Percolation-Based Compiling for Evaluation of Parallelism and Hardware Design Trade-Offs

Technical Report 91-80

Roni Potasman
Department of Information and Computer Science
University of California, Irvine CA, 92717

Dedication

To my late grandparents Adel and Moshe Herman

for teaching me the first steps in my life.

Contents

List of Figures

List of Tables

Acknowledgements

Many people have helped me to reach this point that ends almost 22 years of formal study. My advisor, Professor Alex Nicolau, opened for me a window to the world of compilers and parallel processing, spending endless hours discussing new ideas and problems. His support, guidance and constant encouragement made my research fruitful and so enjoyable.

I would like to thank my committee members, Professor Daniel Gajski, Professor Nader Bagherzadeh and Professor Fadi Kurdahi for their interest in my research.

The e-mail and phone communication with Mauricio Breternitz helped me a lot during the toughest hours of the research. Haigeng Wang shared with me a lot of frustrating as well as bright moments writing two important parts of the compiler and debugging dozens of benchmarks. Steve Novack and David Kolson spent hours reviewing my drafts.

Of course, special thanks are due to my wife Sarah who made the last three years possible and to our son Doron who brought a new meaning and dimension to our life. Their daily warm welcome made this Ph.D. worthwhile.

Finally, thanks to my parents Emma and William and to my brother Ico for their everlasting love and support and for giving me the opportunity to be a part of their pride.

Curriculum Vitae

November 3, 1954	Born Nahariya, Israel.
1976	BSc. in Electrical Engineering, Technion, Israel Institute of Technology, Haifa, Israel.
1983	MSc. in Electrical Engineering, Technion, Israel Institute of Technology, Haifa, Israel.
1991	PhD. in Electrical and Computer Engineering, University of California, Irvine, California. Dissertation "Percolation-Based Compiling for Evaluation of Parallelism and Hardware Design Trade-Offs".

Publications

R. Potasman, J. Lis, A. Nicolau, D. Gajski. "Percolation Based Synthesis". *Proceedings of the ACM IEEE 27th Design Automation Conference*, Orlando, FA, June 1990.

A. Nicolau, R. Potasman. "Realistic Scheduling: Compaction for Pipelined Architectures". *Proceedings of the 23rd Annual Workshop on Microprogramming*, Orlando, FA, November 1990.

A. Nicolau, R. Potasman. "Incremental Tree Height Reduction for High Level Synthesis". *Proceedings of the ACM IEEE 28th Design Automation Conference*, San Francisco, CA, June 1991.

A. Abnous, R. Potasman, N. Bagherzadeh and A. Nicolau. "A Percolation Based VLIW Architecture". *Proceedings of the 1991 International Conference on Parallel Processing*, St. Charles, IL, August 1991.

A. Nicolau, R. Potasman and H. Wang. "Register Allocation, Renaming and Their Impact on Fine-Grain Parallelism". *Proceedings of the 4th International Workshop on Languages and Compilers for Parallel Processing*, Santa Clara, CA, August 1991.

H. Wang, A. Nicolau and R. Potasman. "A New Technique for Induction Variable Removal". *Proceedings of the 24th Annual Workshop on Microprogramming*, Albuquerque, NM, November 1991.

Abstract of the Dissertation

Percolation-Based Compiling for Evaluation of Parallelism and Hardware Design Trade-Offs
by
Roni Potasman
Doctor of Philosophy in Electrical and Computer Engineering
University of California, Irvine, 1991
Professor Alexandru Nicolau, Chair

This thesis investigates parallelism and hardware design trade-offs of parallel and pipelined architectures. To explore these trade-offs we developed a retargetable compiler based on a set of powerful code transformations called Percolation Scheduling (PS) that map programs with real-time constraints and/or massive time requirements onto synchronous, parallel, high-performance or semi-custom architectures.

High-performance is achieved through extraction of application inherent fine-grain parallelism and the use of a suitable architecture. Exploiting fine-grain parallelism is a critical part of exploiting all of the parallelism available in a given program, particularly since highly irregular forms of parallelism are often not visible at coarser levels and since the use of low-level parallelism has a multiplicative effect on the overall performance.

To extract substantial parallelism from both the hardware and the compiler, we use a clean, highly parallel VLIW-like architecture that is synchronous, has multiple functional units and has a single program counter. The use of a hazard-free and homogeneous architecture does not result only in a better VLSI design but also considerably increases the compiler's ability to produce better code. To further enhance parallelism we modified the uni-cycle VLIW model and extended the transformations such that *pipelined* units that provide extra parallelism are used.

Another approach presented is of resource constrained scheduling (RCS). Since the RCS problem is known to be NP-hard, in practice it may be solved only by a heuristic approach. We argue that using the heuristic *after* extraction of the unlimited-resources schedule may yield better results than if the heuristic has been applied at the beginning of the scheduling process.

Through a series of benchmarks we evaluate hardware design trade-offs and show that speed-ups *on average* of one order of magnitude are feasible with sufficient functional units. However, when resources are limited we show that the number of functional units needed may be optimized for a particular suite of application programs.

Chapter 1

Introduction

1.1 Parallelism

The ever-growing need for computation power combined with the advances in VLSI technology created a new research domain called parallel processing. Although parallel processing is achieved quite differently by various approaches—all have the same goal: to increase the performance of a computer system as much as possible by performing activities concurrently. Basically, there are four ways to increase performance:

- Increase the inherent parallelism in the application.
- Use of a better technology.
- Use of hardware parallelism.
- Algorithmic change of the application.

For years, researchers seemed to concentrate only on the last 3 methods: Among the parallel computers the vector machines appeared first. They were targeted to enhance the performance for vector and matrix scientific applications, where the computation could be easily distributed. They included a high-speed

pipelined floating-point unit which reduced the overhead needed in vector computation by applying the same instructions to many different vector elements. These machines are classified according to [Fl66] as SIMD machines. By definition, these machines are powerful only for limited (vector) applications and the fact that one has to issue the same instruction to all functional units limits, naturally, its potential speed-up. The second class of parallel machines was the class of asynchronous multiprocessors or MIMD. The idea behind MIMD was to distribute the computation among several communicating processors to achieve parallelism in the application level.[1] In order to reduce the interprocessor communication cost, which is the bottleneck in MIMD machines, much effort was put into developing parallel algorithms in order to decrease the communication needs of each processor, thereby reducing the overall interprocessor communication overhead. At the present it appears that SIMD's and MIMD's have reached some saturation point due to physical limitations and complexity of developing parallel algorithms.

The RISC [He85] approach was a consequence of the physical limitations and the complexity involved in implementing a memory hierarchy to alleviate the memory access-time problem. The idea was to concentrate on boosting the processor's performance as well as simplifying the memory system. To achieve this a *load/store* architecture was proposed which intergrates a simple and small instruction set (enabling less instruction decoding time, thus reducing the clock cycle) with a fast memory. Furthermore, the simplicity of the instruction set combined with the "cleanliness" of the hardware enabled the use of pipelined functional units with relatively high efficiency. The high degree of architecture efficiency is most likely the main reason for the success of the RISC approach.

[1] Also known as coarse-grain parallelism.

However, RISC is based on a single functional unit. Therefore, the natural following candidate of parallel architecture had been a tightly-coupled synchronous machine with a simple instruction set and *several* functional units. The need for several units coincided well with the fast advances in VLSI technology manifested by much higher integration. These advances made the integration of several functional units on one chip possible and eliminated the inter-processor communication overhead. A representative of the tightly-coupled synchronous architecture is the VLIW (Very Large Instruction Word) architecture [Fi83]. As implied by its name, VLIW enables issuing and execution of several independent operations concurrently. Although the VLIW architecture may be regarded as a horizontally-microcoded engine, it had not been considered as a realistic architecture because of the lack of automatic tools which find these independent operations so that the functional units might be kept busy. Scheduling for these machines by hand, even short programs, turned out to be tedious and error prone task.

Discouraging results published by [TjFl70] and [RiFo72], which showed that the potential parallelism in ordinary scientific programs is on the order of 2-3, prevented for years further research in this area. However, these results were derived looking for parallelism only within the basic blocks of the program. Obviously, if one considers only operations within basic blocks, since the average number of operations within these basic blocks is 4-5, one cannot expect a speed-up greater than 2-3! In addition, there is Amdahl's Law which states that unless the *entire* program can be parallelized—one cannot expect significant parallelism. Fortunately, experiments done by Nicolau and Fisher [NiFi84] showed that there is substantial parallelism available when one goes beyond basic blocks boundaries. Similar results were obtained by [Ku88]. The available parallelism found was one to two orders

of magnitude. These encouraging results led to a growing interest in searching for the *automatic* extraction of potential parallelism in the programs. This parallelism is also known as *fine-grain parallelism.* Fine grain parallelism is the parallelism extracted by our compiler and discussed in this thesis.

1.2 Motivation

The compiler presented in this thesis is geared towards mapping *general* scientific problems with real-time constraints and/or massive time requirements onto high-performance architectures. Since we were looking for considerable amounts of parallelism for *general* scientific problems we have chosen to do it by extracting fine-grain parallelism. Exploiting fine-grain parallelism is a critical part of exploiting all of the parallelism available in a given program, particularly since highly irregular forms of parallelism are often not visible at coarser levels and since the use of low-level parallelism has a multiplicative effect on the overall performance. However, low-level parallelism can be effective only when communication overhead is negligible. Consequently, a synchronous, parallel hardware configuration is required. In this context, VLIW architecture seemed to be an appropriate paradigm for our compiler. VLIW, which emerged as a result of Fisher's [Fi83] work has some nice properties: it supports fine-grain parallelism by enabling the issue of several RISC-like operations each clock cycle; it is completely synchronous (each action takes, a statically predicted, fixed amount of time unlike *superscalars*), thus the communication penalty is zero since data transfers are scheduled statically by the compiler; and it supports homogeneous, conflict-free data paths and its control flow is simple (one thread of control, unlike MIMD machines). VLIW can be thought of as

a "perfect parallelism" architecture: it integrates temporal (functional pipelining) with spatial (several functional units) parallelism. Furthermore, with unlimited resources, VLIW can be considered as a "static data-flow" machine since all operations can be scheduled as soon as their source arguments are available. But while data-flow machines exhibit complex (and expensive) hardware, VLIW may be implemented as a clean and relatively cheap machine due to its simple homogeneous design and since synchronization is handled during compile-time. The original model of VLIW assumes a single program counter and synchronous functional units that execute all operations in one cycle. In order to take advantage of temporal parallelism we extended the original machine model and our transformations to make pipelining feasible. In fact, by selecting the VLIW architecture and by using low-level parallelizing transformations, we take advantage of the first three methods of increasing performance mentioned at the beginning of the chapter: we increase the inherent parallelism by scheduling operations in parallel, we use an advanced technology and we implement hardware parallelism. The last method is achieved in our compiler on a modest scale suitable for integration into a compiler by a new *local* and *incremental* Tree Height Reduction algorithm that rearranges the program, while preserving semantic correctness, such that more parallelism is exposed.

While this dissertation presents a compiler based on advanced compilation techniques that may be used independently, we were also interested in using the compiler in the context of application-specific *system design.* We believe that when considerably high *system* performance is required, problem-specific peculiarities have to be taken into consideration during the design even if it violates the generality property (otherwise, the overhead incurred may dominate the overall

performance or, at least, may significantly constrain parallelism). For example, one application performs best with a hardware configuration that includes a multiplier with 3 pipeline stages while another application performs better with a 4-stage multiplier. In order to achieve high *system* performance, we propose in this thesis an approach which enables *gradual, two dimensional system synthesis and analysis.* This is in contrast to the traditional approach where compilers were written after a system's hardware had been specified. In other words, they were optimizing the performance using only one degree of freedom (the compiler). Naturally, the prespecified hardware may restrict the ability of the compiler to produce good-quality code. We, on the other hand, are trying to optimize the system's performance using two degrees of freedom (compiler and hardware) by allowing gradual fine-tuning of both. By starting from a canonical paradigm, we can iterate on different hardware configurations (all of which are derivations of the canonical form) in order to achieve the best performance for a specific problem. Since a system's hardware and compiler are two interdependent domains, being able to synthesize the system while controlling both may, potentially, yield better performance than when one of them is predefined.

Another objective of this research was to investigate different hardware and parallelization trade-offs in high-performance, parallel architectures. Therefore, the retargetability of our compiler that allowed scheduling for various system configurations was especially important. In this way we were able to investigate the following: What is the impact of register renaming on the overall performance?, How many registers are needed?, How many functional units?, How many pipeline stages for each unit? and How parallelism increases with number of functional units?

1.3 Previous Related Work

Multiflow's Trace compiler can be considered as the first commercial fine-grain compiler capable of extracting parallelism beyond the basic block boundaries. It is based on Fisher's [Fi81] Trace Scheduling (TS) transformations that convert programs to more parallel ones by concentrating on the most probable paths (called traces) taken during their execution. The program is divided into numerous execution paths and then, according to their priority, each path is compacted using semantic-preserving code transformations. First, independent operations are scheduled concurrently and then compensation code is introduced to preserve program correctness at exits to this trace. Trace Scheduling was mainly targeted for scientific code and for applications in which the flow-of-control can be predicted. For these applications experiments found a considerable speed-up of 10-20. However, the TS transformations are inflexible since they are monolithic and dependent on the success of the single trace selection heuristic and limited since the program is compacted one trace at a time and traces cannot be combined. Therefore, for unpredictable flow-of-control applications, usefulness of TS may decrease. Furthermore, the overhead paid by compensation-code on non-selected paths degrades the performance when the run-time paths executed were not those given high priority during scheduling.

Other approach differences between the Trace compiler and our compiler are in the way multi-cycle and pipelined operations are dealt with and in the application of higher-level compaction strategies as discussed in detail in Chapter 3.

IBM's VLIW [Eb88] is a machine currently being built at IBM Yorktown Heights. Its compiler is based on Percolation Scheduling transformations which

were slightly changed to fit the architecture. The interesting features of IBM's compiler are: a conditional execution model (see Chapter 2), a loop pipelining technique [Eb87] to enhance parallelism and a Software Lookahead Window technique [NaEb90] to manage code-explosion and reduce compilation-time.

IBM's compiler assumes that all operations take 1 cycle (no pipelining of operations). In order to meet this constraint, the machine clock was extended such that the "fastest" operation takes the same amount of time as the "slowest" one. This kind of "worst case design" may be unacceptable in some situations.

Breternitz's [Br91] architecture synthesis approach is conceptually similar to our compiler (although the goals are different).[2] Their methodology utilizes an architecture paradigm and a set of tools to generate the structure of customized data and control paths. The synthesis is done in three major tasks: Architecture Synthesis, Specification Synthesis and Implementation Synthesis. The first defines an architectural template for the application. The second task tries to parallelize the application given this architecture template and the Implementation Synthesis phase finds the best implementation for the (previously) parallelized code. The major differences between the architecture synthesis approach and our approach can be summarized as: (a) while the architecture synthesis approach is targeted towards *automatic* generation of architecture for application-specific programs, our compiler is an evaluation compiler. We are interested in exploring hardware and compilation trade-offs to generate the best schedule for a given application and affordable architecture. (b) our approach deals with pipelined or multi-cycle operations, which is a major issue in high-performance, parallel architectures. (c)

[2]In fact, the first prototype of our compiler was jointly developed with John Shen's group in Carnegie Mellon University.

our approach supports modest algorithmic changes of the application program (like Tree Height Reduction). (d) we use different register allocation and resource constrained scheduling algorithms.

1.4 Thesis Overview

This dissertation is organized into 10 chapters. Chapter 2 gives a brief overview of the compiler, its different layers and the approach taken during its design. In Chapter 3 we detail all transformations used in our compiler from the lowest-level transformations to the highest-level compaction strategies. In this chapter we try to answer the question "Why Percolation Scheduling, as defined by Nicolau in [Ni85a], cannot deal with pipelined (multi-cycle) operations?", we present some transformation-related problems and show examples why our approach has a better potential over other approaches for scheduling with pipelined operations.

In Chapter 4 we discuss one of the main issues in parallelism extraction in high-performance architectures: register allocation and register renaming. We show how parallelism is determined by the number of registers used and explain our approach for local register renaming during the parallelization process. Since our technique involves, in general, addition of copy operations into the code (and in the case of loops some of these copies become redundant induction variable) it is crucial to be able to remove them. For completeness purposes, we describe a technique, written at UC Irvine by Haigeng Wang, that removes copy operations

generated during renaming. Other (non-induction-variable generated) copies are removed by a copy elimination technique that is described at the end of the chapter.

Chapter 5 discusses the way we perform resource constrained scheduling. We give the motivation for our approach which involves scheduling with unlimited resources and then mapping of this schedule to the given (constrained) architecture. In Chapter 6 we present a novel incremental Tree Height Reduction algorithm. This algorithm compacts programs even when data-dependency prevents further compaction. We show several examples that clarify the algorithm and justify our claims.

Chapter 7 is devoted to specific issues that are peculiar to the compiler and are not directly related to any of the other chapters. We describe our "on-the-fly" dead-code-elimination technique, we show how we perform inter-procedural live analysis and describe a code optimization called "load-after-store optimization" that removes unnecessary memory loads for memory-traffic reduction. We then address the loop detection problem and explain how memory reference disambiguation, which is critical for significant parallelism extraction, is done in our compiler. Our simulator is described at the end of this chapter. The results of benchmarking the compiler in various configurations and the analysis of these results are discussed in Chapter 8.

In Chapter 9 we explain why our compiler can be considered a High Level Synthesis (HLS) tool, we give an example to illustrate our *gradual, two dimensional design approach* and suggest some future extensions. We conclude the dissertation in Chapter 10 with a summary, discussion and future work.

Chapter 2

Overview of the Compiler

When building a fine-grain, parallelizing, optimizing compiler, many trade-offs and design decisions have to be made. This chapter lays out the approach taken during the compiler's design and the main trade-offs that inspired it. We also detail the machine model used throughout the thesis.

2.1 Layered Structure

While some commercial compilers are built monolithically with the essential procedures merged into a single module, our compiler is implemented in several layers as shown in Figure 2.1. The layered structure is especially important in our context since one of the primary objectives of this research was to evaluate different configurations and design alternatives. It is possible that a monolithic compiler would be more efficient but it would be harder to reconfigure and debug. If the interfaces between the layers are well-defined, the reconfiguration price is usually that of replacing one layer by another (compatible) one, which is relatively easy.

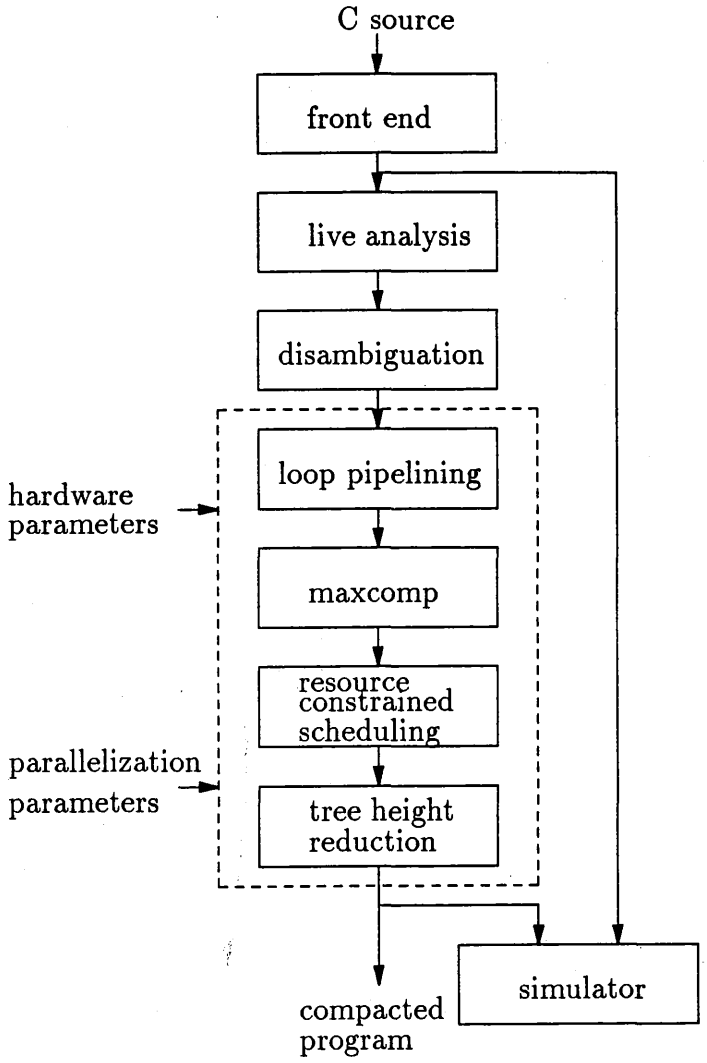


Figure 2.1: Compiler's Layers

For example, in order to check another loop pipelining algorithm, the only change needed is to replace the layer called loop pipelining. All others remain intact. Another by-product of this philosophy is that it is easier to reconfigure the compiler even without recompiling it: we keep a configuration file in which all layers are specified. The compiler decides which layers to execute by reading this file. In this way, disabling a layer means commenting out the corresponding line in the configuration file.

The front end is a *modified* GNU front end that converts a C source input to an abstract, machine-independent RISC-like load/store three-address code.[1] All optimizations and transformations are done on this code. As an example, consider the following C program:

for $(i = 1; i < 20; i++)$
  $A[i] = A[i-1] + B[i];$

The corresponding formal three-address code for this program is shown below. Next to it we give a readable format which is used throughout the thesis: (For readability reasons we use *base*, i, a, b, $cc0$ as virtual hardware registers.)

```
        (PROC_BEGIN main
(LABEL main)
        (iconstant base 176)        base := 176
        (iconstant i 0)             i := 0
(LABEL L1)
        (ivload a base -88)         a := M[base - 88]
```

[1]Which is a subset of N-address code, NADDR. We used the MIPS implementation of the GNU as a basis for our front-end and modified it to produce our abstract format.

(ivload *b base* -164)	$b := M[base - 164]$
(iadd *a a b*)	$a := a + b$
(ivstore *base a* -84)	$M[base - 84] := a$
(iadd *base base* 4)	$base := base + 4$
(iadd *i i* 4)	$i := i + 4$
(ile *cc*0 *i* 76)	$cco := (i <= 76)?$
(if *cc*0 (LABEL L1))	
(LABEL exit)	
(igoto $31)	return
(PROC_END main))	

The destination register is specified first followed by the source variables. Each operation is preceded by the letter "i" or the letter "f" indicating whether the operation is an integer operation or floating-point one. The last operation in each procedure (igoto $31) represents a return from the procedure through an address specified in register $31. Since store operations have no register destination, their notation is a bit different than the others: (ivstore *base a* -84) means M[*base*-84]:= *a*. Comparison operations always set *condition-code registers* (*cc*). In the example above, *cc*0 is assigned with the *logical* value of the (integer) comparison between *i* and 76. Subsequently, a conditional jump operation controls the program's flow by testing the updated condition code register.

The disambiguation layer is responsible for determining whether two indirect (array) memory references of two memory-accessing operations (loads and stores) are actually referring to the same memory location in which case their schedule-ordering matters. Like other non-memory-accessing operations, memory loads and

stores may cause data-dependencies. Since data-dependencies restrict the parallelization process and since indirect memory references are very common in many application programs, it is very important to be able to disambiguate (anti-alias) these memory references. While in the case of non-memory-accessing operations it is quite easy to check for a dependency, the dependency test when loads and stores are involved requires quite complex computations. Consider, for example, this code segment:

1. $a := b + c[ref1];$
2. $c[ref2] := d + e;$

If we are unable to show that the two references ref1 and ref2 are always different, these two operations should not be scheduled independently. The "easy" solution of assuming that these references could be equal without trying to disambiguate causes a considerable degradation in performance [Ni84]. Our compiler uses a powerful and efficient fine-grain disambiguation technique to enable considerable parallelism extraction. The technique, which was implemented in our compiler by Haigeng Wang, works by computing the *symbolic derivations* of all loads and stores in the program. The symbolic derivations are the corresponding addresses expressed in symbolic terms that enable simpler comparison of addresses.

Loop pipelining is a class of techniques for extracting parallelism by overlapping execution of consecutive iterations of the same loop. In other words, the next iteration may be initiated before the current iteration is completed. This causes a pipelining effect similar to the one found in hardware pipelining. Since ordinary programs tend to spend most of their time executing loops, the ability to parallelize loops has a major impact on the overall parallelism.

Maxcomp is a transformation to maximally compact the non-inner-loop parts of the program. After this layer the program is maximally compacted and the parallelism is restricted only by data-dependency.

The resource-constrained scheduling layer is the one responsible for the mapping of the unlimited-resources schedule (produced by previous layers) to the given machine. The schedule derived up to this layer assumes that all needed resources are available. In practice, machines have fewer functional units than required by the unlimited resources schedule. In order to "fit" the schedule for the given machine, the schedule is adjusted so that no more operations are issued per cycle than allowed by the hardware constraints. Since the RCS problem is known to be NP-hard it must be solved in practice by heuristics. The lack of optimality raises the question: "What is a good RC schedule?". Another common question regarding RCS is "When to apply the heuristics?". We present in Chapter 5 a new algorithm for performing RCS that is applied *after* the extraction of the unlimited-resources schedule. We show in [PLNG90] that applying RCS after unlimited-resources scheduling (which is in many cases the optimal schedule and does not involve application of heuristics) rather than scheduling *from the beginning* with RCS, is one of the most important factors in scheduling. The are four reasons for this: First, when parallelism is limited it may be critical to do very well. Hence, too early application of heuristics may further decrease performance. Second, the application of heuristics *after* the extraction of unlimited resources parallelism offers flexibility to tune *only the heuristics* to get better speed-up. Third, this approach gives a good lower bound for the total execution time. Without this bound it is sometimes very difficult to estimate what would be the best performance of an application,

making the heuristic algorithm harder (When to stop?, What is a "good" schedule?). Fourth, with this approach it is easy to compare different RCS heuristics and schedule for a variety of different resource constraints, starting from the same unlimited-resources schedule.

All transformations applied to this point do *not* change the code structure. All operations in the compacted code are either copies or derivatives of operations in the input (serial) code. The next layer, the Tree Height Reduction layer, *does* change the code. When data-dependency restricts the parallelization, Tree Height Reduction is applied to further compact the program by performing an algorithmic change on the code in case there are unused resources. Although this process involves structural change of the code, it provably preserves the semantics of the original code.

As shown in Figure 2.1, several hardware and parallelization parameters may be tested by inputing different constraints to the compiler. The constraints include the number of pipeline stages needed for each multi-cycle operation, the number of functional units and registers available and whether a conditional execution model is supported by the machine or not. Parallelization parameters may include renaming, induction variable removal, copy elimination, load-after-store optimization, compaction without moving conditional jumps and turning disambiguation off.

The simulator is a very important layer in our compiler. As mentioned in Chapter 1 one of our objectives in this research was to explore hardware/software trade-offs by measuring the performance of the compiler under variety of configurations/conditions. While a "traditional" compiler can be evaluated and tested by

running it on its target architecture, we needed a simulator to test our compiler in these different environments. The simulator served also as a validation tool. It turns out that, even for very short programs, it is impractical to check the correctness of the results produced by the compiler. The presence of conditional jumps in the code and the fact that our transformations compact numerous operations into trees, formed by these conditionals, make hand-debugging (or hand-checking) non realistic. The simulator checks the correctness by comparing the results obtained by running the serial program on a set of input data and the results obtained by the compacted program, running on the same data. In this context, we also tuned our simulator to serve as an efficient debugging tool.

2.2 Hierarchical Approach

The compiler is built hierarchically. On the lowest level there are the core Pipelined Percolation Scheduling (PPS) transformations that convert an original *program graph* into a more parallel one. The transformations define the rules for moving operation between two adjacent nodes in the graph (see Section 3.4). Since even for small programs the number of transformations required to compact the graph is considerable, a set of higher-level transformations is required. The higher-level transformations "guide" the low-level transformations in an attempt to expose as much parallelism as possible. The hierarchical approach provides easier examination of different control strategies by abstraction of the parallelizing process: the system, rather than the user, deals with the burdensome details of correctness-preservation while the user can concentrate on different higher-level

control strategies. The layered structure of the compiler integrates well with the hierarchical approach. Since the interface with the low-level transformations and the interfaces between the layers are well-defined—exploring a new higher-level technique, usually, means a substitution of a present layer with the new one; thereby eliminating the need to make drastic changes in several layers.

The two main higher-level transformations used are loop pipelining and maxcomp. Loop pipelining invokes the core transformations in order to extract parallelism both across and within inner-loop iterations in the program. Maxcomp is a transformation to maximally compact the non-loop or non-inner-loop portions of the code. These transformations are described in detail in Sections 3.5.2 and 3.5.1.

2.3 Incremental Register Allocation and Renaming

As in most existing compilers, the GNU front-end that we use also tries to make an efficient use of registers by using the minimal number possible. This is achieved by assigning the same registers to variables whose life-spans do not overlap. While saving registers may be a good strategy for sequential machine, for parallel, high performance architectures the re-use of registers introduces false dependencies (either "anti" or "output") that may severely decrease the performance since it limits the compiler's ability to compact.

In the past, when VLSI technology limited the number of registers that would be integrated into one chip, efficient use of registers was a primary issue in any chip design. However, today, with the recent advances in VLSI technology, the number of registers is considered a less-serious bottleneck (although the cost of routing to and from multi-ported register-files increased). When building a high performance architecture, one cannot afford to limit the number of registers to the number that was used only a few years ago.

Ideally, in parallel architectures, only true data-dependency and resource availability should limit the parallelism. On the other hand, register allocation has to be done—we simply do not have an infinite number of registers. In order to solve this problem we propose a new approach for register allocation. We begin with GNU's register allocation and during the parallelization process we allow renaming of registers to eliminate false dependencies that *prevent otherwise desirable/feasible code transformations*, provided that a register is available at that point.

2.4 Machine Model

Throughout this thesis we assume a modular architecture which serves as a paradigm for our target architecture. This architecture is illustrated in Figure 2.2. The paradigm has a single flow-of-control (single PC), it is *totally* synchronous, has multiple functional units and a multi-ported register-file. Each functional unit may have a variable number of pipeline stages. The model, called the *conditional execution model*, allows the presence of multiple conditional jumps (as well as other operations) per cycle. This model is similar to the one used by the IBM

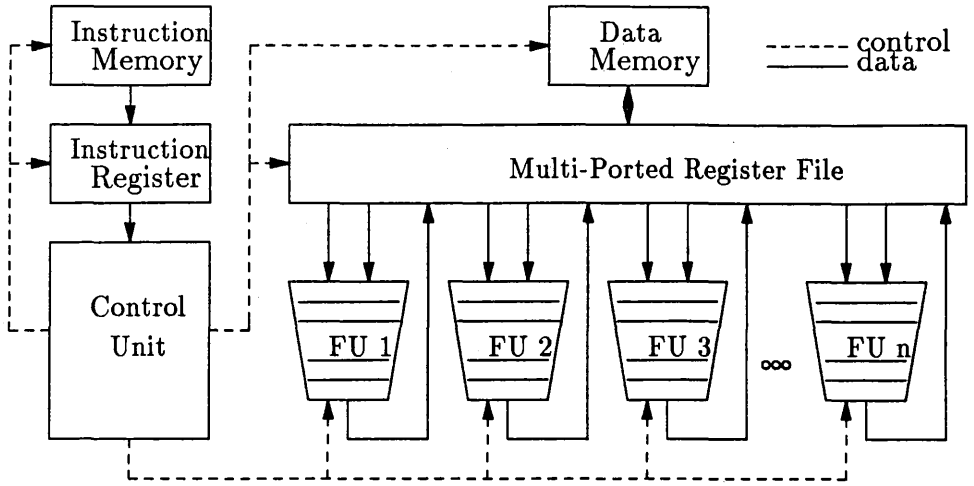


Figure 2.2: Modular Architecture

VLIW machine [Eb88] and allows a direct mapping of the schedule into horizontal microcode.

