` should not be bypassed in the first iteration of the loop which means that the first occurrence of `r4 -> ld_t` in the prologue should not be changed. This is shown in figure 5.11c, `r4` is fetched from the RF in the first cycle of the prologue, and after that it is bypassed.

Requiring a different schedule for the first iteration[5] conflicts with modulo scheduling where all iterations are supposed to have the same schedule. Our scheduler handles this situation by assigning two sets of transport resources to the move whose source is bypassed. One set for the un-bypassed version for the first iteration and one set for the bypassed version for the remaining iterations. This is not an ideal solution to the problem; the transport resources for the un-bypassed version are not used in the kernel of the loop where most of the execution time is spent. In the example code of figure 5.11c, the move bus and the RF read port used by `r4 -> ld_t` in the first cycle of the prologue are unused in the last cycle of the kernel. Improving this situation requires a significant amount of engineering effort.

Unscheduling

Unscheduling of operations is also more complex for TTAs. Unscheduling an operation O_1 with an operand bypassed from O_2 requires that O_2 is unscheduled as well if the result move of O_2 has become dead due to bypassing from O_2 to O_1. Similarly, if operation O_1 is unscheduled and an operation O_3 has an operand that is bypassed from O_1, O_3 needs to be unscheduled as well because bypassing is no longer valid. This may lead to a small 'chain reaction.' A better solution for the first case might be to try to 'un-kill' the result of O_2 move by finding transport resources for it. For the second case it might be better to undo

[4]As described in section 3.2.3, sockets are assigned during scheduling and move buses after scheduling. Only the possibility of a move bus assignment is checked during scheduling.

[5]The same situation occurs with the last iteration. A result move can be dead in all iterations except the last one. This occurs for example if variable `p` allocated in `r4` of the code shown in figure 5.11 would have been live on exit of the loop. In such a situation our compiler keeps `ld_r -> r4` alive. Killing `ld_r -> r4` in all iterations except the last one is quite complicated. It requires an extra move in the epilogue which a problem when there is no epilogue, such as in figure 5.11, or there are no transport resources available in the cycle where it should be placed.

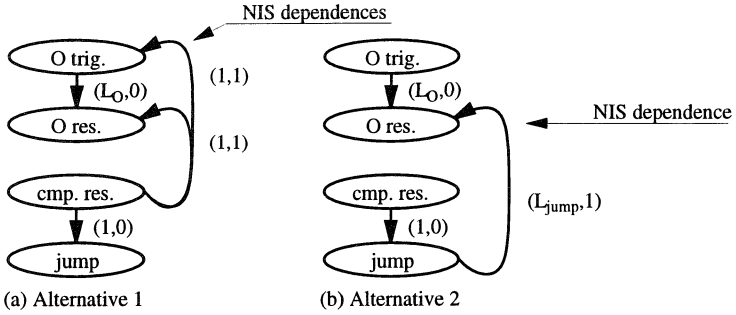


Figure 5.12: NIS dependences for an operation O scheduled on a hybrid pipelined FU

the bypassing and to try to find transport resources. Both solutions are significantly more complex to implement and may fail in which case unscheduling is still required.

If-conversion

If-conversion guards operations as described in section 4.4; operations scheduled on hybrid pipelined FUs have both their trigger and result moves guarded, and operations scheduled on VTLP FUs have only their result move guarded. The latter leads to lower *RecMII*s. NIS dependences for operations scheduled on VTLP FUs are handled similar; only their result move is guarded. For an operation O scheduled on hybrid pipelined FUs we have two alternatives shown in figure 5.12.

1. NIS dependences between the result move of O_{cmp} and the trigger and result moves of O. The effect is that the result move of O cannot be executed earlier than $1 + L_O$ cycles after the result move of O_{cmp}, where L_O is the latency of O.

2. A NIS dependence between O_{jump} and the result move of O with a delay equal to the jump latency L_{jump}. In this case the result move of O cannot be executed earlier than $1 + L_{jump}$ after the result move O_{cmp}.

Clearly, which alternative is preferred depends on L_O and L_{jump}. Option 1 is preferable for $L_O < L_{jump}$ and option 2 for $L_O > L_{jump}$. For $L_O = L_{jump}$ the scheduler uses option 2 which may reduce the live time of the boolean defined by O_{cmp}.

Architecture and Compiler Evaluation

6

Measurements play an important role in computer architecture. Architectural features, implementation alternatives, and compilation techniques have to be evaluated to understand their behavior and to determine whether it is cost effective to incorporate them. This chapter describes several measurements related to TTAs, compilation techniques for TTAs, and compilation for ILP processors in general.

Section 6.1 describes the methodology, consisting of the used benchmark set, architectural parameters, and measurement trajectory. Section 6.2 describes the experiments and their results. Section 6.3 describes bottlenecks in ILP exploitation found by analysis of scheduled code.

6.1 Methodology

Table 6.1 shows the architectural parameter values that we shall use for our experiments. All parameters values are fairly realistic with exception of the fully connected interconnection network and the perfect memory system. The influence on performance of a partially connected interconnection will be measured in section 6.2.11. The influence of a non-perfect memory system cannot be measured with the used measurement trajectory.

In most experiments we shall vary the number of move buses in order to measure how the evaluated feature depends on the amount of ILP provided by the hardware.

For our experiments we use a benchmark set of 30 programs; 20 of them are workstation-type applications, the other 10 are DSP applications. Table 6.2 lists the benchmarks, together with a short description, their dynamic operation

Parameter	Value			
Move buses	12, 64-bits wide			
Functional units:	**Number**	**Latency**	**Type**	**Operations**
• LSU	2	2	VTLP	load and store
• ALU	3	1	VTLP	int. w/o mul. and div.
• MUL	1	3	VTLP	int. multiply
• DIV	1	8	non-pip.	int. divide and modulo
• FPU	1	3	VTLP	floating point
Immediates:				
• short	12 × 8-bits signed			
• long	2 × 32-bits			
Register files:	**Number**	**Registers**	**Ports**	
• boolean	1	4	1W	
• integer	1	48	3W+3R	
• floating point	1	48	3W+3R	
Jump latency	2 cycles (1 delay slot)			
Guard expressions	*simple, and,* and *or* expressions; see section 4.5			
Interconn. network	fully connected			
Memory system	perfect (no cache and TLB misses)			

Table 6.1: Architectural parameters

count, their source code size, their *average degree of superpipelining* for the machine described in table 6.1, and the used input data set. The average degree of superpipelining (ADSP) of a machine m and an application a is the average latency of the operations of m weighted with their relative frequency in a [122]:

$$adsp(m,a) = \sum_{o \in operation\_set(m)} latency(o,m) \cdot frequency(o,a)$$

This number corresponds to the average number of operations that are in execution when the machine issues one operation per cycle. Multiplying it by the average number of operations issued per cycle gives the average number of operations in execution per cycle, i.e., the amount of exploited ILP.

Most of the workstation-type applications are GNU utilities; the exceptions are compress, djpeg, mpeg_play, and virtex. These applications have been selected based on their computational requirements and the absence of system calls that are hard to support by our simulator (process management, signal handling, etc.). The DSP applications are from [75]. They have been included in the benchmark set because of their resemblance with typical ASP applications. In order to facilitate memory reference disambiguation, a few annotations have been added to the DSP applications. This is usually acceptable for DSP/ASP applications. The workstation-type benchmarks have not been modified. All DSP applications except g722 and mulaw operate on single precision floating point data.

Benchmark	Description	Operations	Lines[a]	ADSP	Input Data
a68	68K assembler	2805K	13565	1.61	stanford.s
bison	Parser generator	5153K	9636	1.57	flex.y
cmp	File compare	855K	1747	1.67	2×563K C code
compress	File compression	37M	1516	1.59	563K C code
cpp	C preprocessor	1983K	7657	1.63	stanford.c
diff	File compare	29M	11424	1.53	2×563K C code
djpeg	JPEG decoding	27M	8856	1.54	jp.jpg (512×683)
expand	Tab expansion	29M	1510	1.60	563K C code
flex	Scanner generator	12M	10454	1.60	flex.l
gs	Postscript interpreter	39M	42134	1.59	pipeline.ps
gzip	File compression	108M	9064	1.52	563K C code
mpeg_play	MPEG decoding	54M	3132	1.48	anim.mpg (24 fr)
od	Octal dump	20M	3132	1.57	stanford.c
sed	Stream editor	46M	11971	1.55	563K C code
sort	Sort lines	80M	2410	1.53	563K C code
sum	Checksum computation	11M	1297	1.59	563K C code
tr	Translate characters	8555K	3005	1.60	563K C code
uniq	Report repeated lines	27M	1887	1.48	563K C code
virtex	Text formatting	49M	20313	1.60	man.tex (42K)
wc	Word count	7192K	1313	1.68	563K C code
arfreq	Autoregressive freq. estim.	13M	367	1.84	audio sample
equaliz	Equalization	3460K	525	1.82	audio sample
g722	Adaptive differential PCM	18M	891	1.61	audio sample
instf	Frequency tracking	3140K	436	1.86	audio sample
interp3	Sample rate conversion	3900K	504	1.81	audio sample
mulaw	Speech compression	330K	207	1.76	audio sample
music	Music synthesis	44M	321	1.64	audio sample
radproc	Doppler radar processing	29M	387	1.82	audio sample
rfast	Fast convolution using FFT	3098K	559	1.92	audio sample
rtpse	Spectrum analysis	2090K	388	1.86	audio sample

[a]Excluding header files and library code.

Table 6.2: Benchmark characteristics

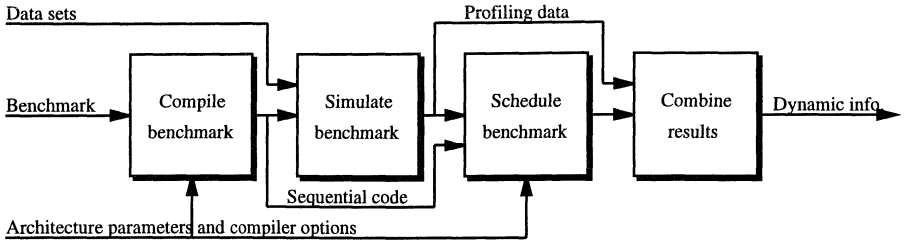


Figure 6.1: Measurement trajectory

The measurement trajectory, shown in figure 6.1, consists of the following four steps: (1) compiling the benchmark applications to sequential code with the highest optimization level of gcc-move, (2) simulation of the sequential code with representative data sets listed in table 6.2, (3) scheduling the sequential code for a specified configuration and according to specified scheduling options, and (4) combining information from the parallel code with the profiling data in order to obtain dynamic information such as cycle counts. To speed up our measurements, we schedule the procedures that are responsible for at least 99% of the operation count. This introduces a very small error in our results, but allows us to perform a large number of experiments (more than 22,000 scheduling steps) within a reasonable time.

6.2 Experiments

Each of the following 13 subsections describes an experiment that evaluates an aspect of TTAs, compilation for TTAs, or compilation for ILP processors in general. Subsection 6.2.14 summarizes the experiments.

6.2.1 Speedup

Table 6.3 shows results of scheduling the benchmarks for a 12 move bus configuration. The speedup relative to a single move bus configuration varies between 3.14 and 7.67, with averages of 4.30 and 6.38 for workstation-type and DSP-type applications, respectively. The average number of operations executed per cycle varies between 1.91 and 4.58, with averages of 2.52 and 3.28. Multiplying these numbers by the ADSP gives the average number of operations simultaneous in execution, i.e., the amount of exploited ILP. This varies between 3.04 and 7.51, with averages of 3.95 and 5.90 for workstation-type and DSP-type applications, respectively. The DSP-type applications contain clearly more exploitable ILP than the workstation-type applications. This is due to the well known fact that many DSP applications are characterized by loops with a

significant amount of relative easy to exploit inter-iteration parallelism.

Benchmark	Operations/cycle	Speedup relative to 1 move bus
a68	2.15	4.03
bison	2.26	3.83
cmp	3.01	5.98
compress	1.91	3.45
cpp	2.59	3.28
diff	2.35	4.26
djpeg	2.54	5.15
expand	2.53	3.87
flex	2.45	3.84
gs	2.49	3.82
gzip	2.50	3.14
mpeg_play	2.19	3.79
od	3.01	5.62
sed	2.41	4.13
sort	2.86	4.56
sum	2.52	5.11
tr	3.40	6.57
uniq	2.41	4.08
virtex	2.19	3.49
wc	2.62	4.05
arfreq	3.46	7.67
equaliz	2.72	4.79
g722	3.41	6.52
instf	3.11	6.35
interp3	3.40	7.35
mulaw	3.20	6.60
music	4.58	6.75
radproc	3.06	5.92
rfast	2.83	5.76
rtpse	3.02	5.99
avg. WS-type	2.52	4.30
avg. DSP-type	3.28	6.38

Table 6.3: Operations per cycle and relative speedups

6.2.2 Scheduling Scope

In this section we shall measure the effect of scheduling scope on performance. The considered scheduling scopes are the three scheduling scopes described in the preceding three chapters, together with a restricted variant of extended basic block scheduling. In this variant, the scheduler has no non-trapping versions of operations that may cause exceptions at its disposal. Without these operations the scheduler cannot schedule trapping operations above compares on which they are control dependent. Figure 6.2 shows the results. The y-axis shows the speedup relative to a single move bus configuration. The arithmetic mean is used to combine the speedups of the 30 benchmarks.

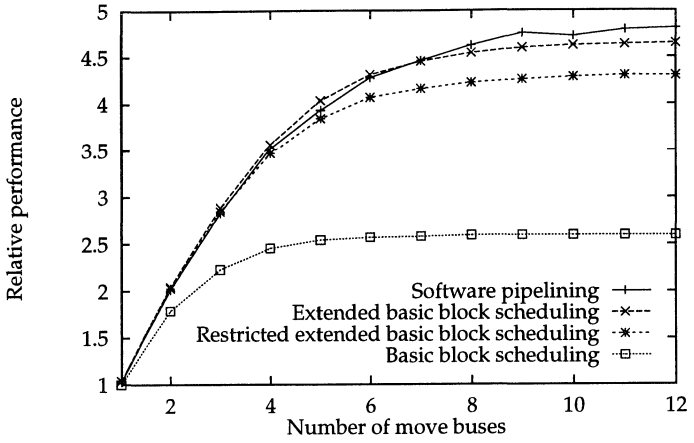


Figure 6.2: Speedup curves for four scheduling scopes

Basic block scheduling is not able to utilize more than 4 move buses. Extended basic block scheduling is therefore a necessity. Extended basic block scheduling is 79% faster for a 12 move bus configuration. Restricted extended basic block scheduling is 65% faster. The benefit of software pipelining above extended basic block scheduling is small, 3.5% for a 12 move bus configuration. For a small number of move buses (≤ 5), software pipelining is even performing worse than extended basic block scheduling. This is caused by the if-conversion selection heuristic described in section 5.2.1 that does not consider the amount of ILP provided by the target machine. For better results, the user should lower the if-conversion aggressiveness parameter, or the selection heuristic should be modified.

Benchmark	Software pipelining ratio	Speedup
arfreq	0.814	1.146
cmp	0.997	1.298
djpeg	0.365	1.022
g722	0.505	1.089
instf	0.833	1.039
interp3	0.793	1.146
mulaw	1.000	1.579
od	0.425	1.087
radproc	0.324	1.047
rfast	0.727	0.994
rtpse	0.264	1.004
sort	0.206	1.003
tr	0.994	1.078

Table 6.4: Software pipelining ratio and speedup due to software pipelining

There are several reasons for the modest improvement of software pipelining over extended basic block scheduling. First, without software pipelining loops are unrolled in order to increase intra-iteration parallelism which is already quite effective. Second, only a fraction of the loops are suitable for software pipelining. Table 6.4 shows the fraction of the execution time spend in software pipelined loops, called the *software pipelining ratio*, and the speedup due to software pipelining. Only the benchmarks with a software pipelining ratio higher than 20% are shown. A more flexible software pipelining algorithm that can handle loops with multiple backward and exit control flow edges is necessary in order to increase the software pipelining ratio. A third reason is the low trip counts of many loops; software pipelines are optimized for throughput instead of latency.

6.2.3 Scheduling Freedom

TTAs offer extra scheduling freedom; operand and trigger moves do not have to be scheduled in the same cycle, and a result move can be scheduled after the result it fetches becomes available. In order to measure the benefit of this freedom we define four scheduling models:

1. **OTR:** No freedom between operand, trigger, and result moves; operand and trigger moves are scheduled in the same cycle and result moves as soon as the result becomes available.
2. **OT:** No freedom between operand and trigger moves; operand and trigger moves are scheduled in the same cycle.
3. **TR:** No freedom between trigger and result moves; result moves are scheduled as soon as the result becomes available.
4. **Free:** No scheduling restrictions.

Figure 6.3 shows the results of scheduling the benchmarks for the four scheduling models. The extra scheduling freedom results in a better performance when transport resources are constraining the performance. For the used benchmark set and architectural parameters, the effect of extra scheduling freedom becomes negligible for more than eight move buses. This point will 'shift to the right' for applications with more exploitable parallelism since these applications are able to utilize more move buses. Figure 6.3 furthermore shows that the freedom between operand and trigger moves is more valuable than the freedom between trigger and result moves (TR performs better than OT).

Figure 6.4 illustrates scheduling freedom in a different way. The relative performance is shown as function of the *OT-freedom* and *TR-freedom*. OT-freedom is the maximal distance between corresponding operand and trigger moves, and TR-freedom is the maximal distance between the moment a result becomes

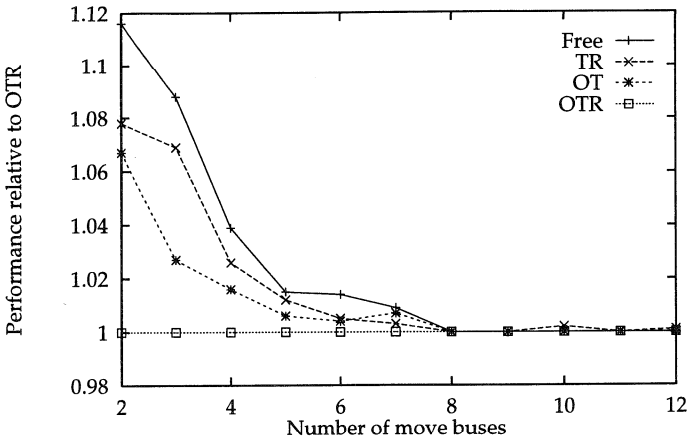


Figure 6.3: Effect of scheduling freedom

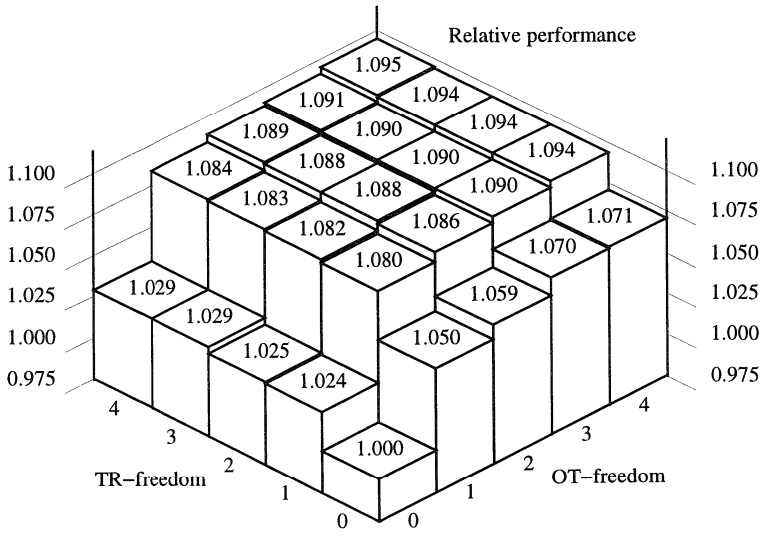


Figure 6.4: Effect of scheduling freedom for 3 move buses

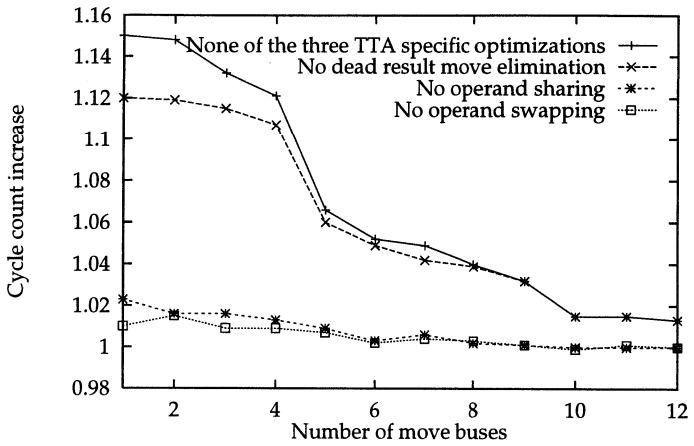


Figure 6.5: Effect of TTA specific optimizations

available and the result move that fetches it. Figure 6.4 shows again that OT-freedom is more important than TR-freedom.

6.2.4 TTA Specific Optimizations

To measure the effect of operand swapping, operand sharing, and dead result elimination, we have scheduled the benchmark set without these optimizations enabled. Figure 6.5 shows the results. Similar to the scheduling freedom experiment in the previous section, these TTA advantages are most valuable when resource constraints are limiting the performance. Dead result move elimination is clearly the most profitable TTA specific optimization; the speedup is more than 8% for small TTAs. The benefit of operand sharing and operand swapping is small, 0 – 2%, but of course still valuable.

Again, similar to scheduling freedom, the TTA specific optimizations will become more important when applications contain more exploitable parallelism.

6.2.5 Register File Port Requirements

TTAs have a lower RF port requirement than OTAs. This is because (1) not all operations produce a result and use two register operands, (2) bypassing, dead result move elimination, operand sharing, and operand swapping save RF accesses, and (3) RF ports are not coupled to FUs.

Table 6.5 shows the results of several measurements that were performed to get some insight in the RF port requirements of TTAs. The first two measurements show that the RF port requirement of the sequential code is already lower than

Measurement	Result
# Register operands per operation for sequential code	1.03
# Results per operation for sequential code	0.63
# Register operands per operation for parallel code	0.62
# Results per operation for parallel code	0.37
% Operand and trigger moves that are bypassed	42.9%
% Dead result moves due to dead result move elimination	35.8%
% Dead operand moves due to operand sharing	1.7%
# Moves that share the same RF read port (socket sharing)	1.21

Table 6.5: Results of various measurements.

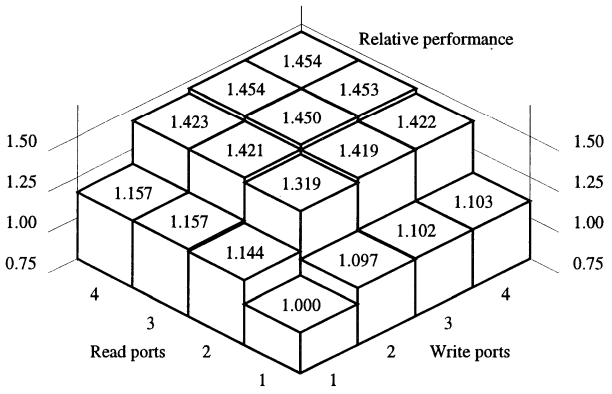


Figure 6.6: Performance of 16 RF configurations for 12 move buses

two read and one write port per operation; on average 1.03 read and 0.63 write ports. The next two lines show these numbers for scheduled code; 0.62 read and 0.37 write ports[1]. It should be noticed that this is, unlike the two read and one write port requirement of OTAs, an average requirement. In order to be able to execute up to N operations per cycle, the hardware should usually have to provide more than $\lceil 0.62N \rceil$ read and $\lceil 0.37N \rceil$ write ports, unless the smoothability of the application is very good.

The next four lines show results of experiments that quantify contributions to the RF port reduction. 42.9% of all register operands are bypassed. 35.8% of the produced results do not have to be written to an RF. 1.7% of all operand moves are dead due to operand sharing; part of them accesses an RF. Finally, each RF read port is used by 1.21 moves on average due to socket sharing.

In the next experiment we schedule the benchmark set for 16 RF configurations where the number of read and write RF ports on the integer and FP RFs

[1]Excluding port sharing.

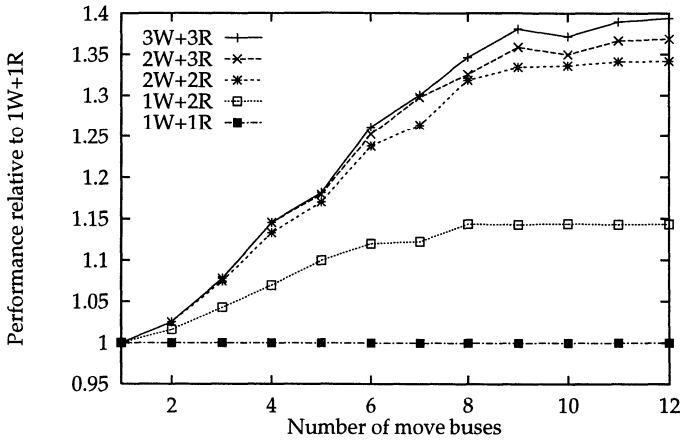


Figure 6.7: Relative performance of 5 RF configurations

is varied between 1 and 4. Figure 6.6 shows the results. The performance is expressed relative to the performance of a 1W+1R ported RF configuration. The results show that a 3W+3R ported RF is a good choice for a 12 move bus configuration. Adding more ports does not increase the performance significantly. Figure 6.6 also tells us which combinations of read and write ports are optimal. For example, the optimal 5 ported RF has 3 read and 2 write ports. Figure 6.7 shows performance of 5 optimal RF configurations relative to a 1W+1R ported RF as function of the number of move buses.

6.2.6 Partitioned Register Files

Section 3.1.9 described a simple method to generate code for partitioned register files. The registers are distributed over the register files such that register $\texttt{r}i$ is placed in RF $i \bmod N$, where N is the number of RFs that are numbered from 0 to $N-1$. Scheduling for partitioned register files is done by constraining the sockets that can be used to access a register. Input/output socket $\texttt{wp}i/\texttt{rp}i$ can be used to write/read register $\texttt{r}j$ if $(i-j) \bmod N = 0$. Figure 6.8 illustrates how a 6W+6R ported RF with 48 registers is partitioned in three 2W+2R ported RFs with 12 registers each.

We shall use 5 RF configurations, shown in table 6.6, to measure the effect of register file partitioning. The configurations differ in the number of RFs and the number of ports per RF. All configurations have 48 registers in total. The number of transistors is an indication of the cost of a configuration. This number is calculated according to the model described in [196] for 32-bit registers. The number of transistors for 64-bit wide RFs is 1.85 – 1.94 times larger. The last column of table 6.6 shows Hellerman's estimated read/write bandwidth of

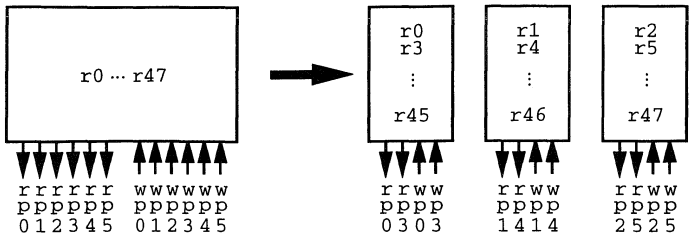


Figure 6.8: Partitioning an RF ($C_{3\times2}$ from table 6.6)

a configuration [100]. This corresponds to the expected number of read/write accesses before the first RF port (resource) conflict occurs assuming that the accesses are uniformly distributed over the RFs[2]. This estimation is optimistic in the sense that accesses are assumed to be uniformly distributed, and it is pessimistic in the sense that no more accesses will be scheduled on an RF after the first port conflict has occurred.

Configuration	RFs	Ports/RF	Regs/RF	Transistors	Bandwidth
$C_{1\times3}$	1	3W+3R	48	19,152	3.00
$C_{2\times2}$	2	2W+2R	24	15,808	3.13
$C_{4\times1}$	4	1W+1R	12	12,016	2.22
$C_{3\times2}$	3	2W+2R	16	16,800	4.05
$C_{8\times1}$	8	1W+1R	6	14,144	3.24

Table 6.6: Five register file configurations

Figure 6.9 shows the results of scheduling the benchmarks for the five RF configurations. Both the integer and floating point RFs are partitioned in this experiment. The boolean RF is left unimpaired. The required number of transistors and the estimated bandwidth are shown on the right hand-side of figure 6.9. The estimated bandwidth appears to be a reasonable performance predictor. The performance impact of partitioning is relatively small. Only the $C_{4\times1}$ configuration exhibits a cycle count increase of 3.2%. The $C_{3\times2}$ configuration performs even better than the more expensive $C_{1\times3}$ configuration. From this experiment we can conclude that it is very well possible to partition RFs in order to improve the cost/performance.

[2]Let X_i be a set of independent discrete random variables with a uniform distribution between 1 and N. Let Y be the maximum K such that $X_1 \ldots X_K$ do not have the same value more than M times. The estimated bandwidth of configuration $C_{N\times M}$ corresponds to $E(Y)$.

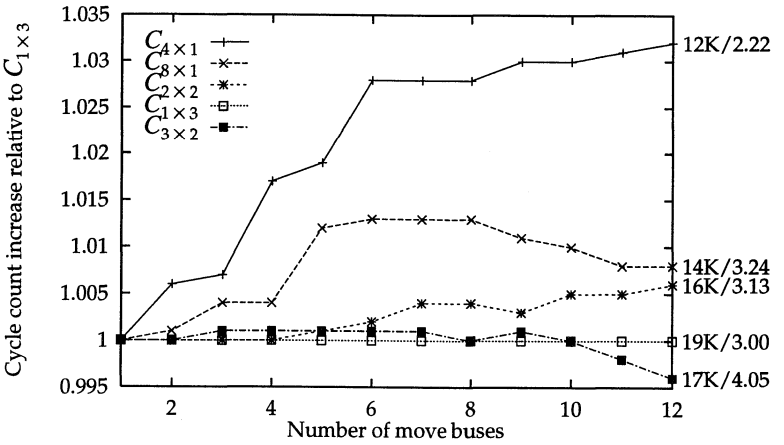


Figure 6.9: Effect of partitioning RFs

6.2.7 Multi-Way Branching and Guarding

Our scheduler supports a restricted form of multi-way branching; two guarded jumps can be scheduled in the same cycle which results in a three-way branch. Figure 6.10 shows the effect of multi-way branching. The average improvement for a 12 move bus configuration is 4.9%. Most of this improvement comes from wc (26.0%), a68 (13.4%), cpp (10.5%), and gzip (8.8%). Seven of the ten DSP application exhibit no improvement, and the other three DSP applications exhibit an improvement of less than 0.8%. This is obviously due to the fact that many DSP applications have a high software pipelining ratio.

To measure the effect of guarding support, we have scheduled the benchmarks with and without and/or guard expressions (see section 4.5). Without and/or guard expressions, the scheduler is not able to generate multi-way branches, it cannot schedule an operation above more than one branch on which it is control dependent and for which it should be guarded, and it is limited in its if-conversion capabilities. Figure 6.11 shows the performance improvement due to and/or guard expressions. The average improvement for 12 move buses is 6.4%. Most of this difference comes from wc (22.8%), od (15.1%), a68 (14.1%), and cpp (12.2%).

The 6.4% performance improvement is not decisive to either reject and/or guard expressions and use a software method that achieves the same with simple guard expressions [123, 160] or to adopt it, possibly in combination with the method sketched in section 4.5 to reduce code size. Further research is required to make this decision.

The required number of booleans appeared to be very small. For simple guard expressions, 2 booleans will suffice; adding a third boolean improved the av-

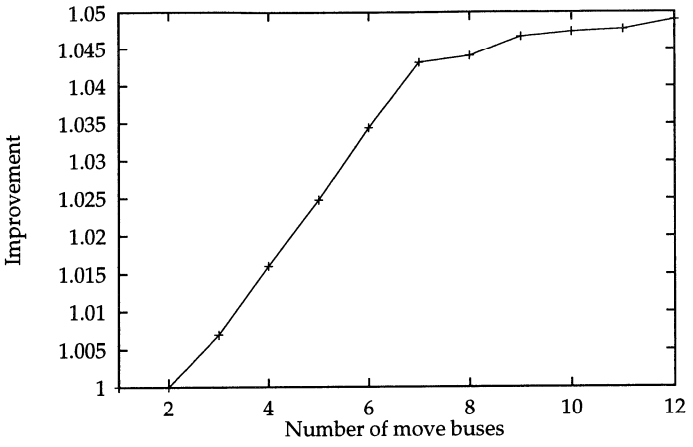


Figure 6.10: Performance improvement due to multi-way branching

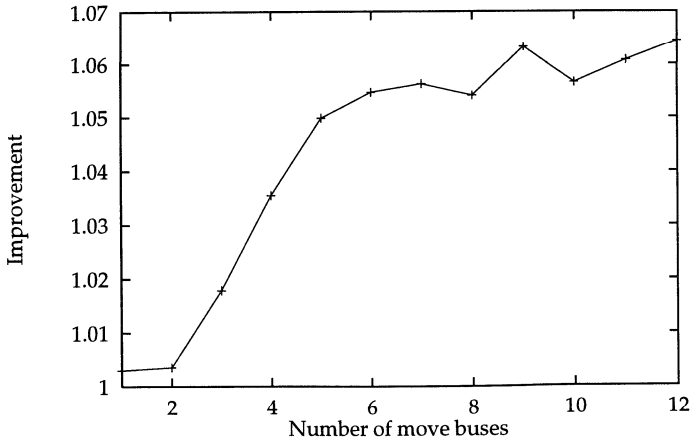


Figure 6.11: Performance improvement due to and/or guard expressions

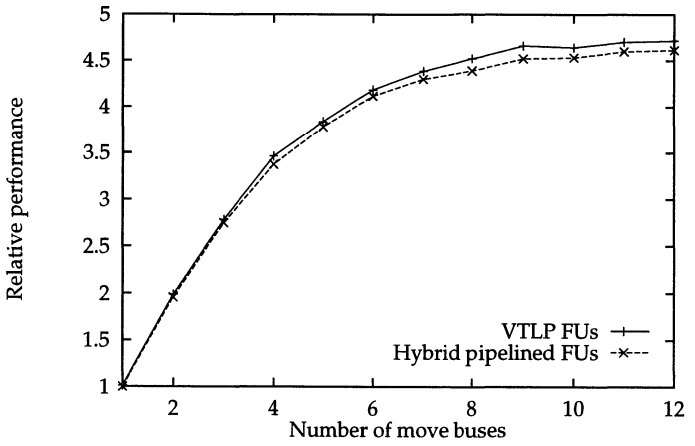


Figure 6.12: Hybrid pipelined vs. VTLP FUs

erage performance by 0.15%. For and/or guard expressions, 2 booleans is also sufficient; adding a third boolean improves performance by 0.59% and a fourth boolean gives another 0.32% performance improvement.

6.2.8 Functional Unit Pipelining

Section 2.3.7 described two alternatives for pipelined FUs: hybrid pipelined FUs and VTLP FUs. Hybrid pipelined FUs give more scheduling freedom while VTLP FUs are easier to implement and are more convenient in combination with speculative execution. In order to determine which FU type is preferable, we have scheduled the benchmarks for an architecture with hybrid pipelined FUs and one with VTLP FUs. Figure 6.12 shows the results. VTLP FUs are clearly preferable; VTLP FUs are easier to implement, and perform up to 4.5% better than hybrid pipelined FUs.

6.2.9 Memory Reference Disambiguation

In order to measure the effect of memory reference disambiguation on performance, we introduce three levels of memory reference disambiguation capabilities:

1. **No disambiguation:** Two memory operations are dependent unless both accesses are load operations.
2. **Realistic disambiguation:** Two memory operations are independent if they can be disambiguated by the memory reference disambiguator described in section 3.1.8.

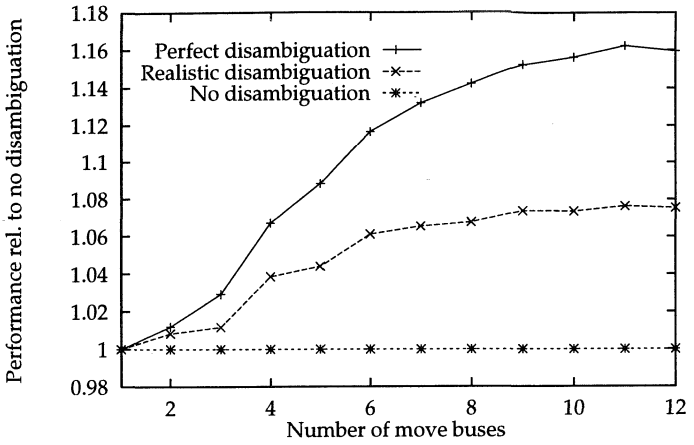


Figure 6.13: Effect of memory reference disambiguation

3. **Perfect disambiguation:** Two memory operations of which at least one is a store operation are dependent if they have referred to a common memory location during simulation of the sequential code.

Perfect disambiguation detects only the dependencies that occurred during simulation of the sequential code and is therefore dependent on the used data sets. It is used to give an upper bound on what can be achieved by improving the memory reference disambiguator. Figure 6.13 shows the results of scheduling the benchmarks with the three levels of disambiguation capabilities. Only the workstation-type benchmarks are used in this experiment because of the annotations for memory reference disambiguation have been added to the DSP applications. The difference in performance between no disambiguation and realistic disambiguation is 7.5%. The difference between realistic and perfect disambiguation is 7.8%. The latter is likely to increase when the target machine becomes wider or load latencies increase. Furthermore, the gap between realistic and perfect disambiguation varies greatly between the benchmarks. For example, 0% for cmp, tr, and wc, 23% for expand, and 35% for mpeg_play.

6.2.10 Multicasts

A multicast is a one-to-many data transport over a single move bus. It reduces the move bus requirements at the expense of extra multicast destination ids. To get an idea on what can be gained by multicasts we have modified the move bus assignment part of our scheduler[3]. Multiple moves can share a move bus

[3]We have implemented this in an older version of our scheduler. This version performs first-fit move bus assignment during scheduling instead of the bipartite matching based assignment algorithm described in section 3.2.3.

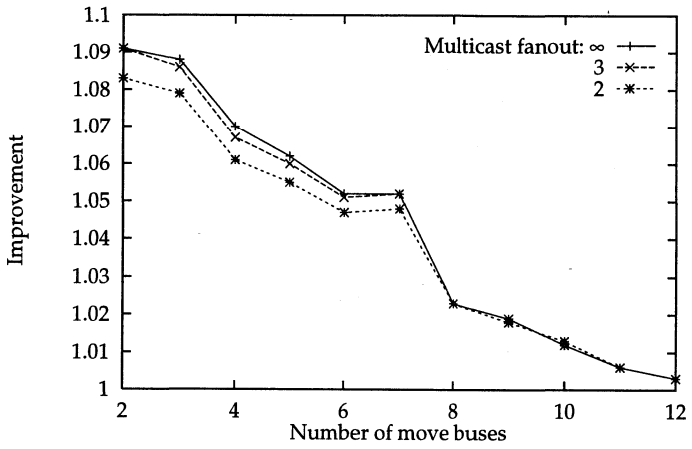


Figure 6.14: Effect of multicasts

if they have the same source. The maximum number of moves that can share a move bus is called the *multicast fanout*.

Figure 6.14 shows the results of scheduling the benchmarks for TTAs with multicast support with multicast fanouts 2, 3, and infinity. In this experiment there are no constraints on the combinations of data transports that can be performed by multicasts. Such constraints would be required in realistic implementations. The results of the experiment are clear; multicasts reduce the move bus requirement, and improve therefore the performance when move buses constrain the performance. This experiment also shows that a multicast fanout of three is sufficient.

Further research is necessary to determine how multicasts can be incorporated in the scheduler (section 3.2.3 gave already a possible solution), which multicasts are required, how multicasts affects guarded execution (in the performed experiment we assumed one guard expression per multicast destination), and whether it is cost effective.

6.2.11 Partial Connectivity

Realistic TTAs will have a partially connected interconnection network. This leads to the following questions: (1) How much connectivity is needed, i.e., how many connections can be removed from a fully connected network before the cycle counts starts to increase? (2) Which connectivity is required for an application or an application domain? The first question will be treated in this section; the latter question in section 7.1.2.

To answer the first question, we schedule the benchmarks for a sequence of configurations with decreasing connectivity starting with the fully connected

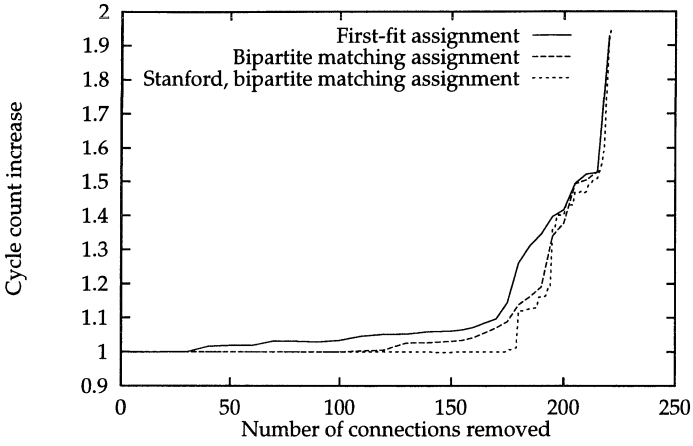


Figure 6.15: Effect of partial connectivity

configuration. This sequence is optimized for the stanford benchmark and is computed by means of the algorithm described in section 7.1.2. The stanford benchmark is a collection of ten small benchmarks such as queen, hanoi, quicksort, and FFT. In order to reduce measurement time, we use a smaller configuration with fewer connections than the configuration described in table 6.1. The used configuration contains 2 LSUs, 2 ALUs, 1 FPU, 8 move buses, a 3W+3R ported integer RF, and a 2W+2R ported FP RF. This configuration has 256 connections when it is fully connected. This connectivity can be reduced to 45 connections; the stanford benchmark cannot be scheduled anymore when more connections are removed.

Figure 6.15 shows the results of this experiment. Three lines are shown. The bottom line shows the cycle count increase of the stanford benchmark as function of the number of removed connections. The cycle count of stanford starts to increase rapidly after 175 connections have been removed. The middle line shows the average cycle count increase for the 30 benchmarks when they are scheduled for TTAs with interconnection networks that are optimized for the stanford benchmark. In this case the cycle count starts to increase after 120 connections have been removed. The top line shows the cycle count increase for the 30 benchmarks in case a first-fit move bus assignment algorithm would have been used. In this case the cycle count starts to increase after 40 connections have been removed. This experiment shows that the bipartite matching move bus assignment algorithm performs substantially better that the first-fit assignment algorithm; 4% on average.

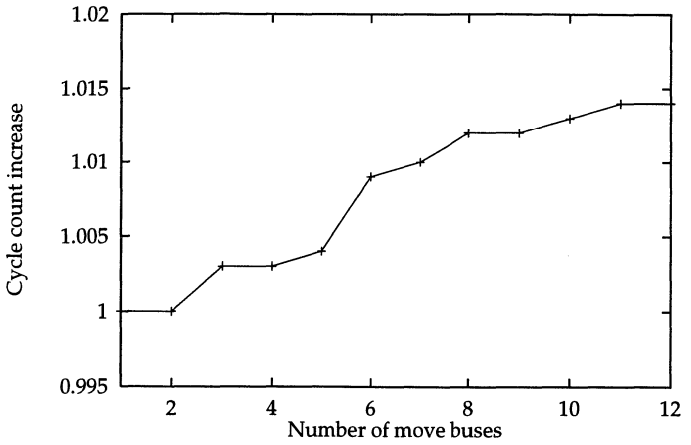


Figure 6.16: Performance degradation due to bypass conflicts

6.2.12 Bypass Conflicts

TTAs perform bypassing at compile-time. This works only if the compiler can determine whether bypassing is required and, if it is required, from which FU. In chapters 4 and 5 we have seen that this is not always possible. We have called these situations bypass conflicts. Bypass conflicts are resolved by our scheduler by delaying the move being scheduled. This may increase the cycle count. In order to measure the performance degradation due to bypass conflicts, we schedule the benchmarks with an option that instructs the scheduler to ignore bypass conflicts. Obviously, the produced code is therefore no longer correct.

Figure 6.16 shows the cycle count increase due to bypass conflicts. This is 1.4% for a 12 move bus configuration. The main contributors to this increase are: compress (6.5%), gzip (4.2%), expand (3.1%), and gs (3.1%). The cycle count increase for the DSP applications is negligible; the only exception is music which exhibits an increase of 1.2%. As described in section 4.4, resolving bypass conflicts without delaying the move being scheduled is hard. It requires extra resources which makes it hard for the compiler to determine whether it is worthwhile to use these resources to resolve the bypass conflict. Furthermore, it requires extra input sockets on FU inputs if the instruction pipelining scheme does not allow for input socket sharing (see section 4.4).