The input program is represented by a control flow graph. The vertices (nodes) in the graph correspond to operations executed in each cycle. The edges represent flow of control from one node to its successor. Initially all nodes contain a single operation. Making a program "more parallel" involves compaction of several operations into one node (or microinstruction) while preserving the semantics of the sequential code. In the original PS model [Ni85a] a node represented a large instruction word containing several operations (all of which executed in parallel) and a tree-like structure of conditional jumps. The actual value of these condition code registers determined the next instruction. The model described in [Eb88] is a refinement on the original model. The difference is that two operations writing to the same destination are allowed to be packed into one node (the conditional codes are used to allow only operations on the actual execution path to write their results).

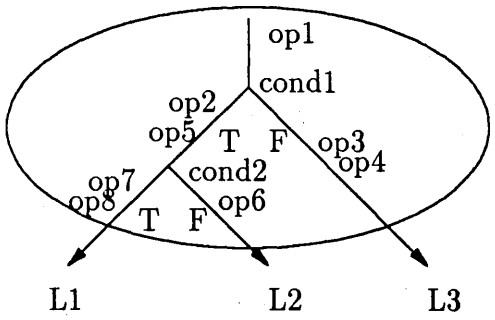


Figure 2.3: A Node in Our Machine

Operations form a tree-like structure in the node. Exactly one execution path is selected from the root down to the unique successor of the node. This path is selected according to the results of the conditional operations in the tree. For example, if we assume that cond1 is true and cond2 is false in Figure 2.3, then op1, op2, op5 and op6 are to be executed and the successor of this node is L2. The execution of the node can be conceptualized as three steps (on the IBM machine, these three steps are executed as part of one basic machine cycle):

1. Operands (for all operations) and condition code registers are read.
2. All operations are executed, condition codes are evaluated and a path to the (unique) successor instruction is chosen.
3. The results of operations on the path chosen (for single cycle operations) and the results of (pipelined) operations which were issued in previous nodes and completed their execution in this node, are written back to the register file/data memory.

This model is similar to any microcode-based architecture supporting a multiway jump mechanism.[2] A simple mapping to other machines is also feasible. If the

[2]In fact, work done at UC Irvine [Ab91, APBN91] implements this multiway branching and conditional execution model.

machine does not support this sort of conditional execution, the transformations can be simply modified to disallow moving multiple operations writing to the same destination and multiple conditional jumps into the same node.

The pipeline model we use assumes that each operation can take several cycles (or nodes) to complete. In order to reflect the effect of multi-cycle operation on the program graph, we assume that each operation (with latency of k cycles) reads its source arguments during the first cycle (in which it is specified). Then, for $k-1$ cycles, it uses the stages of its functional unit. In the last cycle, it writes back the result to its destination. All operations which use the result of this operation should be scheduled at least k cycles later than the beginning of this operation. This model encompasses both pipelined functional units (where we can issue a new operation every cycle) and multi-cycle functional units (where we have to flush the unit before issuing the next operation).

The model described here (as opposed to superscalars, for example) assumes that everything can be determined statically. It is the compiler's responsibility that all data paths will be conflict-free (i.e. multi-ported register file and memory accesses). The only exception is when a cache miss occurs. Then, the processor's activity is frozen until the data is fetched from memory. Throughout this thesis we assumed a "clean" model with only one register-file and a single data memory. When implementation issues require *several* register files or memory banking we further assume that all accesses are uniform (i.e. each functional unit can access register-file and memory bank). When the actual hardware is unable to support such accesses, it is again the compiler's job to orchestrate these implementation-hazards. While superscalars can take advantage of the time lost by VLIWs during cache misses, they are certainly more complicated and therefore their hardware

is more complex. More complex hardware usually means a slower machine. In addition, since superscalars schedule operations dynamically their ability to do a good job is constrained by the cycle time of the processor.

Chapter 3

The Parallelizing Transformations

3.1 Fine-Grain Transformations for Pipelined Architectures

Compile-time techniques to extract parallelism from ordinary programs can be grouped into two classes: coarse-grain parallelizing techniques and fine-grain parallelizing techniques. While coarse-grain techniques expose parallelism at the implementation level (this parallelism is often visible and easier to detect), they cannot take advantage of highly irregular forms of parallelism exhibited in the fine-grain (instruction) level. However, exploiting fine-grain parallelism is critical since low-level parallelism has a multiplicative effect on the overall achievable parallelism. By definition, fine-grain transformations are low-level transformations. Therefore, even for short programs, the number of transformations needed can be considerable and that is why fine-grain transformations need to be very efficient. We have chosen Percolation Scheduling (PS) transformations as a basis for our compiler since they are *efficient* (need only local data), *atomic* (applied to two

adjacent nodes in the program graph), and virtually *complete* with respect to the set of all possible local, dependency-preserving transformations on program graphs [Ai88]. The atomicity is especially important since it enables easy integration with a wide variety of higher-level transformations which may expose coarse-grain parallelism as well. The transformations described in this chapter deal with both levels of parallelism—we first detail the low-level transformations and continue with the higher-level ones.

When trying to extract substantial parallelism for high-performance, parallel architectures, one of the promising ways to do this is by exploiting both spatial and temporal parallelism. To enhance temporal parallelism, machines are built with pipelined functional units, therefore, a pipeline-supporting set of transformations is required. Neither Trace Scheduling (TS) nor PS integrate multi-cycle operations within their transformational model. Ellis [El86] uses a partial solution by insertion of a *partial schedule* into the code for multi-cycle operations for the trace selected for compaction. While this approach may yield good results for the selected traces—for off-trace paths the *partial schedule* can cause poor compaction. The poor compaction in off-traces is a result of inherent TS limitations[1] amplified by the fact that operations are multi-cycled (more no-ops in the code). Since functional unit pipelining is particularly important when loop pipelining is applied and since loop pipelining is a major parallelizing technique in our compiler, we had to extend the original transformations to be able to deal with pipelined operations. The extended transformations are called Pipelined Percolation Scheduling (PPS).

[1]TS can not move two syntactical copies of an operation above conditional jump while the unify transformation in PS enables it.

Two previous approaches have been taken to deal with the differences in operations' latencies. In the first approach, a parallelizing compiler (assuming that *all* operations take the same amount of time, i.e. one time-unit) was built and then a supporting architecture was fitted. In such a way the machine cycle time was extended to satisfy the slowest operation's latency. This approach was obviously inefficient since both the fastest and slowest operations required the same amount of time to execute. The other approach schedules in two phases. First, the code was compacted as if each operation took one cycle. Then, in a second phase, empty cycles were inserted between adjacent nodes, such that latency times of the operations were satisfied. Again, this strategy assumes worst case design, because the number of empty cycles inserted is the maximal latency on the path connecting the two nodes—a fact that caused frequent idle time for some functional units. To our knowledge no technique to date allows uniform compaction to go beyond straight line code. In other words, there are no previous general approaches to exploit spatial and temporal parallelism. This is precisely the role of PPS.

To substantiate our claims we show in Section 3.7 representative examples where PPS produces better schedules than those achieved by the other two approaches mentioned.

3.2 So Why is PPS Needed?

In the original PS model, all nodes contain operations that are issued and executed at the same time in one cycle, hence, in PS only a node and its immediate successors need to be considered to preserve correctness of the transformations.

However, when dealing with the movement of multi-cycle operations (whose execution can span several nodes beyond the one in which they issue), care must be taken to keep correct information in all these successors on the execution path of the operation, making the correctness preservation issue much harder. The cause of difficulty is that we want to refrain from using pipeline interlocks. We want to guarantee that whatever execution path is taken (even if the path is determined *after* the operation has been issued)—the correctness is preserved on all possible paths.

Suppose, for example, in Figure 3.1 that we try to apply *move-op* to *op* (i.e. schedule *op* in node m instead of node n). Let's assume that there is no data-dependency between *op* and all other operations in node m and that a is dead at the top of node p. In this case PS would allow the motion. But, when dealing with multi-cycle operations, these operations may have effect on those successors other than the immediate ones. In this example, if *op* takes 5 cycles to complete and *op1* takes 2 cycles, by moving *op* to node m, we change the value of a used in nodes following node k. Hence, in addition to conditions which are required by PS we need to impose further restrictions on the move.

3.3 Definitions

In this section we define terms used later in this chapter.

- *latency* and *distance*

  The latency of an operation is the number of pipeline stages needed to execute

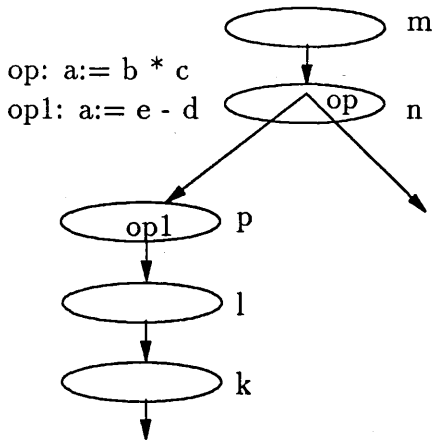


Figure 3.1: Why is PPS Needed?

this operation. The distance between some node s and some node n is the number of nodes between them plus 1.

- *pipeline analysis*

  The movement of a multi-cycle operation from one node to its predecessor may affect not only these two nodes but also other nodes in the program. The affected nodes are those within a distance less than or equal to this operation's latency. The procedure to check correctness (in all these nodes) when trying to move the operation is called *pipeline analysis.*

- *pipe_set*

  The original PS transformations use data-dependency analysis and live/dead analysis to check whether the movement of an operation is allowed. The *live_set* of a node is the set of variables live at the top of the node. The *kill_set* of a node is the set of variables written (or defined) in the node. For PPS, we also need to keep (locally, in each node) information about the effect of multi-cycle operations on successor nodes. Each node in the graph contains a set of elements called *pipe_set.* Each element in the set

represents an operation which is in the process of evaluation in this node. The operation is represented either by its destination register (for non-store operations) or by its memory location (for store operations). If, for example, an operation writes into register a, issued in node n and has a latency of 5, then in another node k, which is 2 cycles later, it is represented by the value a and an integer equal to 2 which is the number of cycles still needed to complete its execution. The *pipe_set* information is added to the program graph when new empty nodes are inserted in the original graph between adjacent nodes to satisfy pipeline latencies. This information is updated incrementally during the movement of operations from one node to another.

- *induced_field*

  The *induced_field* of a node is information similar in notion to the *pipe_set* information. The difference is that while *pipe_set* represents *all* operations which are under evaluation in this node, the *induced_field* information excludes operations which have been issued in this node. In other words, it represents the "net" effect of operations in preceding nodes on this node. The name *induced_field* reflects the fact that operation "induces" a field (which is its destination-register or memory location) on successor nodes which can *not* be seen by simply looking at the operations specified in this node.

  An illustration is given in Figure 3.2. Assume that op1 and op2 take 3 cycles to complete and op3 and op4 take 2 cycles. The *pipe_set* of node n includes the values {r1,r2,r3,r4}. This means that operations 1, 2, 3 and 4 are in the process of evaluation. The *induced_field* in node n includes the values {r1,r2}. These values ({r1,r2}) represent the effect of operations in predecessor nodes of n on this node.

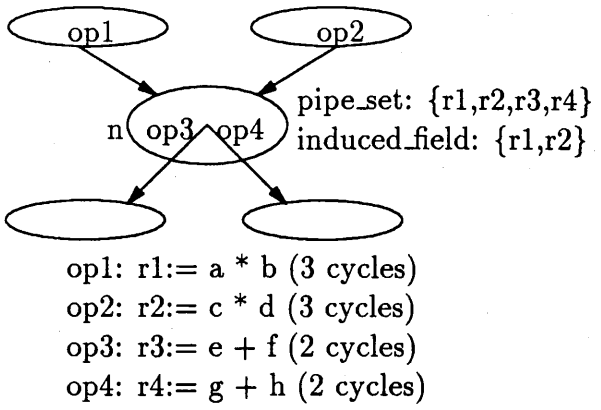


Figure 3.2: Pipe_set and Induced_field Of a Node.

- *trail*

  The *trail* of an operation is the path(s)/branch(es) in the node originating at the node's entry point and encompassing the operation. In Figure 2.3, the *trail* of op6 is the path from the entry point of the node leading to instruction L2. The *trail* of op2 and op5 are the paths from the node's entry to L1 and L2.

3.4 PPS Transformations

PS uses four core transformations (*move-op, move-cj, delete* and *unify*) to compact programs. The actual code motion is done by the first two transformations while the other two can be considered as optimization transformations. We detail PPS transformations as an extension of the PS transformations described in [Ni84, Ni85a, AiNi88a, EbNi89].

3.4.1 Move-op

Move-op is a transformation to move an operation up from a node n to one of its predecessors m. The original *move-op* does not allow the motion of *op* when its destination is either read by another operation in *from-node* or when its destination is live on a path in *from-node* other than *op*'s *trail*. We use throughout this thesis the modified version of *move-op* that allows the motion by doing renaming. For more detail refer to [EbNi89].

The move-op of an operation *op* from a node n to node m is possible whenever:

1. No operation in m writes one of *op*'s source variables (data-dependency test).
2. None of *op*'s source variables is an element in m's *induced_field* (pipeline dependency).
3. No successor s of m through a path *not* in *op*'s *trail* which is at distance less than *op*'s latency includes *op*'s destination in its *pipe_set*.

   OR

   If one of these successors does include *op*'s destination in its *pipe_set*, then, either the *pipe_set* for all successors of n at distance equal to $(latency - 2)$ includes at least one element of *op*'s destination or *op*'s destination is dead at all nodes that do not include it. (Pipeline dependency and correctness in all paths other than *op*'s *trail*).
4. All successors s of n in *op*'s *trail* and at a distance equal to $(latency - 2)$ have in their *pipe_set* one and only one element of *op*'s destination. (correctness in all successors through *op*'s *trail*).

We refer to the last three conditions as pipeline dependency. Condition 1 tests the data-dependency between the operation we want to move and all operations in the target node.

Condition 2 guarantees that an operation which reads a value won't read it while the operation producing the value is still being executed in the functional unit. (Remember that *induced_field* is an indication that an operation is still under evaluation).

Condition 3 is to ensure that the movement of *op* up does not interfere with the same destination register read by operations in successor nodes which are not in *op*'s *trail.* In PS this test is done by checking that *op*'s destination is dead on these paths (otherwise we need to rename *op*), but since PPS deals with operations which may have different latencies, *op*'s destination may be dead but it still may overwrite the destination of another operation (due to larger latency), thus changing the semantics of the program. To understand the first part of condition 3 refer again to Figure 3.1. Let's assume that op takes 3 cycles and op1 takes 2 cycles to complete. In this case there is no problem in moving op from n to m because all successors of n at a distance equal to $latency - 2$ (node p in this case) include at least one element of *op*'s kind in their *pipe_set.* It means that op1 modifies the register-file after op which guarantees a correct order of modification. If, on the other hand, op take 5 cycles the move is not allowed since op modifies the register-file in node k (after op1 has modified it) and therefore all nodes following k will read op's value instead of op1's. The second part of condition 3 is to guarantee that if there *is* a conflict due to the fact that two operations modify the same register in the register-file in the same node, the order of modification will be preserved. If all successors of n at a distance equal to $(latency - 2)$ have in their *pipe_set* at

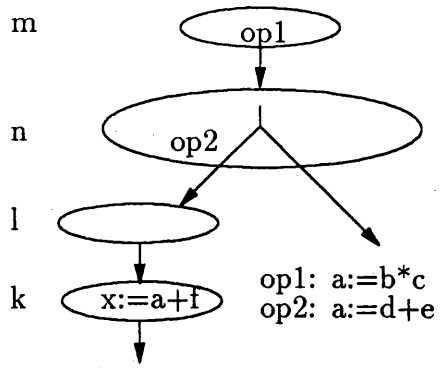


Figure 3.3: Explanatory Example for Condition 4

least one element of *op*'s destination, it means that another operation is inducing the same field on the same successors as *op* induces. After the move, the other operation is guaranteed to modify its destination later than *op* so there is no reason to prevent the motion.

Condition 4 is necessary to keep the correct order of modification of *op*'s destination in successors of *op*'s *trail*. This addresses the problem caused by two different operations that complete their execution in the same node. For example, in Figure 3.3, op1 and op2 both complete in node *l*, assuming multiplication takes 3 cycles and addition takes 2 cycles. Remember that in our model, the value of *a* in node *k* is determined by op2 (which reads its source variables later), so the operation in node *k* should read the value computed by op2). If we move op2 up to *m* we may expose node *k* to a "new" value which is op1's result, causing a change in the program semantics. In order to prevent this situation, we check *all* successors of *n* in *op*'s trail and at distance $(latency - 2)$ from *n*. It is clear that if *all* these nodes have a *pipe_set* which includes more than 1 of *op*'s destination registers, two different operations might complete in the same node *k*. Hence, the move is disallowed.

The move-op version we use allows renaming of variables wherever there is a false dependency (anti-dependency) between operations. In this context, it is important to note that variable renaming is of major importance in pipelined machines. While in single clock machines this "artificial" dependency prevents an operation from moving up one step (cycle), in pipelined machines, it blocks the move up of several cycles. This could cause a substantial degradation in parallelism extraction. The code outline for the transformation follows.

```
procedure move-op(op: operation, n: from_node,m: to_node, tp: path_to)
  /* latency= the latency of op */
  /* dest= destination-register or memory location of op */
  /* dest'= renamed destination-register */
  if (all 4 conditions met) begin
    if /* write-live conditions */
      (there is an op' in n other than op such that
      intersection( reads(op'), writes(op) ) ! = nil)
      or
      (for one successors s of n op is live and not killed in path to s)
        rename_flag= TRUE;
    /* actual move */
    create a copy n' of n;
    unify(n',op); /* unification */
    move op into tail of tp in m;
    make m go to n' instead of n (on path tp only!);
    if (there exist op' in m s.t. writes(op')= writes(op)) and
      (op' is on a path in m going through the newly added op) begin
```

```
        delete op' from these paths;
        push it down to the branch not leading to op;
    end
    /* modify pipeline characteristics */
    modify the pipe_set of m to reflect addition of op;
    modify the pipe_set of n' to be equal to:
      the induced_field of n plus
      all destinations of operations issued in n';
    for all successors s of m in paths reached through n'
      but NOT in op's trail and in distance <= latency
      add dest to pipe_set(s);
    if (rename_flag= TRUE) begin
      for all successors s of n' at distance= latency
        add copy operations: dest:= dest';
      for all successors s of m at distance < latency
        add dest' to pipe_set(s);
    end
    if (n has no predecessors)
      delete(n);
    if (deletion succeeded)
      for all successors s of n at a distance <= latency
        remove dest from pipe_set(s);
    update live-dead information in m;
  end /* move */
end (move-op)
```

3.4.2 Move-cj

Move-cj performs the movement of a conditional jump from a node n to a predecessor node m. Since a conditional jump operation does not modify any variable, the last two conditions of move-op are irrelevant here. So, the movement of a conditional jump from node n to node m is possible if:

1. No operation in m writes one of op's source variables (data-dependency test).
2. None of op's source variables is an element in m's induced_field (pipeline dependency).

```
procedure move-cj(op: cj, n: from_node, m: to_node, tp: path_to)
  /* latency= the latency of op */
  if (condition1 and condition2 are met) begin
    create a copy nT of n, that behaves as if op in n
      always took the true branch, and unify(nT,op);
    create a copy nF of n, that behaves as if op in n
      always took the false branch, and unify(nF,op);
    create a new (cj1: copy of op) conditional jump and place
      it as the last operation on path tp in m and make:
      the target of the true branch of cj1 is nT;
      the target of the false branch of cj1 is nF;
    /* modify pipeline characteristics */
    modify pipe_set(nT); /* see n' in move-op */
    modify pipe_set(nF); /* see n' in move-op */
    for all successors s of nT at a distance<=latency
      modify pipe_set(s);
```

```
    for all successors s of nF at a distance<=latency
      modify pipe_set(s);
    /* try to delete n */
    if n has no predecessors
      delete(n);
    if the deletion succeeded
      for all successors s of n at a distance<=latency
        reflect the deletion of n in pipe_set(s);
    update live-dead information in nT;
    update live-dead information in nF;
  end /* move */
end (move-cj)
```

3.4.3 Delete

The PPS delete transformation is very similar to the PS delete transformation. Delete removes nodes that do not contain any operation or nodes that have no predecessors from the program graph. It is called by move-op and move-cj after a successful move. The only difference is that a node which does not include any operation can not be simply removed because it might serve as a delay node for satisfying pipeline latencies. Hence, in addition to checking predecessors, one has to check whether the *pipe_set* field in this node is empty, which means that no operation is under evaluation during this node.

procedure delete (n: node)

```
if ( (n has no predecessors) or (n has no operations) ) and
  (pipe_set(n) is empty) begin
    for all successors s of n at a distance<=latency
      remove all corresponding pipeline elements from pipe_set(s);
    remove n from the program and adjust links;
  end
end (delete)
```

3.4.4 Unify

Unify merges syntactical copies of an operation from different branches of a node into a unique copy placed in a mutual predecessor node. It is called by move-op and move-cj after a successful move. The difference between the original unify transformation and PPS unify is that now, we have to reflect the deletion of all syntactical copies of op in all successors of these branches. In other words we have to modify the pipeline information in all relevant successor nodes.

```
procedure unify (n: node, op: operation)
  for all copies op' of op in n
    for all successors s of n in trail of op' at a distance <= latency
      remove corresponding pipeline element from pipe_set(s) and induced_field(s);
end (unify)
```

3.5 Higher-Level Transformations

The transformations described in previous sections are the lower-level transformations that perform the actual motion of operations while preserving correctness. However, some higher-level transformations are required in order to direct the low-level transformations and in order to extract coarser-grain parallelism which is sometimes exhibited only in higher-level program constructs. Maxcomp is an example of a transformation required to "guide" the low-level core transformations in order to maximaly compact the program. It works by moving operations as high as possible in the program graph. The goal of Maxcomp is to schedule all operations as soon as possible, subject only to data-dependencies. Another example of a higher-level transformation that extracts parallelism in loops is called Loop Pipelining. It exploits parallelism across as well as within loop iterations.

3.5.1 Maxcomp

```
procedure Maxcomp(program)
  /* real successor= successor not through backedge */
  /* fence= global list of nodes; */
  for each instruction n in the program filled(n)= FALSE;
  let fence= {header instruction of the program};
  let newfence= {empty set};
  while fence is not empty begin
    for each instruction n in fence
      fill_instr(n);
```

```
    for each instruction n in fence begin
      for all successors s of n
        if ( (s is a real successor of n) and
             (not filled(s)) and
             (s is not in fence) and
             (s is not in newfence) )
          add s to newfence;
      mark filled(n)= TRUE;
    end
    let fence= newfence;
    let newfence= {empty set};
  end (while)
end (Maxcomp)
```

The idea behind Maxcomp is to "push" operations up by filling all nodes in the program with operations in their successors. Fill_instr tries to move operations from all successors of n into n using move-op and move-cj. In order to guarantee that an operation will move up as high as possible, whenever an operation has moved into n all of n's predecessors are added to the (global) *fence*. Consequently, when nodes in the new *fence* are filled, this operation can move up from n to its predecessor.

```
procedure fill_instr(n: to_node)
  /* fence= global list of nodes */
  for each successors s of n in paths t from n to s begin
    for each operation op in s
```

```
      move-op(op, s, n, t);
    for each conditional jump cj in s
      move-cj(cj, s, n, t);
    if (any operation moved out of s)
      for each predecessors p of s
        if (p is not in fence)
          add p to fence;
  end
  if (any operation moved into n)
    for each predecessors p of n
      if (p is not in fence)
        add p to fence;
end (fill_instr)
```

3.5.2 Loop Pipelining

Since ordinary programs tend to spend most of their time executing loops, the ability to parallelize loops has a major impact on the overall parallelization of the program. Loop pipelining is a class of techniques for extracting parallelism within and across iterations, by overlapping execution of operations from multiple iterations of the same loop. This causes a pipelining effect similar to that found in hardware pipelining.

The loop pipelining algorithm used by our compiler was inspired by the Perfect Pipelining (PP) algorithm described in [AiNi88c] and the enhanced pipeline

scheduling technique detailed in [EbNa89]. Loop pipelining is applied to all innermost loops in the program one by one.

The main goal of any software loop pipelining technique is to schedule different iterations of the same loop in an overlapped fashion so that an iteration may start before the previous one has ended. In our compiler the overlapping is achieved by allowing operations from the next iteration to percolate up into nodes containing operations from the current iteration. The loop pipelining works as follows. We keep a global list of instructions called *pipeline_fence* which is sorted originally in a depth-first order. Initially the list contains the loop header only. First, all operations in the loop are allowed to move up as high as possible by move-op and move-cj transformations (so that the loop body is fully compacted). Then, we virtually unwind the loop header by increasing its depth-first number so that it is greater than the depth-first number of all its successors. This makes the header a *real successor* of all its predecessors and allows operations from the header (which now represent the next iteration) to percolate up into nodes from previous iteration. Concurrently, the *pipeline_fence* is replaced by all successors of nodes that are in the *pipeline_fence*. During this stage of compaction, nodes that were previously added to the *pipeline_fence* are not allowed to break apart. In other words, either *all* their operations can move up or none. Only later, in a post-pass, we allow operations from *all* nodes to percolate up in the loop. This process continues with modification of the *pipeline_fence* until no nodes can be added to the *pipeline_fence*.

The algorithm, as presented, enhances the technique in [EbNa89] to be equivalent to PP. Since Ebcioglu and Nakatani are allowing operations to move *only into nodes that are currently in the pipeline_fence* (in other words, nodes that are not in

the *pipeline_fence* can not be parallelized) and since they do not allow the splitting of those nodes that have already been in the *pipeline_fence*, their algorithm tends to converge too fast. While in the case of uni-cycle operations the fast convergence is not a determinant factor, with multi-cycle operations it significantly affects the speed-up. We alleviated this problem by integrating two changes: first, we allow *all* nodes in the loop to be filled (more aggressive compaction) and second we apply the post-pass compaction phase to *all* nodes such that we actually "simulate" the PP technique (in the limit) in a controlled and simpler fashion. In the algorithm that follows, Maxcomp-for-pipelining() performs the same function as Maxcomp() except that nodes that have already been in the *pipeline_fence* cannot be split.

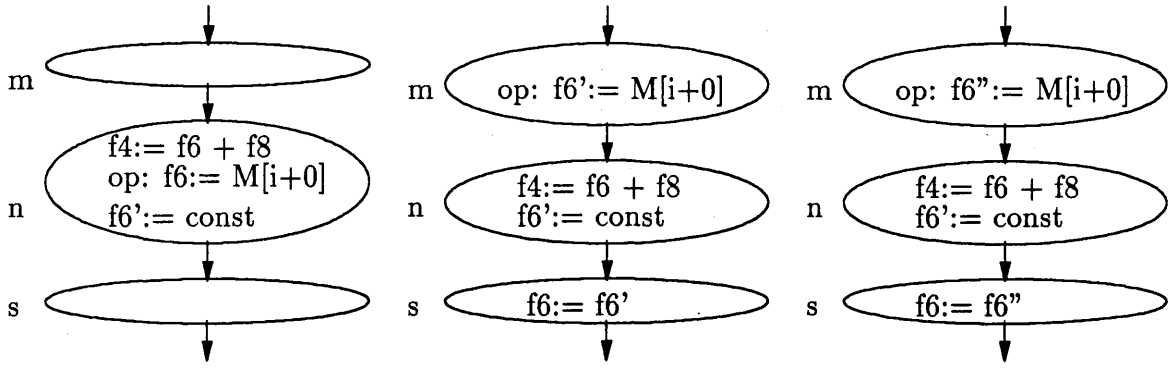
```
procedure Pipeline(loop)
  /* s is a real successor of n if s's DFS number is larger */
  /* than n's DFS and s is part of the loop */
  for each instruction n in loop
    mark n as not part of pipeline_fence;
  let pipeline_fence= {header instruction of the loop};
  let pipeline_newfence= {empty set};
  while pipeline_fence is not empty begin
    Maxcomp-for-pipelining(loop);
    for each instruction n in the pipeline_fence begin
      mark n as part of pipeline_fence;
      for all successors s of n
        if ( (s is a real successor of n) and
             (not filled(s)) and
             (s is not in pipeline_fence) and
```

```
            (s is not in pipeline_newfence) )
          add s to pipeline_newfence;
        /* modify n's DFS number to be larger than all its successors */
        make n a real successor of all its predecessors;
        mark filled(n)= TRUE;
      end
      Maxcomp(loop);
      let pipeline_fence= pipeline_newfence;
      let pipeline_newfence= empty;
    end (while)
end (Pipeline)
```

3.6 PPS-Related Specific Problems

Besides some implementation differences, the extension of PS to PPS transformations required a solution of several delicate problems that are inherent to compaction with multi-cycle operations. Pipelined operations (unlike uni-cycle operations) have an impact on nodes as far away as their latency. This has to be taken into consideration during compaction, otherwise either significant performance-degradation or wrong results may occur. These problems include:

- Disambiguation: while for PS the move up of a load operation to a predecessor node depends only on the predecessor's store operations, in PPS one has to further verify that the predecessor's *pipe_set* does not include an element representing a reference to the possibly same location caused by a



(a) before move-op (b) after move-op (wrong) (c) after move-op (correct)

Figure 3.4: Before and After Move-op

store issued in previous nodes. Therefore we need to disambiguate not only with store operations but also with symbolic derivations induced by store operations kept locally in each node.

- Renaming: while in PS renaming requires extraction of local live information stored in the node from which we move the operation and its immediate successors, in PPS we have to extract this information from all nodes that are in the operation's *latency* distance. Refer to Figure 3.4(a) and suppose we want to move *op* from n to m and assume that the load takes two cycles. Since $f6$ is read by another operation in n it has to be renamed.

  With PS we would pick $f6'$ to be the renamed register ($f6'$ is dead in n) and get the segment shown in Figure 3.4(b). However, this code is obviously wrong since $f6$ in s uses a the value computed by $f6 := f6'$. Therefore, for PPS we have to find a register that is dead *not only* in n but also in its successors. Doing this yields the code shown in Figure 3.4(c).

- Resource Constrained Scheduling: when deferring pipelined operations (see Chapter 5) care must be taken to preserve correctness in order of modification.
- Simulator: when two pipelined operations modify the same register in the register-file in the same cycle—only the one that was issued later determines the value of the register.

In the remainder of this section we present two other problems encountered with PPS during the parallelization process. Although the problems may be regarded as implementation-specific, we believe that they apply in general to any local transformation that handles multi-cycle operations.

3.6.1 Blocking of Loop Pipelining by Operations Outside the Loop

Suppose the following loop is given:

for (i=0; $i < 10$; i++)
  $x[i] = x[i] + 2.0$

We assume in this example that additions and memory loads take 2 cycles, while all other operations take one cycle. This loop transforms into the three-address-code shown in Figure 3.5 (for simplicity, we omitted the exit test).

After $i := i + 4$ moves across the backedge and percolates up, we get the program illustrated in Figure 3.6. However, without any further change the loop pipelining process terminates here because the *induced_field* of $i := i + 4$ (which is

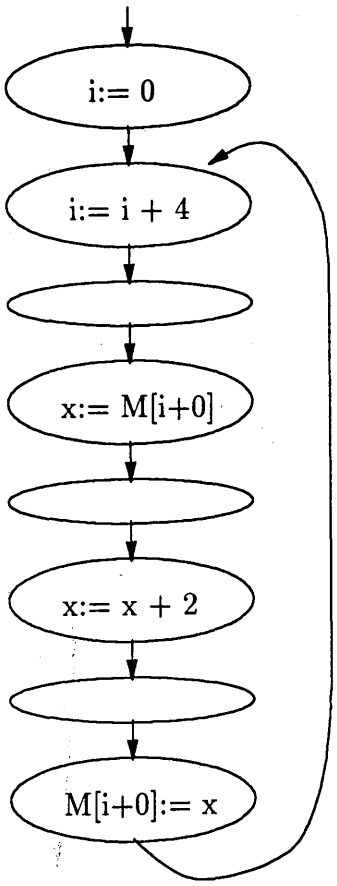


Figure 3.5: Three Address Code for The Loop

outside the loop, now) reaches nodes inside the loop. In particular, $x := M[i+0]$ is blocked from moving into node m and it blocks the motion of other operations as well. In other words, operations *which are not part of the loop* limit the parallelism inside the loop. To alleviate this problem, we trade parallelism outside the loop for parallelism inside (which is much more critical). This is done by the insertion of extra empty nodes (up to the latency of the operation leaving the loop) between the operation that has left the loop (during loop pipelining, in this case $i := i + 4$) and the loop as shown in Figure 3.7. This guarantees that no operation outside the loop will prevent motion inside the loop. The solution proposed here enables compaction of the loop into 2 nodes as shown in Figure 3.8, while without insertion the loop would be compacted into 6 nodes.

3.6.2 Renaming During Pipelining

We illustrate in this section another example of a problem related to compaction with multi-cycle operations. Consider the partially compacted code shown in Figure 3.9. When trying to move the operation $x := x + 2.0$ from n to m we have to rename x (since it is used by the other operation in n. According to move-op, when performing a renaming of variable x into x', we add a copy operation $x := x'$ in all successors of m in a distance equal to this operation's latency. Let's assume that a floating-point addition takes 3 cycles. In this case, it seems that the copy operation should be added to node l as shown in Figure 3.10, but this may lead to a incorrect code since node l is executed before node m. Hence, in order to guarantee a correct renaming, whenever the copy operation is added to a

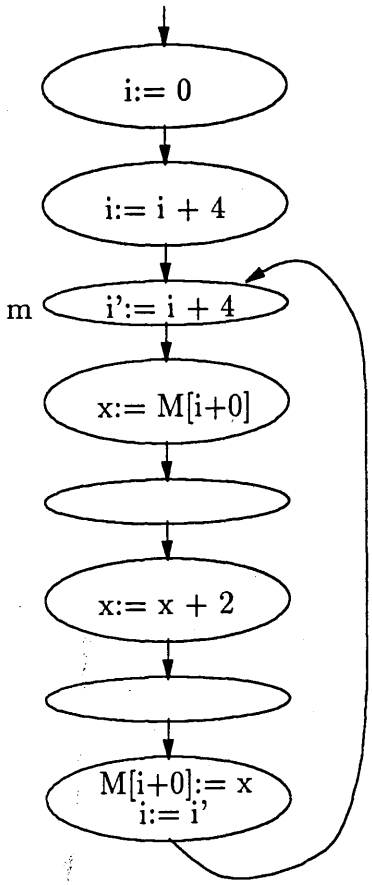


Figure 3.6: Code After $i := i + 4$ Percolated Up

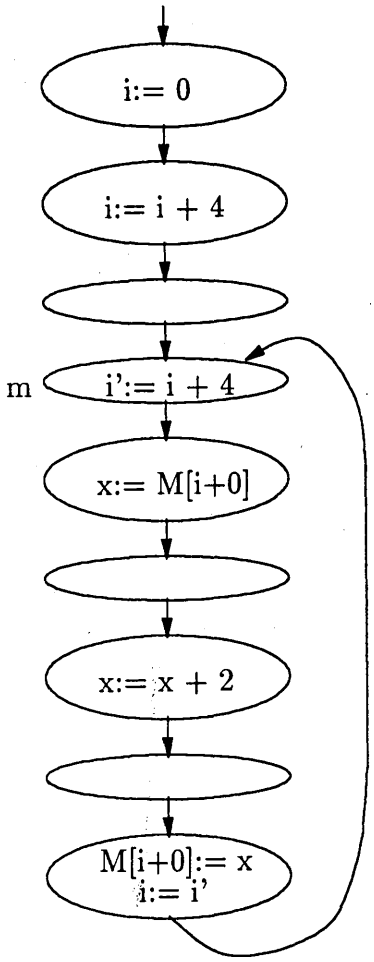


Figure 3.7: Code After Insertion of an Empty Node

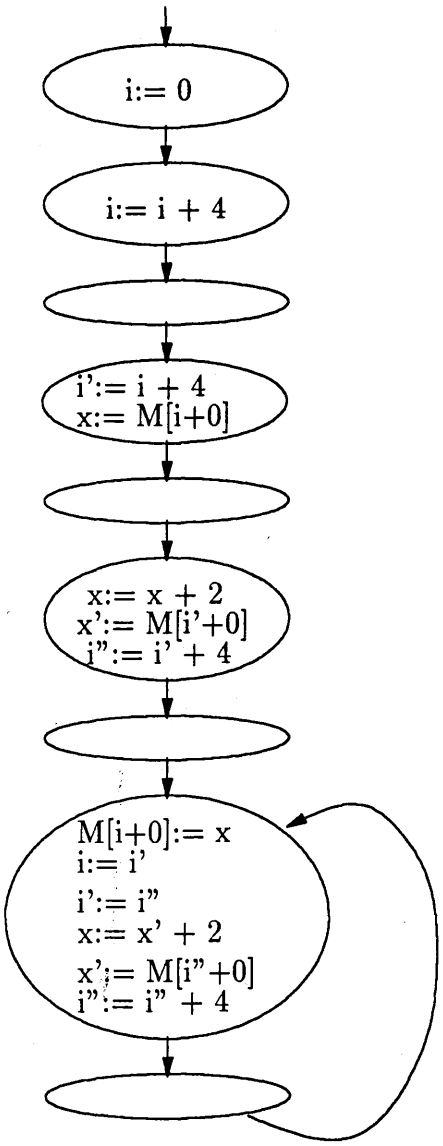


Figure 3.8: Final Code

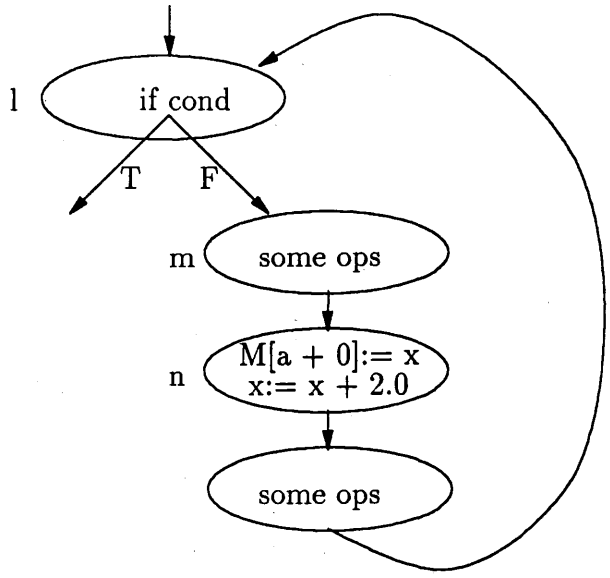


Figure 3.9: Initial Code

node that may be executed before node m, an extra node is added before the loop header to compensate for this copy. This is shown in Figure 3.11.

3.7 Examples

The two examples presented in this section show why our PPS transformations yield better results than PS (with insertion of empty nodes after compaction) and Trace Scheduling (TS) with multi-cycle operations.

3.7.1 Example 1

Assume the following code segment is given:

LABEL L1:

$a := c * 1.0;$ 3 cycles

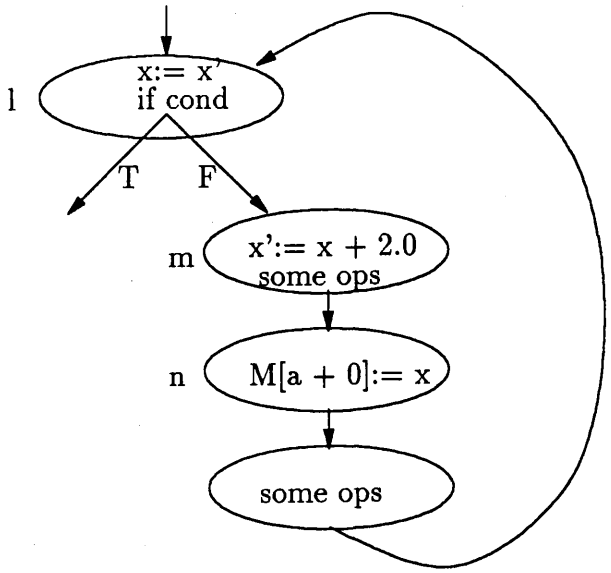


Figure 3.10: Code After Renaming

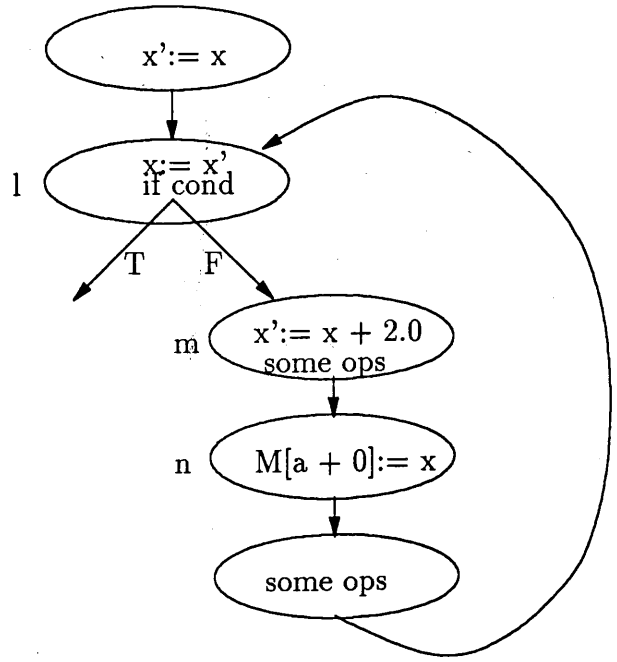


Figure 3.11: Final Code

$c := M[0 + b];$	2 cycles
$b := 2.0;$	1 cycle
$d := a * 3.0;$	3 cycles
if $d <= 0$ GOTO L2;	1 cycle
$e := b * 4.0;$	3 cycles
$f := c * 5.0;$	3 cycles
$g :=$ fix(e);	1 cycle
$M[f + 1] := g;$	1 cycle
GOTO L3	

LABEL L2:

$e := b * 4.0;$	3 cycles
$f := c * 5.0;$	3 cycles
$g := fix(e);$	1 cycle
$M[f + 2] := g;$	1 cycle

LABEL L3:

EXIT

The latencies of the operations are listed next to the serial code. The serial execution time of any of the two paths is 18 cycles. By applying PPS we get the code depicted in Figure 3.12 below. This shows that code is compacted into 7 cycles. On the other hand, TS would pick one of the traces (let's say the TRUE branch) and would compact only this trace. But after compacting this trace all other operations in the FALSE branch can NOT move any more into the compacted trace which results in execution time of 7 cycles on one trace but 12 cycles on

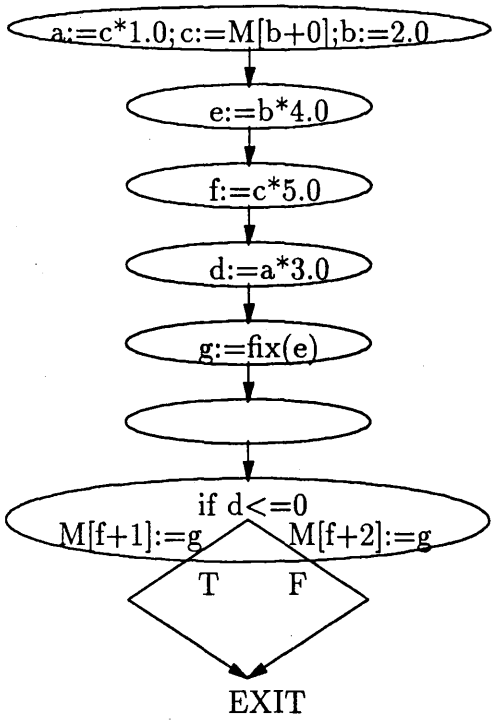


Figure 3.12: Example 1: Compacted Code by PPS

the other. This difference is partly due to PS, but is exacerbated by operations' latencies.[2]

3.7.2 Example 2

The following example shows the power of PPS over PS, assuming each operation takes 1 cycle, and then inserting empty cycles to satisfy the real latencies of operations:

LABEL L1:

$a := c * 1.0;$ 3 cycles

[2]Presumably there are other paths going through L2 implying that normal code hoisting would fail to eliminate the same operations on different branches.

$b := a * 3.0;$	3 cycles
$c := -a;$	1 cycle
$d := a + 2.0;$	2 cycle
if $c <= 0$ GOTO L2;	1 cycle
$g :=$ fix(b);	1 cycle
$M[e + 1] := d;$	1 cycle
GOTO L1	

LABEL L2:

$g :=$ fix(b);	1 cycle
$M[e + 2] := d;$	1 cycle
EXIT	

Again, the latencies of the operations are given in the code. Figure 3.13 shows the compaction derived by PPS—a sequential loop with execution time of 12 cycles is compacted into a parallel loop with execution time of 4 cycles. If we apply PS on the same loop, assuming each operation takes 1 cycle, we get a compacted graph which also has 4 cycles in the loop. Now, if after compacting the code with one-cycle operations, we insert empty nodes to satisfy real latencies and then attempt to locally compact, we get a loop which takes 6 cycles to complete (compared to 4 with PPS) as shown in figure 3.14. The reason for that difference is that while PPS fills nodes as tightly as possible, allowing general motion of multi-cycle operations, PS followed by insertion of empty cycles does not allow the movement of operations out of the basic block (after the initial compaction has been done) and thus prevents better compaction. Of course, we could try to remove

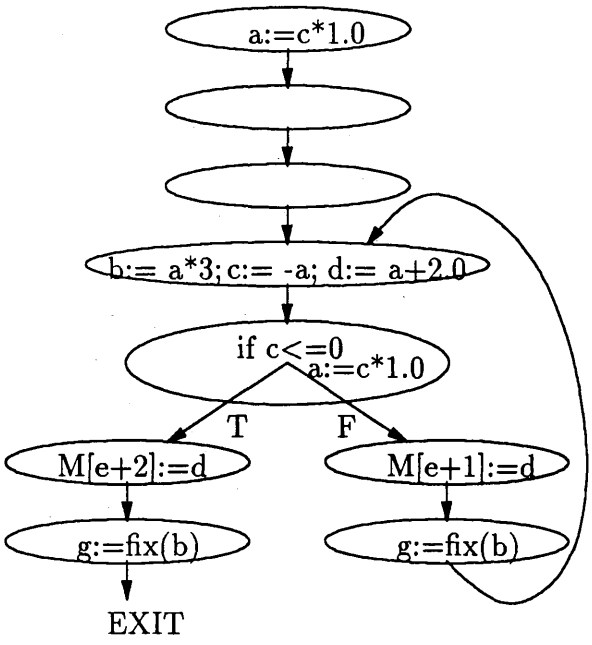


Figure 3.13: Example 2: Compacted Code by PPS

these constraints, but to do this with any generality would require transformations akin to those provided by PPS.

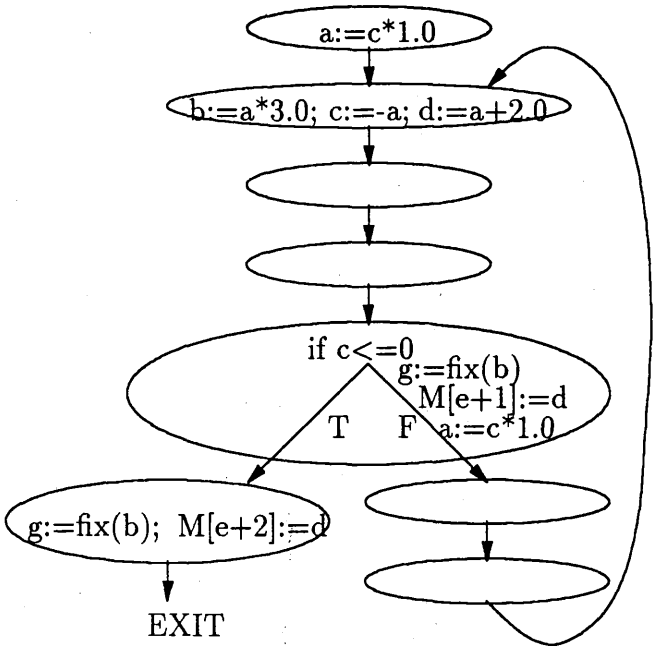


Figure 3.14: Example 2: Compacted Code by PS

Chapter 4

Register allocation, renaming and their impact on our compiler

4.1 When Should Register Allocation Be Done?

Registers have been considered for years as one of the most precious resources in any architecture design. Integration (space) and timing (address decoding) limitations restricted the total number of registers on chip. Accordingly, a good style of programming and/or a good policy of register allocation for a compiler has been one that makes efficient use of registers. This has been accomplished by compiling source programs into machine language using a minimal number of registers, assigning the same registers to variables whose runtime life-spans do not overlap. Such use of registers is beneficial for sequential machines. However, the re-use of registers in high-performance, pipelined, fine-grain parallel architectures may severely decrease performance since it limits the ability of fine-grain parallelizing compilers to compact programs [El86, CKV85].

In parallel architectures, ideally, only true data-dependency (and resources availability) should limit the parallelism. Re-use of registers limits the achievable

parallelism since it introduces a false ordering (dependency) between operations.[1] This could cause a substantial degradation in performance. Consider, for example, the following program segment:

1: $a := b + c;$

2: $d := a * e;$

3: $a := f + g;$

4: $h := a + k;$

Here, $d := a*e$ has to be executed *after* $a := b+c$. That is a strict data-dependency. On the other hand, $a := f + g$ is prevented from being scheduled in parallel with $a := b + c$, *only because* it re-uses register a which is read by $d := a * e$. This dependency is called an *anti-dependency* [CKV85]. But this dependency need not limit the parallelism. Renaming register a to a' would yield the following:

1: $a := b + c;$ $\qquad$ $a' := f + g;$

2: $d := a * e;$ $\qquad$ $h := a' + k;$

which is two cycles shorter. Thus by using an additional register (a') we have increased the parallelism of the program. Another source of false dependency is called *output dependency* which occurs when two operations write to the same output register. In the example above, operations $a := f + g$ and $a := b + c$ modify the same registers. Without renaming, $a := f + g$ cannot move above $a := b + c$. As will be shown in Section 4.2, the effect of renaming becomes even more important during loop pipelining [RaGl82, GrLa86, Eb87, AiNi88c] when

[1] While in uni-cycle operations this artificial dependency prevents an operation from moving up one step, for multi-cycle operations it blocks a move of several cycles (equal to the latency of the operation which causes this anti-dependency).

operations are allowed to move out of their original iteration in order to achieve parallelization both across and within loop iterations.

Thus fine-grain parallelizing compilers face two mutually contradictory constraints: on the one hand registers should be aggressively re-used to avoid spilling and its ensuing inefficiency, while on the other hand anti/output dependencies should be eliminated to increase parallelism. In this context the time when register allocation is done becomes critical to the quality of the code produced.

If register allocation is carried out before compaction, the best strategy to avoid undully limiting parallelism is to use as many registers as necessary (essentially equivalent to transforming the code into quasi static-single-assignment (SSA) form [CFRWZ89]). Unfortunately, the total number of registers needed is usually greater than the number physically available. Hence, some registers have to be re-used. Ideally, we would like to refrain from re-use in places where it actually inhibits the compaction, and allow re-use where it is insignificant. However, if register allocation precedes the compaction phase, this information is unavailable and we may end up with re-use of registers in the wrong places.

A second approach is to perform register allocation after compaction has been completed. This method allows unlimited virtual registers during compaction.[2] In a post compaction pass, the virtual registers are mapped to the actual architecture's registers. Of course, if the number of registers needed exceeds the number

[2]That is, a new virtual register is used for each computed operand and its associated use(s). This may be achieved simply during translation of the input into intermediate code, for non-loop code. However, for maximizing parallelism in loops (in the context of loop pipelining), renaming is still necessary as will be discussed in Section 4.2.

of actual registers—spilling to memory is required. Performing spilling after compaction may severely damage the carefully parallelized ("packed") code, yielding inefficient schedules. In fact, the performance obtained by this approach on its own can be so poor that compilers using it (e.g. Cydrome, Fujitsu) resort to repeated compaction of the new program whenever a spill occurs. While after such iterative compaction the code quality is very good, this repeated computation can be very time costly in cases where many spills occur.

Ideally, we would like to do register allocation during compaction which would make meaningful trade-offs possible. However, this implies dealing with two NP-hard problems (functional units allocation and register allocation) *during* compaction, presenting an exceedingly complex task. Because of the difficulty involved, this approach has been avoided by existing compilers which instead chose one of the simpler approaches above.

In this chapter we propose a simple alternative that allows the flexibility of renaming during compaction while avoiding spilling and the complexity of performing full register allocation during the parallelization process. We start by performing conventional register allocation before compaction. However, during compaction we allow renaming to remove false dependencies *that prevent otherwise desirable/feasible code transformations*—provided a register is available at that point. So, extra registers are used only when actually needed to enhance parallelism, and only if the benefits are not offset by spilling cost.

4.2 Renaming in Fine-Grain Parallelizing Compilers

The price paid for register renaming is not only the increase in number of registers. If renaming is to be done during fine-grain compaction, it needs to be very efficient. In particular, we simply cannot afford to do global searches for uses of the registers being renamed, and potentially complex code transformations to allow semantically correct renaming. Consider again the previous example. Register a in $a := f+g$ has been renamed to $a' := f+g$. Consequently, in all following operations using a, we substituted a'. In general there might be multiple, distant, uses of register a requiring a global search throughout the program. Furthermore, even if all uses are located, renaming may not be immediately feasible, as shown in the example in Section 4.2.2. To avoid these problems during compaction, renaming can be carried out by leaving a copy operation in place of the renamed operation, to reassign the value computed to the original register. Thus, by adding an extra copy to the code, renaming is converted into a local, efficient transformation and the need for search and issues of semantic preservation are eliminated:

1:	$a := b + c;$	$a' := f + g;$
2:	$d := a * e;$	$a := a';$
3:	$h := a + k;$	

However, the copies introduced may create their own set of problems. The effect of introducing extra copies is especially critical when loop pipelining is performed and a considerable number of operations (from multiple iterations of the loop) are exposed to renaming. This may cause significant code-explosion. Furthermore, if these copies are left in the code, their execution results in a waste of functional

units. So, when considering renaming as a parallelization aid in fine-grain compilers, special care must be taken to maintain a good schedule taking into account all the resources available (functional units and registers)—otherwise renaming may not be beneficial.

4.2.1 Local Copy Bypassing

The renaming process, using copy operations in the code, does not (by itself) significantly increase parallelism. Another local substitution is required to tap the full potential of renaming. Refer again to the previous example. Here, $h := a + k$ cannot move up into cycle 2 since it depends on $a := a'$. However, since this data-dependency is generated by a copy operation, we can always substitute a' in $h := a + k$ and rewrite the schedule as:

1: $a := b + c;$ $a' := f + g;$

2: $d := a * e;$ $a := a';$ $h := a' + k;$

which is one cycle shorter. This optimization is *local* (applied during one of the local PS transformations) and therefore simple and efficient. Obviously, if all operations using a move above the copy during PS, $a := a'$ becomes dead and can be removed *locally* from the code.

4.2.2 Renaming During Loop Pipelining

Loop pipelining (see Section 3.5.2) involves unwinding the loop body and compacting the resulting code. This incremental process repeats until data-dependencies force the emergence of a repeating pattern which then becomes the compacted loop

body. In general, the amount of unwinding cannot be precisely predetermined, particularly in the presence of conditional jumps. Thus, static renaming (e.g. on the loop body or on some small and fixed number of unrolled iterations) is not satisfactory. Dynamic renaming is needed.

The overlapping effect is achieved in our technique by allowing operations from the next iteration to percolate up into nodes containing operations from the current iteration. To illustrate the application of renaming during the process of loop pipelining consider the following example:

$for(i = 1; i < 20; i++)$

$A[i] = 4 * (A[i] + 8) + A[i];$

This loop translates into the following three-address-code (all operations listed in the same node number are executed in parallel):[3]

PROG_BEGIN:

node 1: $i := 0;$

LABEL LOOP:

node 2: $a := M[i + 4];$

node 3: $b := a + 8;$

node 4: $c := b * 4;$

node 5: $d := a + c;$

node 6: $i := i + 4;$

node 7: $M[i + 0] := d;$

node 8: $cc0 := i <= 80;$

[3]For simplicity, we do not draw the program graph but use labels to denote nodes.

node 9: IF $cc0$ GOTO LABEL LOOP;

LABEL EXIT:

node 10: RETURN;

PROG_END

Also, in order to simplify the explanation we ignore the exit test operations (nodes 8 and 9). Omitting these two operations and compacting the loop body yields the (partially) compacted code:

PROG_BEGIN:

node 1: $i := 0;$

LABEL LOOP:

node 2: $a := M[i + 4]; \quad i := i + 4;$

node 3: $b := a + 8;$

node 4: $c := b * 4;$

node 5: $d := a + c;$

node 6: $M[i + 0] := d;$ GOTO LOOP

LABEL EXIT:

node 7: RETURN;

PROG_END

The next step is to unfold the next iteration of the loop and try to percolate its operations upwards. For example, after operations from the first node of the second iteration have percolated the code would be:

PROG_BEGIN:

node 1: $i := 0;$

node 2: $a := M[i + 4]; \quad i := i + 4;$

LABEL LOOP:

node 3:	$b := a + 8;$			
node 4:	$c := b * 4;$			
node 5:	$d := a + c;$	$a := M[i + 4];$		
node 6:	$M[i + 0] := d;$	$i := i + 4;$	GOTO LOOP	

LABEL EXIT:

node 7: RETURN;

PROG_END

Without renaming, neither $a := M[i+4]$ nor $i := i+4$ can move any further. Both a and i are used by other operations in their corresponding nodes. This sort of dependency is common in loops since the index (as well as other variables) are often used repeatedly in successive iterations, greatly restricting parallelism. However, with renaming further motion is allowed. This process continues and further iterations are unfolded and percolated until a repeating pattern emerges in the schedule. For our example the final schedule (after pipelining of the next iterations) results in:

PROG_BEGIN:

node 1:	$i := 0;$			
node 2:	$a := M[i + 4];$	$i := i + 4;$		
node 3:	$b := a + 8;$	$a' := M[i + 4];$	$i' := i + 4;$	
node 4:	$c := b * 4;$	$b := a' + 8;$	$a'' := M[i + 4];$	
	$i'' := i' + 4;$			
node 5:	$d := a + c;$	$a := a';$	$a' := a'';$	$c := b * 4;$
	$b := a'' + 8;$	$a'' := M[i'' + 4];$	$i''' := i'' + 4;$	

LABEL LOOP:

node 6:	$M[i+0] := d;$	$i := i';$	$i' := i'';$	$i'' := i''';$
	$d := a + c;$	$a := a';$	$a' := a'';$	$c := b * 4;$
	$b := a'' + 8;$	$a'' := M[i''' + 4];$	$i''' := i''' + 4;$	GOTO LOOP

LABEL EXIT:

node 7: RETURN;

PROG_END

The final schedule shows that the whole loop (6 cycles in sequential form) is compacted into one cycle (speed-up of 6) *given enough resources.*[4] On the other hand, if renaming were not performed the overlap of iterations would have been minimal, yielding a speed-up of only 2. However, renaming resulted in numerous copy operations. If only two functional units were available, no speed-up would be obtained without eliminating these copies. Since 5 extra copies are added to the original 6 operations in the loop, 6 cycles are required to issue the 11 operation in the new loop body.

Conventional copy propagation techniques will fail to remove the copies generated in this example. In node 6, i is both defined and used and two different definitions of i are reaching the node. Consequently, conventional copy propagation (and/or induction variable elimination) techniques will not work (it is not possible to substitute for i).

[4]The speed-up achieved when considering the exit test operation would be 8.

4.3 Previous Work

The effect of storage allocation on parallelism and storage requirements for Fortran programs is discussed in [Ku87]. It is shown that "anti-dependency" inhibits the parallelism exhibited in scientific programs, therefore renaming (or storage reallocation) is needed.

[BEH91] compared three code generation approaches on three RISC processors. The approaches varied from a complete separation between scheduling and register allocation to execution of the two tasks interdependently. They found that completely separating register allocation from code scheduling produced inefficient code while performing pre-scheduling followed by register allocation when the scheduling phase (that was done first) was restricted by the same constraints as the register allocator yields the best cost-performance results.

Cytron and Ferrante [CF87] show that any imperative, Fortran-like language can *always* be transformed into a program whose only constraint on the order of execution is the direct flow of values (i.e. a dataflow graph). In addition, a polynomial-time algorithm to allocate registers (for a scalar processor), requiring no more than the maximum number of simultaneously live registers in the original program is given. It is further shown that the number of extra copies introduced into the program is O(M) where M is the total number of multiple definitions of variables in the original program.

In IBM's VLIW machine [Eb88], whose compiler performs register renaming during the parallelization process, an intermediate approach similar in spirit to the one proposed here is taken. Instead of rewriting the whole program in a

single-assignment-form to allow maximal compaction— loops are unrolled several times (the amount of unrolling is determined empirically) so that each iteration uses a new set of registers. Consequently, renaming is achieved for registers which are defined and used inside the loop body. Although the amount of unrolling is determined heuristically in an extra (preliminary) phase—by unrolling the source program, the running time of the compiler increases considerably due to code explosion. Furthermore, this is wasteful in terms of the number of registers required.

In order to alleviate this problem [Br91] suggests a refinement on this idea. Instead of unrolling the loop at the source level, the loop is unrolled in its pipelined form (renaming using copies is done during loop pipelining). The amount of unrolling can be determined from the length of the longest chain of copy operations *on each path through the loop.* In this way there is no need for multiple iterations of compaction, however, code duplication is still problematic.

4.4 Our Approach

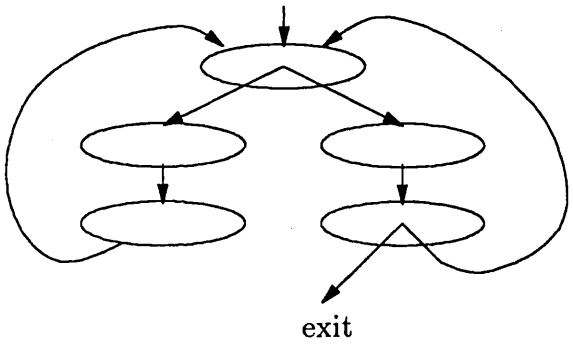
Copies created by renaming during the parallelization process may become dead and be removed locally (see Section 4.2.1) and thus only the remaining copies need to be considered for elimination after compaction. Furthermore, another reason for delaying the application of copy elimination is that sometimes these copies do not affect the resource-constrained schedule (if there are enough functional units)—and thus removal may be unnecessary.

It is convenient to differentiate between two types of renaming candidates during loop pipelining: loop induction variables (IVs), and non-induction variable

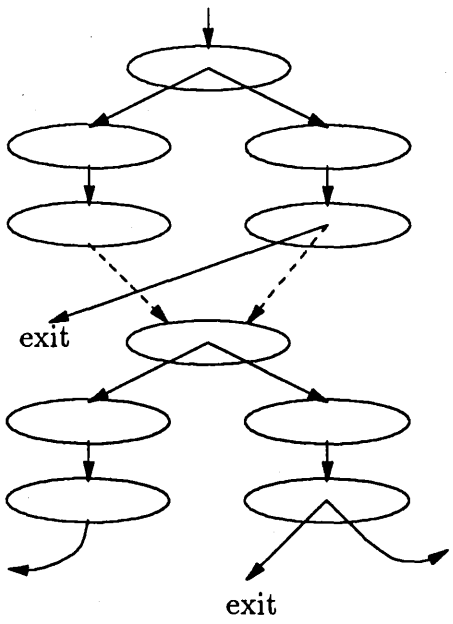
operations which are eliminated by loop unwinding. Furthermore, we were looking for an algorithm that, while removing copies generated by renaming of IVs, will remove redundant IVs as well. The approach described here includes two parts, each corresponding to a different source of renaming. Since the technique for removal of IV-generated copies does not involve code duplication it is applied first.