6.2.13 Register Allocation

In section 3.1.9 we motived our decision for post-pass scheduling and described a method to deal with the problem of post-pass scheduling: false de-

pendences due to register re-use that reduce scheduling freedom. In this section we want to measure the effectiveness of this method. To do this we compile the benchmarks in three different ways:

1. Without a register allocator. In this case every live range is placed in a different register.
2. With a register allocator that does not generate caller/callee save/restore code and without a FDPG (false dependence prevention graph).
3. With a register allocator that does not generate caller/callee save/restore code and with a FDPG.

Dividing the cycle counts of (2) by (1) gives the cycle count increase due to spilling and register re-use when no FDPG is used. These results are shown in column 2 of table 6.7. Dividing the cycle counts of (3) by (1) gives the cycle count increase when a FDPG is used. These results are shown in columns 3 and 4 of table 6.7, for 48 and 32 registers, respectively. Since 32 and certainly 48 registers should be enough in order to avoid spilling for most of the benchmarks, the reported cycle count increase is a tight upper bound on the cycle count increase due to register re-use, whereby the upper bound for the 32 register measurement is less tight than the upper bound for the 48 register measurement.

From table 6.7 we can see that usage of a FDPG to prevent false dependences reduces the cycle count increase from 25.5% to 1.3% for a 48 register configuration, and to 2.6% for a 32 register configuration. Only, a few benchmarks exhibit a significant cycle count increase: sed (7.4%), gzip (5.2%), wc (3.6%), and expand (3.0%). Further research has to find out how the false dependence prevention mechanism can be improved and whether pre-pass scheduling or integrated scheduling and allocation give better results.

6.2.14 Conclusions

Table 6.8 summarizes the results from the experiments and the conclusions that can be drawn from them.

6.3 ILP Exploitation Bottlenecks

Why is the speedup of a 12 move bus configuration over a 1 move bus configuration less than 12 times even though most applications have enough ILP for 12 move buses, in other words, what are the ILP exploitation bottlenecks? To answer this question, we have spent some time to analyze the code produced by the scheduler, especially the parts where the move bus utilization is low. The main bottlenecks we encountered are listed below.

Benchmark	Cycle count increase due to register re-use and spilling		
	48 GPRs, w/o FDPG	48 GPRs, w/FDPG	32 GPRs, w/FDPG
compress	1.155	1.007	1.007
a68	1.020	1.015	1.014
mpeg_play	1.125	1.001	1.070
virtex	1.229	1.001	1.056
bison	1.070	1.006	1.006
diff	1.111	1.022	1.027
uniq	1.208	1.018	1.023
sed	1.360	1.074	1.201
flex	1.126	1.020	1.020
gs	1.197	1.007	1.009
gzip	1.249	1.052	1.087
sum	1.221	0.994	0.994
expand	1.259	1.030	1.041
djpeg	1.136	1.023	1.039
cpp	1.057	1.009	1.026
wc	1.092	1.036	1.036
sort	1.072	1.003	1.028
cmp	1.000	1.000	1.000
od	1.096	1.013	1.019
tr	1.397	1.000	1.000
equaliz	1.044	1.000	1.000
rfast	1.532	1.000	1.000
rtpse	1.449	1.008	1.001
radproc	1.417	1.018	1.047
instf	1.389	0.999	1.000
mulaw	2.299	1.000	1.000
interp3	1.157	1.009	1.009
g722	1.508	1.018	1.020
arfreq	1.277	1.000	1.000
music	1.392	1.000	1.000
average	1.255	1.013	1.026

Table 6.7: Effect of register allocation

Experiment	Results and conclusions
Speedup	The average speedup of a 12 move bus configuration relative to a single move bus configuration is 4.30 for workstation-type applications and 6.38 for DSP-type applications. Average number of operations initiated per cycle: 2.52 (WS) and 3.28 (DSP). Average number of operations in execution per cycle: 3.95 (WS) and 5.90 (DSP).
Scheduling scope	Basic block scheduling cannot utilize more than 4 move buses. Extended basic block scheduling gives a performance improvement over basic block scheduling of 79%. Software pipelining gives another 3.5% improvement.
Scheduling freedom	Valuable when transport resources are constraining the performance. The performance improvement is 2 – 10% for TTAs with a small number of move buses.
TTA specific opt.	Valuable when transport resources are constraining the performance. Dead result move elimination: 4 – 10% improvement. Operand swapping and operand sharing: 0 – 2% improvement.
RF port requirement	The average RF port requirement per operation is 0.63 read ports and 0.37 write ports. A 3W+3R configuration is sufficient to perform 2.77 operations per cycle on average.
Partitioned RFs	The presented method seems to work well. RFs can be partitioned in order to reduce their costs without a significant performance loss.
Guarding	Multi-way branching: 4.9% improvement. And/or guard expressions instead of simple guard expressions: 6.4% improvement. A 2 – 3 boolean RF is sufficient.
FU pipelining	VTLP is preferable; easier to implement and up to 4.5% faster than hybrid pipelining.
Memory ref. disamb.	7.5% performance improvement in comparison with no memory reference disambiguation. Up to 7.8% improvement can be achieved by a better memory reference disambiguator.
Multicasts	5 – 9% performance improvement when transport resources are constraining the performance.
Partial connectivity	With the bipartite matching move bus assignment algorithm it is possible to remove a large percentage of the connections before the cycle count starts to increase. 47% percent of the connections of a fully connected configuration can be removed before the cycle count increase becomes more than 1%.
Bypass conflicts	1.4% performance degradation on average.
Register allocation	The used method to prevent false dependences introduced by register allocation before scheduling seems to work well. The average cycle count increase caused by register re-use *and* spilling is 1.3% for 48 registers, and 2.6% for 32 registers.

Table 6.8: Results and conclusions

1. **Ambiguous memory references.** The inability to disambiguate memory references results in serialization of memory references. This occurs mainly when components of address computations are loaded from memory or passed via procedure arguments.

2. **The latency of conditional jumps.** A conditional jump takes four cycles for the architecture described in table 6.1.

```
r2 -> cmp.eq_o; r3 -> cmp.eq_t  /* trigger compare operation */
cmp.eq_r -> b2                  /* move result to boolean RF */
b2: target -> jump              /* guarded jump */
...                             /* delay of guarded jump */
```

Reducing this to three or two cycles improves performance. The jump delay can be removed by some form of dynamic branch prediction. The delay between the define of a boolean and using it can be reduced to zero cycles by bypassing the boolean RF. This is illustrated in figure 6.17. Moves can be guarded by booleans in the boolean RF and the booleans currently written to the boolean RF. Whether this is possible without affecting the cycle time has to be researched. A quick experiment produced the following results:

Improvement	Cycle count decrease (%)		
	Average	Maximum	
Boolean RF bypassing	4.54	11.79	(uniq)
Single cycle jumps	5.03	17.24	(a68)
Both	10.27	29.33	(wc)

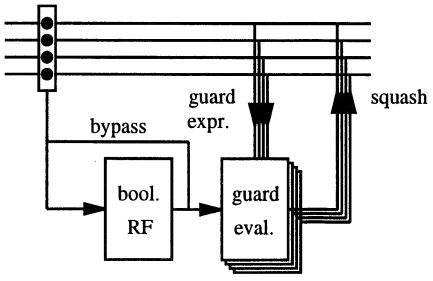


Figure 6.17: Boolean RF bypassing

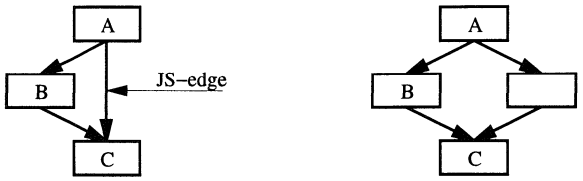
3. **The latency of load operations.** A load with offset calculation takes three cycles whereas most RISC processors do this within one or two cycles. An unsigned sub-word load operation takes another cycle for zero extension (a bitwise-and operation). Reducing the load latency, by a faster data cache or memory, has a significant influence on the performance, especially in code where load operations show up in critical paths due to

ambiguous memory references. A quick experiment produced the following results:

Load latency	Cycle count increase (%)	
	Average	Maximum
1	0.00	0.00
2	8.79	18.08 (sed)
3	23.53	39.62 (sed)
4	37.22	69.65 (sed)

4. **The SCP rule.** The SCP rule discussed in section 4.3.1 constrains inter basic block code motions. Bernstein describes in [28] a solution to this problem. He 'breaks' all *JS-edges* by inserting empty basic blocks. A JS-edge is a control flow edge from a fork point to a join point. After all JS-edges have been eliminated, the set of basic blocks where duplicates of the operation being imported have to be placed ($D(b, b')$ in section 4.3.1) satisfies always the SCP rule. Figure 6.18 illustrates JS-edge breaking; after inserting an empty basic block between A and C it becomes possible to import operations from C to B.

   JS-edge breaking prior to scheduling is not directly possible in our scheduler. The problem is that the scheduler removes basic blocks whenever they become empty. A better solution would be to break JS-edges on-the-fly during scheduling whenever the SCP rule restricts a code motion. Implementing this requires a significant amount of engineering effort.

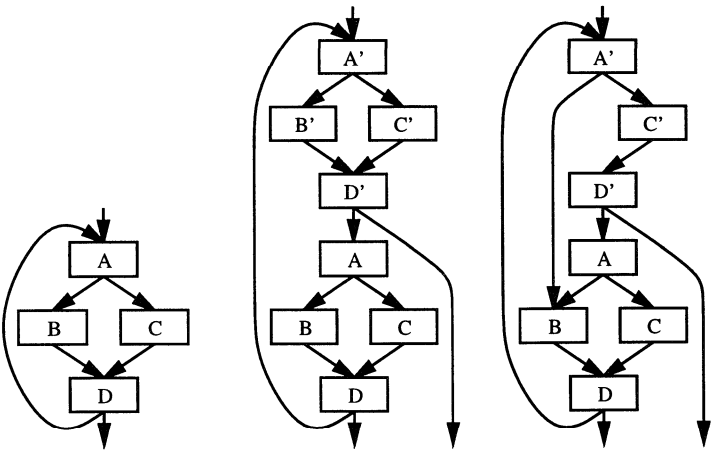


(a) Before JS–edge breaking (b) Ater JS–edge breaking

Figure 6.18: Breaking JS-edges

5. **Short frequently taken paths though large loops.** There are many loops where frequently taken paths though the loop are short and do therefore not contain sufficient ILP, but where the loop is too large for loop unrolling. The solution to this problem is *partial loop unrolling* illustrated in figure 6.19. Assume that B is a large sub-CFG and $A \rightarrow C \rightarrow D$ is the frequently taken path through the loop. Figure 6.19c shows the loop of figure 6.19a unrolled two times without duplicating B.

   Partial loop unrolling is described and evaluated in appendix A.



(a) Before loop unrolling (b) Full loop unrolling (c) Partial loop unrolling

Figure 6.19: Full and partial loop unrolling

7 Design Space Exploration

Designing an ASP by means of a templated ASP consists finding a proper *configuration* for the given application, where a configuration corresponds to a set of values for the architectural parameters of the templated ASP. The design objectives are to minimize the cycle count, cycle time, cost, and sometimes power consumption. These objectives are conflicting and the relative importance depends on the design situation. In this thesis we will not consider power consumption. However, power consumption is more or less proportional to chip area and therefore costs, so minimizing costs minimizes power consumption as well. Furthermore, we will assume that FUs, RFs, and the instruction fetch unit are designed for the same cycle time, e.g., 20ns. This means that the cycle time is only dependent on the number of connections on the move buses.

It should be clear that it is very unlikely that examining an application or its features is adequate to find a proper configuration for a non-trivial application. This means that the design process should be based on quantitative feedback of the compiler and processor generator. Design space exploration consists therefore of evaluating the interesting areas of the design space. Since the design space of a realistic templated ASP becomes very large and manual exploration is tedious and error prone work, it is highly desirable to automate the search procedure.

This chapter presents the developed design space exploration method in section 7.1 and describes a case study in section 7.2. Section 7.3 concludes with related work.

7.1 The Design Process

The *MOVE framework* consists of a set of tools together with a method to design ASPs based on TTAs by means of these tools. The design procedure consists of the following steps:

1. **Development of the application and implementing it in C/C++.** This can be done on any workstation that offers a C or C++ programming environment similar to the compiler of the MOVE framework (e.g., same word size). The reason for not using the framework compiler is the execution speed; binaries produced by the framework compiler have to be simulated, which is about 5–50[1] times slower than execution on real hardware.

2. **Identification of critical procedures.** These procedures, where most of the execution time is spent, will be used to identify special FUs and to reduce the design space exploration time. Standard profiling tools such as prof, gprof, and pixie or MOVE framework tools can be used for this purpose.

3. **Identification of special FUs.** The MOVE framework offers support for special FUs (SFUs) for user defined operations. Critical program fragments which are relatively easy to implement in hardware but cannot be executed efficiently by traditional operations can by implemented by SFUs. To identify SFUs, the designer should have some knowledge about what is efficiently implementable in hardware. An example of SFUs will be given in section 7.2.

   The MOVE framework requires that user defined operations are explicitly coded in the C/C++ source code; the compiler is not able to recognize user defined operations by itself. This situation is usually acceptable for an ASP design environment.

4. **Compilation to sequential code.** After the application is coded in C/C++ it is compiled by means of gcc-move to sequential TTA code. This should be done for several operation sets. For example, with and without FP operations, with and without sub-word support, and with or without user defined operations; see table 3.1. The MOVE framework asks some insight from the designer to determine which operation sets are useful, e.g., it makes no sense to compile an FP intensive application without FP operations.

[1] The MOVE framework offers two possibilities for simulation (1) an instruction interpreter, and (2) a translator that converts TTA code to C code which can be compiled and executed on the host system.

5. **Profiling of the sequential code.** Each of the binaries produced by the previous step needs to be simulated to obtain profiling information. Obviously, this needs to be done with representative data sets.

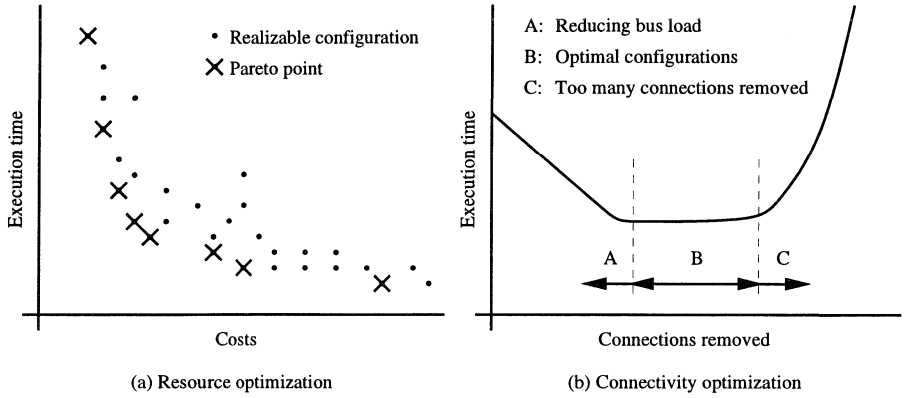
6. **Resource optimization.** This is the first step of the design space exploration. A search algorithm explores the design space and presents a number of 'interesting' configurations to the designer. All these configurations have a different cost/performance ratio. It is up to the designer to choose a configuration that is the best for his/her situation. Resource optimization will be detailed in section 7.1.1.

7. **Connectivity optimization.** The result of resource optimization is a fully interconnected TTA. Connectivity optimization reduces the connectivity of the TTA chosen in the first step and presents again a set of interesting configurations to the designer from which he/she can choose. Connectivity optimization will be detailed in section 7.1.2.

8. **Generation of the processor and executable.** Finally, after the architectural parameter values have been determined, a TTA processor is generated/implemented with these parameter values and an executable is generated for it. A processor generation system for TTAs is described in [58].

As described above, design space exploration is divided into two steps, resource optimization and connectivity optimization. The goal of resource optimization is to find a configuration, which is fully connected, with the right cost/performance ratio. During this step the influence of the full connectivity on the cycle time will not be taken into account. It is assumed that the cycle time of the final configuration, after connectivity optimization, will not be determined by bus load. The goal of the second step is to reduce the bus load, by removing socket move bus connections, such that it does not longer determine the cycle time. Reducing costs is a secondary goal.

The motivation for performing design space exploration in two steps is that combining them is problematic and would be computationally too expensive. The problem is how to connect, for example, an FU to an existing configuration. When it is connected too strongly, e.g., fully connected, the cycle time degradation may be more than the cycle count improvement. On the other hand, when it connected too loosely, the cycle count improvement may not be large enough to justify the costs of the FU. In both cases the explorer will not add the FU, although it could be good decision if it would be well connected. Similar problems also occur when FUs are removed from a configuration.



(a) Resource optimization

(b) Connectivity optimization

Figure 7.1: Results of resource and connectivity optimization

7.1.1 Resource Optimization

The design space is infinite, discrete, and has a large number of dimensions. An evaluation function maps the design space to a two-dimensional cost/execution time space. A configuration is evaluated by invoking the compiler and processor generator. The compiler compiles the profiled application for the configuration and returns a cycle count; and the processor generator returns a cycle time and cost estimation.

Only a subspace of the cost/execution time space will be realizable (for example configurations with zero costs are not realizable), and of that subspace only a subspace will be of interest to the designer. These are the so called *Pareto points* [35, 64]. A configuration is a Pareto point if it is realizable and there are no other realizable configurations that are both faster and cheaper, i.e.,

$$P = \{(c, e) \in R \mid \forall (c', e') \in R, c' \geq c \vee e' \geq e\}$$

where P is the set of Pareto points, R the set of realizable configurations, and (c, e) represents a configuration with cost c and execution time e. Figure 7.1a illustrates the Pareto point concept.

The objective of resource optimization is to present a large set of Pareto points to the designer from which he/she can make a choice. Instead of presenting Pareto points we could confine with minimizing a user specified quality function. We prefer presenting Pareto points since it is usually very difficult for the designer to define a quality function for his/her situation.

We find Pareto points by means of a local search algorithm shown in figure 7.2. Exploration starts with a user specified oversized configuration (*init*), i.e., 6 integer FUs, 20 move buses, and 128 integer GPRs. From this configuration we go to a neighbor configuration by removing one resource of it. This is repeated

until we reach the minimum configuration that is needed to compile the application successfully. Which resource is removed is determined by a quality function.

$$quality(config) = \frac{1}{costs(config)^{\alpha}.execution\_time(config)^{\beta}}$$

where α and β are constants that are reflecting the importance of cost and performance respectively. After the initial configuration has been reduced to the minimum configuration, the process is reversed by putting the removed resources back until the initial configuration is reached again. Which resource is put back is also determined by the quality function. Several of these reduce/extend passes are made between the initial and minimum configurations for different values of α and β. For reduce and extend passes we use the following set of 2-tuples (α, β):

$$\begin{aligned} \text{reduce:} &\quad \{(1,1),(1,1.5),(1,2),(1,2.5),(1,3)\} \\ \text{extend:} &\quad \{(1,1),(1.5,1),(2,1),(2.5,1),(3,1)\} \end{aligned}$$

During the reduce passes we want to stay as close as possible to the cost axis. This is achieved by $\beta > \alpha$. Similarly, during the extend passes we want to stay close to the execution time axis which is achieved by $\alpha > \beta$. This set of used α and β values was obtained by means of experimentation. Adding more pairs to it did not lead to finding significantly more Pareto points, it only increased the exploration time.

The resources that are considered by the explorer are currently: (1) FUs, (2) move buses, (3) GPRs (integer, floating point, and boolean), and (4) RF ports. Removing a move bus from a configuration is done incrementally by reducing its width in steps from 64 bits to 32 bits to 1 bit to 0 bits. Adding move buses proceeds in the other direction. This can lead to configurations such as two 64-bit buses for floating point numbers, six 32-bit buses for integers, and one 1-bit bus for booleans. GPRs are removed/added in groups such that the number of GPRs per RF is always 2^n or $2^n + 2^{n+1}, n \geq 0$. This is because the difference in cost and performance of, for instance, a 32 and a 31 integer RF configuration is very small.

The explorer has several profiled sequential executables of the application compiled for different operation sets at its disposal. When it needs to evaluate a configuration it chooses the executable with the lowest dynamic operation count that uses only operations that are supported by the configuration. For example, if there are two executables, one of 1M operations with FP operations and one of 1.3M operations without FP operations, it chooses the first one if the configuration does support FP operations, and it chooses the second configuration if it does not support them.

The exploration time of the algorithm shown in figure 7.2 can be fairly long, especially if the initial configuration is very large and the evaluation time of

proc ResourceOptimization(*init*)
beginproc
  $curr = init$
  for each $\alpha \in \{1, 1.5, 2, 2.5, 3\}$ **do**
    repeat
      $neig = \{n \in \mathcal{P}(init) \mid n = curr - \{r\},\ r \in curr, \text{IsValid}(n')\}$
      $cand = \{n \in neig \mid \forall n' \in neig, \text{Quality}(n, 1, \alpha) \geq \text{Quality}(n', 1, \alpha)\}$
      if $cand \neq \emptyset$ **then**
        $curr = \text{Select}(cand)$
      endif
    until $cand = \emptyset$
    repeat
      $neig = \{n \in \mathcal{P}(init) \mid n = curr \cup \{r\},\ r \notin curr\}$
      $cand = \{n \in neig \mid \forall n' \in neig, \text{Quality}(n, \alpha, 1) \geq \text{Quality}(n', \alpha, 1)\}$
      if $cand \neq \emptyset$ **then**
        $curr = \text{Select}(cand)$
      endif
    until $cand = \emptyset$
  endfor
endproc

proc Quality($config, \alpha, \beta$)
beginproc
  return $\text{Costs}(config)^{-\alpha} \cdot \text{CycleCount}(config)^{-\beta} \cdot \text{CycleTime}(config)^{-\beta}$
endproc

Figure 7.2: The algorithm for resource optimization. A configuration is represented as a set of resources. IsValid(c) determines whether configuration c has sufficient resource to compile the application successfully. Select(s) returns a member of s. Costs(c), CycleCount(c), and CycleTime(c) return the costs, cycle count, and cycle time, respectively, by invoking the compiler back-end and hardware modeler.

a configuration is long. Fortunately, there are several methods to reduce the exploration time.