4.4.1 Definitions

- A *loop* is a set of nodes in the program graph such that there is a path from each node in the loop to another node in the loop. The loop may be irreducible.
- *Loop unwinding* means duplicating the whole loop and directing the backedges of the previous iteration to the appropriate nodes of the unrolled iteration. The two backedges of the loop presented in Figure 4.1 are directed as shown by the dotted lines.
- A variable i is *defined using* (defined by) j iff $i = j + a$ or $i = j$.
- Variables $i_1, \ldots, i_k$ $(k >= 1)$ are *induction variables* (IVs) in loop L iff i_1 is defined exactly once by $i_2, \ldots, i_{k-1}$ is defined exactly once by i_k and i_k is defined exactly once by i_1. $i_1, \ldots, i_k$ are said to be circularly defined.
- A variable i is an *induction variable (IV)* in loop L iff i is defined only once in L by one of the operations $i = i + a, i = j, i = j + a$, where j is an IV and a is a loop invariant or a constant. An operation that assigns a value to an iv is also called a definition of that IV.
- An IV whose definition in L is of the form $i = i + a$ is called a *basic IV*, otherwise it is called a *non-basic IV*.



(a) Original loop



(b) Loop unwound once

Figure 4.1: Loop Unwinding

- Two IVs i and j are in the same *IV family* iff:

  1. there exist IVs $i_1, \ldots, i_k$ $(k >= 0)$ such that i is defined by i_1, i_l is defined by i_{l+1} for for $l = 1, \ldots, k-1$, and i_k is defined by j, or,

  2. there exist IVs $i_1, \ldots, i_k$ $(k >= 0)$ and $j_1, \ldots, j_m$ $(m >= 0)$ and i_0 such that i is defined by i_1, i_l is defined by i_{l+1} for $l = 1, \ldots, k-1$, and i_k is defined by i_0, and j is defined by j_1, j_n is defined by j_{n+1} for $n = 1, \ldots, m-1$, and j_m is defined by i_0.

- An IV is said to be an *effective IV* iff it is used as the memory address register in some memory access operation (load or store). Otherwise it is called an *ineffective IV.*

4.4.2 Elimination of Copies Generated by Induction Variable Renaming

In this section we describe briefly our technique to remove redundant induction variables and copies generated during renaming of IVs. The technique was developed by Haigeng Wang and described in detail in [NPW91].

The technique

The goal of the technique is to remove as many redundant IVs as possible and all copy operations generated by renaming of IVs (during loop pipelining) for each IV family in a given loop L while preserving the semantics of loop L. The conventional IV elimination algorithm described in [ASU86] cannot remove redundant IVs from IV families that have no basic IV since that algorithm assumes the existence of a basic IV for each IV family.

Consider the following example:

PROG_BEGIN:

$i := 5;$

$i' := i + 4;$

$i'' := i' + 4;$

$i''' := i'' + 4;$

LABEL LOOP:

$r := M[i + 0];$

$r := r + 1;$

$M[i''' + 0] := r;$

$i := i'$;

$i' := i''$;

$i'' := i'''$;

$i''' := i'' + 4$;

GOTO LOOP;

PROG_END

In this example there is no basic IV but i, i', i'' and i''' form an IV family. However, this loop can be transformed into compatible one that includes only one IV. Since all these IVs are related, by expressing each IV in terms of an iteration count[5] we can find the exact difference between all IVs and the one selected to remain. This IV derivation problem can be stated as: given a loop L and IV set $\{i_1, \cdots, i_n\}$, where each IV is defined only once in L by $i_k = i_l + a$, $k = 1, \ldots, n$, $l = 1, \ldots, n$, $a \in Z$, and let I be the iteration count of loop L, we want to express all IVs in the form $i = \delta_i I + i_0$, where δ_i is the *progression* of i in each iteration and i_0 is the initial value of i upon entering the loop.

After selecting which IV is kept and after deriving expression for all IVs, we replace all memory expressions in the loop that are using removed IVs with the base IV and adjust the offset accordingly. To preserve correctness on all paths outside the loop, for each removed IVs that are live at the loop exit, we add a compensation operation to reflect the new variables. For example, if i''' was replaced by i, i''' is still live outside the loop and i''' is larger than i by 12. In this case we have to add $i''' := i + 12$ to restore the value of i''' as in the original L.

[5]The iteration count of a loop counts the number of iterations executed. It can be thought of as a canonical IV.

4.4.3 Copy elimination through unwinding

Previous section presented a technique to remove copies generated by renaming of IVs. However, some copies cannot be eliminated without loop unwinding. In this section we present a technique to remove other copies generated during loop pipelining by loop unwinding.

The technique

Copies generated during renaming do *not* produce new values but rather serve to "shift" (already computed) values for future uses. An operation may only define one new value for each iteration. However, the copy operations serve to preserve values produced by this operation over multiple iterations. The reason that loop unwinding is done is to match an operation with its corresponding use, so that copy operations are not needed to keep the produced value live until its subsequent use.

To begin with, we split all nodes in the loop that have multiple reaching definitions of registers involved in copy operations. For example, if register a is involved in one of the copies in the loop, and node n is part of the loop it is split as shown in Figure 4.2.

Following the split we perform a *reaching value analysis* for the loop [ASU86]. During the *reaching value analysis* we build, for each node that is part of the loop, a *mapping table* which represents the correspondence between all operations' symbolic *values* and registers used to store their results. A *value* is a tuple representing

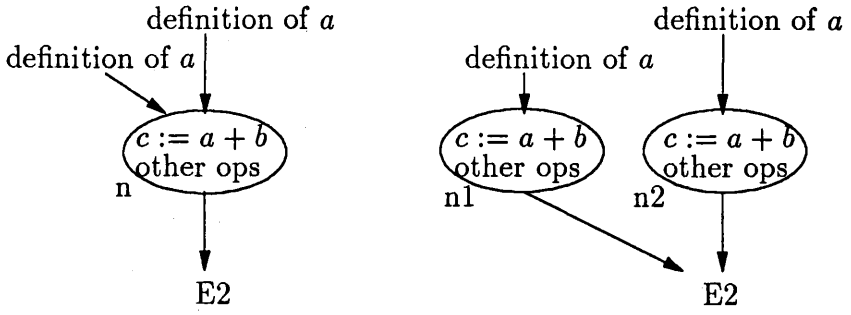


Figure 4.2: A Node Splitting

the *operation number* and the iteration to which the operation belongs.[6] For example, suppose that a representative node includes the following operations:

00: $f6 := f6'$

12: $f6' := M[\$2 + 1000]$

17: $f4 := f6 + f8$

23: $f2 := f8 - f2$

Then, the *mapping table* may include the following entries: {$f6$,12:1}, {$f6'$,12:0}, {$f4$,17:0}, {$f2$,23:0} which means that register $f6$ holds the value produced by operation 12 in the previous iteration, register $f6'$ holds the value produced by operation 12 in the current iteration, register $f4$ holds the value produced by operation 17 in the current iteration and register $f2$ holds the value produced by operation 23 from the current iteration. The next step is to perform a *complete* loop unwinding as shown in Figure 4.1.

All freed registers (of removed copies) are used later when needed for substitution. Then, once the *reaching value analysis* has been applied to this loop iteration, all undirected backedges of that loop iteration are directed to any previous loop headers with a matching *mapping table.* In other words, a backedge from

[6]Naturally, operations are uniquely numbered.

node s may be directed to a node t only if the *mapping table* at the output of s matches the *mapping table* at t. If there still exist backedges with no matching headers, the loop is unwound again.

Since the number of copies is finite, as is the number of different paths in the loop, the number of possible permutations of operations vs. registers (in the *mapping table*) is also finite (we do *not* add new registers during the substitution process). Therefore, the number of unwindings is bound and the algorithm converges. In practice, we found that the number of unwindings required is small compared to the theoretical bound.

Chapter 5

Resource Constrained Scheduling

5.1 Resource Constrained Scheduling and Local Transformations

Resource constrained scheduling (RCS) is the process of mapping an unlimited-resources schedule onto the given architecture, taking into account all its peculiarities and constraints. Since The RCS problem is known to be NP-hard, in practice it may be solved only by a heuristic algorithm. This raises questions like: "What is a good RC schedule?" or "When to apply the heuristics?". Basically, there are two conceptual alternatives to perform RCS: either consider the constraints during the parallelizing process, not allowing the motion of operations into nodes that are already filled, or perform an unlimited-resources parallelizing process and then apply the constraints in another pass.

The main problem in integrating RCS with any *local* transformations (like PPS), is their different nature. While PPS is a set of *local* transformations, based on local information available at each node, RCS would yield, potentially, better results when using *global* information. In other words, when scheduling for a specific node n in the graph, one would like to move into this node the best

choice among *all* operations in the graph rather than among operations which are closer to n. Since operations are moved from one node to another by higher-level transformations, there is no guarantee that an operation *op1* which is closer to n will not block another, farther, operation *op2* from moving into n, while the better choice would be to move *op2* first. Hence, a *global* RCS technique should be applied during the RCS process.

That is exactly what Ebcioglu and Nicolau propose in [EbNi89]. Their technique integrates resource limitations into the Percolation Scheduling transformational model without sacrificing the generality and completeness of PS. The technique relies on information called *unifiable_ops* that is kept locally for each node in the graph. The unifiable_ops of a node is the set of all operations in the program that can *potentially* be scheduled in it. This information is computed for the serial (uncompacted) program and updated incrementally during the parallelizing process. Hence, the RCS problem can be stated as: "among all operations in the unifiable_ops find the best k choices and move them up into this node". Since the information in unifiable_ops is incrementally updated during compaction the "best" global information is available for making the choices at every step.

Another approach, which is used by our compiler, is to extract first the maximal parallelism, ignoring resource constraints and then, in another pass, map this schedule with the given resources by "splitting" nodes with more operations than allowed to meet the constraints.

The main argument against the latter technique is that it may have to undo some of the code motions performed in the unrestricted phase, thus increasing the compaction time. In addition, since some of the transformations do not have a

unique inverse—undoing them may produce worse code than the original program. In order to have a measure for these claims, we have implemented at the early stages of our project both techniques (the global technique was implemented in CMU [Br91]). It turned out that the compaction time with the former technique was *much* higher than with the latter. That was mainly due to the time spent in computing and modifying the unifiable_ops information.

On the other hand, we show in [PLNG90] that starting from an unlimited-resources schedule, which is in most cases the optimal schedule, is one of the most important factors in scheduling because it offers a good lower bound for the total execution time. Without this bound it is sometimes very hard to estimate what would be the best performance of an application, making the heuristic algorithm harder (when to stop?, what is a "good" schedule?).

Another important argument in favor of our technique is that it actually enables exposure of all optimizations since operations are compacted first without taking resource constraints into account. On the other hand, when resource constraints are inherent to the parallelization process, the order in which operations are selected for scheduling implies whether an optimization is invoked or not. By (heuristically) preferring some operations, we may loose some important optimizations. For example, in Section 7.3 we present the load-after-store optimization. In this optimization a load operation is eliminated when the two nodes involved in the move-op transformation have a specific pattern (the node to which the operation moves contains a store into a memory location and the node from which the operation moves includes a load from the *same* location). The unlimited-resources schedule will always expose this pattern when it exists. However, if RC heuristics are integrated in the scheduler only a specific selection of operations during

resource constraints scheduling will yield the same effect. Furthermore, once this load is eliminated, there is obviously no need to schedule it later with resource constraints.

5.2 Our Technique

Using a heuristic to solve the NP-hard RCS problem implies a priority function to select the "most important" operations to be scheduled in a node. The priority function we have selected is a weighted priority function. It includes two measures that characterize the operation. First is the *mobility* of the operation. The *mobility*[1] of an operation is the "freedom" one has in scheduling this operation without stretching the whole schedule [PaGa86]. When an operation can be deferred by one cycle and does not cause a subsequent increase in the total execution time its *mobility* is one. In other words, operations on critical paths have a zero *mobility* (delaying these operation lengthens the schedule) while operations not on critical paths may have positive *mobility*. The mobility itself is prioritized: critical paths in inner-loops are more important then critical paths in non-inner-loops or non loop sections of the program. Naturally, we tend to defer operations with highest *mobility*. The other measure characterizing the priority of an operation is its *precedence_number* in the original (non-compacted) program. The *precedence_number* indicates its original "distance" from the program header. Basically, our algorithm is a list-scheduling-based RCS with *mobility* and *precedence_number* as a heuristic.

[1]This is the term that is used in High Level Synthesis.

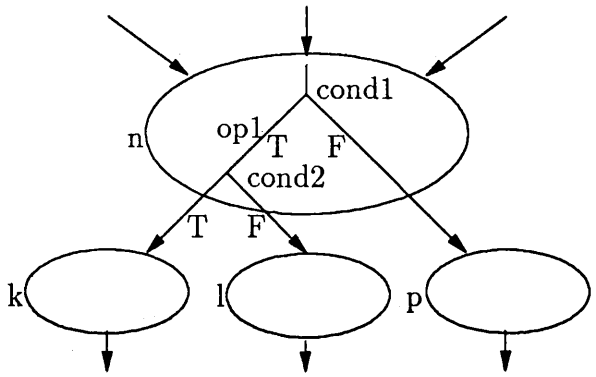


Figure 5.1: Code Segment Before Deferring of op1

The resource constraints include the maximal number of functional units of each kind (adders, multipliers etc.) that are available per node and the total number of operations (if less than the sum of partial constraints) executable per node. The RCS process is carried out by traversing all nodes in the (unlimited-resources) compacted program. For each node we compare the number of operations scheduled with the allowable number and when some operations have to be deferred—they are selected in an order according to their priority. When deferring operations care must be taken not to violate program correctness. An operation selected for deferring has to be deferred on *all* paths it belongs to. In Figure 5.1, suppose that op1 is selected for deferring. In this case two new nodes are created and inserted between nodes n and k and l. Op1 is deferred from n to the newly created nodes k' and l'. The resulting segment is shown in Figure 5.2.

When a conditional jump has to be deferred, the lowest one in the operation tree is always the one selected for deferring. After the conditional jump has been deferred, the code segment of Figure 5.1 will be transformed into the one shown in Figure 5.3.

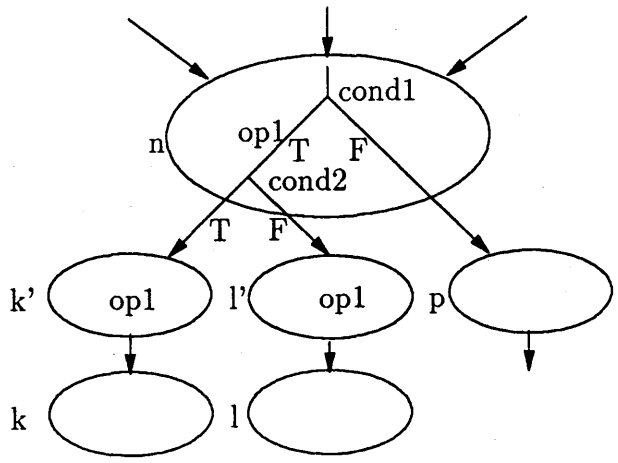


Figure 5.2: Code Segment After Deferring of op1

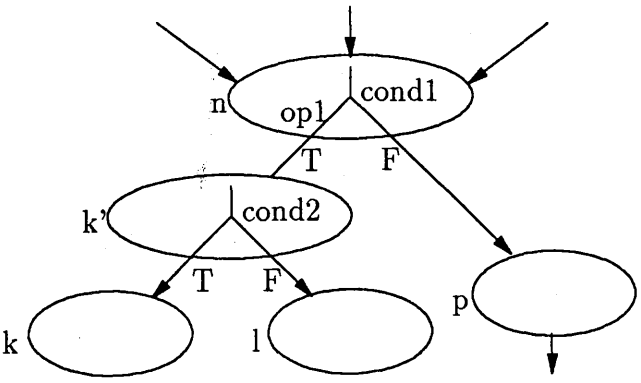


Figure 5.3: Code Segment After Deferring of Conditional

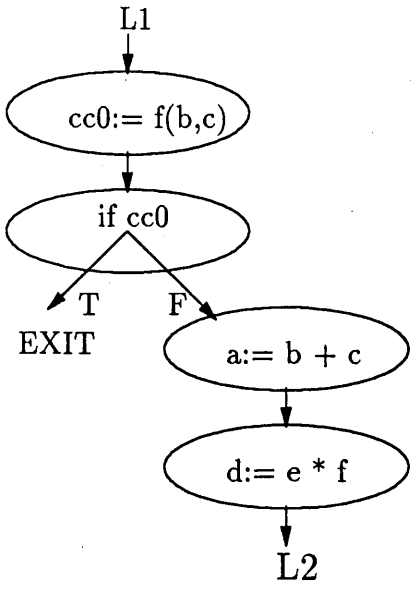


Figure 5.4: Code Segment Before Compaction

The second priority measure of an operation, its *precedence_number*, is an important one since omitting it may yield a compacted program whose execution time is worse than the original (sequential) program. We clarify this by the example illustrated in Figure 5.4. After (unlimited-resources) compaction this segment transformed to the one shown in Figure 5.5. Now assume that two operations are executable per node and assume that priorities of all three operations in node n are such that $cc0 := f(b, c)$ was selected to be deferred. In this case we get the segment illustrated in Figure 5.6.

This schedule is worse than the original (uncompacted) schedule despite that two operations are executable per node!. In the original schedule it took two steps to execute the path from L1 to EXIT while in the compacted program it takes 3 steps. This happens because PPS allows *speculative execution* of operations (i.e. operations are scheduled to be executed before the condition to execute them has been resolved). In order to prevent such cases, each operation in the

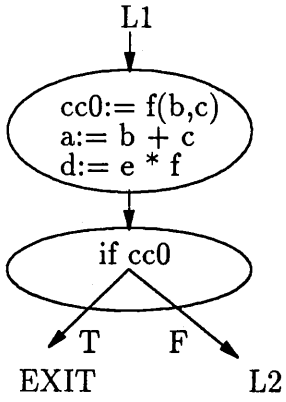


Figure 5.5: Code Segment After Compaction

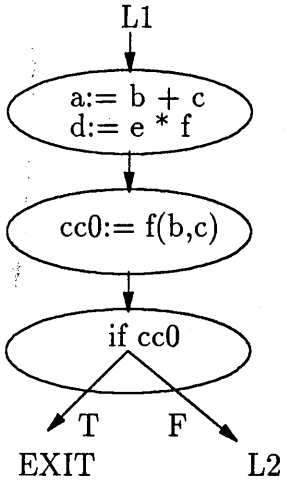


Figure 5.6: Code Segment After RCS

original program graph is assigned a *precedence_number*. All operations below a conditional jump are assigned a *precedence_number* which is greater by one than the *precedence_number* of operations above this conditional. Thus, the rule for deferring can be stated as: "among all operations in the node choose to defer the ones with the greatest *precedence_number*. Among all operations having the same *precedence_number* choose the ones with the highest *mobility*.

5.3 Resource Constrained Scheduling Algorithm

The RCS process is detailed hierarchically in this section.

```
procedure RCS()
  /* real successor= successor not through backedge */
  assign mobility and precedence_number to all operations in the program;
  for each instruction n in the program
    RC_filled(n)= FALSE;
  let RC_fence= {header instruction of the program};
  let RC_newfence= {empty set};
  while RC_fence is not empty begin
    for each instruction n in the fence
      RC_schedule_inst(n);
    RC_maxcomp(program);
    for each instruction n in the fence begin
      for all successors s of n
        if ( (s is a real successor of n) and
```

```
            (not RC_filled(s)) and
            (s is not in RC_fence) and
            (s is not in RC_newfence) )
          add s to RC_newfence;
      mark RC_filled(n)= TRUE;
    end
    let RC_fence= RC_newfence;
    let RC_newfence= {empty set};
  end (while)
end (RCS)
```

RCS(), which is the highest-level RCS procedure, uses two other procedures: RC_schedule_inst() and RC_maxcomp(). The former schedules each node in the program such that it meets the resource constraints while RC_maxcomp() tries to maximally compact the program after all nodes in RC_fence have been scheduled. The procedure RC_maxcomp() is different from the Maxcomp() procedure described in 3.5.1 in that nodes that have already been scheduled with RC (RC_filled(n)=TRUE) are *not* involved in the parallelizing process anymore (they are presumably full).

```
procedure RC_schedule_inst(n: instruction)
  while RC violation begin
    op_to_defer= select_candidate(n);
    create_new_nodes_in_successors_of(op_to_defer);
    defer_op(op_to_defer);
    update number of operations in n;
```

end (while)

end (RC_schedule_inst)

Select_candidate(n) chooses the best operation to defer according to the criteria explained above. Create_new_nodes_in_successors_of() creates new nodes at *all* successors of op_to_defer as explained earlier and defer_op() performs the actual deferring of the operation.

Chapter 6

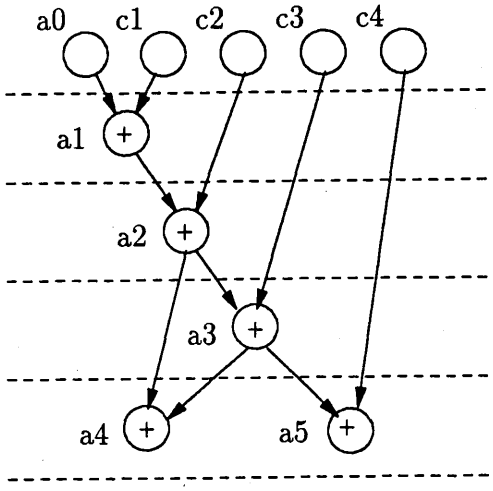
Tree Height Reduction

PPS transformations are parallelizing, semantic-preserving transformations that convert a serial program to a parallel one while maintaining the original data-dependencies between operations. Consequently, the compaction is limited only by data-dependencies and resources' availability. In this context two interesting questions are:

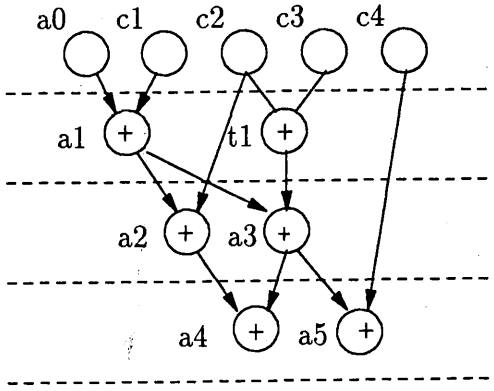
- Is there any way to further compact the code (while preserving correctness) at the expense of additional computation?
- With the given resource constraints, can the (PPS-compacted) code be further parallelized?

The questions are especially important when very high performance is needed, even at the expense of more hardware, and when one wishes to maximize the utilization of a given design (i.e. for given resource constraints to achieve shorter schedules). Suppose the schedule illustrated in Figure 6.1(a) is given and the resource constraints are such that the use of only 2 adders is allowed.

Without any semantic transformation four steps are required to execute this segment. In three out of the four cycles resources are not fully used. However, this segment may be rescheduled as shown in Figure 6.1(b) which is one cycle



(a) schedule before THR



(b) schedule after THR

Figure 6.1: Better Utilization of Resources

shorter *under the same resource constraints.* This is achieved through additional computation (two operations in this case). This schedule length reduction is called Tree Height Reduction (THR).

THR is a well known technique [KuMuCh72, Ku78], which was introduced many years ago, for reducing the height of an expression tree. The question is: "If so, why has THR not been widely used by other scheduling systems?". The answer to this question is two fold. First, THR is effective only when there is a long-enough chain of operations that are data-dependent. Unfortunately, the original THR was only applicable to operations within basic blocks. Since the average number of operations within a basic block is 4-5 [TjFl70] (and not all of these will always form a chain) potential speed-up in basic blocks is limited. Second, the traditional implementation of THR required global information about the whole expression to be reduced. This prevented integration of THR into any of the existing local and incremental parallelizing transformations (List Scheduling, Trace Scheduling, PS etc.).

Obviously, if one considers only basic blocks, the chain of dependencies is not long enough to expose the strength of THR, but by looking at the global RTL-level parallelism we are able to go past conditional jumps and have a longer chain of operations which improves the potential parallelism. This is particularly noticeable when combined with loop pipelining, when operations from different iterations make this chain even longer.

By designing a set of incremental transformations for THR that integrate into our system of local transformations we overcome the previous problems associated with THR. In this context incremental and local THR has some important

advantages: it is not as ad hoc as the global one, it has more general application, it is easier to implement and it interfaces very well with other *local* parallelizing transformations and enables better control of resources.

Furthermore, the local and incremental aspect of our technique will exploit potential opportunities wherever they are interspersed in the program; so even in a program that is not as regular as in the above example, we may still benefit from local opportunities interspersed throughout the program.

The application of our THR is controlled by the resources available such that it only "fills" *unused* resources. Thus, the traditional concern that THR may degrade performance by generating redundant code that cannot translate into speed-up (due to limited resources) is completely eliminated by our incremental approach.

Besides the fact that THR is able to compact programs when other techniques fail due to data-dependency between operations, it has another interesting property: while known compaction techniques yield a constant factor of speed-up (even with unlimited resources), THR has a potential speed-up of $O(n/\log n)$ [Ku78].

In performing THR care must be taken not to violate the *numerical stability* of the code. This problem may occur when the code includes two operations like:

$$a := b - c$$
$$d := a * e$$

This may be transformed during THR into $d := b * e - c * e$. If the values of b or c are too large but the value of their difference is still small, the order in which

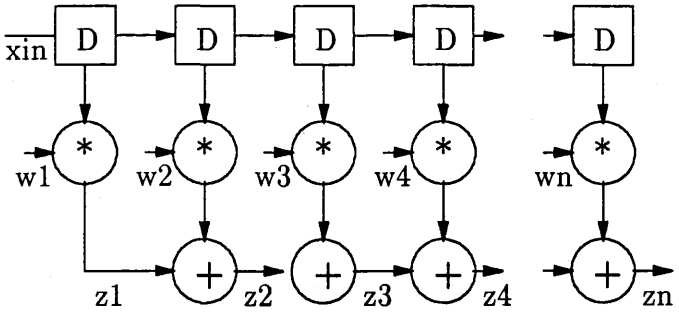


Figure 6.2: Sample Digital Filter

the expression is evaluated may be significant. However, the algorithm presented in this chapter may be used selectively and in cases where numerical stability is violated the algorithm would disallow it. We believe that in most cases THR can be applied without detrimental side effects.

Although our implementation of THR can handle pipelined operations, for simplicity, we assume throughout the algorithm description that all operations are uni-cycle. The extension of the incremental THR algorithm to pipelined operations is straightforward.

6.1 THR Applications

Although [Ku78] claims that applying THR to multi-operation machines "would be quite disappointing", we found a wide range of applications for THR. Digital filters are potential candidates for THR since they have a chain of additions (resulting from the different delay elements), as shown in Figure 6.2, and since they are usually implementing loops that may be pipelined, so that many more operations may be exposed to THR.

Another common candidate for THR is the computation of array elements. This includes sum of vector elements, dot product and simple recurrences where one can find a chain of dependent operations. Although these chains of dependencies are simple–they prevent any reduction to parallel form without an *algorithmic* change. Given enough resources, THR will reduce the computation time for all these examples from O(n) to O($\log n$).

6.2 Algorithm Description

6.2.1 Background

The idea behind tree height reduction is to try to compact a program at the expense of additional computation (that results, sometimes, in an increase in design area). In a design where execution of more than one operation per cycle is possible, it is natural to utilize all available (unused) resources in order to increase performance. Hence, THR is adding more operations to the program that can be executed by these "free" resources such that the total execution time of the program is reduced.

THR takes advantage of the associativity and distributivity properties of arithmetic operations. For simplicity, we only present the algorithm with addition, multiplication and subtraction. It can be extended easily to programs with divisions and logical (AND, OR) operations as well.

6.2.2 Definitions

In this section we define some notations used later in the algorithm:

- *Program:*

  The program is the one defined in Section 2.4.

- *Operation:*

  Each operation has a type (op_type) and variables which are called *uses* variables (for operands read) and a *def* variable (for operands written). For the operation: $a := b * c$ the *def* is a and the *uses* are b and c. The op_type is multiplication.

- *Current_op:*

  The operation currently being examined (or the operation we are trying to schedule earlier than its current cycle).

- *Selected_path:*

  The path selected for THR.

- *Later_definer and earlier_definer:*

  The operations defining the *uses* of *current_op*. In $a := b * c$, b and c are called the "definers" of a. Suppose the following program is given:

  cycle (k) : $b := d + e$; $h := d * g$;

  cycle (k+1): $c := h - e$;

  cycle (k+2): $a := b * c$;

  We will call the operation ($c := h - e$) the *later_definer* of operation ($a := b * c$) while the operation ($b := d + e$) is called the *earlier_definer* of the *current_op*.

- *Available variable:*

  A variable is said to be available in cycle (k) if it is defined at cycle (k-1) or

earlier. In the example above c is available in cycle (k+2) while b is in cycle (k+1).

- *Percolate Operations:*

  Compact the program using PPS transformations (see Chapter 3).

6.2.3 Algorithm in Detail

Our local and incremental THR algorithm can be invoked in one of two ways. If during the incremental process, in which PS is trying to move an operation up from a node to its predecessor, a dependency is encountered then THR is invoked to incrementally change the code to allow the motion. Alternatively, incremental THR can be invoked in the final phase of the compaction process, after all data-independent operations have moved up as high as possible and there are still unused resources to "fill". In either of the former cases only those nodes in the graph that are not full, may be considered for incremental THR.

When activated for a particular operation the algorithm checks whether it could be scheduled earlier than its current cycle by introduction of a new operation that can be performed early enough to be used to eliminate the dependency on the *later_definer* and advance the schedule of the *current_op*. Since each node may have more than one predecessor node (several incoming paths), incremental THR should be performed with respect to *selected_paths* in the program. On different

paths, each operation may have different *later_definer* and different *earlier_definer*, thus each path should be considered separately.[1]

Although it is usually sufficient to check only adjacent nodes in the program, thus preserving locality, it turns out that in order to achieve *optimality* (in the presence of sufficient resources), the whole chain of operations on the path has to be checked. This process is not needed when the resources are limited. The following algorithmic description refers to the *optimal* reduction on each path.

The algorithm analyzes two cases differently. The first is when the associativity property of operations is used, which happens whenever the *current_op* and its *later_definer* constitute one of the following pairs: ADD/ADD, ADD/SUB, SUB/ADD, SUB/SUB and MUL/MUL. The other case is when *current_op* is MUL and its *later_definer* is either ADD or SUB where the distributivity property is used. In any of these cases we try to hoist *current_op* from its current node (cycle) to a predecessor node, which eventually may reduce the length of the program.

Necessary and sufficient conditions for an operation to be hoisted:

1. One of its *definers* must be available at least two cycles earlier than itself on the path selected.
2. *current_op*'s *later_definer* has a *definer* which is available at least two cycles earlier than *current_op*'s cycle on that path.

[1]This *does not* mean that the algorithm needs to consider all paths; we may simply concentrate on only one or several important paths. Due to the incremental nature of the transformations we can stop at any point in the process and still have correct code.

3. If *current_op* is ADD or SUB then *later_definer* has to be either ADD or SUB. (These legal combinations constitute a "legal" chain). If, on the other hand, *current_op* is MUL, the *later_definer* might be either MUL, ADD or SUB.
4. Both *current_op* and its definers have two *uses* variables.
5. All relevant nodes on the path (into which new operations are added) have free resources.

Procedures

The procedures are described in this section in a top-down manner.

```
procedure THR_Analysis(selected_path)
  for each node n in the selected_path begin
    reset back_track flag;
    for each operations in n begin
      if current_op meets the conditions begin
        switch
          case associativity:
            Associativity_Analysis(current_op);
          case distributivity:
            Distributivity_Analysis(current_op);
        end
        percolate operations on the path;
      end
    end
```

```
    if back_track is set
      recheck predecessor node;
    else
      check next node;
  end
end (THR_Analysis)
```

The back_track flag causes backtracking to the previous node. This node has to be rechecked due to the possible creation of "new" legal chains of operations following the "pushing" of multiplications upward. These chains may create further THR opportunities. See example 1.

```
procedure Associativity_Analysis(current_op)
  if current_op is SUB and later_definer is its subtrahend
    set sign_flag;
  earliest_op= Find_Highest_Avail_Op(current_op);
  if succeeded to find such an operation begin
    /* add new operations recursively to path */
    Climb_Up(modified op_type, earliest_op's earlier_definer,
      current_op's later_definer);
    remove current_op from list;
  end
end (Associativity_Analysis)
```

Sign_flag controls the correct addition of SUB operations into the program. We need to flip the operands whenever we find a SUB and its *later_definer* is its subtrahend.

procedure *Distributivity_Analysis*(*current_op*)

/* the procedure is called when *current_op* is of the form $d := a * (b + c)$.
In this case we do not try to hoist d—but rather use the distributivity
property and convert d into $d := a * b + a * c$. */

/* add first additive $(a * b)$ */

add new operation with (MUL type, *later_definer*'s *earlier_definer*,
current_op's earlier_definer) into *later_definer*'s node;

/* add second additive $(a * c)$ */

add new operation with (MUL type, *later_definer*'s *later_definer*,
current_op's earlier_definer) into *later_definer*'s node;

/* add modified *current_op* (d) */

add new operation with (*later_definer*'s type, first_additive, second_additive)
into *current_op*'s node;

remove *current_op* from list;

set *back_track* flag;

end (*Distributivity_Analysis*)

procedure *Find_Highest_Avail_Op*(*selected_path*)

/* the procedure is searching along the *selected_path* for the earliest
operation which meets the conditions explained in section 4.3.1. For
correctness preservation, each time a SUB is found and its *later_definer*
is the subtrahend—the operation's sign is flipped. */

end (*Find_Highest_Avail_Op*)

procedure *Climb_Up*(type, first_op, second_op)

/* the procedure adds new operations into *selected_path* after the earliest

operation that meets the conditions has been found by previous procedure.

Calls itself recursively until it reaches the *later_definer* of *current_op*. The

addition of the modified *current_op* is done by a higher level procedure. */

add new operation with (type, first_op, second_op);

if (didn't reach *current_op*'s *later_definer*)

Climb_Up(first_op's type, first_op's *later_definer*, the newly added operation);

end (*Climb_Up*)

6.3 Examples

We present in this section three examples on which THR is applied. The first is to clarify the algorithm, the second to show how incremental THR works across basic blocks of a program and the last to show how THR, combined with loop pipelining, may exposes more operations for compaction, thus yielding better parallelization.

6.3.1 Example 1

Suppose a code segment, as illustrated in Figure 6.3, is given and assume that a0 and all c's are available at the first cycle.

Step 1:

Let us begin, for example, with the third operation ($a3 := a2 - c3$). Its *earlier_definer* is *not* defined in the previous instruction, so execute *Associativity_Analysis()*. The op_type is SUB—so set sign_flag and call *Find_Highest_Avail_Op()*. But, since

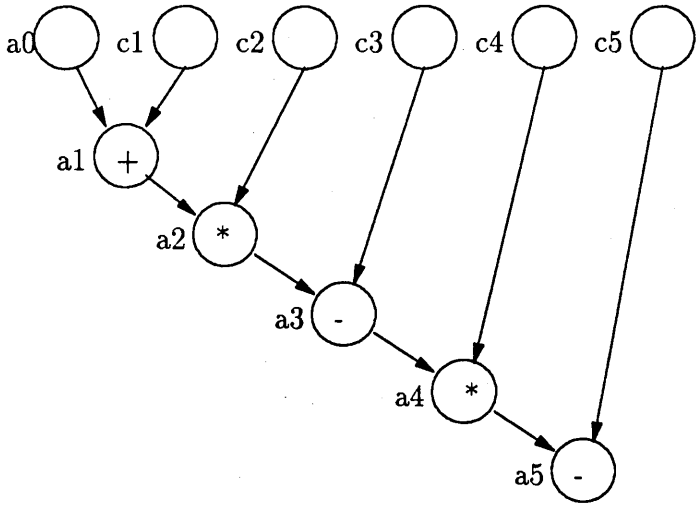


Figure 6.3: Original Code of Example 1.

current_op violates condition 3 quit the procedure.

Step 2:

Current_op is $(a4 := a3 * c4)$. It's type is MUL and its *later_definer* is SUB so *Distributivity_Analysis()* is called. Three operations are added into the tree:

1. A MUL operation whose *uses* are *later_definer's earlier_definer* (c3) and *current_op*'s *earlier_definer* (c4). This operation gets a new *def* (t1) and is inserted into *later_definer*'s cycle.
2. Another MUL whose *uses* are (a2) and (c4) and its *def* is t2. It is inserted into *later_definer*'s cycle.
3. The reconstruction of *current_op* with the type of *later_definer* (SUB) and with *uses* which are the operations just added. Its *def* is *current_op's def.*

The *back_track* flag is set. After this step **and percolation**, we get the code shown in Figure 6.4:

Step 3:

Since *back_track* flag is set cycle 3 is rechecked. $(a3 := a2 - c3)$ cannot be hoisted

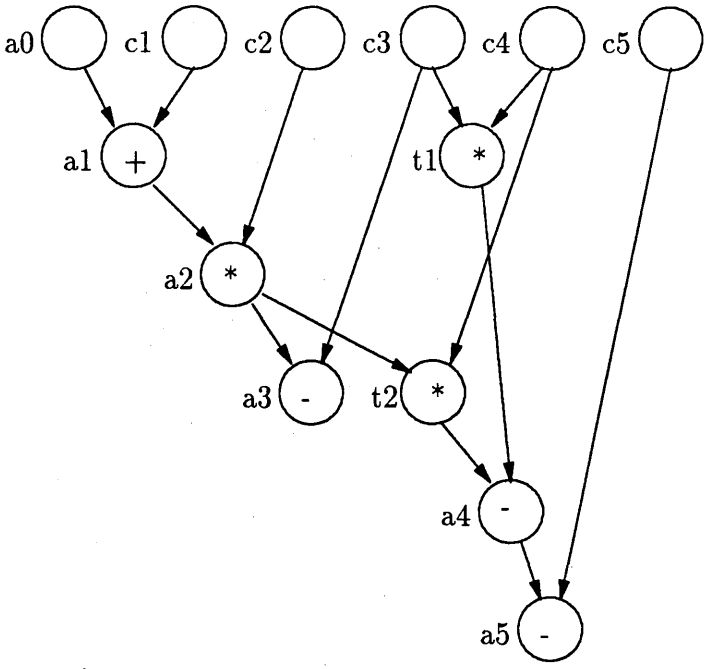


Figure 6.4: Example 1 After Step 2.

for the same reason mentioned in step 1 above—so check the next operation in this cycle ($t2 := a2 * c4$). The operation is MUL and its *later_definer* (a2) is MUL, hence by *Find_Highest_Avail_Op()* the highest op which is ($a2 := a1 * c2$) is found. Now, using *Climb_Up()*, operations are added as follows: a MUL (t3) operation whose *uses* are highest op's *earlier_definer* (c2) and *current_op*'s *earlier_definer* (c4) is added. Then another MUL, whose *uses* are *later_definer*'s *later_definer* (a1) and t3 is added. After this step and percolation we get:

$a1 := a0 + c1;$ $t1 := c3 * c4;$ $t3 := c2 * c4;$

$a2 := a1 * c2;$ $t2 := a1 * t3;$

$a3 := a2 - c3;$ $a4 := t2 - t1;$

$a5 := a4 - c5;$

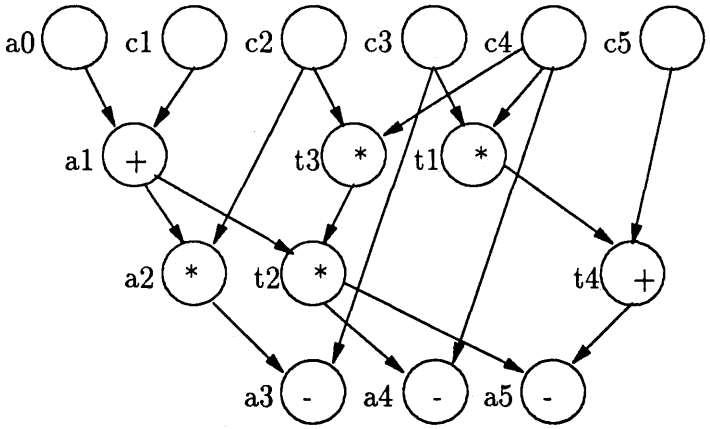


Figure 6.5: Compacted Program for Example 1.

Step 4:

Consider ($a5 := a4 - c5$) as the *current_op*. Since it is SUB we use *Associativity_Analysis()* and get the final compacted code as illustrated in Figure 6.5:

Note that if resources didn't allow one of the steps (e.g. if only two subtracters were available per cycle) the incremental THR would have stopped without allowing $a5$ to move up, but still would produce a one cycle gain.

6.3.2 Example 2

This example shows how incremental THR works across basic blocks. Consider the program shown in Figure 6.6.

This program segment has 3 basic blocks separated by conditional jumps. "Conventional" THR (within basic block boundaries) on this program fails since there are not enough operations in each of these 3 chains to produce any speed-up. But applying our incremental THR beyond the conditionals yields a significant compaction (from 8 cycles to 3) as shown in Figure 6.7.

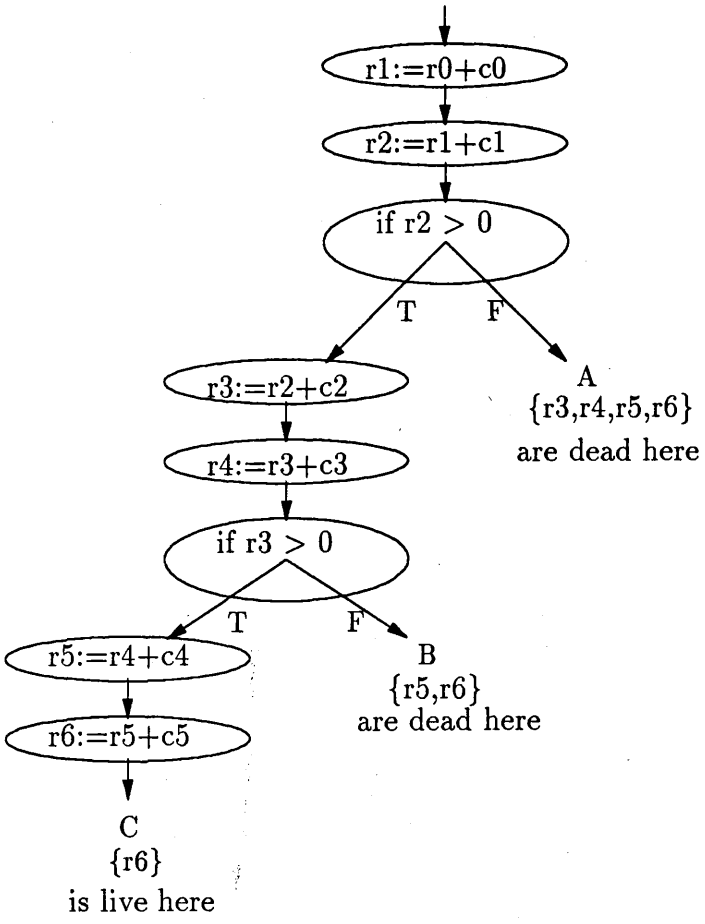


Figure 6.6: Original Program for Example 2.

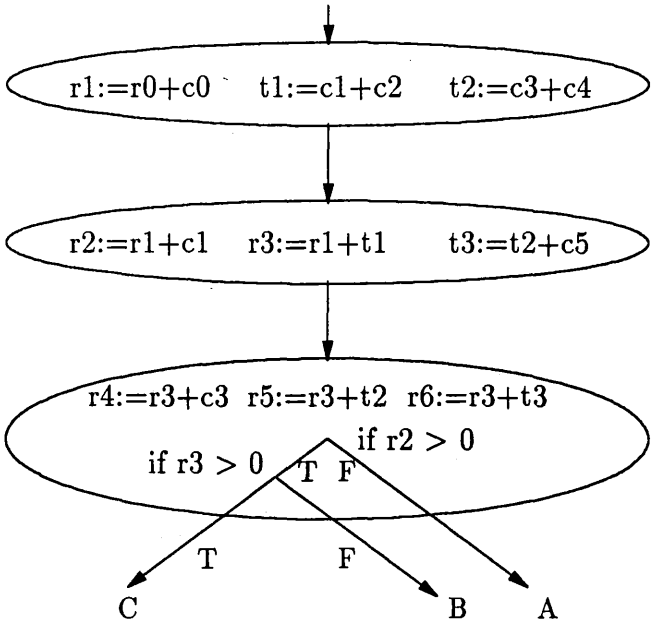


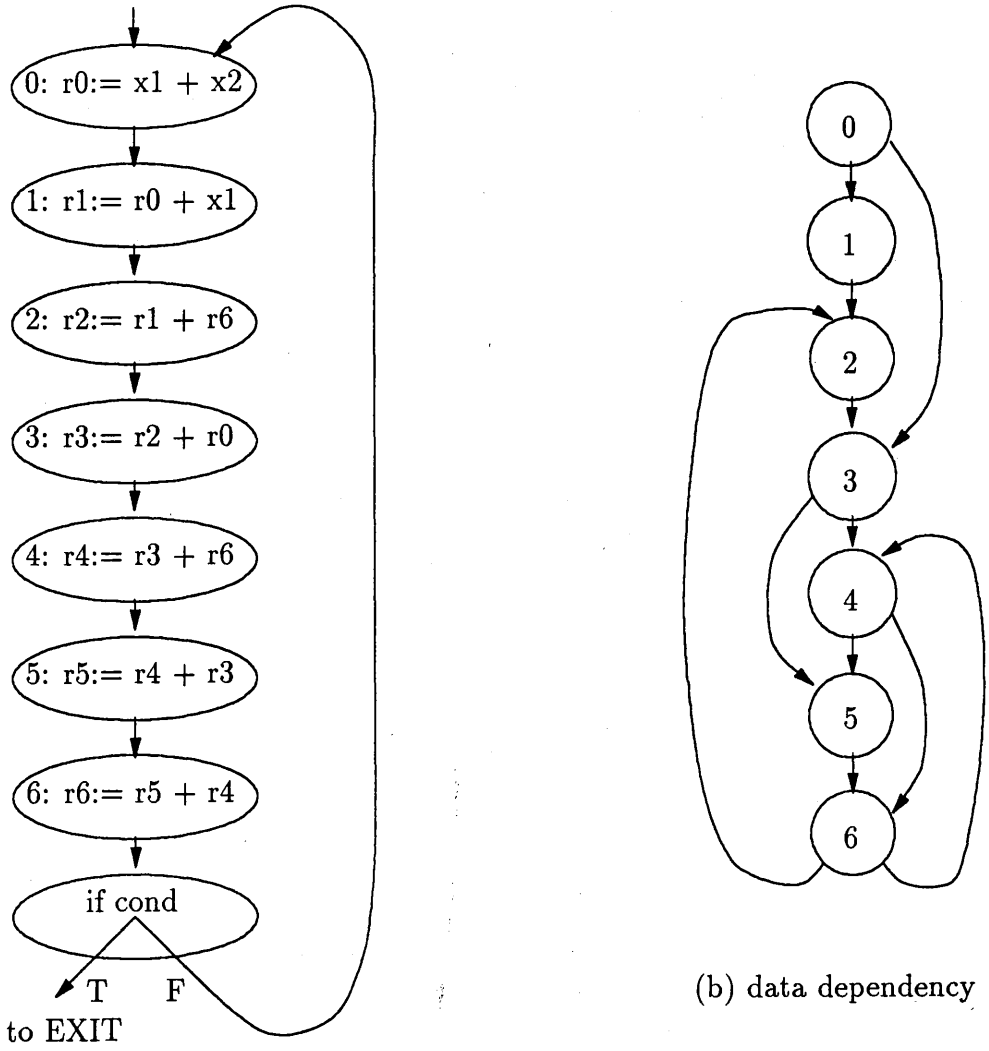
Figure 6.7: Compacted Program for Example 2.

6.3.3 Example 3

THR is especially powerful when combined with loop pipelining since pipelining exposes more operations for THR. Consider, for example, the loop described in Figure 6.8

For simplicity we omitted the conditional jump (loop exit test). Loop pipelining converts this loop into the one shown in Figure 6.9 parallelizing it from $7 * n$ cycles into $5 * n$ cycles.

That is the *optimal* schedule for this loop, preserving data-dependencies between operations. As seen, two adders are needed to execute this loop *optimally.* However, this loop can be further compacted by THR by adding an extra operation as shown in Figure 6.10.



(a) original loop

(b) data dependency

Figure 6.8: Original Program for Example 3 and Corresponding Data-Dependency.

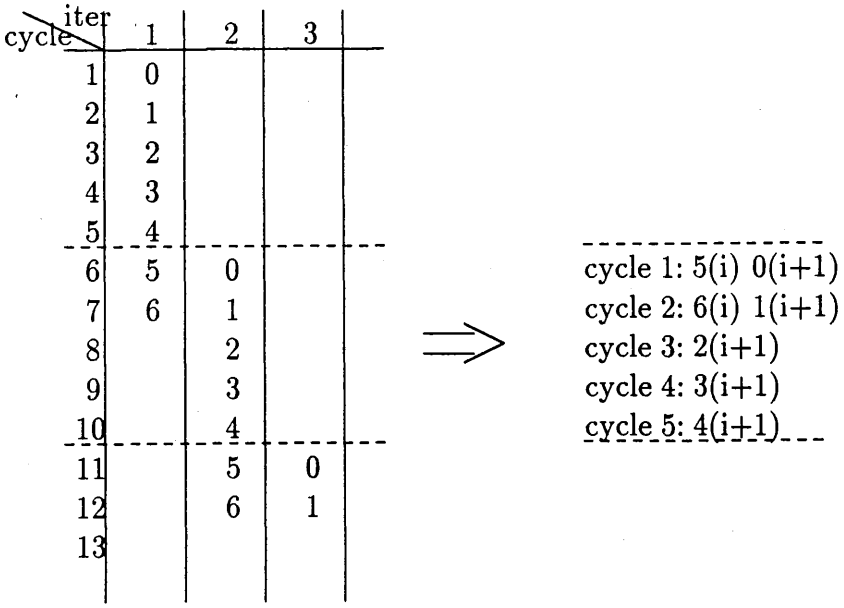
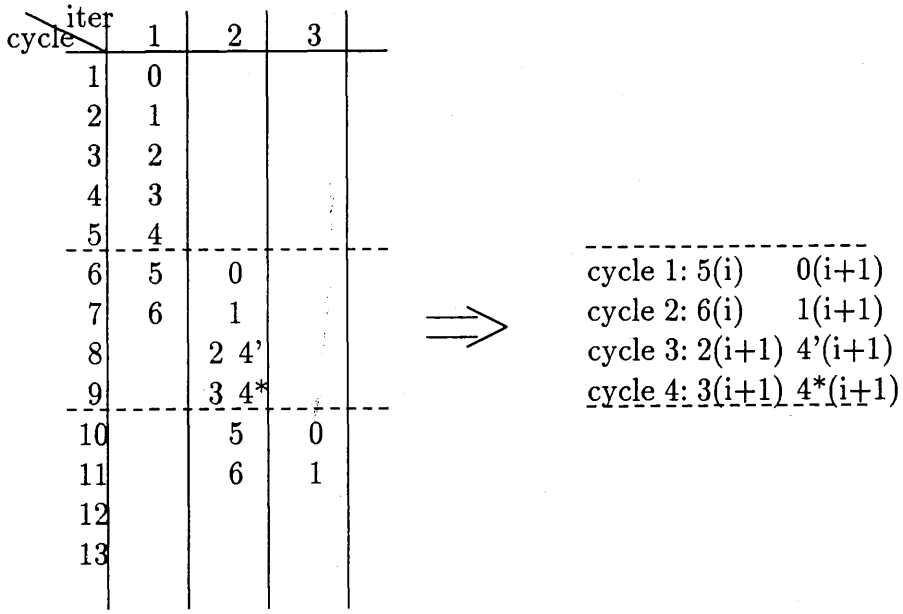


Figure 6.9: Pipelined Loop of Example 3.



4': t4:= r6 + r0
4*: r4:= r2 + t4

Figure 6.10: Pipelined Loop of Example 3 After THR.

So, *using the same resources*, THR reduces the total execution time by 20%. While this example is quite simple—it demonstrates the potential effectiveness of THR combined with loop pipelining.

6.4 THR Experiments

This section details the results obtained by applying the incremental THR on the fifth order elliptic filter example [PaKn89] and the Sehwa example presented in [PaPa88]. In the following tables FDS and FDLS stand for Force Directed (List) Scheduling, PBC for Percolation Based Compiler [PLNG90] and PBCT for PBC with THR.

A. Fifth Order Elliptic Filter:

Table 6.1 refers to the non pipelined case where the model assumes that the execution unit has to be flushed before the succeeding operation can be issued.

Table 6.1: Fifth Order Elliptic Filter—Non-Pipelined

	Without loop pipelining				With loop pipelining	
Res	*FDS*	*FDLS*	*PBC*	*PBCT*	*PBC*	*PBCT*
3+, 3*	17	17	17	16	16	16
3+, 2*	18	NA	18	17	17	16
2+, 2*	19	18	18	17	17	17
2+, 1*	21	21	21	20	20	19

Table 6.2 is for the pipelined case where the functional units units can accept new input each cycle. The results for the elliptic filter show that even though

Table 6.2: Fifth Order Elliptic Filter—Pipelined

	Without loop pipelining				With loop pipelining	
Res	*FDS*	*FDLS*	*PBC*	*PBCT*	*PBC*	*PBCT*
3+, 2*	17	17	17	16	16	16
3+, 1*	18	18	18	17	16	16
2+, 1*	19	19	19	18	18	18

incremental THR is powerful when applied to the loop body–it may yield further parallelization when combined with loop pipelining.

B. Sehwa:

The Sehwa example is an implementation of a digital filter with 16 points. Using the same semantics as [PaPa88], our system reduces the schedule from 6 time steps to 5. Using structural pipelining rather then functional pipelining (see [PLNG90]) incremental THR reduces the schedule from 10 steps into 8 as shown in table 6.3.

Table 6.3: Sehwa's Digital FIR Filter - Pipelined

Pipelining	*SEHWA*	*PBCT*
Functional	6	5
Structural	10	8

Chapter 7

Specific Issues

In this chapter we describe some issues that are peculiar to our compiler. Although design and implementation of a parallelizing compiler usually involves many specific optimizations, enhancements and techniques, we detail here only the more characteristic ones that have an important impact on the performance of the compiler.

7.1 Dead-Code-Elimination "On The Fly"

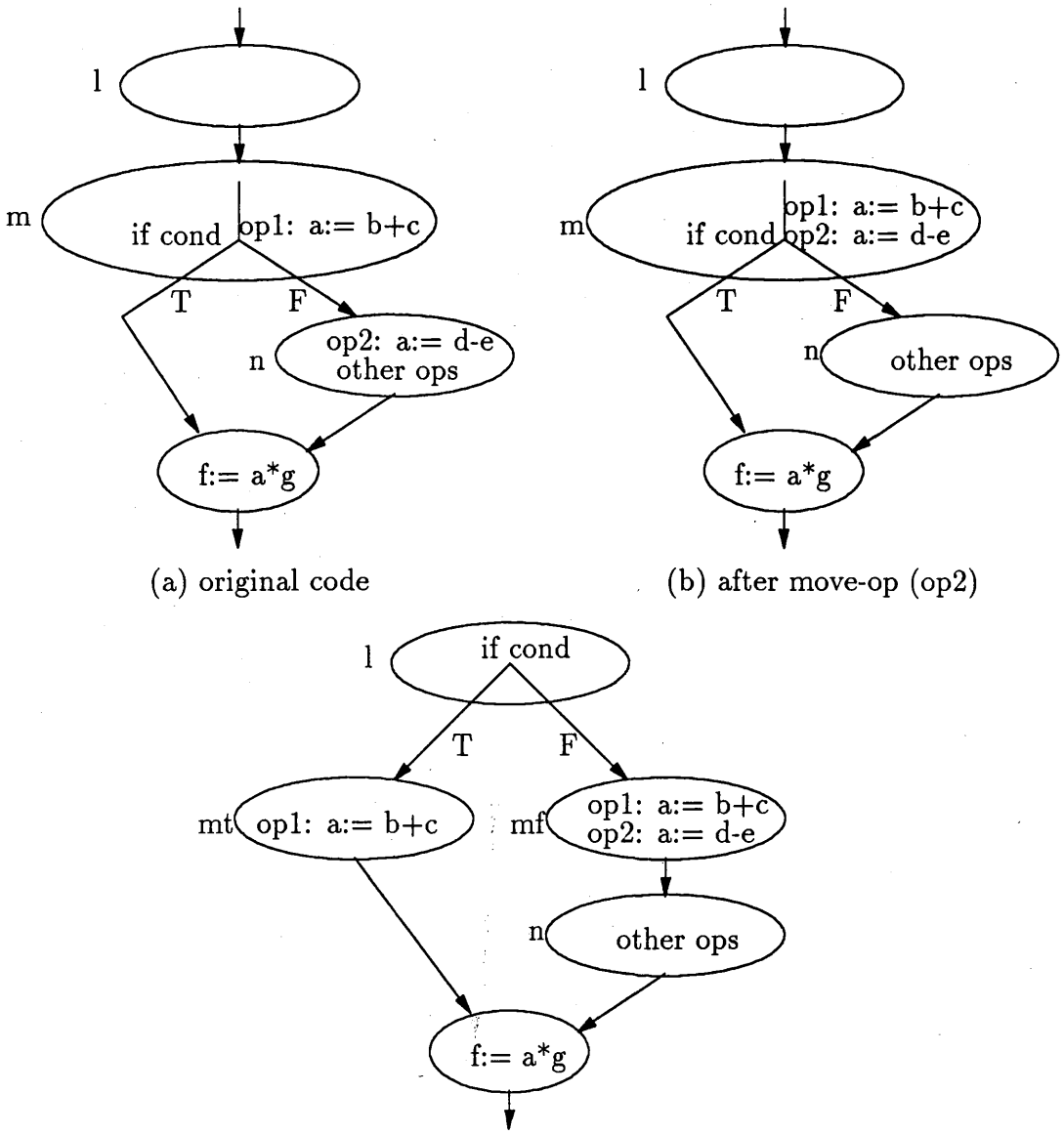
Dead-code-elimination [ASU86] is a well known technique to remove redundant operations from the code. Operations which become redundant (or dead) as a by-product of the parallelizing transformations should be removed. However, applying the traditional (global) dead-code-elimination techniques may become very costly if applied frequently. Thus, a local and efficient technique is required. Fortunately, the fact that we keep the live/dead information, locally, in each of the program's nodes enables us to apply a *local* dead-code-elimination transformation "on the fly" during the parallelization process.

One of the two most common scenarios in which dead code is generated is described in Section 4.2.1. The other happens when moving up an operation *op2* from node n to m as shown in Figure 7.1(a). Notice that *op1* is live since a is used by $f := a * g$ on the true branch of m. After move-op we get the segment shown in Figure 7.1(b). When a move-cj is performed we get the code shown in Figure 7.1(c). Thus, *op1* in node *mf* becomes dead and should be removed.

Since the live/dead information in each of the nodes is kept and updated locally, carrying out dead-code-elimination means checking, before move-op and move-cj are performed, that the relevant paths in the node do not contain any dead operations and removing such operations if needed. This task is relatively easy and efficient.

7.2 Procedure Calls and Inter-procedural Live Analysis

The "programs" discussed so far throughout this thesis assumed implicitly that the input is a single procedure and consequently represented by a single control graph. However, procedure calls are an integral part of any higher-level language and therefore should be considered by our compiler. Procedure calls exhibit not only a change in the control flow of the program but also affect the live/dead information in higher and lower levels of the calling hierarchy. Since C allows recursive calls, we allow them in our compiler too.



(a) original code

(b) after move-op (op2)

(c) after move-cj

Figure 7.1: Generation of Dead-Code by Move-op

There are several published works on inter-procedural flow analysis [He77, Ba78, JoMu81]. Like other techniques, the algorithm presented here is also inspired by the idea described in detail in [ASU86]. For each procedure we perform "the traditional" live analysis but changes in each procedure are also reflected in higher and lower level procedures (in the calling hierarchy) until all levels have settled down.

Procedure calls are represented in the three-address-code by lines with the called procedure name. Procedure call operations are never compacted into nodes with other operations. In other words, nodes which contain procedure call operations do not contain any other operation.

The initial live/dead analysis is performed on the sequential (input) code (where each instruction holds a single operation). Then, during the parallelizing process (see Chapter 3) the live/dead information is locally and incrementally updated. The algorithm presented relies on the fact that the set of registers live at the top of the call operation is the union of two sets: the set of registers live at the top of the successor node and the set of registers live at the top of the called procedure. For example, in Figure 7.2 the set of registers live at the top of "call proc1" (in main) is the union of the set of registers live at the top of instr1 and the set of registers live at the top of proc1().

7.2.1 Definitions:

- IN-SET(l): Set of all registers live at instruction l.
- LIVE-SET(proc): Set of all registers live at entry of procedure proc.
- GEN-SET(p): Set of all registers used in path p.

- KILL-SET(p): Set of all registers killed in path p.

7.2.2 Algorithm Description

```
procedure live_analysis(proc, live-set)
  for each instruction l in proc
    IN-SET(l)= new-set;
  changes= TRUE;
  while (changes)
    changes= FALSE;
    for each instruction l in proc backwards begin
      if l holds a CALL operation begin
        /* assume the operation is CALL call-dest */
        if ( (call-dest!= proc) and changes ) begin
          /* non-recursive call */
          stack-pointer++;
          live-stack[stack-pointer]= live-set;
          live-set= LIVE-SET(call-dest);
          live_analysis(call-dest,live-set);
          live-set= live-stack[stack-pointer];
          stack-pointer--;
        end
        else if (call-dest== proc)
          /* a recursive call */
          live-set= live-set ∪ LIVE-SET(proc)
```

```
        temp= IN-SET(successor of l);
      temp= temp ∪ LIVE-SET(call-dest);
      end
      else /* not a CALL operation */
      temp= new-set;
      for each path p in l begin
        temp= GEN-SET(p);
        temp1= IN-SET(successor of l in path p);
        temp1= temp1 \ KILL-SET(p);
        temp= temp ∪ temp1;
      if ( l is a RETURN instruction)
        temp= live-set;
      end
      if ( temp != IN-SET(l) ) begin
        IN-SET(l)= temp;
        changes= TRUE;
      end
    end
  end
end (live_analysis)
```

7.2.3 Example

Figure 7.2 illustrates a program with two regular calls and a recursive one. Beginning with the main procedure we compute the live set of all nodes that do

not hold call operations. For these nodes the live set is derived by subtracting registers killed in this node from the live set of their successors (see [ASU86]). When a call is encountered we either proceed to analyze the called procedure (in a non-recursive call) or add the set of registers live at the top of this procedure to the set of registers live at the call's (only) successor (in a recursive call case).

Refer to Figure 7.2: we compute, backwards, the live set of all nodes in main(), beginning from the end (igoto), until we hit the call to proc1(). Since IN-SET(call proc1) equals to the union of IN-SET(instr1) and LIVE-SET(proc1) we have to compute first the latter set. In order to do that, we then proceed to compute the live set of all nodes in proc1() starting from its last (igoto) instruction. When call proc2 is found we leave proc1() and begin to compute LIVE-SET(proc2). This time, since proc2 calls itself, IN-SET(call proc2) equals to the union of IN-SET(instr3) and LIVE-SET(proc2).

This iterative process continues until there is no change in any of the live sets of any node in the program.