1. Evaluating the configuration only for the most critical procedures of the application. For example, the set of procedures where 99% of the execution time will be spent.
2. Using a rule-based hardware modeler instead of a processor generator to estimate cycle time and costs. Figure 7.3 shows an example of a hardware model file used by the hardware modeler.
3. Removing resources from the initial configuration that have no effect on the cycle count before the actual exploration begins. These resources will not be added during the extend passes.
4. Memorizing all configurations together with their evaluation results in a hash table to prevent unnecessary evaluations.
5. Selecting a neighbor configuration immediately if it has a better quality than the current configuration instead of evaluating all neighbor configurations and selecting the best one. We call this *first fit* instead of *best fit* exploration.

After several reduce and extend passes the explorer determines which configurations are Pareto points and presents them to the designer. The designer selects a few Pareto points that seem to be the most appropriate for his/her situation. The selected configurations are evaluated in more detail by generating processors for them for an accurate cycle time and cost estimation, and evaluating them with other data sets. Finally, the designer has to choose one configuration which is passed to the next stage of the design process: connectivity optimization.

7.1.2 Connectivity Optimization

Connectivity optimization transforms the fully connected configuration found by resource optimization into a partially connected configuration that has less load on the move buses and therefore a shorter cycle time. This is accomplished by removing move bus socket connections from the move buses in a round robin fashion. The move bus socket connection that is removed from a bus is the first connection that has no influence on the cycle count. If no such connection exists we take the connection with the lowest influence on the cycle count. We repeat this process until the cycle time remains constant (limited by other factors than bus load) and the cycle count starts to increase. By removing connections in a round robin fashion we balance the bus load, i.e., all move buses have approximately the same number of connections.

```
define BaseCost =
(
    4.5
);
define MoveBusCost(nbits) =
(
    3.5 + 2.0 * nbits / 32
);
define SocketCost(nbits, nbuses) =
(
    0.02 + 0.1 * nbits * nbuses / 32
);
define RegisterFileCost(nbits, nregs, nwports, nrports) =
(
    nbits * nregs * (0.4 + 0.1 * (nwports + nrports)) / 32
);
define FunctionalUnitCost(operations) =
(
    if operations =~ {'add, 'sub} then 1.0
    else if operations =~ {'mul} then 6.0
    else if operations =~ {'div, 'divu} then 6.0
    else if operations =~ {'shl, 'shr, 'shru} then 3.0
    else if operations =~ {'and, 'ior, 'xor} then 1.0
    ...
    else if operations =~ {'eq, 'gt, 'gtu} then 1.0
    else -1
);

define BaseCycleTime =
(
    20.0
);
define MoveBusCycleTime(nwriters, nreaders) =
(
    12.0 + 0.4 * (nwriters + nreaders)
);
define RegisterFileCycleTime(nbits, nregs, nwports, nrports) =
(
    20.0
);
define FunctionalUnitCycleTime(operations) =
(
    20.0
);
```

Figure 7.3: Excerpt of a hardware model file. The rules for computing the estimated cycle time and costs are written in an expression language called *simple*. Costs are expressed relative to the cost of a 32-bits adder; cycle time is expressed in nano seconds.

Figure 7.1b shows how the execution time of the application depends on the number of removed connections. First, removing connections has a positive effect on the execution time since it reduces the bus load. Next, after sufficient connections have been removed the execution time is no longer determined by bus load. Last, when too many connections are removed the cycle count starts to increase and therefore the execution time as well. These results are presented to the designer who can select a configuration.

Figure 7.4 summarizes the design space exploration trajectory.

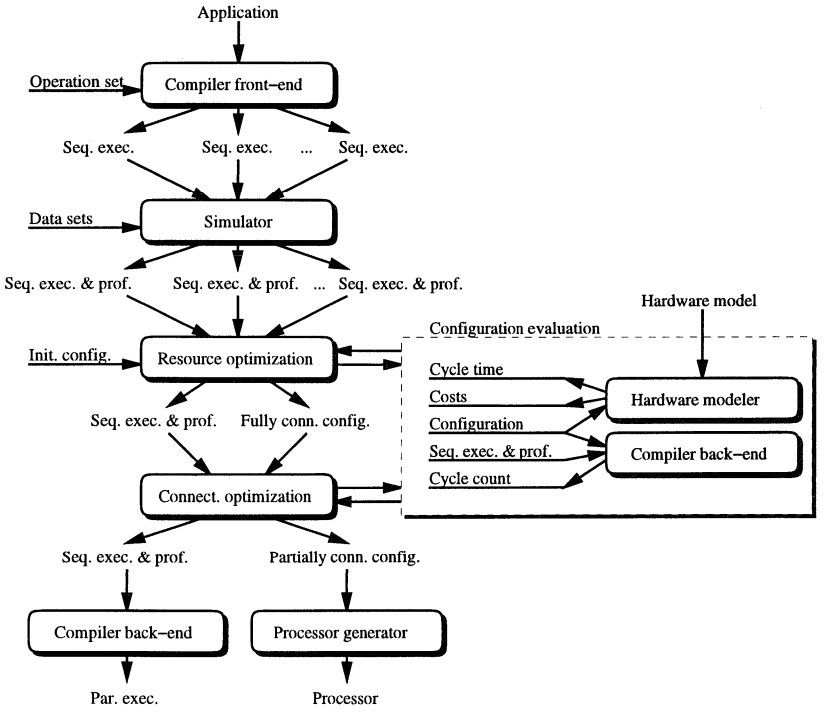


Figure 7.4: The design space exploration trajectory

7.2 Case Study: An ASP for MCCD

In this section we will illustrate the design space exploration method by designing an ASP for minimum cost contour detection (MCCD) [32]. The MCCD algorithm is able to detect a contour in an image. As shown in figure 7.5, it can be used to generate the contour of the heart in an echocardiogram of the heart. This is a useful application in medicine for estimation of the volume of a heart. In practice, images are processed real-time, i.e., 30 images have to be processed

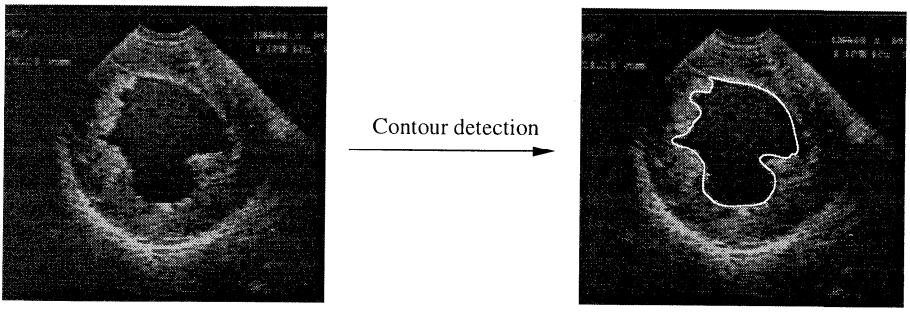


Figure 7.5: Minimum cost contour detection (MCCD)

Characteristic	Value
Lines of C code	2,851
Number of executed operations/image	1,147,561
Average number of operations per basic block	6.51
Percentage add/subtract operations	44.0%
Percentage load/store operations	14.5%
Percentage shift operations	3.1%
Percentage compare operations	12.1%
Percentage logic operations	9.0%
Percentage multiply operations	2.3%
Percentage divide operations	0.0%
Percentage floating point operations	0.0%
Parallelism upper bound based on trace analysis	14.46
Parallelism exploitable by our compiler	3.85
Average number of live integer GPRs	9.41

Table 7.1: Characteristics of the MCCD algorithm

per second. For each image a new contour has to be computed and drawn. The MCCD algorithm uses the contour of the previous image to reduce the search area for the next image.

Table 7.1 lists characteristics of the MCCD algorithm. The parallelism upper bound and average number of live integer GPRs are obtained by means of the instruction trace analysis. As discussed in section 2.1.4, the value of parallelism upper bounds is limited since it is usually far from achievable in practice. The average number of live GPRs gives an indication for the number of required GPRs. The amount of exploitable parallelism was measured by compiling the application for wide TTA with single cycle latencies.

Procedure	Description	Operation count
gvi5	Gray value interpol.	329,741
path	Find min. cost path	144,482
cost	Cost function calc.	108,765
sptr	Spatial transf.	60,176
norm	Normalize costs	46,244
next	Next contour prep.	37,009
ipol	Interpolation	8,738
sqrt	Integer square root	7,168
smth	Contour smoothing	5,565
	Miscellaneous	6,075
Total		747,888

Table 7.2: The operation profile of the MCCD algorithm

```
void gvi5(...)
{
    for(...) {
        sum = 0;

        for(i = -2; i <= 2; i++)
            for(j = -2; j <= 2; j++)
                sum += image[... + i][... + j];

        *results++ = sum;
    }
}
```

Figure 7.6: The original `gvi5` routine

7.2.1 Special Functional Units

Table 7.2 shows the profile of the MCCD application. Procedure `gvi5` is clearly the most critical procedure; 44% of all operations are executed in `gvi5`. Figure 7.6 shows the source code of `gvi5`. It computes the sum of the 8-bit pixel values of a 5×5 matrix within a 512×512 image. It does this for a number of 5×5 matrices. The simplest method to speed up `gvi5` is by unrolling the two inner loops completely. This reduces loop overhead[2]. The result is a chain of 24 additions, 25 byte loads, and 25 address computations.

Further improvement is possible by processing 32-bit words of 4 8-bit pixels instead of one pixel at a time. This technique has been successfully applied in general purpose CPUs, such as the Sun UltraSPARC [94] and HP PA-7100LC [138], to speed up multi-media applications. What we need for `gvi5` is

[2]Loop unrolling is also performed by the back-end. However, the loop unroller of the back-end is not sophisticated enough to detect that the trip counts of both loops are always five times.

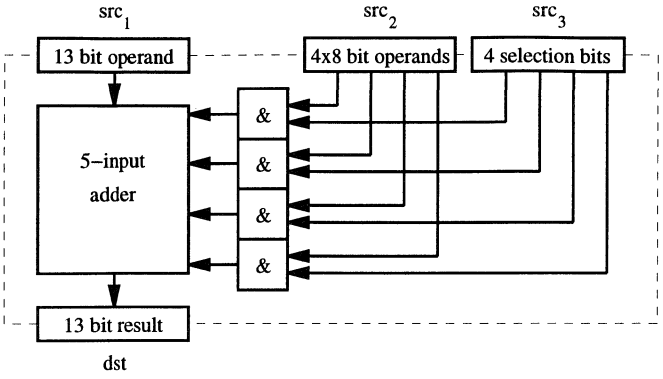


Figure 7.7: An SFU for MCCD

an operation that adds four 8-bit values to a 13-bit partial sum. The additions need to be conditional since only the pixels of the word located within the 5×5 matrix should be added to the partial sum. The result is an operation with the following description:

$$\begin{aligned} dst_{12\ldots0} = src_{1,12\ldots0} &+ \text{if } src_{3,0} \text{ then } src_{2,7\ldots0} \text{ else } 0 \\ &+ \text{if } src_{3,1} \text{ then } src_{2,15\ldots8} \text{ else } 0 \\ &+ \text{if } src_{3,2} \text{ then } src_{2,23\ldots16} \text{ else } 0 \\ &+ \text{if } src_{3,3} \text{ then } src_{2,31\ldots24} \text{ else } 0 \end{aligned}$$

With this operation, which we will call `add4`, we can compute the sum of the 5×5 matrix by means of 10 `add4` operations, 10 word loads, 10 address computations, and a few extra operations to compute the four selection bits. An SFU that implements `add4` is shown in figure 7.7. Its complexity is comparable to a 32-bits adder. Notice that `add4` and therefore the SFU that implements it remains useful when, due to an algorithmic change, `gvi5` is replaced by, for instance, `gvi7` which computes the sum of a 7×7 matrix.

The MOVE framework supports user defined operations but requires that these operations are explicitly coded in the application source code. Figure 7.8 shows how `gvi5` has been modified for `add4` after its two inner loops have been unrolled. A user defined operation is used by means of

```
__userdef_nm__(i, t, o1, ..., on)
```

Where `n` is the number of operands, `m` the number of results, `i` the user defined operation index, `t` a type specifier, and $o_1 \ldots o_n$ the operands of the operation. Type specifier `PURE_FUNCTION` as opposed to `SIDE_EFFECTS` indicates that the operation is free of side effects. User defined operations will usually be used

```
#define add4(x, y, z) __userdef_31__(1, PURE_FUNCTION, x, y, z)

void gvi5(...)
{
    __not_aliased__(image, results);

    for(...) {
        x0 = ... - 2;              /* corner of 5x5 square */
        y0 = ... - 2;

        sel1 = 0x0f >> (x0 & 3); /* sel1 = 1111, 0111, 0011, or 0001 */
        sel2 = 0x78 >> (x0 & 3); /* sel2 = 1000, 1100, 1110, or 1111 */

        x0 &= ~3;                  /* align x0 */

        sum = add4(0,   *(int *) &image[y0 + 0][x0 + 0], sel1);
        sum = add4(sum, *(int *) &image[y0 + 0][x0 + 4], sel2);
        sum = add4(sum, *(int *) &image[y0 + 1][x0 + 0], sel1);
        sum = add4(sum, *(int *) &image[y0 + 1][x0 + 4], sel2);
        sum = add4(sum, *(int *) &image[y0 + 2][x0 + 0], sel1);
        sum = add4(sum, *(int *) &image[y0 + 2][x0 + 4], sel2);
        sum = add4(sum, *(int *) &image[y0 + 3][x0 + 0], sel1);
        sum = add4(sum, *(int *) &image[y0 + 3][x0 + 4], sel2);
        sum = add4(sum, *(int *) &image[y0 + 4][x0 + 0], sel1);
        sum = add4(sum, *(int *) &image[y0 + 4][x0 + 4], sel2);

        *results++ = sum;          /* store result */
    }
}
```

Figure 7.8: The modified `gvi5` routine

via a macro or an inline procedure.

In order to compile and simulate user defined operations, the designer needs to provide five small C++ functions which will be linked with the tools of the MOVE framework. These functions specify (1) how to simulate user defined operations, (2) their names (e.g., `add4`) in machine description and hardware model files, (3) whether the operations are commutative, (4) whether it is allowed to speculate them without proper guarding, and (5) possible dependencies between user defined operations. To illustrate the latter two functions, consider two user defined operations `modify` and `lookup` for accessing a lookup table. Clearly, a `modify` needs to be guarded if it is executed speculatively, and two table lookup accesses are dependent unless both accesses are `lookup` operations.

Figure 7.8 also illustrates the `__not_aliased__` annotation. Without this annotation the memory reference disambiguator is not able to disambiguate the accesses to `image` and `results` which are both incoming procedure arguments. This limits the iteration overlap of the for loop.

Resource		Cost
Functional units:		
• INT	`add, sub, gt, gtu`	1
• MEM	`ld, ldb, ldh, st, stb, sth`	11
• LOG	`and, ior, xor, eq`	1
• SHF	`shl, shr, shru`	3
• MUL	`mul`	6
• SFU	`add4`	1
Move buses		$3.5 + 0.0625W$
Register files		$(0.0125 + 0.003125P)WN$
Sockets		$0.02 + 0.003125WC$
Base (instruction fetch unit, etc.)		4.5

Table 7.3: The hardware cost model. W stands for number of bits, N for number of registers, P for number of ports, and C for number of connections.

By adding a user defined operation to the operation set, the operation count of `gvi5` is decreased by 57% from 329,741 to 144,281 and the total operation count is decreased by 25% from 747,888 to 562,428. This illustrates the power of SFUs; a significant operation count reduction can be achieved by means of inexpensive application specific SFUs.

7.2.2 Resource Optimization

The design exploration starts with compiling the MCCD application to a number of sequential executables with different operation sets and determining an initial oversized configuration. Based on the application characteristics shown in table 7.1, sub-word load and store operations should be included, emulating them would be too expensive, and divide and floating point operations can be excluded. For multiply and the user defined `add4` operations it is not clear whether they should be included. Therefore we compile the application to four operation sets, one without `mul` and `add4`, one with `mul`, one with `add4`, and one with both `mul` and `add4`.

We shall assume six FU types, INT for integer operations, MEM for load and store operations, LOG for logic operations, SHF for shift operations, MUL for multiply operations, and SFU for `add4` operations. The latency of the MEM FU is 2 cycles, the latency of MUL FU is 4 cycles, and all other FUs have a single cycle latency. All FUs are VTLP pipelined. Furthermore, we assume a jump latency of 2 cycles and guard expressions of 2 booleans. As initial configuration we shall use a TTA with 16 32-bit move buses, 3 INT FUs, 3 MEM FUs, 1 MUL FU, 2 LOG FUs, 2 SHF FUs, 2 SFU FUs, a 4W+4R ported integer RF of 64 registers, and a dual ported boolean RF of 8 registers.

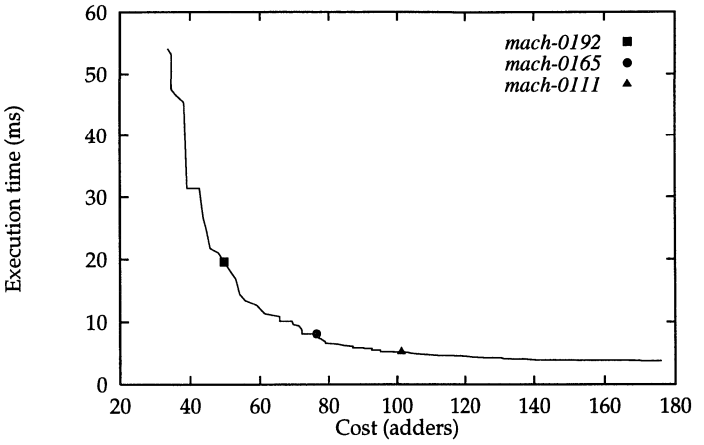


Figure 7.9: The result of resource optimization

Table 7.3 shows the used hardware cost model. This model is based on results produced by a processor generator for 1 micron CMOS technology [58]. The costs are expressed relative to a 32-bit integer FU. As cycle time model we use:

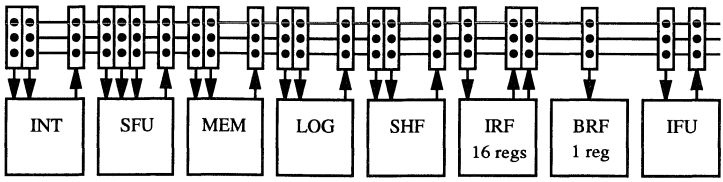
$$T(config) = \max\{20, \max\{12 + 0.4 \times connections(b) \mid b \in buses(config)\}\}$$

This means that cycle time is only determined by the number of connections on the move buses and has a minimum value of 20ns.

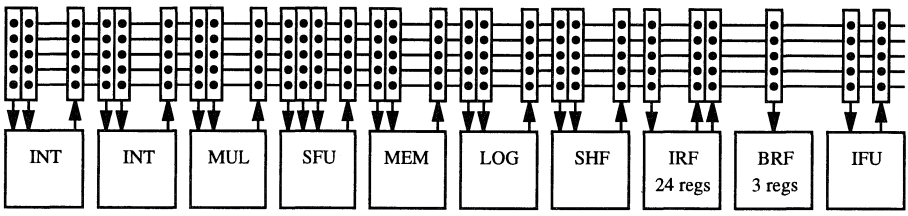
Figure 7.9 shows the result of resource optimization. 76 Pareto points are presented to the designer varying from a configuration with cost 33.6 that processes a frame in 2.7M cycles with 0.51 operations per cycle to a configuration with cost 175.9 that processes a frame in 188K cycles, with 3.23 operations per cycle. Computing this curve required about 1.5 hours on a 47 SPECint92 workstation (1383 evaluations of about 3.5 seconds). All procedures for the computation of a frame were included for the evaluation of a configuration.

Three configurations, named *mach-0192*, *mach-0165*, and *mach-0111*, are marked in figure 7.9 and are shown in figure 7.10. These three configuration differ in the number of move buses, the number of INT FUs, the presence of a MUL FU, the number of integer registers, the number of boolean registers, and the number of ports on the integer RF. Because *mach-0192*, unlike the other two configurations, does not contain a MUL FU, it requires another executable than the other two configurations.

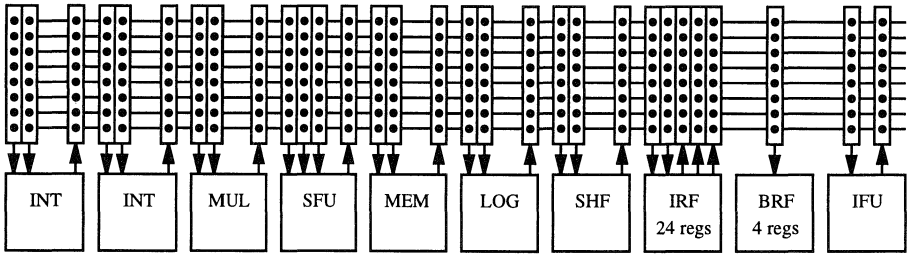
Which configuration the designer will choose depends on the design situation. In case of cost or performance constraints, the designer will simply choose the fastest/cheapest configuration that meets the cost/performance constraints. In other situations the decision will be based on the price the designer is willing



(a) mach–0192: cycle count: 980,465; operations per cycle: 1.18; costs: 49.7



(b) mach–0165: cycle count: 408,073; operations per cycle: 1.46; costs: 76.4



(c) mach–0111: cycle count: 261,197; operations per cycle: 2.34; costs: 101.2

Figure 7.10: Three configurations resulting from resource optimization. IRF stands for integer RF, BRF for boolean RF, and IFU for instruction fetch unit.

to pay for more performance.

7.2.3 Connectivity Optimization

Let us assume the designer chooses for the *mach-0111* configuration shown in figure 7.9c. The next step is connectivity optimization. The result is shown in figure 7.11. The designer has to choose a configuration on the flat part of the curve. The two extremes are the configuration where the cycle time stops falling (100 connections removed) and the configuration where the cycle count starts to rise (160 connections removed). The first configuration is slightly more expensive than the latter (110 vs. 105). On the other hand the first configuration is better proof against changes in the application that change the trans-

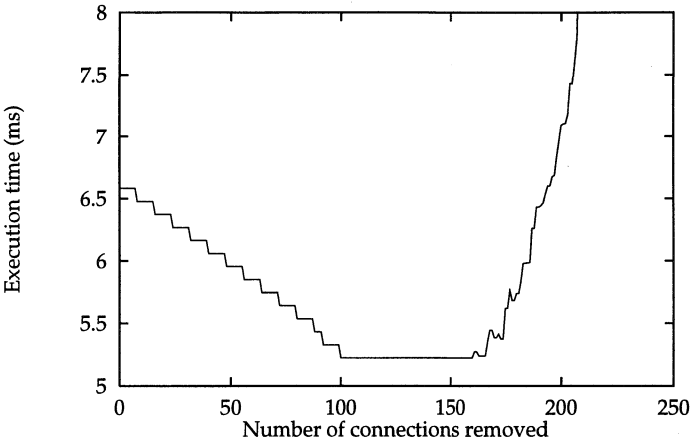


Figure 7.11: The result of connectivity optimization

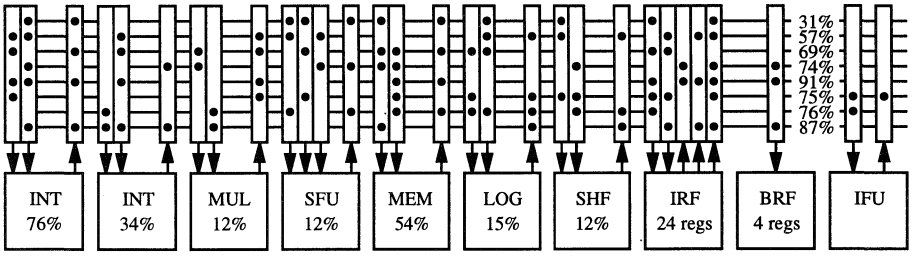


Figure 7.12: A configuration resulting from connectivity optimization

port requirements. The designer can therefore trade-off cost for flexibility.

Figure 7.12 shows *mach-0111* where 140 connections have been removed. FUs and move buses are shown with their utilization obtained by means of simulation of the parallel code. Notice that FUs with a high utilization (the left INT and MEM FUs) have more connectivity to the interconnection network than FUs with a low utilization (the MUL and SHF FUs).

7.2.4 Miscellania

In order to get confidence in our local search algorithm for resource optimization, and in the spirit of Karl Popper's falsification [99], we have implemented a search algorithm based on simulated annealing [126] that tries to maximize the above mentioned quality function for a given α and β. Simulated annealing has been used by others for design space exploration, for example [20, 52, 113].

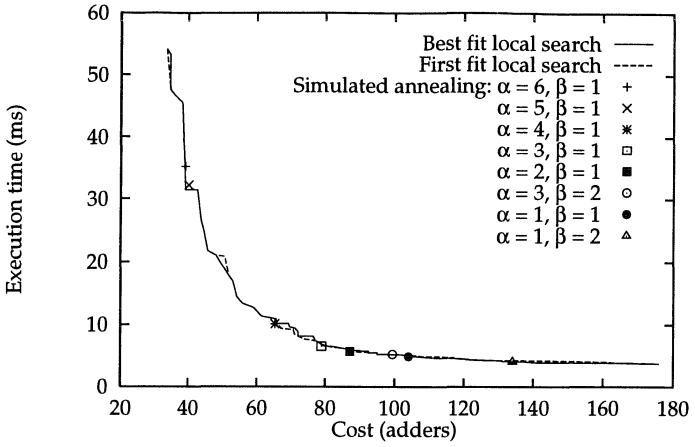


Figure 7.13: Best fit vs. first fit vs. simulated annealing

The results of searching by means of simulated annealing for eight different values for α and β are shown in figure 7.13. All points found by simulated annealing were also found by the local search algorithm which gives us confidence in our local search algorithm.

Figure 7.13 also shows the difference between best fit and first fit local search. The difference is very small. First fit required 856 evaluations instead of 1383, and found 65 Pareto points instead of 76[3]. We have found similar results with other cases [107]. This has led us to the conclusion that first fit quite useful to reduce exploration time.

7.2.5 Limitations

The current tools have several limitations that have to be removed to make them more valuable. The following items need to be incorporated: caches, code size, and real time constraints.

Caches may have a large impact on the cost and performance of a processor. Therefore, cache parameters, such as number of lines, line size, and associativity, have to be considered during resource optimization. The problem with including cache parameters is that cache performance is dependent on how the code is scheduled. Speculative execution of load operations affects the data cache performance [151], and code duplication affects the instruction cache performance. Furthermore, it is not possible to predicted cache performance accurately by means of some sort of profiling of the sequential executables. For accurate performance estimation, the parallel code produced by the back-end

[3]Notice that the number of found Pareto points does not say much about the quality of the exploration algorithm.

has to be simulated for every configuration evaluation. This is usually far too expensive.

Code size is often an important issue for ASPs. It depends on compiler parameters which control function inlining and loop unrolling, and on the utilization of move slots. The latter depends on the discrepancy in ILP of the application and ILP provided by the hardware. To take code size into account, the backend of the compiler should be able to report the code size, the hardware modeler should be able to compute the cost of code size, and the explorer should be able to change compiler parameters that affect code size.

Many ASPs have to guarantee real-time response to external events. This requires that the compiler can guarantee minimal and maximal execution times of certain execution paths. This is in general an undecidable problem. The only way to guarantee real-time responses with the current tools is by means of manual verification of Pareto points found by resource optimization.

7.3 Related Work

Developing tools and methodologies for designing or *synthesizing* hardware and software for a specific application is the objective of many research projects in the industry and academia. This section describes briefly some related work in this area that is known to the author. This work can be divided into three categories: templated ASPs, hardware/software co-design, and high-level synthesis.

Templated ASPs

Breternitz and Chen divide the ASP design process into specification and implementation optimization [36,37]. Specification optimization determines a VLIW configuration with a central RF for the given application. The designer explores the design space manually by compiling the application for configurations with various architectural parameter values. Implementation optimization replaces the central RF by a set of single-ported RFs and a partially connected interconnection network between the RFs and FUs in order to make a cost effective implementation possible. To do this variables should be allocated to RFs such that variables referenced in the same cycle (according to the schedule produced in the specification optimization step) are assigned to different RFs. A graph coloring based algorithm is used for this allocation problem that minimizes the required number of RFs.

The SCARCE project [153,164], the work of Potasman [165], the CASTLE project [40,183], and the LIFE project [129,130] are examples of frameworks for ASP design that provide tools for processor generation and/or code generation for these processors based on an architectural template but do not pro-

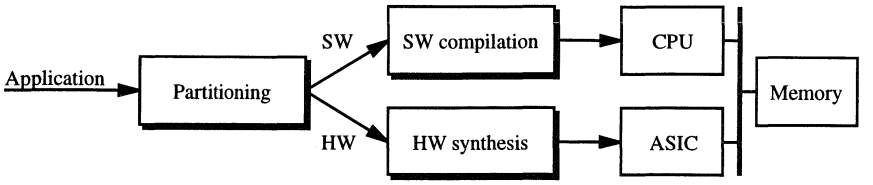


Figure 7.14: Hardware/software co-design

vide tools or methodologies to explore the design space. They leave this task to the designer.

Hardware/software co-design

In the hardware/software (HW/SW) co-design community one partitions an application into a software component, that runs on a standard processor, and a hardware component that is implemented in an ASIC (application specific integrated circuit) [63]. The processor and the ASIC are usually connected via the memory bus of the processor (see figure 7.14); which can be a serious communication bottleneck if there is a lot of communication between the CPU and the ASIC [76]. Two systems are known to the author where the partitioning is performed automatically; the Cosyma system [76] and the Vulcan system [97, 98]. Both systems start with an initial partitioning which is repeatedly evaluated and modified until the partitioning meets the design constraints. The initial partitioning of the Vulcan system is a hardware only partitioning, while the initial partitioning of the Cosyma system is a software only partitioning. Both systems are programmed in a C like programming language.

Razdan and Smith [173], Athenas and Silverman [18], and others describe systems where standard processors are augmented with programmable FUs (PFUs) based on field programmable gate arrays (FPGAs). PFUs are similar to SFUs in that they are used to improve performance by providing special operations for the critical parts of the application. Unlike SFUs, PFUs can be reprogrammed at run-time by downloading a new image into their FPGAs. Primary disadvantage of this approach is the relatively slow FPGA technology and the time required to reprogram FPGAs.

High-level synthesis

High-level synthesis (HLS) is an active area of research where one translates a behavioral description of an application, usually written in a hardware description language such as VHDL, into a structural description, usually a netlist of a data path [41, 63, 87]. This process consists of several partitioning, scheduling, allocation, and library binding steps. The resulting data path is

Issue	HLS	HW/SW co-design	Templated ASPs
Problem size	small	large	large
Programming language	special	general	general
Pipelined control	no	yes	yes
Interruptible systems	no	yes	yes
Communication overhead	-	high	zero
Functionality overhead	zero	medium	medium
Reprogrammable	no	yes	yes

Table 7.4: Primary differences between three methods for system design

controlled by a finite state machine or a microcode engine that is generated after the data path has been synthesized and the application has been scheduled for the data path.

Comparison

Table 7.4 lists the primary differences between three major methods for system design: HLS, HW/SW co-design, and templated ASPs.

1. **Problem size.** Current HLS systems are not able to handle large applications, e.g., a postscript interpreter. The other two design methods do not have problems with large applications.

2. **Programming language.** HLS designs are usually specified in a hardware description language, such as VHDL, or a special purpose language, such as Silage [193]. Designs for the other two methods are usually programmed in a (software) HLL, such as Pascal, C, or C++. These languages provide powerful constructs, e.g., dynamically allocated memory and have a larger user base.

3. **Pipelined control.** HW/SW co-design and templated ASP systems are based on Von Neumann processors whose control can be pipelined in order to achieve a faster cycle time. The controller of an HLS system waits for control signals from the data path before it proceeds to the next controller stage or microcode instruction. There is no equivalent of branch latency in an HLS system.

4. **Interruptible systems.** HW/SW co-design and templated ASP systems are interruptible. HLS systems have to use some form of polling to mimic this functionality.

5. **Communication overhead.** The overhead of the communication between the hardware part of a HW/SW co-design system, that implements the critical sections of the application, and the software part can be

fairly high [76]. In templated ASPs there is no overhead in communication between special FUs (comparable to the hardware part of a HW/SW co-design system) and the general FUs (comparable to the software part). In HLS systems there is no such separation.

6. **Functionality overhead.** HLS designs can be very small. This is due to the fact that there is no functionality overhead. HW/SW co-design systems always contain a standard processor, while templated ASPs always contain some extra hardware that is part of the template, e.g., the program counter and the instruction register.

7. **Reprogrammability.** HW/SW co-design and templated ASP systems are reprogrammable when their instructions are stored in RAM or external ROM. This is very useful when design specifications are not yet stabilized or the system has to be used in other products with (slightly) different specifications.

The overall conclusion can be that the advantage of HLS is the absence of functionality overhead. HW/SW co-design and templated ASPs have a lot in common, they can handle large problems, they are programmed by general programming languages, they provide pipelined control and interrupts, and they can be made reprogrammable. The main difference is the communication overhead between the general part and the application specific part of the system.

8 Conclusions

This chapter concludes this thesis. Section 8.1 provides a summary, section 8.2 describes the current status of the compiler, section 8.3 puts this work in perspective, and section 8.4 describes possible future research.

8.1 Summary

Transport triggered architectures (TTAs) push one on the main principles of RISC and VLIW architectures to its limits: do what you have to do in hardware and let the compiler do the rest. The task of controlling the data transports between the functional units (FUs) and register files (RFs) has been shifted from the hardware to the compiler. TTAs are therefore programmed by specifying data transports instead of operations as is done by traditional *operation triggered architectures* (OTAs). This results in better control over the hardware and better utilization of the transport buses and register file ports, which are critical resources in instruction-level parallel processors. A better utilization improves the cost/performance ratio.

Whether migrating responsibilities from hardware to the compiler is a good idea depends on how well the compiler can handle these responsibilities. The work described in this dissertation demonstrates that it is very well possible to develop and implement a compiler with state-of-the-art compilation techniques for TTAs. The main features of the developed compiler are:

1. Based on GNU compiler of the Free Software Foundation with front-ends for ANSI C, C++, and Fortran 77.
2. Applies extended basic block scheduling in order to exploit inter ba-

sic block parallelism. The used scheduling scope consists of regions, which are single entry, acyclic control flow graphs. Speculative execution, guided by profiling information, is applied to increase parallelism. Guarded execution is applied to facilitate speculative execution. Multi-way branching is applied to improve performance of branch intensive code.

3. Applies software pipelining in order to exploit inter-iteration parallelism. The used software pipelining algorithm belongs to the class of modulo scheduling algorithms and uses if-conversion to software pipeline multi basic block loops. Delay lines are used to deal with long living variables defined within loops that limit the initiation interval of the software pipeline.

4. Highly parameterized. All information about the target TTA is specified in a machine description file. Furthermore, the scheduler offers support for user defined operations provided that these operations are explicitly specified in the source code. These operations are very useful within a design system for application specific processors (ASPs).

5. All TTA specific optimizations are implemented. These are: dead result move elimination, bypassing, operand sharing, operand swapping, and port sharing.

6. Register allocation is performed before scheduling in such a way that the register allocator tries to prevent false dependences that limit the freedom of the scheduler.

7. Functional unit, socket, and immediate field assignments are performed during scheduling. Move bus assignment is performed after scheduling; only the possibility of a move bus assignment is checked during scheduling. This makes it possible to schedule efficiently for TTAs with an irregular interconnection structure.

The compiler has been implemented within a period of approximately 2.5 years by the author. Another 0.5 to 1 year is required to make it operational. The key to a successful development of a compiler for TTAs is making the right trade-offs between expected performance and engineering complexity. Several times we rejected design options because of their complexity and chose for easier to implement options with a lower performance. A few of these cases are reported in chapters 3, 4, and 5. The major design decisions made during the development of the compiler along with their motivations are listed in figure 8.1.

Figure 8.2 shows the progress of the scheduler during its development. It illustrates the step-by-step development. Performance improvements were made every time an optimization or enhancement, such as function inlining, loop unrolling, multi-way branching, or software pipelining, was implemented.

1. The GNU compiler as front-end
The GNU compiler produces production quality code, is stable, and provides front-ends for ANSI C, C++, and Fortran. Drawback is the limited amount of information it can provide for memory reference disambiguation.

2. Register allocation before scheduling
Register allocation during scheduling was considered too complex. Register allocation after scheduling is problematic since: (1) scheduling may increase register pressure, (2) live variable analysis for guarded code is not trivial, and (3) inserting spill/reload code into scheduled TTA code is problematic.

3. FU, socket, and immediate field assignments during scheduling
Delaying all assignments until scheduling has been performed is too complex. Furthermore, in practice there is usually not much to choose, e.g., the number of FUs that support a particular operation is usually one or two.

4. Move bus assignment after scheduling
Necessary for efficient code generation for irregular interconnection networks.

5. Scheduling all moves of an operation in one atomic step
Scheduling the moves of an operations one-by-one interleaved with moves from other operations may lead to inefficient FU usage and scheduling deadlocks. Scheduling heuristics and deadlock prevention mechanisms would be required to prevent this.

6. Operation based list scheduling
A direct consequence of decision 5. Trigger and results moves of an operation have to be scheduled in different instructions. This is not compatible with instruction based list scheduling where one instruction is filled at a time.

7. No scheduling heuristics to stimulate TTA specific optimizations
Attempts to develop these heuristics failed; the average improvement was not worth the complexity. The problem is combining these heuristics with the critical path heuristic of list scheduling.

8. Region scheduling scope for extended basic blocks scheduling
A region is the most general scheduling scope of all scheduling scopes known from the literature. Multiple path parallelization is realized without code duplication prior to scheduling (tail duplication in superblock and decision tree scheduling) and without the if-conversion requirement of hyperblock scheduling.

9. Modulo scheduling type software pipelining
Modulo scheduling has been chosen because: (1) it has been worked out very well, (2) previous positive experience with modulo scheduling, and (3) its performance in comparison with other algorithms. Enhanced pipeline scheduling, an attractive alternative, is not compatible with operation based list scheduling.

Figure 8.1: Major design decisions and motivations

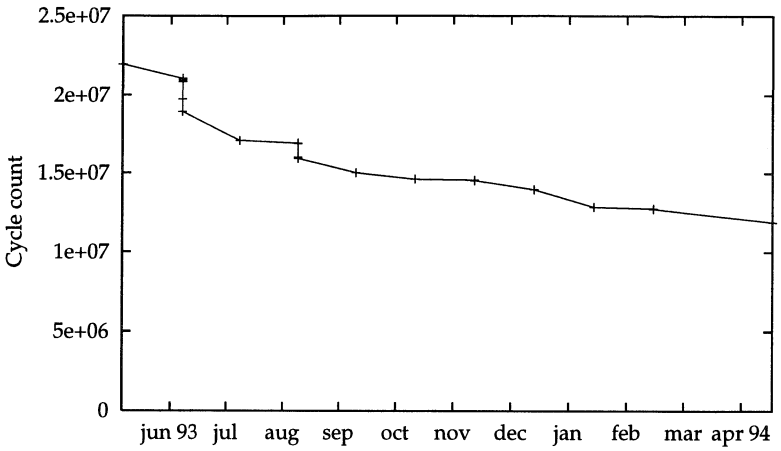


Figure 8.2: Progress of the scheduler during its development. The graph shows the cycle count of the stanford benchmark. This benchmark was frequently used during the development of the scheduler to verify its operation and to evaluate design alternatives.

With a compiler for TTAs at our disposal, we were able to perform experiments to evaluate TTAs and features of TTAs and we were able to develop a method for design space exploration. The results of the performed experiments are summarized in table 8.1.

The design space exploration method, to find an 'optimal' configuration for a given application, consists of two steps as shown in figure 8.3: resource optimization and connectivity optimization. Resource optimization computes a set of so called Pareto points, each corresponding to a configuration with a certain cost and performance for the given application. Each Pareto point is optimal in the sense that resource optimization could not find other configurations that both cost less and have a better performance. The Pareto points computed by the developed design exploration tool vary in the number of move buses, functional units, general purpose registers, and register file ports. All Pareto points have a fully interconnected network. The set of Pareto points is presented to the user who selects a configuration that meets his/her requirements. Next, the selected configuration is passed to the second pass of design space exploration: connectivity optimization.

Connectivity optimization determines the connectivity for the configuration selected in the resource optimization step. It does this by removing connections in an order such that the performance loss is minimal and the bus load is balanced. The result is a sequence of configurations with an increasing number of removed connections. The user is asked again to select a configuration.

The design space exploration method has been demonstrated by designing a

Experiment	Results and conclusions
Speedup	The average speedup of a 12 move bus configuration relative to a single move bus configuration is 4.30 for workstation-type applications and 6.38 for DSP-type applications. Average number of operations initiated per cycle: 2.52 (WS) and 3.28 (DSP). Average number of operations in execution per cycle: 3.95 (WS) and 5.90 (DSP).
Scheduling scope	Basic block scheduling cannot utilize more than 4 move buses. Extended basic block scheduling gives a performance improvement over basic block scheduling of 79%. Software pipelining gives another 3.5% improvement.
Scheduling freedom	Valuable when transport resources are constraining the performance. The performance improvement is 2 – 10% for TTAs with a small number of move buses.
TTA specific opt.	Valuable when transport resources are constraining the performance. Dead result move elimination: 4 – 10% improvement. Operand swapping and operand sharing: 0 – 2% improvement.
RF port requirement	The average RF port requirement per operation is 0.63 read ports and 0.37 write ports. A 3W+3R configuration is sufficient to perform 2.77 operations per cycle on average.
Partitioned RFs	The presented method seems to work well. RFs can be partitioned in order to reduce their costs without a significant performance loss.
Guarding	Multi-way branching: 4.9% improvement. And/or guard expressions instead of simple guard expressions: 6.4% improvement. A 2 – 3 boolean RF is sufficient.
FU pipelining	VTLP is preferable; easier to implement and up to 4.5% faster than hybrid pipelining.
Memory ref. disamb.	7.5% performance improvement in comparison with no memory reference disambiguation. Up to 7.8% improvement can be achieved by a better memory reference disambiguator.
Multicasts	5 – 9% performance improvement when transport resources are constraining the performance.
Partial connectivity	With the bipartite matching move bus assignment algorithm it is possible to remove a large percentage of the connections before the cycle count starts to increase. 47% percent of the connections of a fully connected configuration can be removed before the cycle count increase becomes more than 1%.
Bypass conflicts	1.4% performance degradation on average.
Register allocation	The used method to prevent false dependences introduced by register allocation before scheduling seems to work well. The average cycle count increase caused by register re-use *and* spilling is 1.3% for 48 registers, and 2.6% for 32 registers.

Table 8.1: Results of the experiments described in chapter 6 and the conclusions that can be drawn from them (copy of table 6.8 on page 128)

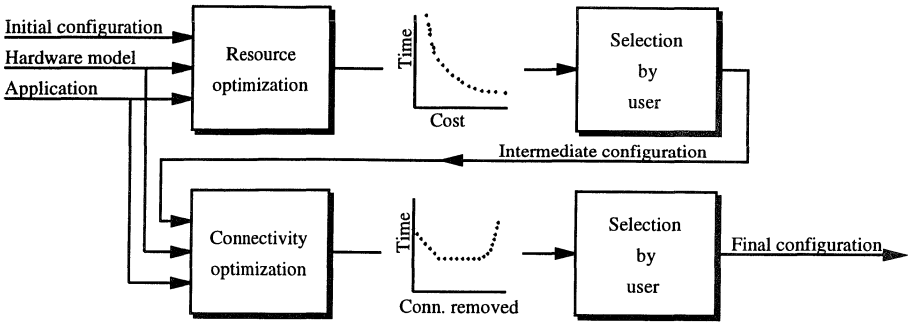


Figure 8.3: Design space exploration

TTA for the minimum cost contour detection application.

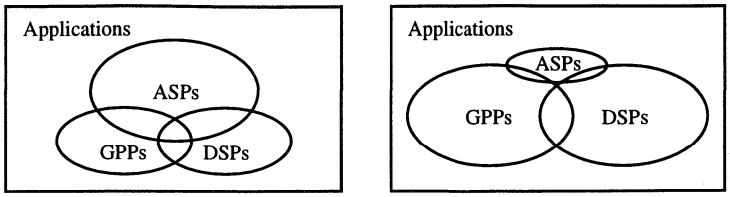
8.2 Current Status of the Compiler

The developed compiler is nearly operational. What is missing is the final step in the compilation process: generating an executable binary. The current version of the compiler generates a textual representation of the scheduled code. Generating a binary involves low-level issues such as instruction encoding and relocation of code and data. None of these issues should present new problems. Besides this work, other partners within the MOVE project are currently (1) developing a graphical user interface for the MOVE framework tools, (2) developing heuristics for multiple RF register allocation, (3) implementing static branch prediction techniques, (4) integrating the compiler and design space explorer with the processor generator tools, and (5) researching issues listed in section 8.4.