7.3 Load-After-Store Elimination

One of the traditional bottlenecks of high-performance systems is the memory-access problem caused by load and store operations. Since memory access time is normally much larger than the CPU cycle time each memory access potentially degrades the overall system's performance. To alleviate this problem two traditional approaches have been introduced. The first (caching) deals with dividing the

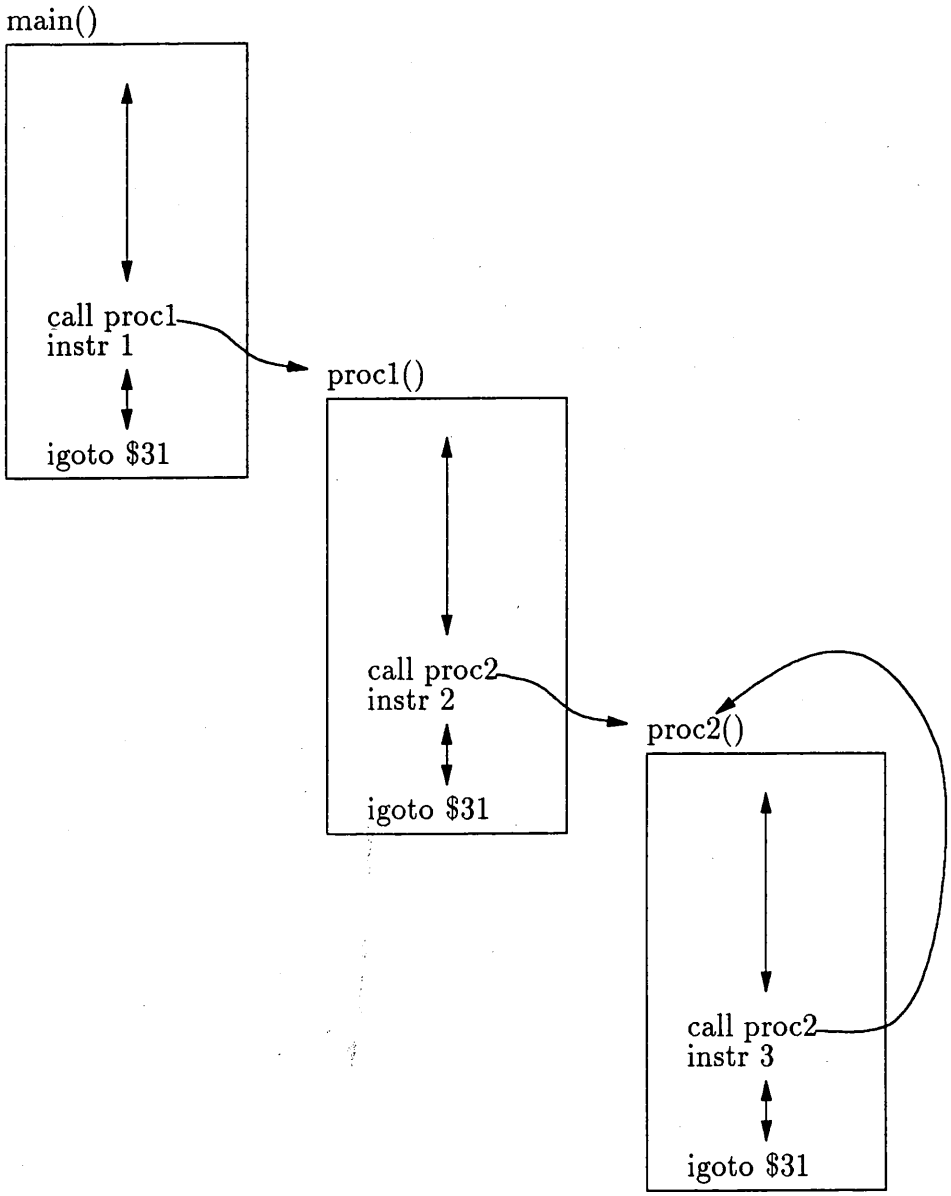


Figure 7.2: An Example of Calling Graph

memory hierarchically such that the CPU accesses only the lowest (fastest) hierarchy. The other approach (memory banking) "parallelizes" the memory such that different memory modules may be accessed concurrently by the CPU. Obviously, both solutions are hardware solutions.

We implemented in our compiler a very powerful (software) technique to reduce the memory transfer rate by eliminating load operations occurring after store operations. Not only are the number of memory accesses reduced, but also the total speed-up may increase dramatically.

We have adopted the idea of *redundant load elimination* ([CCK87]) to fit into our local transformations. In the context of PS, load-after-store elimination happens directly as a result of the move-op transformation. In our compiler this optimization applies to non-loop code and works in the presence of conditional jumps. The original technique ([CCK87]) is not applicable with these two features. Yet another advantage of our technique is that the optimization is done only when needed to enhance parallelism and so register pressure is not unnecessarily increased as it might if the optimization was done separately from PS transformations.

Consider, for example, the following loop:

for $(i = 1; i < 20; i++)$
  $A[i] = A[i-1] + B[i];$

This loop transforms into:

  $base := 176;$
  $i := 0;$

```
(LABEL L1)
        a:= M[base-88];
        b:= M[base-164];
        a:= a+b;
        M[base-84]:= a;
        base:= base+4;
        i:= i+4;
        cc0:= i <= 76;
        if cc0 (LABEL L1)
(LABEL exit)
```

The first load in the second iteration addresses the same memory location as the first iteration's store operation. This reflects the recursion in a[i]. After the loop body is compacted we get the code illustrated in Figure 7.3.

The next step in loop pipelining involves moving up operations from n into m. But since the load in n and the store in m are referring to the same location—instead of moving the load up we simply optimize the loop into the form shown in Figure 7.4. This loop does **not** contain the load anymore.

Since for each memory-access operation we keep its symbolic derivation locally, this optimization is actually a comparison of two symbolic derivations, which is relatively efficient and done as part of the move-op transformation. Therefore, practically, this optimization comes almost for free.

Actually, by using local transformations we do not even have to realize that these operations are inside a loop. Whenever move-op is invoked as in the code

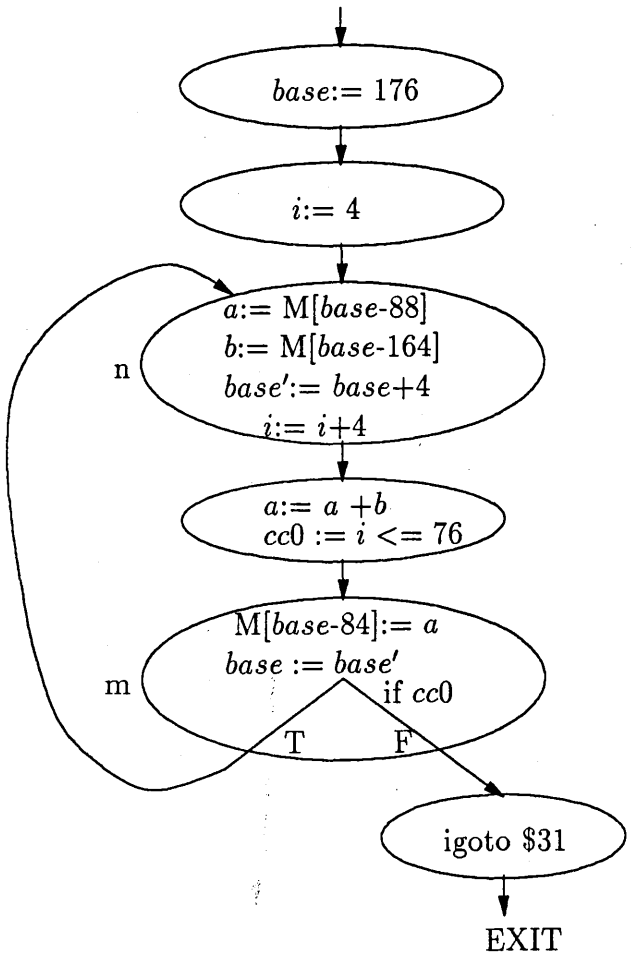


Figure 7.3: Code After Loop Pipelining When Pipe_fence Includes Node *n*.

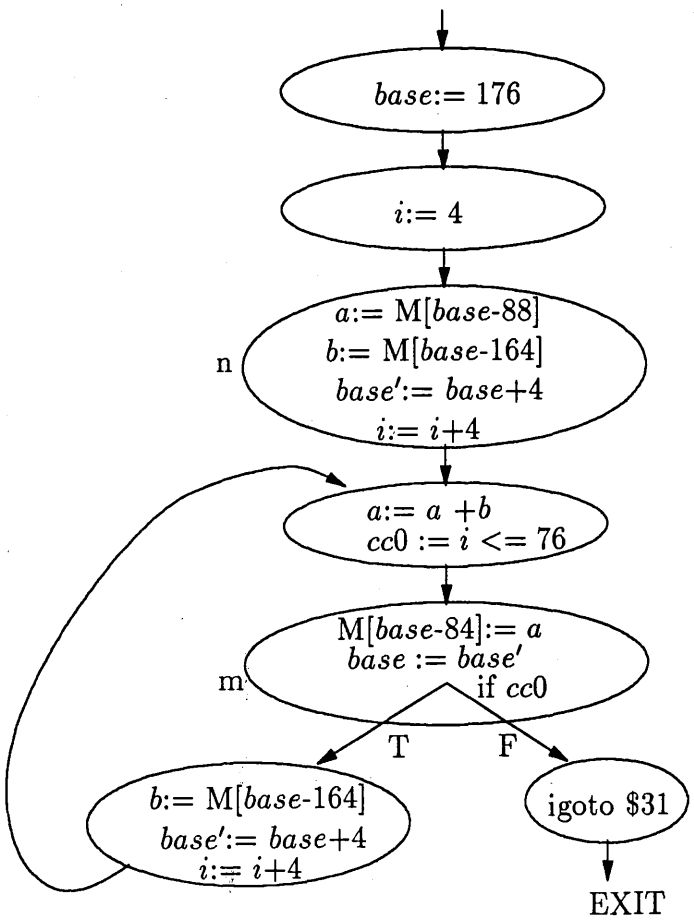
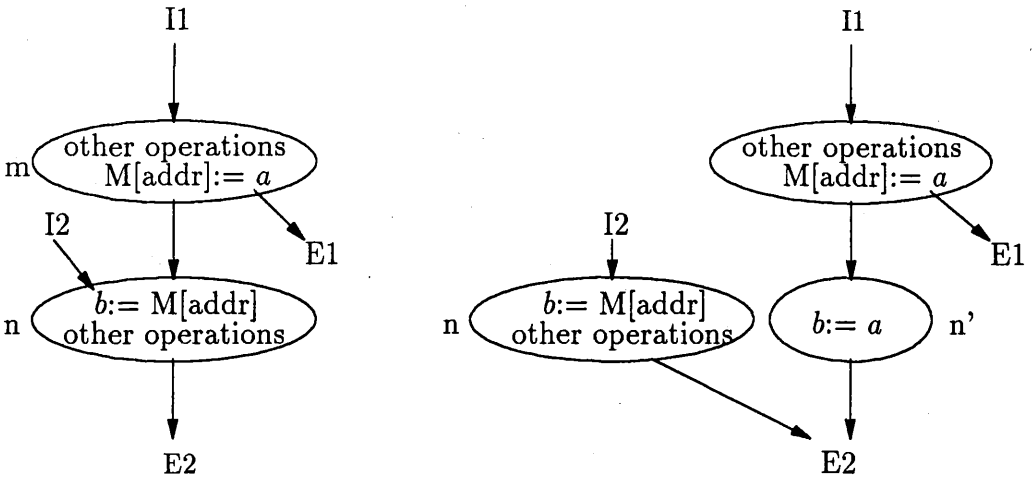


Figure 7.4: Code After Load-After-Store Elimination.



(a) prior to move-op (b) after move-op

Figure 7.5: Code Before Move-op.

segment shown in Figure 7.5 (a) the code transforms into the one illustrated in Figure 7.5 (b).

7.4 Loop Detection and Incremental Update

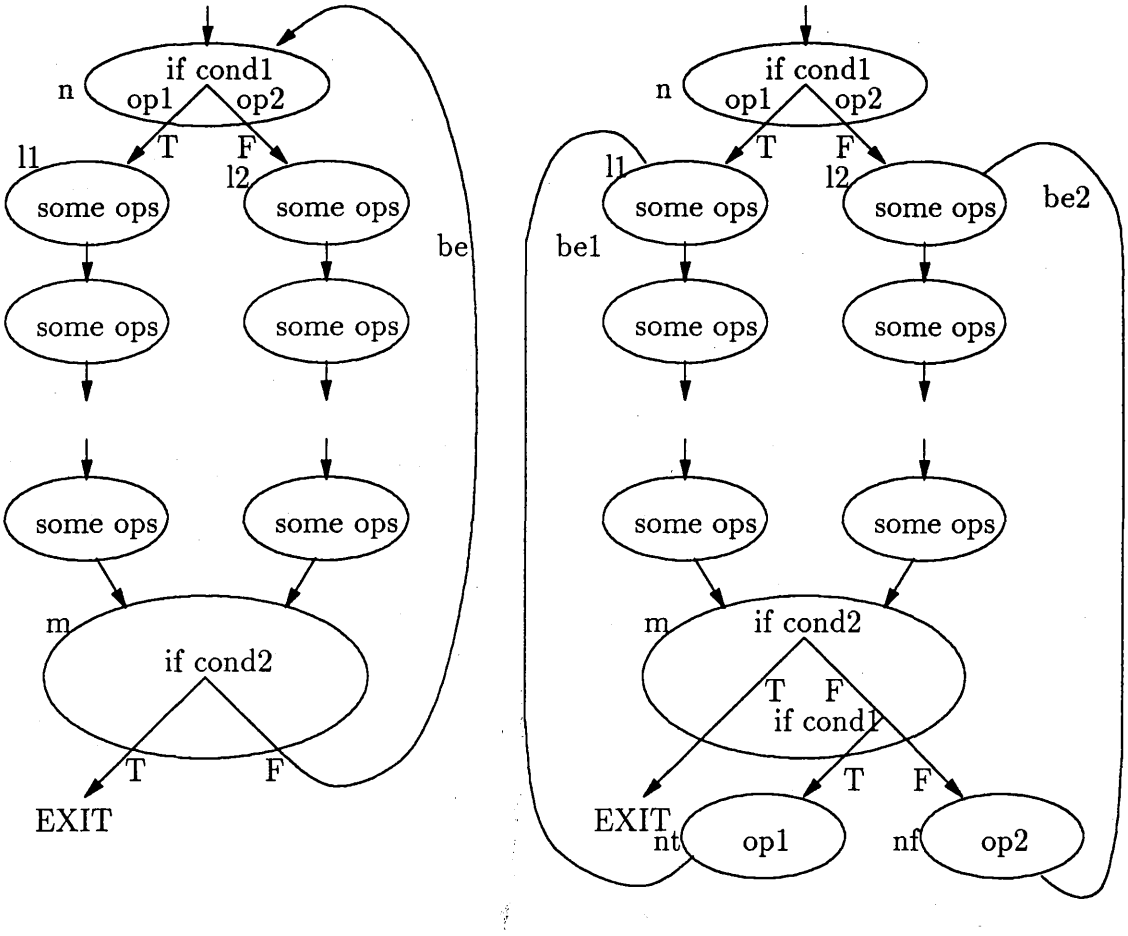
According to the principle of locality, programs tend to spend most of their time executing loops. Therefore compacting loops successfully obviously affects the overall performance of the compiler. Loop information is important not only in the context of loop pipelining (see Section 3.5.2) but also for other optimization techniques and loop-based algorithms like redundant induction variables removal (see Chapter 4), removal of other copy-operations generated by renaming, reaching-definition computation, maxcomp and others.

Programs which are written exclusively with structured flow-of-control statements (if-then-else, for, while-do, continue, break etc.) always create a *reducible*

graph. Even programs written using goto statements are almost always reducible. Intuitively, reducible programs are those in which there is no jump into the middle of a loop from outside the loop (and not through the loop header). Detection of reducible loops is easy and implemented in the compiler according to the conventional algorithm presented in [ASU86]. Unfortunately, during loop pipelining, graphs that are reducible may become irreducible. The loop detection algorithm fails for these loops, therefore, we cannot apply the loop detection algorithm on an already compacted program.

In Figure 7.6(a) a reducible loop is illustrated. There is one node that dominates all other nodes in the loop (node n). However, if the conditional jumps in n are moved up (into node m) across the backedge (see Section 3.5.2) an irreducible graph is generated as illustrated in Figure 7.6(b). This phenomenon happens in other cases as well.

The strategy we have chosen to solve the irreducibility problem was to update the loop information incrementally during the parallelizing process. Since loops are uniquely defined by their backedges (the triplet of: source node, destination node and the backedge connecting these nodes) and since we know precisely when the transformations change the loop information, we modify this information whenever there is a move (move-op or move-cj) across a backedge that changes the backedge triplet. For example, in Figure 7.6(a) the triplet is {m, n, be}. After move-cj we get two loops with the following triplets {nt, l1, be1} and {nf, l2, be2}.



(a) A reducible graph (b) An irreducible graph (after Move-cj)

Figure 7.6: A Reducible Loop Becomes Irreducible

7.5 Memory Reference Disambiguation

When high performance is achieved through aggressive code motion, it is very important not to bottleneck the parallelizing process by an overly conservative approach. Indirect memory references are created by the use of array indexing and pointers. While pointers constitute a complex form of indirection and are often dependent on run-time data (and therefore hard to disambiguate), array references offer the greatest potential for parallelism and can, relatively easily, be precisely analyzed.

As mentioned in Section 3.5.2, the ability to pipeline the inner loops of a program is crucial for the overall performance of the compiler. However, scientific programs include a significant number of indirect references, most of which are due to array references. If for each store-load pair of operations (i.e. a store followed by a load operation), we *always* assumed that they refer to the same memory location (the conservative approach), we would, actually, serialize all memory references and degrade drastically the performance as discussed and shown in [Ni85b]. The performance degradation occurs since the whole chain of operations which are data-dependent on the load's result are prevented from moving up. The inability to move up the load causes a bottleneck in the compaction process.

The disambiguation algorithm used by our compiler is based on the work in [Ni85b] and was implemented by Haigeng Wang. In a pre-compaction pass, all memory accesses in the program are expressed in terms of *symbolic derivations.* A symbolic derivation of an operation is its most primitive presentation in terms of loop invariants, loop indices and other (predefined) variables. The symbolic derivations are derived by variable-folding—each memory address is defined

in terms of previous reaching definitions of its source variables. By a recursive process, all addresses are expressed in terms of variables that cannot be further simplified (folded). The special case of variables called *induction variables*, for which the variables are defined by themselves (and therefore the recursion would run infinitely), is treated differently: before evaluation of the symbolic derivations, all induction variables are detected and substituted by derivations that contain an imaginary loop counter.

Hence, after substitution of all memory addresses by their symbolic derivation, the disambiguation problem is to find whether two addresses have an *integer* solution when their symbolic derivations are equated. These equations are called *Diophantine equations.* These equations can be solved for the linear case but cannot be handled for the non-linear case, therefore, in the variable-folding process, we stop substitution whenever an operation, other than addition, subtraction or multiplication is involved.

7.6 Simulator

The simulator which we built for our compiler was inspired by the lack of existing supporting hardware to validate the derived compaction results. Since we compact programs beyond basic block limits and since we allow compaction of several conditional jumps into one node, it turns out that even for relatively short programs, it is simply impractical to check the correctness of the results produced by the compiler. Traditionally, a compiler is validated by running the compacted code on the target architecture and comparing the results with the results derived

by running the serial code on the same input data. However, since our compiler is mainly a research/study oriented compiler, it was hard to find a specific machine to run our compiled code on. On the other hand, a compiler has to be debugged and validated. This required us to come up with a debugging and validation tool. That is exactly the goal of the simulator.

The simulator works by "executing" the serial (sequential) program, step by step, (as defined in Section 2.4) and recording the results of all registers and memory locations involved for a set of input data. Then, it "executes" the compacted (parallel) program and compares the results to the ones recorded for the sequential code. By "execution" we mean a total emulation of the program as if it was running on the hardware described in Section 2.4. This includes setting and resetting of condition code registers, modification of all memory locations and architecture-registers, loading values from memory locations and branching and returning from procedure calls.

The simulator is implemented, basically, using two arrays: one representing the register-file and the other the memory space. After initialization of all memory locations and registers (by the sample input data), the simulator executes each node in the program in three steps which are part of the basic machine cycle:

1. Operands (for all operations) and condition code registers are read from the two arrays (register-file and memory).
2. All operations are executed by calling a C procedure, that emulates the physical operation. All condition codes are evaluated and the next node is determined.

3. The results of operations on the path chosen (for single cycle operations) and the results of operations which began in previous nodes and completed their execution in this node, are written back to the two arrays.

When completed, all elements of the two arrays are compared with the two arrays derived for the sequential execution.

During the development of the compiler we found out that the simulator can serve not only as a validation tool but also as a very powerful debugging tool. This is due to the fact that we use *semantic-preserving* and *atomic* set of transformations. Suppose we run the compiler on a specific benchmark and get negative results (i.e. the results for the compacted program are different from the results for the sequential program). If the benchmark is non-trivial, it is quite complicated to figure out where and why a bug happens. The compaction process involves sometimes tens of thousands of code transformations and it is not feasible to hand-trace the bug. However, since PPS transformations are semantic-preserving, if at any time (i.e. in any compaction phase) we freeze the program and simulate it the result should be correct if there are no bugs. Conversely, if there *is* a bug, there is a very high likelihood that the simulator will catch it. Since the transformations are atomic, we could concentrate on the transformation which actually converts the program from a correct one to a wrong one. Hence, the debugging procedure can be summarized as follows:

1. Run the compiler to the n-th PPS transformation.
2. Simulate the compacted code.
3. If simulation is correct increment n and repeat from step 1. Otherwise, concentrate on the n-th transformation.

In practice, we found a much faster way to find n: we first derived a m for which the simulation was correct. Then, we found a p for which the simulation results were bad. By bisecting, iteratively, the region [m:p], n could be found easily.

The simulator serves not only as a validation and debugging tool but also as a run-time emulator tool. Dynamic speed-up (see Chapter 8) is only one example of parameters that may be produced by the simulator as a run-time measure.

Chapter 8

Results

In order to investigate hardware design trade-offs and evaluate achievable parallelism we carried out a series of tests on well-known benchmarks. The results of these tests are presented in this chapter. The benchmarks represent a spectrum of applications in different scientific and computer-science domains. The first 24 benchmarks that we use are the Livermore kernels representing different known loop structures. The next 6 benchmarks are known as Stanford benchmarks: *FFT*, *bubble*, *cos*, *minmax*, *quicksort* and *permute*. *Line26*, *crale10*, *unriems50* and *nav7055* are core loops from CFD (Computational Fluid Dynamics) codes.

As mentioned in Chapter 2, the compiler may be reconfigured to use different hardware models and/or parallelization input parameters. For example, we can compact a program with three pipelined functional units, with 20 registers and with the conditional execution model described in Chapter 2. In another run we may use only two functional units (not pipelined), 14 registers and disallow conditional execution. We refer to these parameters as *hardware parameters*. Naturally, each target architecture has its own set of hardware parameters. But there are other parameters that may control the parallelization process: we may use renaming of registers (see Chapter 4) or disallow it, we may use induction variable

removal and copy elimination (see Sections 4.4.2 and 4.4.3) or ignore them. We can further apply load-after-store optimization (see Section 7.3), allow/disallow compaction while moving conditional jumps or check what parallelism is available when disambiguation is switched off. We refer to these evaluation parameters as *parallelization parameters.* Since one of the goals of this research was to evaluate compilation techniques and their impact on parallelism we show the effect of each on the system's performance.

Since the number of parameters checked is relatively high, the best way to evaluate the impact of a specific parameter on the overall performance is by changing this parameter while all others are kept unchanged. Obviously, this prevents presenting *all* possible combinations, but on the other hand makes the comparisons meaningful and tractable.

8.1 Definitions

In this section we define some terms used throughout the chapter.

- *Speed-up*

  The speed-up is the ratio of serial code execution time (in cycles) to the compacted code execution time. (Cycle counts were obtained via the simulator). It is computed by the following formula:

$$Speed\text{–}up = \frac{Number\ of\ cycles\ executed\ by\ serial\ program}{Number\ of\ cycles\ executed\ by\ compacted\ program}$$

  Given this metric, it is worth pointing out that one has to be careful when comparing speed-up achieved with uni-cycle operations vs. speed-up achieved

with multi-cycle operations. In practice (see Section 3.1) the cycle time of uni-cycle machines is much larger than that of pipelined machines. Therefore, when a speed-up of 4 (for example) is obtained for uni-cycle machine and a speed-up of 4 is obtained for multi-cycle machine—these number do not imply that the *actual* performance of the systems is the same. With equal speed-up, pipelined machine will, everything else being equal, execute faster due to shorter cycle time. Hence, the results presented in this section are somehow "unfair" to the pipelined machines. In the context of benchmarking for a particular architecture, a more accurate measure for *system performance* would be speed-up normalized by the cycle time of the actual machine.

When uni-cycle operations are used, in some of the benchmarks (e.g. L1, L2, L3, L4, L7, L9, L11, L12 etc.) we were able to compact the loop into a single node with unlimited resources. Compaction of a loop into a single node was one of the termination conditions to the loop pipelining algorithm. However, in some of these loops there are no loop-carried-dependencies, thus the parallelism that may be achieved by scheduling more iterations per cycle can be much higher than given in the tables. In the context of uni-cycle operations it should be further noted that for some benchmarks the speed-up is greater than 2.00 even when only two resources are available. This happens since during the parallelization process we apply some code optimizations that are able to eliminate operations which become redundant.

- *Weighted Harmonic Mean (WHM)*

  The Weighted Harmonic Mean is a measure which attempts to normalize the speed-ups by taking into account their relative sizes. The WHM is computed

by the following formula:

$$WHM = \frac{\sum_{i=1}^{n} N_i}{\sum_{i=1}^{n} \frac{N_i}{S_i}}$$

Where:

N_i is the number of cycles needed to execute (serial) benchmark i.

S_i is the speed-up derived for benchmark i.

For example: suppose three different benchmarks are given. The first requires 1000 cycles to execute sequentially, the second 5000 cycles and the third 1800 cycles. Therefore, 7800 cycles are required to execute all three serially. Further assume that the speed-up obtained for the first benchmark is 5, for the second is 10 and for the third is 6. Consequently, the compacted programs require, in total, 1000 cycles to complete {(1000/5)+(5000/10)+(1800/6)}. Hence, the *real* speed-up (when all three are executed) is 7.8. This is exactly what WHM measures.

- *unlimited resource constraints*

  In the following sections we detail results for unlimited resources (functional units) as well as for limited resources. By unlimited resource we mean the number of resources required to execute the schedule produced by the compiler "as is". In other words, this is the number of units needed to execute the instruction with the most operations in the compacted program. Note that *unlimited* resources does not mean *infinite* resources.

- *canonical resource constraints*

  Unless otherwise specified, by *canonical resource constraints* we mean that only two operations can be issued in any given cycle but at most one memory load and/or one conditional jump. These constraints are chosen somewhat

arbitrarily, but they are representatives of current superscalar/VLIW technology in general and an actual machine in particular.

8.2 Hardware Parameters

Different hardware model configurations result in different speed-up ratios. In this section we illustrate the impact of four hardware parameters on the overall system's performance. We begin with the impact of the conditional execution model on the speed-up. Then we present results obtained for pipelined operations compared to uni-cycle operations. We measure the effect of the number of registers on compaction and conclude with comparing performance achieved with different numbers of functional units.

8.2.1 Conditional Execution

The conditional execution model explained in Section 2.4 may be a worthwhile enhancement over the standard (non conditional) model since it reduces the time needed for the execution of conditional statements. In the conditional execution model only two serial steps are required for the execution of a conditional operation: (a) evaluation of the condition-code register and (b) if the condition is true execution of operations. In the standard model three serial steps are needed: (a) evaluation of the condition-code register, (b) branching to destination if condition holds and (c) execution of the operation. In branch-intensive code this difference may be important; however, the gain does not come for free since the

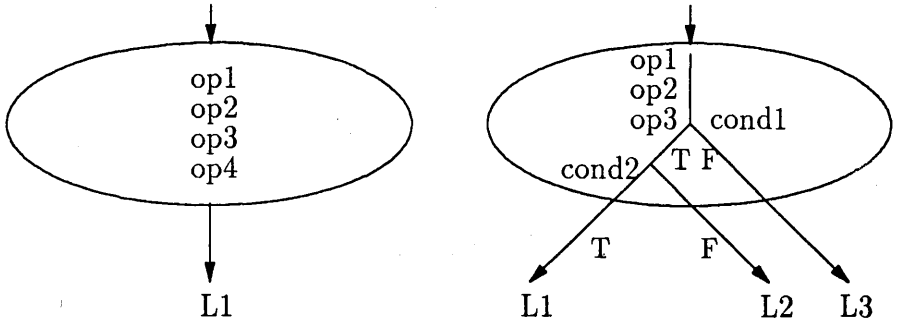


Figure 8.1: Possible Nodes When Conditional Execution Disabled

conditional execution model involves higher degree of complexity[1]. We measured the effect of the conditional execution model by comparing the speed-up obtained with and without conditional execution. When conditional execution is disabled, an operation can move (move-op or move-cj) from node n into node m only if node m does *not* result in conditional execution of the operation. That is, only compaction resulting in nodes like the two shown in Figure 8.1 is allowed.

In Table 8.1 all operations are single cycle operations. For comparison, we give the speed-up with unlimited resources and with limited resources as well. The resource constraints are the *canonical resource constraints* mentioned earlier.

The conditional execution model yields 233% better speed-ups on the average and 213% in WHM with unlimited resources. However, when resources are limited the gain drops to only 10%. The difference in the speed-up between the two models decreases in the latter case since the resource constraints limit the exploitation of substantial parallelism, therefore making both speed-ups comparable.

[1]Operations have to be tagged so that their write-back result can be controlled at run-time.

8.2.2 Pipelining of Operations

As discussed in Chapter 3 one of the features of our compiler is the exploitation of temporal parallelism as well as spatial parallelism. While spatial parallelism is achieved through the use of a number of functional units in parallel, temporal parallelism may be achieved by using pipelined functional units. In this section we illustrate the impact of functional unit pipelining on the overall speed-up. In Table 8.2 we use the *canonical resource constraints* as the limited resources. For the pipelined operations we assume that memory loads take 2 cycles, all floating-point operations take 3 cycles and all other operations take one cycle.

The speed-up achieved with unlimited resources for pipelined operations is less than the corresponding uni-cycle operations for two reasons: (a) it is (objectively) harder to achieve similar compaction for pipelined operations as for uni-cycle operations since multi-cycle operations may affect many more successor nodes through data-dependency and may prevent better utilization. (b) we use a loop pipelining algorithm (see Section 3.5.2) that tends to converge very fast. Remember that we do not allow nodes that have already been in a fence to "break" unless *all* operations from that node can move up. This prevents better pipelining. While in the uni-cycle case this restriction is negligible (i.e. the loop pipelining is limited by other factors like data-dependency, disambiguation etc.), in retrospect, for the pipelined case this is crucial and stops the pipelining well ahead of the other factors mentioned. Other algorithms like Perfect Pipelining [AiNi88c] should produce better results. On the other hand, the results with resource constraints for the pipelined case are better than for uni-cycle operations since the resource constraints "stretch" (lengthen) the schedule in such a way that

resources, rather than operations' latencies, become the dominant factor in compaction. This creates many more opportunities for scheduling and enables better "filling" of nodes. The "empty" nodes (no-ops) that were an artifact of the fast convergence of the pipelining algorithm (they were originally introduced for latency preservation) disappear in the limited resources case.

8.2.3 Number of Registers

As discussed in Chapter 4, register allocation and renaming have a major impact on the achievable speed-ups. In this section we show the effect of the number of registers used on speed-up. In Table 8.3 we present the number of registers used in the (serial) input program followed by the number of registers in the compacted program. The number of registers needed is the maximal number of registers concurrently live at any node. One may see that the number of registers for the compacted program is on average 2–3 times the number for the input program. The two speed-up columns represent the speed-up achieved when we allow the same number of registers as in the input program and with the maximal number needed (unlimited number of registers).