The work described in this thesis has drawn the attention of several industrial R&D groups; among them are: Océ (Venlo, NL), HP labs (Bristol, UK), TNO FEL (The Hague, NL), GMD (Bonn, DE), SPASE (Nijmegen, NL), and Philips Natlab (Eindhoven, NL). Several of them are interested in a fully operational compiler together with other MOVE framework tools.

8.3 Perspective

In this dissertation we advocated templated ASPs and using TTAs as foundation for an ASP design system. Nevertheless, it is good to think critically about the questions whether it makes sense to use ASPs instead of standard processors and to use TTAs instead of OTAs.



(a) Positive scenario (b) Negative scenario

Figure 8.4: Venn diagrams illustrating the territories of ASPs, GPPs, and DSPs. Membership of a processor type class represents that this processor type is most cost effective for the application.

First, does it makes sense to use ASPs instead of standard general purpose processors (GPPs) or digital signal processors (DSPs)? The answer is simple; it only makes sense to use an ASP if an ASP has a better cost/performance. ASPs win cost/performance by providing the hardware for an application that is missing in a standard processor (e.g., special FUs) and omitting the hardware that can be omitted (FP FUs, memory management, etc.). On the other hand, ASPs lose cost/performance on the point of hardware efficiency. The market for a GPP or DSP is usually much larger than for an ASP which makes it possible to put much more effort in the design process. Large parts of the design can be done manually by an experienced design team. The design of an ASP has to be done by means of synthesis tools in order to limit the design effort. The produced quality of these tools is usually significantly lower than (largely) handcrafted designs [58]; the consequence is a longer processor cycle time and a larger die area, i.e., a lower cost/performance. Whether the cost/performance gain of ASPs will be larger than the loss will depend on the 'exoticness' of the application for which the ASP is intended; the more an application deviates from standard (SPECmark-type) applications for which standard processors are designed, the higher the probability that it is cost effective to use an ASP. Case studies have to determine the territories of ASPs, GPPs, and DSPs. These territories are illustrated in figure 8.4. Figure 8.4a shows a positive scenario, from the ASP point of view, where ASPs have the best cost/performance for most of the applications, while figure 8.4b shows a less positive scenario.

The second question, are TTAs a good idea and are they 'better' than VLIW OTAs, is also hard to answer. Figure 8.5 tries to illustrate the differences in cost and performance between TTAs and OTAs as function of the number of FUs, i.e., the amount of ILP. The cycle time of TTAs is expected to increase slower than the cycle time of OTAs as FUs are added; see figure 8.5a. This is motivated by the facts that TTAs have a simpler control and have a lower RF port requirement. The same holds for the costs of TTAs and OTAs. TTAs are easier to design, require less control hardware, fewer buses, and fewer RF ports. Therefore, the cost of TTAs will increase slower than the cost of OTAs; see figure 8.5b. The cycle count of an application is likely to be slightly lower for OTAs than

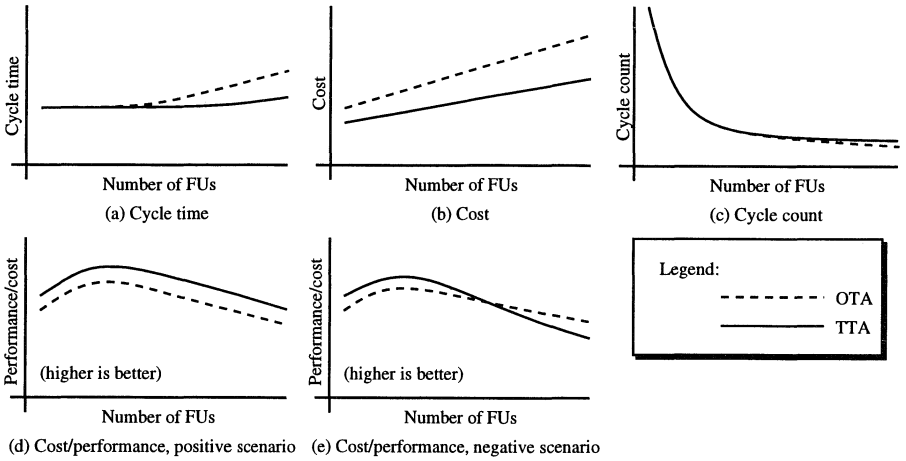


Figure 8.5: Cost, performance, and cost/performance of TTAs and OTAs

for TTAs for configurations with a large number of FUs; see figure 8.5c. This is due to bypass conflicts (1.4% performance degradation on average, see section 6.2.12), and the complexity of code generation for TTAs. When the *same* effort is put into the development of a TTA compiler as in the development of an OTA compiler, it is likely that the OTA compiler will produce better results. This is because an OTA compiler is easier to implement and to debug and it is easier to fine-tune scheduling heuristics, to analyze scheduling inefficiencies, and to extend it with new optimizations.

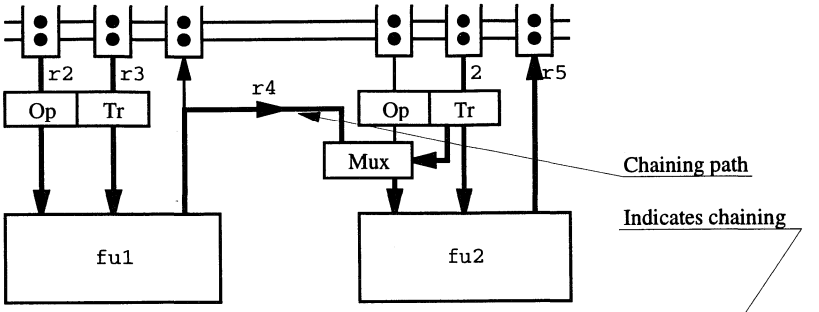
Most of above mentioned contributions to cost and performance are hard to quantify. Therefore, it is hard to state something relevant about differences in cost/performance between OTAs and TTAs. For a small number of FUs, the cost/performance of TTAs will be better. However, for a large number of FUs it is hard to determine whether the cost and cycle time benefits of TTAs will be more than the cycle count disadvantage (figure 8.5d) or less (figure 8.5e).

8.4 Future Work

The work described in this dissertation can be extended in several directions:

1. **Enhancing the software pipeliner.** Possible extensions are: pipelining loops with multiple backward and exit control flow edges, pipelining multiple basic block loops with multiple initiation intervals [203], and performing loop unrolling prior to software pipelining.

2. **Increasing the amount of exploitable ILP.** This can be achieved by better memory reference disambiguation, various code transformations, more




```
Seq. code:  r2 -> add_o; r3 -> add_t; add_r -> r4    /* r4 = r2 + r3 */
            r4 -> shl_o; 2 -> shl_t; shl_r -> r5     /* r5 = r4 << 2 */
Par. code:  r2 -> fu1.add_o; r3 -> fu1.add_t; 2 -> fu2.shl_t*
            fu2.shl_r -> r5
```

Figure 8.6: Chaining. Results of operations executing on FU `fu1` can be passed via a chaining path and a multiplexor to `fu2`. This makes it possible to schedule the `add` and `shl` operations in the same cycle. The multiplexor is controlled by the trigger move; the trigger move specifies whether the 'left' operand comes from the operand register or from `fu1`.

flexible code motions, and better scheduling heuristics.

3. **Evaluating read/write RF ports.** A bidirectional read/write RF port for both read and write accesses has approximately the same costs as a unidirectional read or write RF port but provides more functionality. Scheduling for TTAs with read/write RF ports gives no new problems; it has already been implemented in our compiler. Initial results for the benchmarks listed in table 6.2 and the architectural parameters listed in table 6.1 are encouraging:

RF Ports	Configuration	RF Port type	Speedup over 1W+1R
3	1W+2R	unidirectional	1.143
3	3WR	bidirectional	1.344
4	2W+2R	unidirectional	1.341
4	4WR	bidirectional	1.413

4. **Investigating multicasting.** We have performed initial experiments to quantify the potential benefit of multicasting and we have briefly described how multicasts can be incorporated in the scheduler. However, several questions regarding multicasts are not answered, such as which multicasts are required, how does it affect guarded execution, how does it affect the move bus assignment, and is it cost effective.

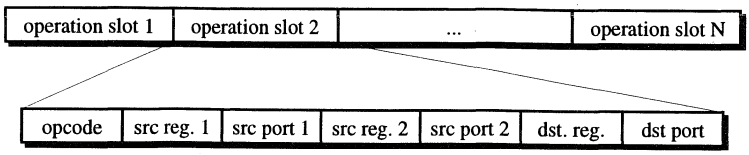


Figure 8.7: Instruction format of an OTA with controllable RF access

5. **Investigating chaining.** Chaining refers to passing data between flow dependent operations without latching it in a register. The goal is to reduce the latency of chains of flow dependent operations. For example, two single cycle flow dependent operations can be scheduled in the same cycle if the time to do both operations after each other is less than the cycle time. Figure 8.6 explains how chaining might be implemented in TTAs. So far, chaining has been used by the high-level synthesis community [41, 87] and for dynamically scheduled superscalar processors [146, 186], but not for statically scheduled VLIWs.

6. **Enhancing the design space exploration tool.** Including cache parameters, code size, and real-time aspects makes the exploration tool more valuable.

7. **Providing compiler feedback.** Feedback about resource utilization, critical paths, and ambiguous memory references in a readable form can help the user to rewrite his/her application in order to improve exploitable ILP.

8. **Investigating a best-of-both-worlds architecture.** Figure 8.7 shows the instruction format of an OTA where each register specifier is accompanied with an RF port specifier. This specifier indicates the RF port through which the register should be accessed. A port specifier belonging to a register operand contains a *null* value if the operand does not need an RF access because it is bypassed. Similarly, a port specifier belonging to a register result contains a null value if it does not need an RF access because the write back is dead. Such an architecture has an RF port requirement very close to the RF port requirement of a TTA. The major reasons that TTAs have a lower RF port requirement (average number of register operands per operation, bypassing, dead write backs, and socket sharing; see table 6.5) hold for this architecture as well. However it does not suffer from bypass conflicts and generating code for it is not significantly more complex than generating code for traditional OTAs.

Bibliography

[1] ABNOUS, A., AND BAGHERZADEH, N. Architectural Design and Analysis of a VLIW Processor. Tech. Rep. UCI ICS-TR-92-79, University of California at Irvine, Information and Computer Science Department, Irvine, CA, 92717, July 1992.

[2] ABRAHAM, S. G., ET AL. Predictability of Load/Store Instruction Latencies. In *Proceedings of the 26th Annual International Workshop on Microprogramming* (Austin, Texas, Dec. 1993), pp. 139–152.

[3] AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers: Principles, Techniques and Tools*. Addison-Wesley Series in Computer Science. Addison-Wesley Publishing Company, Reading, Massachusetts, 1985.

[4] AHUJA, P. S., CLARK, D. W., AND ROGERS, A. The Performance Impact of Incomplete Bypassing in Processor Pipelines. In *Proceedings of the 28th Annual International Workshop on Microprogramming* (Ann Arbor, Michigan, Nov. 1995), pp. 36–45.

[5] AHUJA, R. K., MAGNANTI, T. L., AND ORLIN, J. B. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[6] AIKEN, A., AND NICOLAU, A. Perfect Pipelining: A New Loop Parallelization Technique. Tech. rep., Cornell University, Department of Computer Science, Cornell University, Ithaca, NY 14853, USA, Oct. 1987.

[7] AIKEN, A., AND NICOLAU, A. Optimal Loop Parallelization. In *Proceedings of the SIGPLAN'88 conference on Programming Language Design and Implementation* (Atlanta, Georgia, June 1988), pp. 308–317.

[8] ALLAN, V. H., JONES, R. B., LEE, R. M., AND ALLAN, S. J. Software Pipelining. *ACM Computing Surveys 27*, 3 (Sept. 1995).

[9] ALLAN, V. H., RAJAGOPALAN, M., AND LEE, R. M. Software Pipelining: Petri Net Pacemaker. In *Proceedings of the IFIP Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism* (Orlando, Florida, Jan. 1993).

[10] ALLEN, J. R., ET AL. Conversion of Control Dependence to Data Dependence. In *Proceedings of the 10th ACM Symposium on Principles of Programming Languages* (Jan. 1983), pp. 177–189.

[11] ALPERT, D., AND AVNON, D. Architecture of the Pentium Microprocessor. *IEEE Micro 13*, 3 (June 1993).

[12] ALVERSON, R., ET AL. The Tera Computer System. In *Proceedings of the 1990 International Conference on Supercomputing* (Juny 1990).

[13] AMBROSCH, W., ET AL. Dependence-Conscious Global Register Allocation. In *Programming Languages and System Architectures* (Zürich, 1994), J. Gutknecht, Ed., Springer LNCS 782, pp. 125–136.

[14] *AM29000 32-bit Streamlined Instruction Processor.* Sunnyvale, California, Feb. 1988.

[15] ANANTHA, K., AND LONG, F. Code Compaction for Parallel Architectures. *Software Practice & Experience 20*, 6 (June 1990).

[16] ANDO, H., ET AL. Unconstrained Speculative Execution with Predicated State Buffering. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture* (Santa Margherita Ligure, Italy, June 1995), pp. 126–137.

[17] ANDREWS, K., AND SAND, D. Migrating a CISC Computer Family onto RISC via Object Code Translation. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, Massachusetts, 1992), pp. 213–222.

[18] ATHANAS, P. M., AND SILVERMAN, H. F. Processor Reconfiguration Through Instruction-Set Metamorphosis. *IEEE Computer* (Mar. 1993), 11–18.

[19] BAKER, B. S. An Algorithm for Structuring Flow Graphs. *Journal of the ACM 24*, 1 (Jan. 1977), 98–120.

[20] BAKER, K. R., BROWN, A. D., AND CURRIE, A. J. Optimization Efficiency in Behavioral Synthesis. *IEE Proceedings – Circuits, Devices and Systems 141*, 5 (1994), 399–406.

[21] BALL, T., AND LARUS, J. R. Branch Prediction for Free. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation* (June 1993), pp. 300–313.

[22] BANERJEE, U. *Dependence Analysis for Supercomputing.* The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, Norwell, Massachusetts, 1988.

[23] BANNON, P., AND KELLER, J. Internal Architecture of Alpha 21164 Microprocessor. In *Proceedings of COMPCON '95* (1995), pp. 79–87.

[24] BEATY, S. J. Genetic Algorithms and Instruction Scheduling. In *Proceedings of the 24th Annual International Workshop on Microprogramming* (Albuquerque, New Mexico, Nov. 1991), pp. 206–211.

[25] BECK, G. R., YEN, D. W. L., AND ANDERSON, T. L. The Cydra 5 Minisupercomputer: Architecture and Implementation. *The Journal of Supercomputing 7*, 1/2 (May 1993), 143–180.

[26] BELLMAN, R. On a Routing Problem. *Quaterly of Applied Mathematics 1*, 16 (1958), 87–90.

[27] BERNSTEIN, D., AND COHEN, D. Dynamic Memory Disambiguation for Array References. In *Proceedings of the 27th Annual International Workshop on Microprogramming* (San Jose, California, Nov. 1994), pp. 105–111.

[28] BERNSTEIN, D., COHEN, D., AND KRAWCZYK, H. Code Duplication: An Assist for Global Instruction Scheduling. In *Proceedings of the 24th Annual International Workshop on Microprogramming* (Albuquerque, New Mexico, Nov. 1991), pp. 103–113.

[29] BERNSTEIN, D., AND RODEY, M. Global Instruction Scheduling for Superscalar Machines. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation* (June 1991), pp. 241–255.

[30] BERNSTEIN, D., AND RODEY, M. Proving Safety of Speculative Load Instructions at Compile Time. In *Proceedings of the Fourth European Symposium on Programming* (1992), pp. 344–354.

[31] BERNSTEIN, D., RODEY, M., AND GERTNER, I. On the Complexity of Scheduling Problems for Parallel/Pipelined Machines. *IEEE Transactions on Computers 38*, 9 (Sept. 1989), 1308–1313.

[32] BOSCH, J. G., ET AL. Real-Time Frame-to-Frame Automatic Contour Detection on Echocardiograms. In *Proceedings of Computers in Cardiology* (1994).

[33] BRADLEE, D. G., EGGERS, S. J., AND HENRY, R. R. Integrating Register allocation and instruction scheduling for RISCs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Santa Clara, California, Apr. 1991), pp. 122–131.

[34] BRADLEE, D. G., HENRY, R. R., AND EGGERS, S. J. The Marion System for Retargetable Instruction Scheduling. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation* (Toronto, Ontatrio, Canada, June 1991), pp. 229–240.

[35] BRAYTON, R., AND SPENCE, R. *Sensitivity and Optimization*. Elsevier, 1980.

[36] BRETERNITZ, JR., M., AND SHEN, J. P. Architecture Synthesis of High-Performance Application-Specific Processors. In *Proceedings of the 27th Design Automation Conference* (June 1990).

[37] BRETERNITZ, JR., M., AND SHEN, J. P. Implementation Optimization Techniques for Architecture Synthesis of Application-Specific Processors. In *Proceedings of the 24th Annual International Workshop on Microprogramming* (Albuquerque, New Mexico, Nov. 1991), pp. 114–123.

[38] BREWER, T. A Highly Scalable System Utilizing up to 128 PA-RISC Processors. In *Proceedings of COMPCON '95* (1995).

[39] BRIGGS, P. *Register Allocation via Graph Coloring*. PhD thesis, Department of Computer Science, Rice University, Houston, Texas, Apr. 1992.

[40] CAMPOSANO, R., AND WILBERG, J. Embedded System Design. Tech. rep., GMD, Germany, 1993.

[41] CAMPOSANO, R., AND WOLF, W., Eds. *High-Level VLSI Synthesis*. Kluwer Academic Publishers, 1991.

[42] CAPITANIO, A., DUTT, N., AND NICOLUA, A. Partitioned Register Files for VLIWs: A Preliminary Analysis of Tradeoffs. In *Proceedings of the 25th Annual International Workshop on Microprogramming* (Portland, Oregon, Dec. 1992), pp. 292–300.

[43] CARR, S., DING, C., AND SWEANY, P. Improving Software Pipelining With Unroll-and-Jam. Tech. Rep. May, Michigan Technological University, Department of Computer Science, 1995.

[44] CASE, B. ARM Architecture Offers High Code Density. *Microprocessor Report* (Dec. 1991).

[45] CASE, B. Philips Hopes to Displace DSPs with VLIWs. *Microprocessor Report* (Dec. 1994).

[46] CHAITIN, G. J. Register Allocation & Spilling Via Graph Coloring. In *Proceedings of the SIGPLAN '82 Conference on Programming Language Design and Implementation* (June 1982), pp. 201–207.

[47] CHANG, P. P., ET AL. The Importance of Prepass Code Scheduling for Superscalar and Superpiplined Processors. *IEEE Transactions on Computers 44*, 3 (Mar. 1995), 353–370.

[48] CHEN, W. Y., ET AL. Tolerating Data Access Latency with Register Preloading. In *Proceedings of the 1992 International Conference on Supercomputing* (July 1992), pp. 104–113.

[49] CHOU, H.-C., AND CHUNG, C.-P. An Optimal Instruction Scheduler for Superscalar Processor. *IEEE Transactions on Parallel and Distributed Systems 6*, 3 (Mar. 1995), 303–313.

[50] COHN, R., GROSS, T., LAM, M., AND TSENG, P. S. Architecture and Compiler Tradeoffs for a Long Instruction Word Microprocessor. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, Massachusetts, Apr. 1989), pp. 2–14.

[51] COLWELL, R. P., ET AL. A VLIW Architecture for a Trace Scheduling Compiler. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Oct. 1987), ACM, pp. 180–192.

[52] CONTE, T. M., MENEZES, K. N. P., AND SATHAYE, S. A. A Technique to Determine Power-Efficient, High-Performance Superscalar Processors. In *Proceedings of the 28th Hawaii International Conference on System Sciences* (Maui, Hawaii, Jan. 1995), vol. 1, pp. 324–333.

[53] CONTE, T. M., AND SATHAYE, S. W. Dynamic Rescheduling: A Technique for Object Code Compatibility in VLIW Architectures. In *Proceedings of the 28th Annual International Workshop on Microprogramming* (Ann Arbor, Michigan, Nov. 1995), pp. 208–218.

[54] CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 1992.

[55] CORPORAAL, H. MOVE32INT, Architecture and Programmer's Reference Manual. Tech. Rep. 1-68340-44-(1992)01, Delft University of Technology, Department of Electrical Engineering, The Netherlands, Jan. 1992.

[56] CORPORAAL, H. *Transport Triggered Architectures: Design and Evaluation*. PhD thesis, Department of Electrical Engineering, Delft University of Technology, Delft, The Netherlands, 1995.

[57] CORPORAAL, H., AND HOOGERBRUGGE, J. Code Generation for Transport Triggered Architectures. In *Code Generation for Embedded Processors*, G. Goossens and P. Marwedel, Eds. Kluwer Academic Publishers, 1995, ch. 14, pp. 240–259.

[58] CORPORAAL, H., AND LAMBERTS, R. TTA processor synthesis. In *Proceedings of the First Annual Conference of the Advanced School for Computing and Imaging* (Heijen, The Netherlands, May 1995), J. van Katwijk, J. J. Gerbrands, M. R. van Steen, and J. F. M. Tonino, Eds.

[59] CORPORAAL, H., AND MULDER, H. MOVE: A Framework for High-Performance Processor Design. In *Proceedings of Supercomputing-91, Albuquerque* (Nov. 1991), pp. 692–701.

[60] CORPORAAL, H., MULDER, H., AND SADINSKY, I. Evaluation of Transport Triggered Architectures. Tech. rep., Delft University of Technology, Department of Electrical Engineering, The Netherlands, Nov. 1991.

[61] CORPORAAL, H., AND VAN DER AREND, P. MOVE32INT, a Sea of Gates Realization of a High Performance Transport Triggered Architecture. *Microprocessing and Microprogramming 38* (Sept. 1993), 53–60.

[62] DE GLORIA, A., FARABOSCHI, P., AND OLIVIERI, M. A Non-Deterministic Scheduler for a Software Pipelining Compiler. In *Proceedings of the 25th Annual International Workshop on Microprogramming* (Portland, Oregon, Dec. 1992), pp. 41–44.

[63] DE MICHELI, G. Hardware-Software Codesign. *IEEE Micro* (Aug. 1994), 9–16.

[64] DE MICHELI, G. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[65] DEHNERT, J. C., HSU, P. Y. T., AND BRATT, J. P. Overlapped Loop Support in the Cydra 5. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, Massachusetts, 1989), pp. 26–38.

[66] DEHNERT, J. C., AND TOWLE, R. A. Compiling for the Cydra 5. *The Journal of Supercomputing 7*, 1/2 (May 1993), 181–228.

[67] DEUTSCH, A. Interprocedural May-Alias Analysis for Pointers: Beyond k-Limiting. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation* (June 1994), pp. 230–241.

[68] DOBBERPUHL, D. W., ET AL. A 200-MHz 64-b Dual-Issue CMOS Microprocessor. *IEEE Journal of Solid-State Circuits 27*, 11 (Nov. 1992).

[69] DOLAN, A., AND ALDOUS, J. *Networks and Algorithms: An Introductory Approach*. John Wiley and Sons, 1993.

[70] EBCIOĞLU, K. A Compilation Technique for Software Pipelining of Loops with Conditional Jumps. In *Proceedings of the 20th Annual International Workshop on Microprogramming* (Dec. 1987).

[71] EBCIOĞLU, K. Some Design Ideas for a VLIW Architecture for Sequential Natured Software. In *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Processing* (Pisa, Italy, Apr. 1988), pp. 1–21.

[72] EBCIOĞLU, K., AND NAKATANI, T. A New Compilation Technique for Parallelizing Loops with Unpredictable Branches on a VLIW Architecture. In *Proceedings of the Second Workshop on Programming Languages and Compilers for Parallel Computing* (University of Illinois at Urbana-Champaign, 1989).

[73] EICHENBERGER, A. E., AND DAVIDSON, E. S. Register Allocation for Predicated Code. In *Proceedings of the 28th Annual International Workshop on Microprogramming* (Ann Arbor, Michigan, Nov. 1995), pp. 180–191.

[74] ELLIS, J. R. *Bulldog: A Compiler for VLIW Architectures.* ACM Doctoral Dissertation Awards. MIT Press, Cambridge, Massachusetts, 1986.

[75] EMBREE, P. M. *C Language Algorithms for Real-Time DSP.* Prentice Hall, 1995.

[76] ERNST, R., HENKEL, J., AND BENNER, T. Hardware-Software Cosynthesis for Microcontrollers. *IEEE Design & Test of Computers* (Dec. 1993), 64–75.

[77] ERTL, M., AND KRALL, A. Instruction Scheduling for Complex Pipelines. In *Proceedings of the International Workshop on Compiler Construction* (Paderborn, Germany, Oct. 1992).

[78] ERTL, M., AND KRALL, A. Delayed Exceptions — Speculative Execution of Trapping Instructions. In *Proceedings of the International Conference on Compiler Construction* (Edinburgh, Scotland, Apr. 1994), pp. 158–171.

[79] FISHER, J. A. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Transactions on Computers C-30*, 7 (July 1981), 478–490.

[80] FISHER, J. A., ET AL. Parallel Processing: A Smart Compiler and a Dumb Machine. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction* (Montreal, Canada, June 1984), pp. 37–47.

[81] FISHER, J. A., AND FREUDENBERGER, S. M. Predicting Conditional Branch Directions From Previous Runs of a Program. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, Massachusetts, Oct. 1992), ACM, pp. 85–95.

[82] FISHER, J. A., AND RAU, B. R. Instruction-Level Parallel Processing. *Science 253*, 5025 (Sept. 1992), 1233–1241.

[83] FORD, JR., L. R., AND FULKERSON, D. R. *Flows in Networks.* Princeton University Press, 1962.

[84] FRANKLIN, M. *The Multiscalar Architecture.* PhD thesis, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI, 53706, 1993.

[85] FRANKLIN, M., AND SOHI, G. S. Register Traffic Analysis for Streamlining Inter-Operation Communication in Fine-Grain Parallel Processors. In *Proceedings of the 25th Annual International Workshop on Microprogramming* (Portland, Oregon, Dec. 1992), pp. 236–245.

[86] FREUDENBERGER, S. M., AND RUTTENBERG, J. C. Place Ordering of Register Allocation and Instruction Scheduling. In *Code Generation-Concepts, Tools and Techniques* (1991).

[87] GAJSKI, D., DUTT, N., WU, A., AND LIN, S. *High-Level Synthesis; Introduction to Chip and System Design.* Kluwer Academic Publishers, 1992.

[88] GALLAGHER, D. M. *Memory Disambiguation to Facilitate Instruction-Level Parallelism Compilation.* PhD thesis, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, 1995.

[89] GALLAGHER, D. M., ET AL. Dynamic Memory Disambiguation Using the Memory Conflict Buffer. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, Oct. 1994), ACM, pp. 183–195.

[90] GAREY, M. R., AND JOHNSON, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and company, New York, 1979.

[91] GOFF, G., KENNEDY, K., AND TSENG, C.-W. Practical Dependence Testing. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation* (June 1991), pp. 15–29.

[92] GOODMAN, J. R., AND HSU, W.-C. Code Scheduling and Register Allocation in Large Basic Blocks. In *Proceedings of the International Conference on Supercomputing* (St. Malo, France, July 1988), pp. 442–452.

[93] GOVINDARAJAN, R., ALTMAN, E. R., AND GAO, G. R. Minimizing Register Requirements under Resource-Constrained Rate-Optimal Software Pipelining. In *Proceedings of the 27th Annual International Workshop on Microprogramming* (San Jose, California, Nov. 1994), pp. 85–94.

[94] GREENLEY, D., ET AL. UltraSPARC: The Next Generation Superscalar 64-bit SPARC. In *Proceedings of COMPCON '95* (1995), pp. 442–451.

[95] GRIESEMER, R. Scheduling Instructions by Direct Placement. In *Proceedings of the International Workshop on Compiler Construction* (Paderborn, Germany, Oct. 1992).

[96] GUPTA, R., AND SOFFA, M. Region Scheduling: An Approach for Detecting and Redistributing Parallelism. *IEEE Transactions on Software Engineering 16*, 4 (Apr. 1990), 421–431.

[97] GUPTA, R. K., COELHO JR., C. N., AND DE MICHELI, G. Program Implementation Schemes for Hardware-Software Systems. *IEEE Computer* (Jan. 1994), 48–55.

[98] GUPTA, R. K., AND DE MICHELI, G. Hardware-Software Cosynthesis for Digital Systems. *IEEE Design & Test of Computers* (Sept. 1993), 29–41.

[99] HACKING, I. *Representing and Intervening, Introductory Topics in the Philosophy of Natural Science.* Cambridge University Press, Cambridge, 1983.

[100] HELLERMAN, H. On the Average Speed of a Multiple-Module Storage System. *IEEE Transactions on Computers 15*, 8 (Aug. 1966), 670.

[101] HENDREN, L. J., HUMMEL, J., AND NICOLAU, A. Abstractions for Recursive Pointer Data Structures: Improving the Analysis of Imperative Programs. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation* (July 1992), pp. 249–260.

[102] HENNESSY, J. L., AND GROSS, T. Postpass Code Optimization of Pipeline Constraints. *Transactions on Programming Languages and Systems 5*, 3 (July 1983), 422–448.

[103] HENNESSY, J. L., AND PATTERSON, D. A. *Computer Architecture, a Quantitative Approach.* Morgan Kaufmann publishers, 1990.

[104] HOOGERBRUGGE, J., AND CORPORAAL, H. Comparing Software Pipelining for an Operation Triggered and a Transport Triggered Architecture. In *Proceedings of the International Workshop on Compiler Construction* (Paderborn, Germany, Oct. 1992), pp. 219–228.

[105] HOOGERBRUGGE, J., AND CORPORAAL, H. Register File Port Requirements of Transport Triggered Architectures. In *Proceedings of the 27th Annual International Workshop on Microprogramming* (San Jose, California, Nov. 1994), pp. 191–195.

[106] HOOGERBRUGGE, J., AND CORPORAAL, H. Transport Triggering vs. Operation Triggering. In *Proceedings of the International Conference on Compiler Construction* (Edinburgh, Scotland, Apr. 1994).

[107] HOOGERBRUGGE, J., AND CORPORAAL, H. Automatic Synthesis of Transport Triggered Processors. In *Proceedings of the First Annual Conference of the Advanced School for Computing and Imaging* (Heijen, The Netherlands, May 1995), J. van Katwijk, J. J. Gerbrands, M. R. van Steen, and J. F. M. Tonino, Eds.

[108] HOOGERBRUGGE, J., AND CORPORAAL, H. Resource Assignment in a Compiler for Transport Triggered Architectures. In *submitted for publication* (1996).

[109] HOOGERBRUGGE, J., CORPORAAL, H., AND MULDER, H. Software Pipelining for Transport Triggered Architectures. In *Proceedings of the 24th Annual International Workshop on Microprogramming* (Albuquerque, New Mexico, Nov. 1991), pp. 74–81.

[110] HSU, P. Y. T. Designing the TFP Microprocessor. *IEEE Micro 14*, 2 (Apr. 1994), 23–33.

[111] HSU, P. Y. T., AND DAVIDSON, E. S. Highly Concurrent Scalar Processing. In *Proceedings of the 13th Annual International Symposium on Computer Architecture* (June 1986), pp. 386–395.

[112] HUANG, A., AND SLAVENBURG, G. Speculative Disambiguation: A Compilation Technique for Dynamic Disambiguation. In *Proceedings of the 21th Annual International Symposium on Computer Architecture* (Apr. 1994), pp. 200–210.

[113] HUANG, I. *Co-Synthesis of Instruction Sets and Microarchitectures.* PhD thesis, Advanced Computer Architecture Laboratory, University of Southern California, 1994.

[114] HUFF, R. A. Lifetime-Sensitive Modulo Scheduling. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation* (June 1993), pp. 258–267.

[115] HUMMEL, J., HENDREN, L. J., AND NICOLAU, A. A General Data Dependence Test for Dynamic, Pointer-Based Data Structures. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation* (June 1994), pp. 218–229.

[116] HUNT, D. Advanced Performance Features of the 64-bit PA-8000. In *Proceedings of COMPCON '95* (1995), pp. 123–128.

[117] HWU, W. W., ET AL. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. *The Journal of Supercomputing 7*, 1/2 (May 1993), 229–249.

[118] JAIN, S. Circular Scheduling: A New Technique to Perform Software Pipelining. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation* (June 1991), pp. 219–228.

[119] JANSSEN, J., AND CORPORAAL, H. Partitioned Register Files for TTAs. In *Proceedings of the 28th Annual International Workshop on Microprogramming* (Ann Arbor, Michigan, Nov. 1995), pp. 303–312.

[120] JOHNSON, W. M. *Superscalar Microprocessor Design.* Prentice Hall, 1991.

[121] JONES, R. B. Constrained Software Pipelining. Master's thesis, Utah State University, Logan, Utah, Aug. 1991.

[122] JOUPPI, N. P., AND WALL, D. W. Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, Massachusetts, Apr. 1989), pp. 272–282.

[123] KATHAIL, V., SCHLANSKER, M., AND RAU, B. HPL PlayDoh Architecture Specification: Version 1.0. Tech. Rep. HPL-93-80, Hewlett Packard Computer Systems Laboratory, Palo Alto, CA, Feb. 1994.

[124] KERNIGHAN, B. W., AND RITCHIE, D. M. *The C Programming Language*, second ed. Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[125] KESSLER, R. E., AND SCHWARZMEIER, J. L. CRAY T3D: A new dimension in CRAY research. In *Proceedings of COMPCON '93* (Feb. 1993), pp. 176–182.

[126] KIRKPATRICK, S., GELATT JR., C. D., AND VECCHI, M. P. Optimization by Simulated Annealing. *Science 220*, 4598 (May 1983), 671–680.

[127] KUCK, D. J., ET AL. Dependence Graphs and Compiler Optimization. In *Conference Record of the Eighth ACM Symposium on Principles of Programming Languages* (Jan. 1981), pp. 207–218.

[128] KURPANEK, G., ET AL. PA7200: A PA-RISC Processor with Integrated High Performance MP Bus Interface. In *Proceedings of COMPCON '94* (1994), pp. 375–382.

[129] LABROUSSE, J., AND SLAVENBURG, G. A. A 50MHz Microprocessor with a Very Long Instruction Word Architecture. In *Proceedings of ISSCC '90* (Feb. 1990).

[130] LABROUSSE, J., AND SLAVENBURG, G. A. CREATE-LIFE: A Modular Design Approach for High Performances ASIC's. In *Proceedings of COMPCON '90* (1990).

[131] LAM, M. S. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation* (June 1988), pp. 318–328.

[132] LAM, M. S. *A Systolic Array Optimizing Compiler*. The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, Norwell, Massachusetts, 1989.

[133] LAM, M. S., AND WILSON, R. P. Limits of Control Flow on Parallelism. In *Proceedings of the 19th International Symposium on Computer Architecture* (May 1992), pp. 46–57.

[134] LANDSKOV, D., ET AL. Local Microcode Compaction Techniques. *ACM Computing Surveys 12*, 3 (Sept. 1980), 261–294.

[135] LAVERY, D. M., AND HWU, W. W. Unrolling-Based Optimizations for Modulo Scheduling. In *Proceedings of the 28th Annual International Workshop on Microprogramming* (Ann Arbor, Michigan, Nov. 1995), pp. 327–337.

[136] LAWLER, E. L. *Combinatorial Optimization: Networks and Matroids*. Saunders College Publishing, 1976.

[137] LEE, R. L. Precision Architecture. *IEEE Computer 22*, 1 (Jan. 1989), 78–91.

[138] LEE, R. L. Accelerating Multimedia with Enhanced Microprocessors. *IEEE Micro 15*, 2 (Apr. 1995).

[139] LEVITAN, D., THOMAS, T., AND TU, P. The PowerPC 620 Microprocessor: A High Performance Superscalar RISC Microprocessor. In *Proceedings of COMPCON '95* (1995), pp. 285–291.

[140] LOWNEY, P. G., ET AL. The Multiflow Trace Scheduling Compiler. *The Journal of Supercomputing 7*, 1/2 (May 1993), 51–142.

[141] MAHADEVAN, U., AND RAMAKRISHNAN, S. Instruction Scheduling over Regions: A Framework for Scheduling Across Basic Blocks. In *Proceedings of the International Conference on Compiler Construction* (Edinburgh, Scotland, Apr. 1994), pp. 419–434.

[142] MAHLKE, S. A., ET AL. Compiler Code Transformations for Superscalar-Based High-Performance Systems. In *Proceedings of the 1992 International Conference on Supercomputing* (July 1992).

[143] MAHLKE, S. A., ET AL. Effective Compiler Support for Predicated Execution Using the Hyperblock. In *Proceedings of the 25th Annual International Workshop on Microprogramming* (Portland, Oregon, Dec. 1992), pp. 45–54.

[144] MAHLKE, S. A., ET AL. Sentinel Scheduling for VLIW and Superscalar Processors. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, Massachusetts, Oct. 1992), ACM, pp. 238–247.

[145] MAHLKE, S. A., ET AL. Characterizing the Impact of Predicated Execution on Branch Prediction. In *Proceedings of the 27th Annual International Workshop on Microprogramming* (San Jose, California, Nov. 1994), pp. 217–227.

[146] MALIK, N., EICKEMEYER, R. J., AND VASSILIADIS, S. Interlock Collapsing ALU for Increased Instruction-Level Parallelism. In *Proceedings of the 25th Annual International Workshop on Microprogramming* (Portland, Oregon, Dec. 1992), pp. 149–157.

[147] MAYDAN, D. E., HENNESSY, J. L., AND LAM, M. S. Efficient and Exact Data Dependency Analysis. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation* (June 1991), pp. 1–14.

[148] MCFARLING, S., AND HENNESSY, J. Reducing the Cost of Branches. In *Proceedings of the 13th Annual International Symposium on Computer Architecture* (June 1986), pp. 396–403.

[149] MIRAPURI, S., WOODACRE, M., AND VASSEGHI, N. The MIPS R4000 Processor. *IEEE Micro 12*, 2 (Apr. 1992), 10–22.

[150] MOON, S., AND EBCIOĞLU, K. An Efficient Resource-Constrained Global Scheduling Technique for Superscalar and VLIW Processors. In *Proceedings of the 25th Annual International Workshop on Microprogramming* (Portland, Oregon, Dec. 1992), pp. 55–71.

[151] MOON, S., AND EBCIOĞLU, K. A Study on the Number of Memory Ports in Multiple Instruction Issue Machines. In *Proceedings of the 26th Annual International Workshop on Microprogramming* (Austin, Texas, Dec. 1993), pp. 49–58.

[152] MOUDGILL, M. *Implementing and Exploiting Static Speculation on Multiple Instruction Issue Processors.* PhD thesis, Department of Computer Science, Cornell University, Ithaca, NY, 14853, Mar. 1994.

[153] MULDER, J. M., AND PORTIER, R. J. Cost-Effective Design of Application-Specific VLIW Processors Using the SCARCE Framework. In *Proceedings of the 22nd Workshop on Microprogramming and Microarchitectures* (Aug. 1989).

[154] NAKATANI, T., AND EBCIOĞLU, K. "Combining" as a Compilation Technique for VLIW Architectures. In *Proceedings of the 22nd Annual International Workshop on Microprogramming and Microarchitecture* (Dublin, Ireland, Aug. 1989), pp. 43–55.

[155] NICOLAU, A. Percolation Scheduling: A Parallel Compilation Technique. Tech. Rep. TR 85-678, Cornell University, Department of Computer Science, Cornell University, Ithaca, NY 14853, USA, May 1985.

[156] NICOLAU, A. Run-Time Disambiguation: Coping with Statically Unpredictable Dependencies. *IEEE Transactions on Computers 38*, 5 (May 1989), 663–678.

[157] NICOLAU, A., AND POTASMAN, R. Incremental Tree Height Reduction for High Level Synthesis. In *Proceedings of the 28th ACM/IEEE Design Automation Conference* (San Francisco, California, June 1991), pp. 770–774.

[158] NICOLAU, A., POTASMAN, R., AND WANG, H. Register Allocation, Renaming, and their Impact on Parallelism. *Languages and Compilers for Parallel Computers*, 589 (1991).

[159] NORRIS, C., AND POLLOCK, L. L. A Scheduler-Sensitive Global Register Allocator. In *Proceedings of the International Conference on Supercomputing* (Portland, Oregon, Nov. 1993), pp. 804–813.

[160] PARK, J. C. H., AND SCHLANSKER, M. On Predicated Execution. Tech. Rep. HPL-91-58, Hewlett Packard Computer Systems Laboratory, Palo Alto, CA, May 1991.

[161] PEELING, N. E. ANDF Features and Benefits. Tech. rep., DRA Malvern UK; British Crown, 1992.

[162] PINTER, S. S. Register Allocation with Instruction Scheduling: a New Approach. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation* (June 1993), pp. 248–257.

[163] PORTIER, R. J. GEPS: Global Enhanced Pipeline Scheduling. In *Proceedings of the Sixth Workshop Computer Systems* (Delft, The Netherlands, Jan. 1993).

[164] PORTIER, R. J. *VLIW Processor Architecture Design: Exploration of the Design Space by Means of Compilation Techniques*. PhD thesis, Department of Electrical Engineering, Delft University of Technology, Delft, The Netherlands, 1996, in preparation.

[165] POTASMAN, R. *Percolation-Based Compiling for Evaluation of Parallelism and hardware Design Trade-Offs*. PhD thesis, Department of Information and Computer Science, University of California at Irvine, Irvine, CA, 92717, 1991.

[166] PUGH, W. The Omega Test: A Fast and Practical Integer Programming algorithm for Dependence Analysis. In *Proceedings of Supercomputing-91, Albuquerque* (Nov. 1991), pp. 4–13.

[167] RAMAKRISHNAN, S. Software Pipelining in PA-RISC Compilers. *Hewlett-Packard Journal* (July 1992), 39–45.

[168] RAU, B. R. Dynamically Scheduled VLIW Processors. In *Proceedings of the 26th Annual International Workshop on Microprogramming* (Austin, Texas, Dec. 1993), pp. 80–92.

[169] RAU, B. R. Iterative Modulo Scheduling: An Algorithm For Software Pipelining Loops. In *Proceedings of the 27th Annual International Workshop on Microprogramming* (San Jose, California, Nov. 1994).

[170] RAU, B. R., ET AL. The Cydra 5 Departmental Supercomputer; Design Philosophies, Decisions and Trade-Offs. *IEEE Computer* (Jan. 1989), 12–35.

[171] RAU, B. R., AND FISHER, J. A. Instruction-Level Parallel Processing: History, Overview, and Perspective. *The Journal of Supercomputing 7*, 1/2 (May 1993), 9–50.

[172] RAU, B. R., AND GLAESER, C. D. Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing. In *Proceedings of the 14th Annual International Workshop on Microprogramming* (Oct. 1981), pp. 183–198.

[173] RAZDAN, R., AND SMITH, M. D. A High-Performance Microarchitecture with Hardware-Programmable Functional Units. In *Proceedings of the 27th Annual International Workshop on Microprogramming* (San Jose, California, Nov. 1994).

[174] SITES, R. L. *Alpha Architecture Reference Manual.* Digital Press, Burlington, MA, 1992.

[175] SMITH, J. E., AND SOHI, G. S. The Microarchitecture of Superscalar Processors. *Proceedings of the IEEE 83*, 10 (Dec. 1995).

[176] SMITH, M. D., HOROWITZ, M., AND LAM, M. S. Efficient Superscalar Performance Through Boosting. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, Massachusetts, Oct. 1992), ACM, pp. 248–261.

[177] SOHI, G., AND FRANKLIN, M. High-Bandwidth Data Memory Systems for Superscalar Processors. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Santa Clara, California, 1991), pp. 53–62.

[178] SOHI, G. S., BREACH, S. E., AND VIJAYKUMAR, T. N. Multiscalar Processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture* (June 1995), pp. 414–425.

[179] SOHI, G. S., AND VAJAPEYAM, S. Tradeoffs in Instruction Format Design for Horizontal Architectures. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, Massachusetts, Apr. 1989), pp. 15–25.

[180] STALLMAN, R. M. Using and Porting GNU CC. Tech. rep., Free Software Foundation, Cambridge, MA, 1988.

[181] STEENKISTE, P. The Impact of Code Density on Instruction Cache Performance. In *Proceedings of the 16th Annual International Symposium on Computer Architecture* (Jerusalem, Israel, June 1989), pp. 252–259.