8.2.4 Number of Functional Units

Selecting the right number of functional units for a given architecture is one of the major design decisions. Increasing the number of functional units means not only an immediate increase in area due to the addition of more units but also increase in the number of multiplexers and wires. Therefore, the number

of functional units has to be determined very carefully. The main goal of the results presented in Table 8.4 and in Table 8.5 is to analyze how the speed-up grows as the number of functional units increases. The former table presents the results for uni-cycle operations while the latter shows the speed-up for multi-cycle operations. The multi-cycle operations' latencies are as in Section 8.2.2. For both tables we assume that each functional unit can execute every operation (homogeneous functional units). If, for example, 2 functional units are given, one can issue all possible combinations of 2 operations in this cycle (note: different than the *canonical resource constraints*!).

For each number of functional units, we added two columns. The *utilization* column gives the ratio between the speed-up achieved and the number of functional units available.

$$Util = \frac{Speed\text{–}up}{Number\ of\ functional\ units}$$

This measures how well the units are utilized. The second column is more indicative of the *system's* overall performance: it is the *normalized speed-up*:

$$NS = \frac{Speed\text{–}up}{Unlimited\ resources\ speed\text{–}up}$$

This number is important since it provides information on how well we can enhance the system's performance by increasing the number of functional units.

For the uni-cycle operations the utilization decreases by 2.5 when we increase the number of functional units from 2 to 16. For the pipelined case this ratio is 4.5.[2] The normalized speed-up with uni-cycle operations increases 3.5 times with

[2]Again, since cycle times should be different, direct comparisons between pipelined and uni-cycle cases are questionable.

the increase in the number of functional units from 2 to 16, while in the pipelined case it increases only by 2.5. This relatively low increase is due to fast convergence of the loop pipelining algorithm.

8.3 Parallelization Parameters

In this section we analyze some parallelization parameters and measure their impact on the speed-up. First we test the effect of register renaming (see Chapter 4) on the performance, then we measure the effects achieved by application of the induction variable removal and copy elimination techniques described in Sections 4.4.2 and 4.4.3. We test the effect of load-after-store optimization (see Section 7.3) on the speed-up and then present compaction result without move-cj. We conclude this section with measuring the effect of disambiguation on the overall performance.

8.3.1 Renaming

As mentioned in Chapter 4, renaming of registers is an important optimization in high-performance, parallel architectures. Table 8.6 compares the speed-up achieved with and without renaming.

Clearly, disabling renaming causes a dramatic degradation in the speed-up (54%) in the unlimited resources case. In this table all operations are uni-cycle operations and the resource constraints are the *canonical resource constraints*. When

resources are limited the speed-ups are comparable. Since we have to "fill" only two functional units we can still do well when renaming is disabled.

8.3.2 Induction Variable Removal and Copy Elimination

We presented in Chapter 4 a new technique to remove redundant induction variables and eliminate copies generated by renaming of induction variables. Since loop pipelining frequently involves intensive induction variable renaming it was interesting to measure the effect of induction variable removal and copy elimination techniques on the overall speed-up in the presence of limited resources where removing copies is critical. In Table 8.7 we compare the speed-up achieved when the techniques are applied vs. the speed-up obtained without their application. As before, we assumed here the *canonical resource constraints.*

We see from Table 8.7 that taken together these techniques account for 35.3% improvement. Therefore, without a means for eliminating copies generated during loop pipelining, the gain obtained by renaming may be significantly offset by the introduction of additional copies into the compacted code.

8.3.3 Load-after-store-elimination

Table 8.8 compares the results achieved with load-after-store-elimination optimization (discussed in Section 7.3) vs. the speed-up derived without this optimization.

Comparing WHM, the gain by this optimization is 45.6% for the unlimited resources and 3.5% for the limited resources case. However, in this case the average (or the WHM) may be misleading: this optimization applies to only 12 out of the 35 benchmarks. If we measured the improvement only for the 12 applicable benchmarks we would get 311% improvement in the unlimited case and 13% for the limited resources case compared to the results obtained in the absence of this optimization.

8.3.4 Compaction Without move-cj

One of the "traditional" arguments against Percolation Scheduling has always been code explosion. The main contribution to code explosion is due to the move-cj transformation (see Chapter 3). The penalty of code explosion in parallelizing compilers is not only the extra space needed but also the compilation time which increases significantly as number of nodes/operations increase. One may want to allow limiting the transformations to save space and compilation time. (Consequently, a trade-off exists in determining when it is beneficial to apply move-cj and when it is not.) In Table 8.9 we present the speed-up derived when move-cj is enabled vs. the speed-up obtained when it is disabled. In addition, we give the total number of nodes in the program and the total number of operations in each of these cases. Notice that disabling move-cj tends, sometimes, to make the loop pipelining algorithm converge slower. This happens because much more renaming is required and therefore many copy operations are introduced into the code. Thus, since more unwindings are needed, many more nodes and operations are created.

This makes the difference between the enabling and disabling of move-cj "milder" than one might expect.

The speed-up obtained without move-cj is about 32% less (WHM) as compared to the case when move-cj is enabled. However, on average, 2.5 times more nodes and 2.7 times more operations are involved when move-cj is allowed. For the latter two cases (number of nodes and number of operations) we compare averages, rather than WHM, since these numbers are compile-time measurements.

8.3.5 Disambiguation

Table 8.10 measures the effect of disambiguation on compaction with unlimited resources. The left column corresponds to *automatic* compaction while the disambiguator is in effect. The middle column presents compaction results without disambiguation. The rightmost column is for compaction with disambiguation when user assertions are allowed. In some sense these results may be regarded as "perfect disambiguation" results. Disambiguation is switched off by assuming that whenever *there is a chance* for a memory conflict—the conflict really happens (i.e. we take a pessimistic view). The results show the significant impact of disambiguation on the achievable speed-up. With unlimited resources the speed-up is 2.33 times higher than that obtained when disambiguation is disengaged. When user assertions are allowed, the speed-up increases by 47.6% relative to the speed-up achieved when the compiler uses the automatic disambiguation algorithm.

8.4 Discussion

In the two previous sections we have presented a set of various results derived with different hardware models and parallelization parameters. Naturally, the results cannot (and were not intended to) suggest the "ultimate" hardware model configuration—this depends on the specific application, but we can draw some guidelines and build some intuition based on the results presented. Any inferences from the results presented should, of course, be taken with a grain of salt, particularly since the benchmarks we used are not large enough so that other important issues involved with large benchmarks (e.g. cache effects) are not addressed here.

As mentioned, designing an architecture which implements the conditional execution model may be advantageous over the traditional model. As the results indicate, the conditional execution model yields a greater speed-up. On the other hand it involves significant complexity in its hardware implementation. A conservative conclusion from the results is that the importance of conditional execution increases with the number of functional units available and in the case that the code is branch-intensive.

The use of pipelined operations vs. uni-cycle operations is also a design trade-off as mentioned in Chapter 3. We showed that for limited resources the speed-up achieved is higher when multi-cycle operations are used. After implementation of a better loop pipelining algorithm, we believe that the speed-up for unlimited resources will grow as well. The benefits of pipelined operations become even greater if one keeps in mind that the actual performance of a system can be

expressed by the following proportion:

$$Performance = \alpha \frac{Speed\text{–}up}{Cycle\ time}$$

where α is a machine dependent factor. Since (generally) the cycle time may be reduced as the number of pipeline stages increases the enhancement of pipelining is amplified.

Another problem related to multi-cycle operations and not addressed so far is the compilation-time. Pipelining of operations actually means a "finer" clock. Since each clock (or control step) is represented by a single node, more nodes are involved in the parallelizing process (a three cycle operation spans over three nodes). As the number of nodes increases the compilation-time grows. For the set of benchmarks selected in this thesis the compilation-time for pipelined operations was 2-3 times longer than the corresponding compilation-time for uni-cycle operations.

The selection of the appropriate number of functional units is considered one of the most important design issues in any system that includes multiple functional units. On one hand we would like to have as many functional units as needed. On the other hand the number of functional units affects so many other crucial hardware aspects that one cannot afford to invest in many functional units. Among other issues the number of functional units affects: the width of the instruction word, the number of ports in the register-file, the number of multiplexers needed, the number of bits needed to tag the units, the number of internal wires, the number of pins for packaging etc.. Therefore, a careful selection of the appropriate number of functional units has to be done since we would like to have enough speed-up but be sure that the units are well utilized.

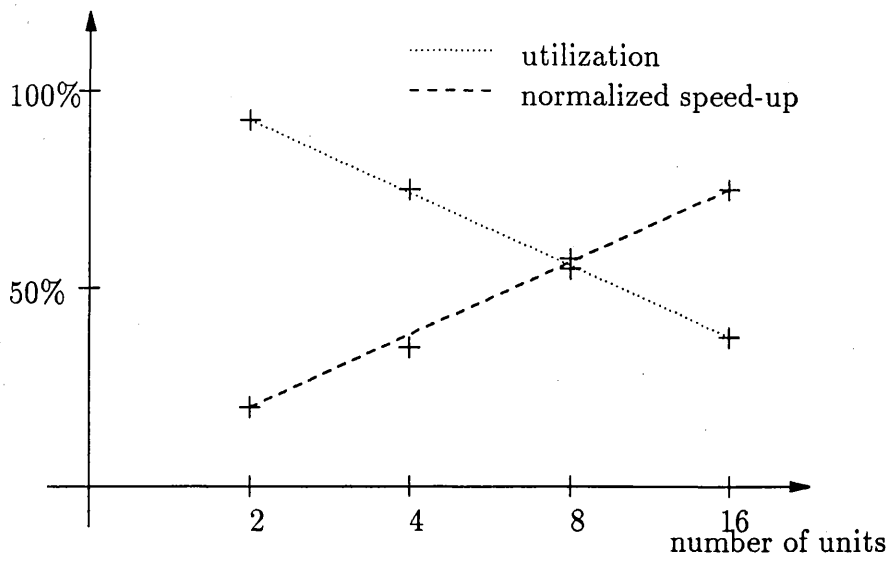


Figure 8.2: Utilization and Normalized Speed-Up for Uni-Cycle Operations.

Tables 8.4 and 8.5 and Figures 8.2 and 8.3 demonstrate two conflicting desires as we increase the number of functional units. First, it becomes harder to keep the units busy, and therefore the utilization drops (in a fair approximation it drops logarithmically). This may suggest an architecture with fewer functional units. Second, at the same time the normalized speed-up increases as the number of units grow. In this sense, the more units we have, the greater opportunity is for potential speed-up.

In order to take into consideration both desires, a good design-compromise (assuming that the cost-function weights equally utilization and normalized speed-up) would be to implement the system with the number of functional units that corresponds to the intersection of the two curves. While for the uni-cycle operations the curves intersect around 8 units, for the pipelined unit the intersection is around 7 units. Special care must be taken not to infer that the best selection for *every* pipelined system is 7-8 functional units. The results for pipelined units were obtained under the assumption that floating-point operations take three cycles to

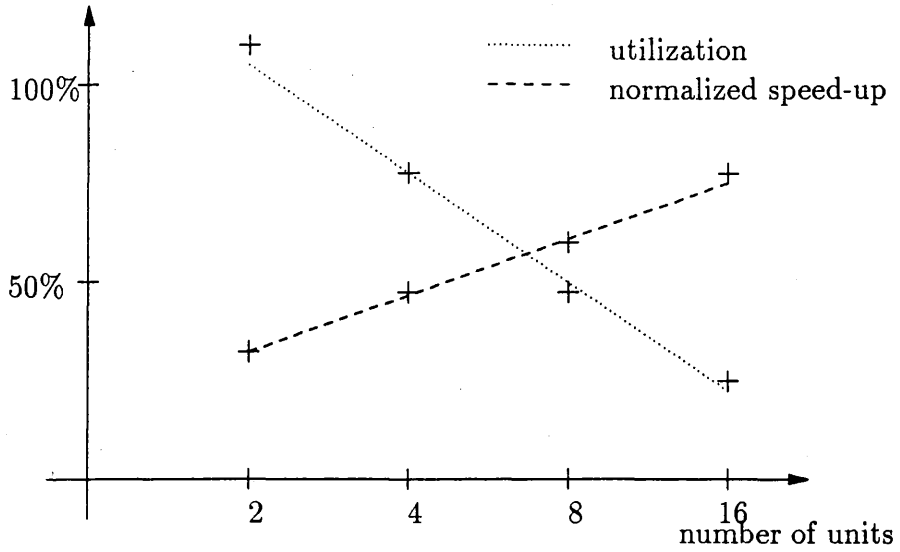


Figure 8.3: Utilization and Normalized Speed-Up for Pipelined Operations.

complete, memory loads take two cycles and all other operations are uni-cycle operations. Different combination of latencies may lead to a different selection of functional units. It should be further emphasized that this applies only to our compiler and for the *whole* set of benchmarks that we selected. For other applications (and even for specific applications out of our selection) these selections may change.

Regarding register allocation and renaming it has been shown that more registers are needed (approximately 3 times) for compaction than available in the input program. The price paid by our compiler for renaming is introduction of extra copy operations into the code. These copies, if not removed, may severely degrade the speed-up. Therefore, carrying out renaming without being able to eliminate these copies may become useless. We have shown the effect of renaming as well as the effect of copy elimination on the speed-up. From the results it is clear that both renaming and copy elimination are essential parts of any parallelizing

compiler. There is still one drawback in the way we perform these two optimizations: while renaming is done incrementally (or locally) *during* the parallelization process the induction variable removal and copy elimination is performed as a post pass *after* compaction (before resource constraints scheduling). Performing copy elimination "on the fly" would be too costly. Consequently, during parallelization copies increase compilation time as well as code space.

The importance of memory reference disambiguation for extraction of significant parallelism in scientific code was reported a few years ago [Ba79, Ni85a]. We have confirmed this observation by comparing the speed-ups obtained with an automatic disambiguation algorithm, without disambiguation and with assertions applied by the user. However, a powerful and precise disambiguation technique not only yields better speed-up by enabling compaction of memory loads and stores but also enables integration of other optimizations which are a direct by-product of the technique. Besides the results shown in Table 8.10 we have also shown in Table 8.7 and in Table 8.8 that further improvements may be achieved by application of redundant IV removal technique and load-after-store elimination. These two techniques rely completely on an accurate (and efficient) disambiguator. The latter even reduces the the memory transfer rate by eliminating load operations and consequently alleviates the memory traffic bottleneck that has posed a real problem in architecture design for years.

We have presented throughout the thesis some higher-level strategies like: loop pipelining, maxcomp, compaction without moving conditional jumps etc. that may be applied on top of the low-level, atomic PPS transformations. Since the interface between those high-level techniques and the low-level transformations is

well-defined it should be quite easy to apply other strategies to check other trade-offs and performance. Compaction without motion of conditional branches was implemented to control compilation-time as well as code space problems. However, since these problems are important enough, a full scale and careful study as well as experimentation with other techniques is still needed.

Table 8.1: Conditional Execution vs. Non Conditional Execution Model

No.	Benchmark	Unlimited resources		Limited resources	
		W/ cond. exec.	W/o cond. exec.	W/ cond. exec.	W/o cond. exec.
1	Livermore L1	13.63	4.64	2.00	1.75
2	Livermore L2	13.09	4.78	1.84	1.66
3	Livermore L3	8.94	3.00	1.80	1.80
4	Livermore L4	11.34	3.97	1.99	1.72
5	Livermore L5	5.48	3.66	2.20	1.57
6	Livermore L6	3.64	2.88	1.63	1.85
7	Livermore L7	27.65	7.54	1.82	1.41
8	Livermore L8	3.98	3.98	1.78	1.48
9	Livermore L9	33.65	9.42	1.78	1.40
10	Livermore L10	4.29	13.37	1.95	1.24
11	Livermore L11	8.93	3.00	2.25	1.80
12	Livermore L12	8.94	3.00	1.80	1.80
13	Livermore L13	2.97	2.97	1.95	1.72
14	Livermore L14	5.04	4.69	1.83	1.76
15	Livermore L15	5.00	3.02	1.69	1.78
16	Livermore L16	2.68	1.21	1.25	1.17
17	Livermore L17	5.75	1.78	1.29	1.44
18	Livermore L18	25.54	11.12	1.96	1.68
19	Livermore L19	5.78	4.97	2.05	1.94
20	Livermore L20	5.29	2.16	1.13	1.62
21	Livermore L21	3.85	2.69	1.73	1.80
22	Livermore L22	7.77	5.23	1.99	1.78
23	Livermore L23	7.20	7.20	1.89	1.38
24	Livermore L24	9.93	1.25	1.43	1.25
25	FFT	3.69	3.03	1.56	1.56
26	bubble	5.12	1.51	1.47	1.28
27	cos	8.93	1.69	1.60	1.56
28	minmax	12.78	1.30	1.30	1.18
29	quicksort	2.31	1.58	1.36	1.29
30	permute	3.15	2.88	1.58	1.57
31	line26	8.62	3.32	1.99	1.70
32	crale10	17.17	5.98	2.00	1.80
33	unriems50	54.11	13.71	1.91	2.12
34	nav7055	24.88	13.68	1.92	2.16
35	dblloop	6.33	1.48	1.73	1.48
	Average	10.78	4.62	1.76	1.61
	WHM	7.63	3.58	1.76	1.66

Table 8.2: Uni-Cycle vs. Pipelined Operations

No.	Benchmark	Unlimited resources		Limited resources	
		Uni-cycle ops	Pipelined ops	Uni-cycle ops	Pipelined ops
1	Livermore L1	13.63	8.82	2.00	3.36
2	Livermore L2	13.09	8.44	1.84	2.71
3	Livermore L3	8.94	4.99	1.80	2.99
4	Livermore L4	11.34	6.18	1.99	3.12
5	Livermore L5	5.48	3.00	2.20	2.57
6	Livermore L6	3.64	2.42	1.63	1.76
7	Livermore L7	27.65	19.07	1.82	3.73
8	Livermore L8	3.98	3.24	1.78	2.19
9	Livermore L9	33.65	21.68	1.78	2.23
10	Livermore L10	4.29	2.54	1.95	1.87
11	Livermore L11	8.93	4.32	2.25	3.24
12	Livermore L12	8.94	4.32	1.80	2.60
13	Livermore L13	2.97	2.58	1.95	2.12
14	Livermore L14	5.04	3.63	1.83	1.99
15	Livermore L15	5.00	3.81	1.69	2.43
16	Livermore L16	2.68	2.19	1.25	1.34
17	Livermore L17	5.75	2.38	1.29	1.59
18	Livermore L18	25.54	11.37	1.96	3.17
19	Livermore L19	5.78	2.94	2.05	2.11
20	Livermore L20	5.29	2.25	1.13	1.35
21	Livermore L21	3.85	2.52	1.73	1.67
22	Livermore L22	7.77	2.80	1.99	2.32
23	Livermore L23	7.20	4.26	1.89	2.32
24	Livermore L24	9.93	2.40	1.43	1.50
25	FFT	3.69	2.84	1.56	1.83
26	bubble	5.12	3.92	1.47	1.85
27	cos	8.93	3.93	1.60	1.53
28	minmax	12.78	4.98	1.30	1.25
29	quicksort	2.31	1.06	1.36	1.05
30	permute	3.15	1.20	1.58	1.11
31	line26	8.62	4.31	1.99	2.23
32	crale10	17.17	4.33	2.00	2.16
33	unriems50	54.11	31.67	1.91	3.66
34	nav7055	24.88	6.96	1.92	2.81
35	dbllloop	6.33	2.78	1.73	2.23
	Average	10.78	5.72	1.76	2.23
	WHM	7.63	4.18	1.76	2.26

Table 8.3: Number of Registers Used

No.	Benchmark	Regs used			Speed-up	
		Org.	Comp.	Ratio	min	max
1	Livermore L1	10	20	2.00	2.00	13.63
2	Livermore L2	13	23	1.77	4.89	13.09
3	Livermore L3	8	11	1.38	8.94	8.94
4	Livermore L4	11	14	1.27	5.89	11.34
5	Livermore L5	7	13	1.86	2.75	5.48
6	Livermore L6	11	17	1.55	3.64	3.64
7	Livermore L7	11	50	4.55	1.72	27.65
8	Livermore L8	17	37	2.18	2.83	3.98
9	Livermore L9	10	92	9.20	3.55	33.65
10	Livermore L10	7	54	7.71	3.58	4.29
11	Livermore L11	6	10	1.67	3.00	8.93
12	Livermore L12	6	9	1.50	3.00	8.94
13	Livermore L13	15	22	1.47	2.39	2.97
14	Livermore L14	16	114	7.13	2.87	5.04
15	Livermore L15	18	33	1.83	2.97	5.00
16	Livermore L16	21	108	5.14	2.27	2.68
17	Livermore L17	16	24	1.50	2.84	5.75
18	Livermore L18	26	84	3.23	6.50	25.54
19	Livermore L19	12	22	1.83	4.96	5.78
20	Livermore L20	15	56	3.73	2.55	5.29
21	Livermore L21	15	17	1.13	3.12	3.85
22	Livermore L22	11	26	2.36	2.29	7.77
23	Livermore L23	15	87	5.80	4.86	7.20
24	Livermore L24	10	20	2.00	3.33	9.93
25	FFT	19	26	1.37	2.79	3.69
26	bubble	10	20	2.00	2.98	5.12
27	cos	9	22	2.44	4.63	8.93
28	minmax	10	23	2.30	3.24	12.78
29	quicksort	15	23	1.53	1.96	2.31
30	permute	15	17	1.13	2.78	3.15
31	line26	9	14	1.56	4.83	8.62
32	crale10	13	23	1.77	8.88	17.17
33	unriems50	15	20	1.33	3.93	54.11
34	nav7055	18	169	9.39	4.22	24.88
35	dblloop	4	9	2.25	2.16	6.33
	Average	12.69	37.97	2.88	3.69	10.78
	WHM	11.66	20.59	1.76	3.51	7.63

Table 8.4: Speed-Up With Different Number of Functional Units—Uni-Cycle case

No.	Bench-mark	Number of functional units												
		∞	2			4			8			16		
		SU	SU	Util	NS	SU	Util	NS	SU	Util	NS	SU	Util	NS
1	Liv. L1	13.63	2.00	100	14.7	3.48	87.0	25.5	6.94	86.7	50.9	13.63	85.1	100
2	Liv. L2	13.09	1.85	92.5	14.1	3.64	91.0	27.8	7.07	88.4	54.0	7.26	45.4	55.4
3	Liv. L3	8.94	1.80	90.0	20.1	3.00	75.0	33.6	4.49	56.1	50.3	8.94	55.9	100
4	Liv. L4	11.34	1.99	99.5	17.5	3.96	99.0	34.9	5.91	73.9	52.1	11.34	70.9	100
5	Liv. L5	5.48	2.20	110	40.1	3.66	91.5	66.8	5.48	68.5	100	5.48	34.3	100
6	Liv. L6	3.64	1.79	89.5	49.1	2.89	72.3	79.4	3.64	45.5	100	3.64	22.8	100
7	Liv. L7	27.65	1.93	96.5	7.0	3.86	96.5	14.0	7.63	95.4	27.6	14.87	92.9	53.8
8	Liv. L8	3.98	1.86	93.0	46.7	2.98	74.5	74.9	3.54	44.3	88.9	3.86	24.1	97.0
9	Liv. L9	33.65	1.96	98.0	5.8	3.90	97.5	11.6	7.73	96.6	23.0	12.82	80.1	38.1
10	Liv. L10	4.29	1.95	97.5	45.4	2.86	71.5	66.7	3.58	44.8	83.4	4.29	26.8	100
11	Liv. L11	8.93	2.25	112	25.2	4.49	112	50.3	8.93	111	100	8.93	55.8	100
12	Liv. L12	8.94	1.80	90.0	20.1	3.00	75.0	33.6	4.49	56.1	50.2	8.94	55.8	100
13	Liv. L13	2.97	1.95	97.5	65.6	2.74	68.5	92.3	2.97	37.1	100	2.97	18.6	100
14	Liv. L14	5.04	1.88	94.0	37.3	3.19	79.6	63.3	4.45	55.6	88.3	4.95	30.9	98.2
15	Liv. L15	5.00	1.73	86.5	34.6	3.00	75.0	60.0	4.65	58.1	93.0	5.00	31.3	100
16	Liv. L16	2.68	1.28	64.0	47.8	1.79	44.8	66.8	2.28	28.5	85.0	2.48	15.5	92.5
17	Liv. L17	5.75	1.29	64.5	22.4	2.31	57.8	40.2	2.88	36.0	50.1	2.88	18.0	50.1
18	Liv. L18	25.54	1.96	98.0	7.7	3.74	93.5	14.6	6.93	86.6	27.1	11.5	71.6	44.8
19	Liv. L19	5.78	2.05	103	35.5	3.74	93.5	64.7	5.78	72.3	100	5.78	36.1	100
20	Liv. L20	5.29	1.23	61.5	23.3	2.26	56.5	42.7	3.86	48.3	73.0	5.29	33.1	100
21	Liv. L21	3.85	1.69	84.5	62.1	2.59	64.8	67.3	3.73	46.6	96.9	3.85	24.1	100
22	Liv. L22	7.77	1.99	100	25.6	3.94	98.5	50.7	7.77	97.1	100	7.77	48.6	100
23	Liv. L23	7.20	1.89	94.5	26.3	3.11	77.8	43.2	4.82	60.3	66.9	6.17	38.6	85.7
24	Liv. L24	9.93	1.67	83.5	16.8	2.00	50.0	20.1	2.50	31.3	25.2	4.99	31.2	50.3
25	FFT	3.69	1.73	86.5	46.8	2.67	66.8	72.4	3.29	41.1	89.1	3.69	23.1	100
26	bubble	5.12	1.80	80.0	35.1	2.32	58.0	45.3	2.53	38.3	54.4	4.82	30.1	94.1
27	cos	8.93	1.69	84.5	18.9	2.78	69.5	33.5	4.03	50.4	45.1	6.25	39.0	70.0
28	minmax	12.78	1.30	65.0	10.2	2.59	64.8	20.3	4.30	53.8	33.6	6.46	40.4	50.5
29	quicksort	2.31	1.37	68.5	59.3	1.93	48.3	83.5	2.19	27.4	94.8	2.31	14.4	100
30	permute	3.15	1.75	87.5	55.6	2.67	66.8	84.8	3.15	39.4	100	3.15	19.7	100
31	line26	8.62	1.99	100	23.1	3.36	84.0	39.0	4.91	61.4	57.0	8.62	53.9	100
32	crale10	17.17	2.00	100	11.6	3.60	90.0	21.0	5.99	74.9	34.9	8.92	55.8	52.0
33	unrms50	54.11	2.11	106	3.8	4.21	105	8.9	8.84	111	16.3	17.36	109	32.1
34	nav7055	24.88	2.18	109	8.8	4.22	106	17.0	8.24	103	33.1	14.24	89.0	57.2
35	dblloop	6.33	1.73	86.5	27.3	2.16	54.0	34.1	3.96	49.5	62.6	3.96	24.8	62.6
	Average	10.78	1.82	90.7	28.9	3.10	77.6	45.9	4.96	62.2	65.9	7.07	44.2	82.4
	WHM	7.63	1.86	93.2	21.2	3.15	78.7	36.4	4.74	59.3	55.8	6.28	39.3	73.4

Table 8.5: Speed-Up With Different Number of Functional Units—Pipelined case

No.	Bench-mark	Number of functional units												
		∞	2			4			8			16		
		SU	SU	Util	NS	SU	Util	NS	SU	Util	NS	SU	Util	NS
1	Liv. L1	8.82	3.36	168	38.1	6.66	167	75.5	8.82	110	100	8.82	55.1	100
2	Liv. L2	8.44	3.33	167	39.4	5.31	133	62.9	8.44	106	100	8.44	53.0	100
3	Liv. L3	4.99	2.99	74.8	59.9	4.99	125	100	4.99	62.4	100	4.99	31.2	100
4	Liv. L4	6.18	3.12	50.5	156	6.15	99.5	154	6.18	77.3	100	6.18	38.6	100
5	Liv. L5	3.00	2.57	129	85.7	3.00	75.0	100	3.00	37.5	100	3.00	18.8	100
6	Liv. L6	2.42	1.76	88.0	72.7	2.36	97.5	59.0	2.42	30.3	100	2.42	15.1	100
7	Liv. L7	19.07	4.36	218	22.9	7.72	40.5	193	11.4	142	59.7	16.21	101	85.0
8	Liv. L8	3.24	2.29	115	70.7	2.76	69.0	85.2	3.07	38.4	94.7	3.24	20.3	100
9	Liv. L9	21.68	2.58	129	11.9	4.37	109	20.1	7.78	97.3	35.9	13.66	85.4	63.0
10	Liv. L10	2.54	1.87	93.5	73.6	2.29	57.3	90.2	2.54	31.8	100	2.54	15.9	100
11	Liv. L11	4.32	3.24	162	75.0	4.32	108	100	4.32	54	100	4.32	27	100
12	Liv. L12	4.32	2.60	130	60.2	4.32	108	100	4.32	54	100	4.32	27	100
13	Liv. L13	2.58	2.12	106	82.1	2.52	63.0	97.7	2.58	32.3	100	2.58	16.1	100
14	Liv. L14	3.63	2.14	107	59.0	2.33	58.3	64.2	3.63	45.4	100	3.63	22.7	100
15	Liv. L15	3.81	2.54	127	66.7	3.68	92.0	96.6	3.81	47.6	100	3.81	23.8	100
16	Liv. L16	2.19	1.27	63.5	58.0	1.85	46.3	84.5	2.09	26.1	95.4	2.09	13.1	95.4
17	Liv. L17	2.38	1.73	86.5	72.7	2.01	50.3	84.5	2.24	28.0	94.1	2.38	14.9	100
18	Liv. L18	11.37	3.17	159	27.9	4.83	121	42.5	6.49	81.1	57.1	7.69	48.0	67.6
19	Liv. L19	2.94	2.11	106	71.8	2.94	73.5	100	2.94	36.8	100	2.94	18.4	100
20	Liv. L20	2.25	1.57	78.5	69.8	1.89	47.3	84.0	2.13	26.6	94.7	2.14	13.4	95.1
21	Liv. L21	2.52	1.67	83.5	66.2	2.25	56.3	89.3	2.52	31.5	100	2.52	15.8	100
22	Liv. L22	2.80	2.15	108	76.8	2.79	69.8	99.6	2.80	35.0	100	2.80	17.5	100
23	Liv. L23	4.26	2.32	116	54.5	3.19	79.8	74.9	4.00	50.0	93.9	4.26	26.6	100
24	Liv. L24	2.40	1.71	85.5	71.3	1.71	42.3	71.3	2.00	25.0	83.3	2.40	15.0	100
25	FFT	2.84	1.98	99.0	69.7	2.50	62.5	88.0	2.82	35.3	99.3	2.84	17.8	100
26	bubble	3.92	1.83	91.5	46.7	2.52	63.0	64.3	3.58	44.8	91.3	3.58	22.4	91.3
27	cos	3.93	1.53	76.5	38.9	2.54	63.5	64.6	3.67	45.9	93.4	3.93	24.6	100
28	minmax	4.98	1.74	87.0	34.9	2.50	62.5	50.2	2.96	37.0	59.4	3.20	20.0	64.3
29	quicksort	1.06	1.06	53.0	100	1.06	26.5	100	1.06	13.3	100	1.06	6.6	100
30	permute	1.20	1.11	55.5	92.5	1.18	29.5	98.3	1.20	15.0	100	1.20	7.5	100
31	line26	4.31	2.60	130	60.3	4.23	106	98.1	4.31	53.9	100	4.31	26.9	100
32	crale10	4.33	2.16	108	49.9	3.25	40.6	75.1	4.33	54.1	100	4.33	27.1	100
33	unrms50	31.67	3.83	192	12.1	6.12	153	19.3	8.21	103	25.9	14.85	92.8	46.9
34	nav7055	6.96	2.81	141	40.4	3.77	94.3	54.2	5.32	66.5	76.4	6.96	43.5	100
35	dblloop	2.78	2.23	112	80.2	2.78	69.5	100	2.78	34.8	100	2.78	17.4	100
	Average	5.72	2.33	111	62.0	3.39	78.8	84.0	4.14	51.7	90.1	4.75	29.8	94.5
	WHM	4.18	2.39	111	31.8	3.18	78.7	47.2	3.65	45.8	59.7	4.00	24.9	78.9