[182] STEVEN, F. L., STEVEN, G. B., AND WANG, L. Using a Resource-Limited Instruction Scheduler to evaluate the iHARP Processor. *IEE Proceedings – Computers and Digital Techniques 142*, 1 (1995), 23–31.

[183] STRAVERS, P. *Embedded System Design.* PhD thesis, Department of Electrical Engineering, Delft University of Technology, Delft, The Netherlands, 1994.

[184] SU, B., DING, S., WANG, J., AND XIA, J. GURPR — A Method for Global Software Pipelining. In *Proceedings of the 20th Annual International Workshop on Microprogramming* (Colorado Springs, CO, Dec. 1987), pp. 97–105.

[185] SU, B., DING, S., AND XIA, J. URPR — An Extension of URCR for Software Pipelining. In *Proceedings of the 19th Annual International Workshop on Microprogramming* (New York, NY, Oct. 1986), pp. 104–108.

[186] SUN MICROSYSTEMS. The SuperSPARC Microprocessor, Technical White Paper. Tech. rep., Sun Microsystems Inc., 2550 Garcia Avenue, Mountain View, CA, 1992.

[187] THEOBALD, K. B., GAO, G. R., AND HENDREN, L. J. One the Limits of Program Parallelism and its Smoothability. In *Proceedings of the 25th Annual International Workshop on Microprogramming* (Portland, Oregon, Dec. 1992), pp. 10–19.

[188] THISTLE, M. R., AND SMITH, B. J. A Processor Architecture fo Horizon. In *Proceedings of Supercomputing '88* (Orlando, Florida, Nov. 1988), pp. 35–41.

[189] TIRUMALAI, P., LEE, M., AND SCHLANSKER, M. S. Parallelization of Loops with Exits on Pipelined Architectures. In *Proceedings of Supercomputing-90* (Nov. 1990), pp. 200–212.

[190] TOUZEAU, R. F. A Fortran Compiler for the FPS-164 Scientific Computer. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction* (Montreal, Canada, June 1984), pp. 48–57.

[191] TULLSEN, D. M., EGGERS, S. J., AND LEVY, H. M. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture* (Santa Margherita Ligure, Italy, June 1995), pp. 392–403.

[192] TYSON, G. S. The Effects of Predicated Execution on Branch Prediction. In *Proceedings of the 27th Annual International Workshop on Microprogramming* (San Jose, California, Nov. 1994), pp. 196–206.

[193] VANHOOF, J., ET AL. *High-Level Synthesis for Real Time Digital Signal Processing.* The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, 1993.

[194] VEEN, A. Dataflow Machine Architecture. *ACM Computing Surveys 18*, 4 (Dec. 1986), 365–396.

[195] VERBERNE, A., AND CORPORAAL, H. Towards Efficient Code Scheduling for Transport Triggered Architectures. In *Proceedings of the Fourth Workshop Computer Systems* (Amsterdam, The Netherlands, Oct. 1991), pp. 31–41.

[196] WADA, T., RAJAN, S., AND PRZYBYLSKI, S. A. An Analytical Access Time Model for On-Chip Cache Memories. *IEEE Journal of Solid-Sate Circuits 27* (Aug. 1992), 1147–1156.

[197] WALL, D. Predicting Program Behavior Using Real or Estimated Profiles. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation* (Toronto, Ontatrio, Canada, June 1991), pp. 59–70.

[198] WALL, D. W. Limits of Instruction-Level Parallelism. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems* (Apr. 1991), pp. 176–188.

[199] WARREN, H. S. Instruction Scheduling for the IBM RISC System/6000 Processor. *IBM Journal Research Development 34*, 1 (Jan. 1990), 85–92.

[200] WARTER, N. J. *Modulo Scheduling With Isomorphic Control Transformations.* PhD thesis, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, 1994.

[201] WARTER, N. J., ET AL. Reverse If-Conversion. In *Proceedings of the ACM SIGPLAN '93 Conference on Program Language Design and Implementation* (June 1993).

[202] WARTER, N. J., HAAB, G. E., AND BOCKHAUS, J. W. Enhanced Modulo Scheduling for Loops with Conditional Branches. In *Proceedings of the 25th Annual International Workshop on Microprogramming* (Portland, Oregon, Dec. 1992), pp. 170–179.

[203] WARTER-PEREZ, N. J., AND PARTAMIAN, N. Modulo Scheduling with Multiple Initiation Intervals. In *Proceedings of the 28th Annual International Workshop on Microprogramming* (Ann Arbor, Michigan, Nov. 1995), pp. 111–118.

[204] WEAVER, D., AND GERMOND, T. *The SPARC Architecture Manual, Version 9*. Prentice Hall, 1994.

[205] WOLF, M. E., AND LAM, M. S. A Loop Transformation Theory and an Algorithm to Maximize Parallelism. *IEEE Transactions on Parallel and Distributed Systems* 2, 4 (Oct. 1991), 452–471.

[206] WOLFE, A., AND BOLEYN, R. Two-Ported Cache Alternatives for Superscalar Processors. In *Proceedings of the 26th Annual International Workshop on Microprogramming* (Austin, Texas, Dec. 1993), pp. 41–48.

[207] WOLFE, A., AND CHANIN, A. Executing Compressed Programs on an Embedded RISC Architecture. In *Proceedings of the 25th Annual International Workshop on Microprogramming* (Portland, Oregon, Dec. 1992), pp. 81–91.

[208] WOLFE, A., AND CHEN, J. P. A Variable Instruction Stream Extension to the VLIW Architecture. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems* (Apr. 1991), pp. 2–14.

[209] WOLFE, M. *Optimizing Supercompilers for Supercomputers*. MIT Press, Cambridge, Massachusetts, 1989.

[210] WOLFE, M. Beyond Induction Variables. In *ACM SIGPLAN '92 Conference on Programming Language Design and Implementation* (San Francisco, California, June 1992), pp. 162–174.

[211] WOOD, G. Global Optimization of Microprograms through Modular Control Constructs. In *Proceedings of the 11nd Annual International Workshop on Microprogramming and Microarchitecture* (1979), pp. 1–6.

[212] WU, Y., AND LARUS, J. R. Static Branch Frequency and Program Analysis. In *Proceedings of the 27th Annual International Workshop on Microprogramming* (San Jose, California, Nov. 1994), pp. 1–11.

[213] ZIMA, H., AND CHAPMAN, B. *Supercompilers for Parallel and Vector Computers*. ACM Press Frontier Series. Addison-Wesley Publishing Company, Reading, Massachusetts, 1990.

Partial Loop Unrolling A

There are many loops that are too large for unrolling but where the frequently taken paths through the loop body are too short to contain sufficient ILP. In chapter 6 we proposed *partial loop unrolling* as a technique to deal with this problem. In this appendix we describe partial loop unrolling and report on an experiment that measures its effectiveness[1].

As far as we know, partial loop unrolling or a similar technique has not been proposed before.

A.1 Motivating Example

A motivating example for partial loop unrolling is shown in figure A.1. This code computes the first `size` primes by means of the well-known sieve of Eratosthenes. Since the probability that a number n is prime is approximately $\ln n/n$ for large n [54], the frequently taken path through the outer loop consists of testing `is_prime[i]`, incrementing `i`, and testing `i < size`. This path contains two basic blocks with six operations, which is too short to contain a sufficient amount of ILP. Full unrolling of the outer loop requires duplication of the inner loop. Partial unrolling makes it possible to unroll the outer loop without duplication of the inner loop. This reduces code size expansion and/or allows for a larger unrolling factor of the outer loop in order to expose more parallelism.

[1] The reason that partial loop unrolling is described in this appendix is that we developed it when the first six chapters of this thesis were already written. During the development of the backend we considered it several times but we thought that it would not be possible without generating irreducible code.

```
memset(is_prime, TRUE, sizeof(is_prime));

for(i = 2; i < size; i++) {
    if(is_prime[i]) {
        for(j = 2 * i; j < size; j += i)
            is_prime[j] = FALSE;
    }
}
```

Figure A.1: The sieve of Eratosthenes to compute primes

A.2 The Algorithm

Partial loop unrolling duplicates every basic block belonging to the loop being unrolled a number of times that depends on its size and the fraction of the operation count spend in it. Our implementation uses the following formula to determine the duplication count N_b of basic block b:

$$N_b = \min\{\lceil \frac{250 \times a \times d_b}{d_t \times s_b} \rceil, 5\} \quad \text{(A.1)}$$

where a is a user specified parameter that controls the aggressiveness (default value 10), d_b the dynamic operation count of b, d_t the total dynamic operation count of the whole application, and s_b the static operation count of b. Next, the found duplication counts are decremented until the following inequality holds for every basic block:

$$N_b \leq \max\{N_p \mid p \text{ is a predecessor of } b \text{ belonging to the same loop}\} \quad \text{(A.2)}$$

The reason for this inequality is to prevent generation of loops with basic blocks without predecessors (dead code) that have to be removed afterwards. Figure A.2 illustrates the actual unrolling. In this example $N_A = N_B = N_D = N_F = 3$, $N_C = 2$, and $N_E = 1$. The loop is unrolled $\max_b N_b$ times which is three times for the example loop. The minor loop bodies[2] are numbered as shown in figure A.2b, i.e., from the 'bottom' of the major loop body to the 'top'. A basic block b is omitted in minor loop body N if $N > N_b$. These basic blocks are shown as 'phantom' basic blocks in figure A.2b. Next, control flow edges arriving at a phantom basic block b are redirected to the 'real' copy of b in minor loop body N_b. For example, the control flow edge from A''' to E''' is redirected to E' since E''' is a phantom basic block and $N_E = 1$.

The final step of partial unrolling is updating the profiling data. This is done as follows. First, we assume that the loop header (A''') of the unrolled loop is executed T times. Next, we compute the execution counts of the other basic blocks

[2]The *major* loop body corresponds to the loop body of the unrolled loop, and a *minor* loop body corresponds to the loop body of the original loop. In this example, a major loop body contains three minor loop bodies.

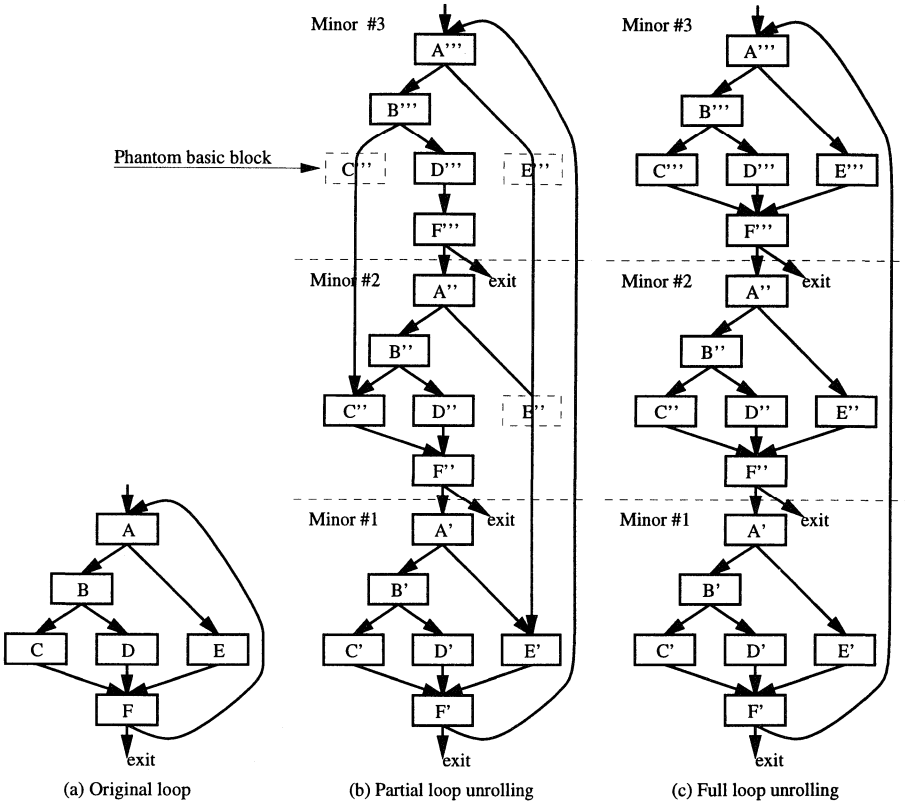


Figure A.2: Partial and full loop unrolling

by traversing the control flow graph in a topological order (ignoring backward control flow edges) and using successor probabilities. For example, if B'' is executed $0.9T$ times and the probability that D is executed after B equals 0.5, then the execution count of D'' is $0.45T$. Next, we sum these execution counts of all copies of the original loop header (A', A'', and A'''), let us say $2.6T$ times, and compare this with the execution count of the loop header of the original loop (A), let us say 1000 times. This gives us the value of T. For the example values this is $T = 1000/2.6 \approx 385$. Finally, we substitute the found value for T in execution counts expressed in T computed in the first step. Notice that updating the profiling data requires approximation. It is assumed that successor probabilities are independent of the loop iteration.

There are also some disadvantages to partial unrolling. First the trip count of a partially unrolled loop depends on the paths taken trough the loop body. This makes it impossible to remove exit control flow edges when the compiler has information about the trip count of the original loop. For example, if the trip count of the original loop in figure A.2a is $3n$ where $n > 0$, it is possi-

Benchmark	Cycle count decrease due to partial unrolling (%)	Code expansion (moves)	
		Full unrolling	Partial unrolling
bison	9.5	3,555	2,208
compress	9.8	296	1,194
cpp	6.1	7,698	2,829
djpeg	5.9	228	3,585
expand	17.2	144	720
flex	5.2	3,979	2,137
gzip	27.2	564	689
mpeg_play	9.6	270	6,034
music	10.6	776	424
sed	7.9	960	405
sort	9.2	1,366	1,299
sum	8.0	220	208
virtex	5.8	2,515	3,849

Table A.1: Partial loop unrolling results

ble to remove two exit control flow edges in the fully unrolled loop shown in figure A.2c. This is not possible for the partially unrolled loop shown in figure A.2b. A second disadvantage related to this is that induction variables may disappear which may reduce the performance of the memory reference disambiguator. A more general induction variable definition is required to deal with this problem [210].

A.3 Evaluation

We compiled the benchmarks listed in table 6.2 for the architecture described in table 6.1 with full and partial loop unrolling. Table A.1 presents the results. Only the benchmarks with a cycle count decrease of more than 5% are listed. None of the benchmarks exhibited a cycle count increase. The cycle count decreases by 4.9% on average while the code size expansion remains approximately the same on average although changes in the amount of code duplication fluctuate heavily. This is due to that (1) some loops that are not unrolled by full unrolling are unrolled by partial unrolling and (2) some loops that are unrolled by full unrolling are unrolled with less code duplication by partial unrolling.

Samenvatting

Code Generatie voor *Transport Triggered Architectures*

Recentelijk (1991) is een nieuwe klasse van *instruction level parallel* (ILP) computer architecturen geïntroduceerd, genaamd *transport triggered architectures* (TTA's), met gunstige eigenschappen op het gebied van schaalbaarheid, ontwerpcomplexiteit, haalbare klokfrequenties en prijs-prestatie verhouding. Deze eigenschappen worden voornamelijk verkregen door het overhevelen van complexiteit van de hardware naar de compiler. De vraag is of de compiler deze complexiteit aan kan. Verder is het wenselijk te weten hoe groot de voordelen van TTA's zijn en of er ook nadelen zijn. Dit proefschrift tracht deze vragen te beantwoorden door een compiler voor TTA's te ontwikkelen en experimenten met deze compiler uit te voeren die eigenschappen van TTA's kwantificeren.

De ontwikkelde compiler is gebaseerd op de GNU C/C++/Fortran compiler en bevat de volgende technieken die noodzakelijk zijn om een aanzienlijke hoeveelheid ILP te kunnen exploiteren:

1. *Extended basic block scheduling:* het schedulen van operaties over basic block grenzen om ILP tussen basic blocks te exploiteren. De scheduler verplaatst operaties tussen basic blocks die tot de zelfde *region* behoren, waarbij een *region* gedefinieerd is als een maximale acyclische *control flow graph* met één *entry* basic block.
2. *Software pipelining:* het schedulen van operaties over loop iteratie grenzen om ILP tussen loop iteraties te exploiteren. Het gebruikte software pipelining algoritme is gebaseerd op *modulo scheduling*.
3. *Speculative execution:* het schedulen van operaties voor operaties waarvan zij *control flow* afhankelijk zijn.

4. *Guarded execution:* het omzetten van *control flow* afhankelijkheden in *data flow* afhankelijkheden om complexe scheduling transformaties mogelijk te maken.
5. *Multi-way branching:* het schedulen van meerdere branch operaties in één instructie.

Verder bevat de compiler de volgende TTA specifieke optimalizaties:

1. *Bypassing:* het doorsluizen van data van *functional unit* (FU) naar FU zonder de data tussentijds op te slaan in een *register file* (RF).
2. *Dead result move elimination:* het verwijderen van zinloze schrijfacties naar een RF.
3. *Operand sharing:* het delen van operand transporten naar een FU met meerdere operaties.
4. *Socket sharing:* het delen van RF leespoorten met meerdere operaties.
5. *Operand swapping:* het verwisselen van operanden van commutatieve operaties om de scheduling vrijheid te verhogen.

In vergelijking met een compiler voor traditionele ILP architecturen is een compiler voor TTA's veel gecompliceerder door de complexere *resource* administratie en de TTA specifieke optimalizaties die gedurende het schedulen van de code worden uitgevoerd. Desalniettemin is gebleken dat het goed mogelijk is om een efficiënte compiler voor TTA's te ontwikkelen.

Met de ontwikkelde compiler zijn een groot aantal experimenten uitgevoerd om inzicht te krijgen in de eigenschappen van TTA's. Gebleken is dat TTA's veel efficienter met *resources* omspringen dan traditionele architecturen. Dit betekent dat TTA's minder hardware nodig hebben dan gelijk presterende traditionele architecturen, of anders gezegd, TTA's presteren beter dan gelijk kostende traditionele architecturen. Met andere woorden, TTA's hebben een betere prijs-prestatie verhouding.

Naast het ontwikkelen van een TTA compiler is in dit onderzoek aandacht besteed aan de vraag: gegeven een applicatie en een aantal ontwerprandvoorwaarden, wat is een goede TTA configuratie voor deze applicatie? Om deze vraag te beantwoorden is een methode ontwikkeld waarmee naar een 'optimale' configuratie voor een applicatie kan worden gezocht binnen de ontwerpruimte van TTA's. Deze methode bestaat uit twee stappen. In de eerste stap worden de hoeveelheden FU's, RF poorten, registers en transport bussen bepaald. In de tweede stap wordt bepaald hoe de FU's en de RF's op de transport bussen worden aangesloten. De methode is gedemonstreerd door deze toe te passen op een beeldverwerkings applicatie.

Curriculum Vitae

Jan Hoogerbrugge was born on April 7, 1967 in Capelle aan de IJssel. From 1979 to 1987 he attended three levels of technical education (LTS, MTS, and HTS). In 1991 he received his masters degree *cum laude* from the Computer Science department of Delft University of Technology. In 1992 he joined prof. Ad van de Goor's computer architecture group at the Electrical Engineering department where he performed the research described in this dissertation.

Jan's research interests include all aspects of parallel processing and in particular compilation techniques for instruction level parallel processors.

Stellingen behorende bij het proefschrift 'Code Generation for Transport Triggered Architectures'

Jan Hoogerbrugge

5 februari 1996

1. Wie een aanzienlijke hoeveelheid (instructie level) parallellisme wil exploiteren, zal inefficiënt hardware gebruik moeten accepteren.

2. Wie denkt dat TTA's en superpipelining een ideale combinatie zijn heeft nooit grondig over compilatie voor TTA's nagedacht. Diep ge-pipelinede TTA's zijn alleen verstandig voor applicaties met weinig control flow veranderingen of een zeer reguliere control flow.

3. Dat 'compile time' met gratis en 'run time' met kostbaar wordt geassocieerd is een veel voorkomend misverstand.

4. Eén van de grootste taboes onder programmeurs is het gebruik van goto's.

5. Het is in het algemeen gemakkelijker om een traag correct programma te versnellen (d.m.v. profiling) dan om een snel incorrect programma te corrigeren (d.m.v. debugging).

6. Een overheid / operating system dient zich te beperken tot die zaken waarvoor zij noodzakelijk is. De overige taken dienen te worden overgelaten aan het particuliere bedrijfsleven / user programs.

7. Computing power laat zich beter delen (time sharing) dan optellen (parallel processing). Liever één 250 SPECint machine dan tien 30 SPECint machines.

8. Bescheidenheid en het publiceren van wetenschappelijk onderzoek gaan moeilijk samen.

9. Daar het algemeen bekend is dat wetenschappers nu eenmaal weinig management, marketing en commerciele inzichten hebben, probeert menig wetenschapper zijn/haar wetenschappelijke kwaliteiten aan te tonen door een onderneming op te richten en deze in no-time failliet te laten gaan.

10. Het huidige verkeers/mileubeleid komt overeen met het instellen van een belasting op liftgebruik en het subsidieren van traplopen in een gebouw van 20 verdiepingen. De mensen die werken/wonen op de laagste verdieping hebben een meevaller, echter de meerderheid heeft nauwelijks een keuze en wordt onterecht op kosten gejaagd.

11. Het veelvuldig publiceren van dezelfde resultaten van eigen onderzoek kan gezien worden als een bijzondere vorm van plagiaat.

12. Wie de arbeidsproductiviteit van zijn/haar organisatie wil verlagen, moet zijn/haar medewerkers Internet faciliteiten aanbieden.

13. Behalve een vooruitziende blik vergt het ook vaak moed om de put te dempen voordat het kalf verdronken is. Voorbeeld: de onbetaalbaar wordende sociale zekerheid.

14. We zullen er mee moeten leren leven dat complexe software nooit vrij van fouten zal zijn. Naast het spreekwoord 'een vergissing is menselijk' wordt het tijd voor het spreekwoord 'een *bug* is *computerlijk*'.

15. Het voordeel van recursie is het ontbreken van loop overhead. Met soortgelijke redeneringen wordt veel wetenschappelijk onderzoek gemotiveerd.

16. Twee van de grootste problemen uit de informatica zijn: (1) computers zijn te traag en (2) computers zijn moeilijk te programmeren. Parallelle computers geprogrammeerd d.m.v. programmeertalen met expliciet parallellisme trachten het eerste probleem op te lossen, maar vergroten het tweede probleem vaak aanzienlijk.