Table 8.6: Impact of Register Renaming on Speed-Up

No.	Benchmark	Unlimited resources		Limited resources	
		W/ renaming	W/o renaming	W/ renaming	W/o renaming
1	Livermore L1	13.63	2.00	2.00	1.75
2	Livermore L2	13.09	4.89	1.84	1.87
3	Livermore L3	8.94	8.94	1.80	1.80
4	Livermore L4	11.34	5.89	1.99	1.99
5	Livermore L5	5.48	2.75	2.20	1.83
6	Livermore L6	3.64	3.64	1.63	1.87
7	Livermore L7	27.65	1.72	1.82	1.63
8	Livermore L8	3.98	2.83	1.78	1.87
9	Livermore L9	33.65	3.55	1.78	1.51
10	Livermore L10	4.29	3.58	1.95	1.96
11	Livermore L11	8.93	3.00	2.25	1.80
12	Livermore L12	8.94	3.00	1.80	1.80
13	Livermore L13	2.97	2.39	1.95	1.81
14	Livermore L14	5.04	2.87	1.83	1.98
15	Livermore L15	5.00	2.97	1.69	1.94
16	Livermore L16	2.68	2.27	1.25	1.38
17	Livermore L17	5.75	2.84	1.29	1.76
18	Livermore L18	25.54	6.50	1.96	1.87
19	Livermore L19	5.78	4.96	2.05	2.05
20	Livermore L20	5.29	2.55	1.13	1.01
21	Livermore L21	3.85	3.12	1.73	1.78
22	Livermore L22	7.77	2.29	1.99	1.78
23	Livermore L23	7.20	4.86	1.89	1.91
24	Livermore L24	9.93	3.33	1.43	1.66
25	FFT	3.69	2.79	1.56	1.62
26	bubble	5.12	2.98	1.47	1.41
27	cos	8.93	4.63	1.60	1.62
28	minmax	12.78	3.24	1.30	1.44
29	quicksort	2.31	1.96	1.36	1.50
30	permute	3.15	2.78	1.58	1.54
31	line26	8.62	4.83	1.99	2.02
32	crale10	17.17	8.88	2.00	2.00
33	unriems50	54.11	3.93	1.91	2.04
34	nav7055	24.88	4.22	1.92	2.10
35	dblloop	6.33	2.16	1.73	1.53
	Average	10.78	3.69	1.76	1.76
	WHM	7.63	3.51	1.76	1.81

Table 8.7: Impact of IV Removal and Copy Elimination on Speed-Up

No.	Benchmark	W/ IV and copy elimination	W/o IV and copy elimination
1	Livermore L1	2.00	1.27
2	Livermore L2	1.84	1.47
3	Livermore L3	1.80	1.80
4	Livermore L4	1.99	1.99
5	Livermore L5	2.20	1.38
6	Livermore L6	1.63	1.61
7	Livermore L7	1.82	1.11
8	Livermore L8	1.78	1.48
9	Livermore L9	1.78	1.00
10	Livermore L10	1.95	1.20
11	Livermore L11	2.25	1.80
12	Livermore L12	1.80	1.50
13	Livermore L13	1.95	1.72
14	Livermore L14	1.83	1.17
15	Livermore L15	1.69	1.62
16	Livermore L16	1.25	1.11
17	Livermore L17	1.29	1.22
18	Livermore L18	1.96	1.56
19	Livermore L19	2.05	1.66
20	Livermore L20	1.13	0.66
21	Livermore L21	1.73	1.41
22	Livermore L22	1.99	1.16
23	Livermore L23	1.89	0.94
24	Livermore L24	1.43	1.43
25	FFT	1.56	1.42
26	bubble	1.47	1.44
27	cos	1.60	1.44
28	minmax	1.30	1.08
29	quicksort	1.36	1.17
30	permute	1.58	1.55
31	line26	1.99	1.46
32	crale10	2.00	1.39
33	unriems50	1.91	1.20
34	nav7055	1.92	0.92
35	dbllloop	1.73	1.73
	Average	1.76	1.37
	WHM	1.76	1.30

Table 8.8: Impact of Load-After-Store-Elimination on Speed-Up

No.	Benchmark	Unlimited resources		Limited resources	
		W/ optim.	W/o optim.	W/ optim.	W/o optim.
1	Livermore L1	13.63	13.63	2.00	2.00
2	Livermore L2	13.09	13.09	1.84	1.84
3	Livermore L3	8.94	8.94	1.80	1.80
4	Livermore L4	11.34	2.99	1.99	1.71
5	Livermore L5	5.48	2.75	2.20	1.83
6	Livermore L6	3.64	3.64	1.63	1.63
7	Livermore L7	27.65	27.65	1.82	1.82
8	Livermore L8	3.98	3.98	1.78	1.78
9	Livermore L9	33.65	33.65	1.78	1.78
10	Livermore L10	4.29	4.29	1.95	1.95
11	Livermore L11	8.93	3.00	2.25	1.80
12	Livermore L12	8.94	8.94	1.80	1.80
13	Livermore L13	2.97	2.56	1.95	1.72
14	Livermore L14	5.04	3.12	1.83	1.56
15	Livermore L15	5.00	5.00	1.69	1.69
16	Livermore L16	2.68	2.68	1.25	1.25
17	Livermore L17	5.75	5.75	1.29	1.29
18	Livermore L18	25.54	25.54	1.96	1.96
19	Livermore L19	5.78	2.70	2.05	1.59
20	Livermore L20	5.29	3.55	1.13	1.08
21	Livermore L21	3.85	3.85	1.73	1.66
22	Livermore L22	7.77	7.77	1.99	1.99
23	Livermore L23	7.20	4.42	1.89	1.82
24	Livermore L24	9.93	9.93	1.43	1.43
25	FFT	3.69	3.69	1.56	1.56
26	bubble	5.12	5.12	1.47	1.47
27	cos	8.93	8.93	1.60	1.60
28	minmax	12.78	12.78	1.30	1.18
29	quicksort	2.31	2.31	1.36	1.36
30	permute	3.15	3.15	1.58	1.58
31	line26	8.62	8.62	1.99	1.99
32	crale10	17.17	17.17	2.00	2.00
33	unriems50	54.11	7.95	1.91	1.89
34	nav7055	24.88	4.08	1.92	1.82
35	dbllloop	6.33	6.33	1.73	1.73
	Average	10.78	8.10	1.76	1.68
	WHM	7.63	5.24	1.76	1.70

Table 8.9: Compaction With Move-cj vs. Compaction Without Move-cj

No.	Benchmark	W/ move-cj			W/o move-cj		
		Speed-up	nodes	ops	Speed-up	nodes	ops
1	Livermore L1	13.63	30	141	13.63	13	88
2	Livermore L2	13.09	44	226	12.58	26	188
3	Livermore L3	8.94	9	33	8.93	10	41
4	Livermore L4	11.34	20	71	11.20	19	73
5	Livermore L5	5.48	11	35	5.48	11	39
6	Livermore L6	3.64	38	116	3.48	20	83
7	Livermore L7	27.65	76	759	27.45	17	277
8	Livermore L8	3.98	98	319	3.95	48	220
9	Livermore L9	33.65	92	890	33.65	18	396
10	Livermore L10	4.29	26	112	3.58	19	143
11	Livermore L11	8.93	10	29	8.92	11	36
12	Livermore L12	8.94	9	30	8.93	10	38
13	Livermore L13	2.97	52	148	2.97	32	100
14	Livermore L14	5.04	509	3570	4.21	57	602
15	Livermore L15	5.00	233	1644	3.53	121	707
16	Livermore L16	2.68	63	563	2.05	75	457
17	Livermore L17	5.75	27	345	2.56	20	85
18	Livermore L18	25.54	221	2510	6.05	57	1332
19	Livermore L19	5.78	37	158	5.70	26	124
20	Livermore L20	5.29	75	517	4.25	29	209
21	Livermore L21	3.85	18	68	3.40	20	74
22	Livermore L22	7.77	29	178	7.65	19	97
23	Livermore L23	7.20	111	475	7.10	24	284
24	Livermore L24	9.93	15	106	4.98	17	128
25	FFT	3.69	66	232	3.69	42	231
26	bubble	5.12	13	83	3.66	15	96
27	cos	8.93	20	690	3.57	19	339
28	minmax	12.78	11	256	2.41	34	560
29	quicksort	2.31	37	129	2.31	38	122
30	permute	3.15	42	122	3.07	40	119
31	line26	8.62	11	52	8.18	10	54
32	crale10	17.17	25	167	17.09	14	119
33	unriems50	54.11	78	1293	5.59	18	172
34	nav7055	24.88	273	5250	3.99	28	220
35	dblloop	6.33	13	83	3.80	8	47
	Average	10.78	69.8	611.4	7.24	28.1	225.7
	WHM	7.63	26.9	117.4	5.18	18.9	105.2

Table 8.10: Impact of Disambiguation on Speed-Up

No.	Benchmark	W/ disambiguation	W/o disambiguation	W/ assertions
1	Livermore L1	13.63	2.33	13.63
2	Livermore L2	13.09	3.01	13.09
3	Livermore L3	8.94	8.94	8.94
4	Livermore L4	11.34	2.99	11.34
5	Livermore L5	5.48	2.75	5.48
6	Livermore L6	3.64	2.18	10.05
7	Livermore L7	27.65	3.10	27.65
8	Livermore L8	3.98	3.39	20.23
9	Livermore L9	33.65	3.53	33.65
10	Livermore L10	4.29	2.69	37.22
11	Livermore L11	8.93	3.00	8.93
12	Livermore L12	8.94	3.00	8.94
13	Livermore L13	2.97	2.06	22.09
14	Livermore L14	5.04	2.01	5.04
15	Livermore L15	5.00	4.26	5.00
16	Livermore L16	2.68	2.68	2.68
17	Livermore L17	5.75	3.82	5.75
18	Livermore L18	25.54	3.43	25.54
19	Livermore L19	5.78	2.70	5.78
20	Livermore L20	5.29	3.55	5.29
21	Livermore L21	3.85	2.72	8.45
22	Livermore L22	7.77	2.30	7.77
23	Livermore L23	7.20	3.68	7.20
24	Livermore L24	9.93	9.93	9.93
25	FFT	3.69	2.73	12.72
26	bubble	5.12	4.20	5.12
27	cos	8.93	8.93	8.93
28	minmax	12.78	12.78	12.78
29	quicksort	2.31	2.31	2.31
30	permute	3.15	3.15	3.15
31	line26	8.62	2.55	8.62
32	crale10	17.17	2.27	17.17
33	unriems50	54.11	3.67	54.11
34	nav7055	24.88	2.25	24.88
35	dblloop	6.33	2.79	6.33
	Average	10.78	3.76	13.31
	WHM	7.63	3.28	11.26

Chapter 9

The compiler as a High Level Synthesis tool

High Level Synthesis (HLS) is the process of designing a structure that implements the behavioral description of a given problem. This process involves three phases [McPC88] which are *compilation* of the behavioral description into an internal representation, *scheduling*, and *allocation* (which includes binding). Our compiler includes similar phases—we compile from a high-level representation (C) into internal representation (three-address-code), then we schedule and then do some higher level hardware allocation. Thus, a natural questions is: "Can our compiler be considered as a HLS tool?" and if so, "What is the relationship between our compiler and other HLS systems?". We are focusing on these issues in this chapter. To illustrate how the compiler can serve as a HLS tool we give an example of our *gradual, two dimensional design approach.* Another aspect we address here is what extensions should and could be made in order to make this compiler a better HLS tool.

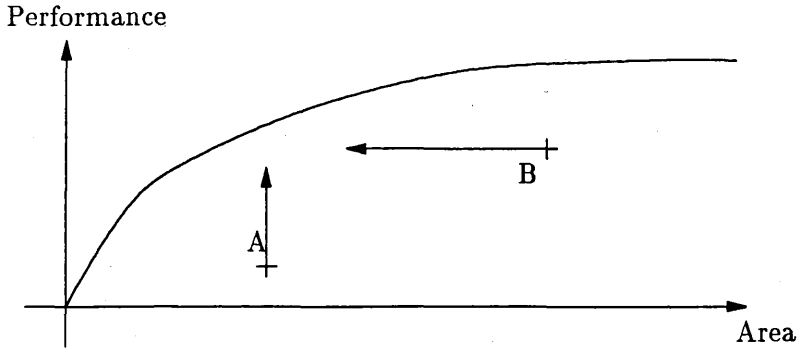


Figure 9.1: Performance vs. Chip Area.

9.1 Design Space Exploration

While trying to synthesize a hardware structure out of a behavioral description there is always a trade-off between the achievable performance and the chip's area. Intuitively, the more area we have, the better performance we can get, as illustrated in Figure 9.1. The graph represents the set of all optimal points in the design space for a given technology (i.e. it represents the best achievable performance for a specific architecture for a given technology). Obviously *not* all the points are feasible. The architecture design problem is to reach one of the curve's points starting from an arbitrary point in the design space.

Consider, for example, the traditional serial design process in which the hardware is designed first and then a compiler is written for this machine. This approach is adopted by most of the microprocessor designing companies. The approach is represented by the path A in Figure 9.1. When the compiler is written the silicon area is already determined and there is no way to add another adder or other functional units which may be required for enhancing the performance for a specific problem.

On the other hand, a typical HLS methodology is represented in the figure by path B. Here, during the scheduling phase of the synthesis, one always schedules with the same scheduler but searches for the design with least area. In both approaches one can explore the design space in one axis only.

The flexibility of our compiler, exhibited in allowing compaction with different constraints, software optimizations and hardware models, enables exploration in two axes and makes our compiler a good candidate for use in HLS. By being able to choose several hardware parameters (conditional execution or not, number of functional units, number of pipeline stages for each functional unit, total number of registers used), and increment them gradually we can "tune" the compiler to maximize performance and actually explore the design space in both axes.

9.2 Application-specific Design

Application-specific design is one of the important arguments in favor of HLS methodology. It is well known that a big penalty (in terms of excess hardware) is incurred during the design of general-purpose (GP) architectures for enabling their use for a large spectrum of applications, but when a very high performance architecture is needed these GP designs are unable to provide the performance required; Therefore, by fine-tuning the design and making it application-specific one can save some of the area spent for generality and devote it to other critical parts needed for the specific design.

The approach we take in our compiler coincides with the application-specific design approach. Since we have complete control on the compiler we can parameterize the design. We can check different options with different hardware parameters and choose the best one as the final schedule. As hardware technology approaches its physical limitations, fine-tuning becomes even more important.

9.3 Design Feedback

Another feature provided by our approach is that of *immediate design feedback*. To demonstrate this argument we'll refer to one of the major decisions during the architecture design process: the selection of the system-clock. Having a fast system-clock makes the whole design faster but will probably increase the number of pipeline stages needed for the execution of the slow operations. On the other hand, stretching the system clock will decrease, naturally, the number of stages but will increase the time wasted in the execution of fast operations. This is a "pure" design trade-off issue. With our approach, we may estimate the execution times of all combinations of clocks and pipeline stages and get the best choice.

9.4 Gradual, 2-Dimensional Design Example

The idea behind our gradual, 2D design approach is to gradually change the compiler's configuration and the hardware constraints to best fine-tune the performance for a specific problem. In other words, we propose a methodology to solve an optimization problem while the problem is N-dimensional (we may have control

on N different parameters) and the optimization function is maximal performance in minimal chip area. The approach is gradual since it allows incremental change of each of these N dimensions and it is 2D because we allow design in both axes of the design space mentioned in Section 9.1. The different problem dimensions may be: the number of functional units, the number of pipeline stages of each unit, the number of registers used and the model of execution (with or without conditional execution).

To simplify the example below, we refer in this section to the performance in terms of dynamic speed-up (as defined in Section 8.1) although in *real* system design the cycle time may vary for different pipeline depths and therefore should be accounted for. Furthermore, the only two dimension we allow to change (for this example) are the number of functional units and the number of stages in each unit.

Suppose the following program is given:

```
main()
{
  int i;
  float x[10],y[10];

  for (i = 1; i < 10; i++)
    x[i] = x[i] * (x[i - 1] + y[i]);
}
```

The intermediate code representing the loop body is:

(LABEL L1)

$2 := $3 − 44;

$f6 := M[$3 − 48];

$f8 := M[$3 − 84];

$f4 := $f6 + $f8;

$f6 := M[$2 + 0];

$f4 := $f4 ∗ $f6;

M[$2 + 0] := $f4;

$3 := $3 + 4;

$4 := $4 + 4;

$cc0 := $4 <= 36;

if $cc0 (LABEL L1)

The loop body has three memory loads, one floating-point addition and one floating-point multiplication. The design goal is to come up with the best speed-up *under the following assumptions:*

- Only floating-point operations and memory loads need to be pipelined. All other operations can be completed in one cycle.
- Floating-point addition and multiplication are executed by the same functional unit (and therefore have the same latency). Memory loads are carried out by another functional unit and all integer and conditional operations are executed on a third unit.
- Pipeline latencies for floating-point operations and for the load may vary from two stages to five stages.
- At most 3 functional units are allowed for this design.

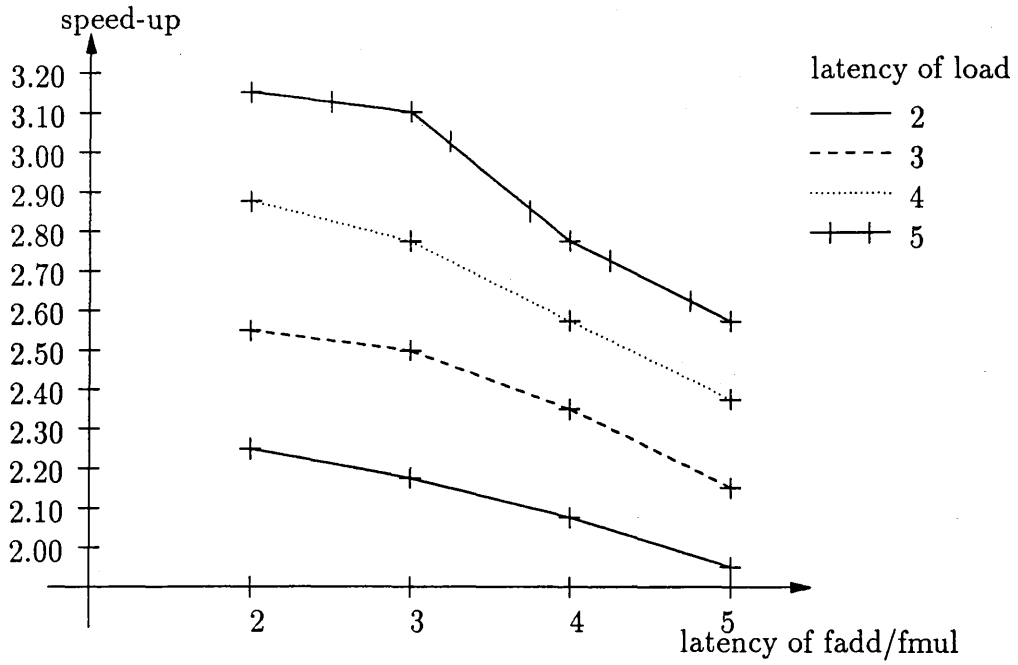


Figure 9.2: Speed-up With Different Latencies for the Gradual Design With 2 Functional Units.

To begin with, let us assume that the latencies of both loads and floating-point operations is two cycles and that only two functional units are available. Using the metric of speed-up, we then gradually change only one of the dimensions, the latency dimension, and measure the speed-up obtained for each combination of latencies. The results summarized in Figure 9.2 assume that only two functional units are available and that no more than one load and/or conditional jump can be executed in each cycle.

Figure 9.2 shows how the speed-up changes with a change in the pipeline depth (when the number of units is kept constant). Intuitively, we would expect the speed-up to decrease with an increase in operations' latency. However, while

an increase in the latency of the memory load unit *increases* the speed-up (keeping the latency of the addition/multiplication unit constant)—an increase in the addition/multiplication units results in a *decrease* in speed-up.

Similar gradual change on the other axis (number of functional units), while maintaining constant latency (of 2) for the addition/multiplication operations, results in the numbers presented in Table 9.1. Same gradual change can be carried out with latencies of 3–5 for the addition/multiplication operations.

Consequently, we get the best speed-up within the given pipeline limits and with 2 functional units when the load takes 5 cycles and the floating-point functional unit takes 2 cycles.

Table 9.1: Speed-Up With Two and Three Functional Units

Load latency	2	3	4	5
2 functional units	2.25	2.55	2.87	3.14
3 functional units	2.98	3.35	3.76	3.60
Improvement	32.5%	31.8%	31.0%	14.6%

From Table 9.1 one can see that, except for the case when load takes five cycles, the improvement in speed-up with 3 units is significant over the speed-up with 2 units. However, for this specific case the speed-up was maximal with 2 units. Therefore, the decision whether to use 3 functional units or stay with 2 is a pure optimization problem: if three functional units are feasible the best choice is with memory load unit that has 4 pipeline stages and floating-point unit that has 2 stages. On the other hand, if the design of 3 functional units is non-realistic, the best selection is that of load unit with 5 pipeline stages and addition/multiplication units with 2 stages.

In general, for each problem and for each set of feasible constraints one can always find the best implementation using the approach described here. Naturally, the selection of the best hardware choices is application-specific.

9.5 Future Extensions

9.5.1 Interactive Compaction

The parallelizing process discussed throughout the thesis is an automatic one based on low-level transformations directed by higher-level "guidance" rules (see chapter 3) and its goal is to generate a schedule that is better (i.e. more parallel) than the one that can be produced by human experts. However, due to the complexity of schedule-generation problems (which are NP-hard), a compiler must rely on heuristics, as explained in chapter 5, which sometimes fail to produce good (i.e. optimal or nearly optimal) schedules.[1] The automaticly-generated schedule may not be the best achievable implementation under the given constraints. Hence, allowing the user to interact with the compiler and direct the application of parallelizing transformations, while the compiler takes care of the tedious aspects of compaction, may yield even better schedules than those generated automatically.

Together with researchers at CMU we have integrated a graphical interface, through which the user can suggest what should be done in parallel, while the

[1]This is true even for sequential architectures, but is critical for architectures utilizing substantially both spatial and temporal parallelism, where the complexity of the schedule-generation problem is much greater. Furthermore, the penalty for a bad schedule in parallel architectures is proportional to the parallelism of the architecture, and thus could be very large.

compiler performs the actual changes using the PPS transformations. If a request cannot be satisfied, the compiler reports the problem causing the failure. The user may then help eliminate the problem by supplying guidance or information not explicit in the program. In such a way the user may add his insight and experience to change the schedule such that all constraints are met and the best, fine-tuned design is achieved.

A typical interactive session proceeds as follows. The user starts by requesting the automatic parallelization of the program (i.e. invoking the built-in heuristic aggregate transformations). If the resulting schedule and hardware are satisfactory, then no further work is needed and the interaction is complete. However, if the user is not satisfied, he may chose to interactively parallelize critical kernels. The user then refines the schedule by requesting specific modifications to the program (e.g. moving some operations from one point in the graph to another. When such a request is made, the compiler tries to instantiate it by a series of transformations. If the instantiation succeeds, the schedule is changed accordingly. Otherwise, the compiler reports the cause of the failure (e.g. a dependency violation). Alternatively, a transformation may sometimes fail due to the inability of the compiler to eliminate spurious dependencies (see Section 7.5). Two indirect references could appear to refer to the same memory location (thus causing a dependency) when in fact the references are distinct. The user may realize this, based on information available from the problem domain but not explicit in the program. In this case, the user may choose to ignore the conflict and direct the compiler to perform the transformation.

Most of this interactive interface has been implemented. We are now looking for ways to integrate the automatic compilation with the interactive session.

9.5.2 Consideration of Allocation Issues

As explained in chapter 5, when performing resource-constrained scheduling we use a priority function to decide which operation to defer in the case that a node contains more operations than allowed. This priority function takes into account the operation's *mobility* as well as its *precedence_number*. Currently, this priority does not consider any allocation aspect of the operation chosen. We believe that choosing a priority function that weights the allocation-aspects of the different options could result in a better overall design. The only change required is to rewrite the priority function (and of course an efficient algorithm to evaluate the "allocation-price").

Chapter 10

Summary, Discussion and Future Work

10.1 Thesis summary

The compiler presented in this thesis is targeted to map application-specific programs onto high-performance, parallel architectures instantiated by a VLIW template. When a high-performance *system* is designed, it is critical to have a perfect matching between its hardware and its compiler, otherwise considerable parallelism may be lost. However, one cannot design a general-purpose compiler to match the architecture for all possible applications. We propose in this thesis an approach to fine-tune a very powerful compiler to an affordable hardware-design by reconfiguring it, iteratively, so that the best matching is achieved. Naturally, the best matching is achieved when the speed-up is maximized for a particular, realizable design.

In order to extract substantial parallelism from both the hardware and the compiler, we use a clean, highly parallel architecture paradigm as well as advanced

compilation techniques. The architecture is VLIW-like in that it is totally synchronous, has multiple functional units which can simultaneously access a shared register-file and has a single program counter (single control thread). To further enhance parallelism, we modified the "pure" VLIW model so that instead of single-stage functional units we use *pipelined* functional units that provide extra (temporal) parallelism. The use of clean (interlock-free) and homogeneous (all units have the same structure) architectures does not only result in a better VLSI design but also considerably increases the compiler's ability to produce better code.

Our compiler's high-performance is achieved through the integration of several transformations, techniques and optimizations:

- The compiler uses a set of transformations called Pipelined Percolation Scheduling (PPS) that extract parallelism across basic blocks of the programs. This is crucial since, unfortunately, there is not enough parallelism within the basic block limits. Any attempt to schedule operations only within basic blocks boundaries is potentially limited. On the other hand, by using transformations that schedule operations from different basic blocks we are able to obtain significant speed-up. These transformations are especially powerful since they allow scheduling for pipelined (multi-cycle) architectures that use temporal parallelism.
- It is well-known that program execution exhibits the "90/10 locality rule" namely, that a program spends about 90% of its time executing 10% of the code. That 10% typically consists of the inner-loops in a program. Hence, being able to significantly parallelize loops is an important factor in overall parallelism extraction. Parallelism extraction in loops is carried out by an enhanced loop pipelining technique that is implemented in the compiler.

- We use a powerful disambiguation technique to determine whether two (indirect) memory references can be scheduled concurrently. This is especially important when the references are due to array indexing in inner-loops. Turning off this technique (by assuming that the two accesses *may* always refer to the same location) considerably limits the parallelism. Relying on the disambiguation information, we applied two other code optimizations that removed redundant induction variables from loops and eliminated redundant memory loads that consequently reduced the memory access traffic.
- The compiler implements a very efficient technique to rename (reallocate) registers that otherwise cause false dependencies that may cause major degradation in achievable parallelism. To get rid of copies generated during renaming we applied redundant IV removal and copy elimination techniques which are very important to achieve a good resource constrained schedule.
- Sometimes, an algorithmic change of the application (i.e. expressing the *same* problem in a different way) can enhance parallelism. A new local and incremental Tree Height Reduction algorithm is integrated in the compiler. The algorithm rearranges the application such that more operations are executed but their (total) execution time is reduced. THR can yield dramatic speed-up when enough resources are available.

The scheduling process for a specific architecture is done in two phases. First, the compiler extracts the maximal achievable parallelism as if all resources were available. Second, it performs resource constrained scheduling which maps the unconstrained schedule into the given architecture.

In order to be able to evaluate the complier's performance and validate its correctness we built a simulator which serves both as an emulator for the target

architecture (we run our compacted code on the simulator rather on the "real" hardware) and as debugging and verification tool. By comparing results derived by running the serial (uncompacted) program on various input combinations with the results derived by running the compacted program on the same input we were able to verify the compiler's compaction correctness.

10.2 Discussion

During the endless hours I spent writing and debugging the compiler (and even before...) a lot of ideas inspired my research. Some simply came as a result of previous work and others grew as by-products of this project. Some of these perspectives, which I see as this research's contribution, are discussed in this section.

- Since digital computers were introduced in the early 1960's there is everlasting dispute: "Who is to blame for the insufficient performance of the machine?". Throughout these years the machines have changed but the dispute remained. Hardware designers were happy to blame the software people who "were not able to write an appropriate compiler for this wonderful machine" while the software designers used to claim that "with such an architecture not much can be done." Who was right depends on whom you have asked, but the result was the same: machines were built with worse performance than could have been achieved *with the same technology.*

  From an objective point of view, both sides were right: there was no problem with the designers but there *was* a problem with the approach: separating

the machine into software and hardware was the key. What actually matters is the *system*'s performance and performance is the integration result of the *compiler and the hardware.* When a hardware engineer designs a chip one cannot expect him to foresee all compiler-related problems. On the other hand, when the hardware is predetermined the compiler's ability to produce good-quality code is also limited. What is needed is an interactive *concurrent* design in both domains so that there is immediate feedback from one to the other. This is especially true when very high performance is required and the compiler is allowed to take advantage of application-specific peculiarities.

The approach taken in this thesis is that of *gradual, two dimensional system design* where the compiler and the hardware are concurrently reconfigured to optimize the system's *overall* performance.

- When this research began there were only two published systems of code transformations that exploit parallelism across basic blocks: Trace Scheduling (TS) and Percolation Scheduling (PS). However, both TS and PS did not have a way of integrating multi-cycle operations within their transformational model. The assumption that each operation takes one cycle implies that one cannot take advantage of the architecture's temporal parallelism capability which is very important in high performance architectures.

  Since the current technology enables design of architectures with pipelined functional units, it seemed to me that using transformations that cannot handle pipelined operation may be a major drawback of our compiler. That inspired the introduction of the modified transformations called Pipelined Percolation Scheduling (PPS) which are used by our compiler.

- When scheduling for a specific architecture two possible approaches can be taken: either take the architecture's resource constraints into consideration from the beginning of the scheduling process or find the unlimited resources schedule and then map the resulting schedule onto the given architecture. The pro and con arguments are discussed in Chapter 5. Our compiler uses the second approach. The main conceptual advantage of this approach is that the unlimited resources schedule adds extra information to the resource constrained scheduling (RCS) task.

  We believe that by having the unlimited resources schedule first, we provide a good (lower) bound on how well we can expect our constrained schedule to be. In addition, this approach separates the heuristic part of the compiler from the non-heuristic part thereby enabling evaluation of different heuristics.

- It is well known that one of the promising ways to increase parallelism in parallel systems is to perform algorithmic changes of the application. The way an application is written determines its inherent achievable parallelism. In this context, while we did not want to cross over into auto-programming(!), we asked ourselves: "Is there any general way to rewrite code such that *the compiler* can compact it better while preserving the original code's semantics?". We found that further compaction can be derived by using a new local and incremental Tree Height Reduction algorithm. The THR algorithm which is implemented in the compiler can reduce (sometimes significantly) the total execution of the program at the expense of more computation. In other words, when enough resources are available, by changing data-dependencies between operations, we can achieve better parallelism than exhibited in the original program.

10.3 Future Work

As mentioned in Chapter 2 one of the approaches taken in designing the compiler was to build it in layers such that algorithms and techniques within each layer that are currently implemented may be easily replaced by other compatible alternatives. Following this idea, we intend to integrate into the compiler different higher-level strategies that control the low-level PPS transformations. In particular, we intend to integrate, at least, two other loop pipelining algorithms (OPT [AiNi88b] and PP [AiNi88c]) that, potentially, may yield better results than the current algorithm we use. In the context of loop pipelining we intend to extend the current algorithm to pipeline not only inner-loops but also outer-loops. While Loop Quantization [Ni88] allows unwinding of both inner-loops and outer-loops concurrently, thus yielding good pipelining effect, it may be difficult to implement and may result in expensive code duplication. We want to relax this method by pipelining first all inner-loops and then consider the (pipelined) inner-loop as one unit during the pipelining process of the outer-loops ([AiNi89]).

An extension of our disambiguation technique such that more sophisticated assertions can be added to the automatic algorithm is also one of our future tasks. Unfortunately, the compiler's analysis of a program cannot capture the user's knowledge of the general problem since this knowledge is usually not fully encoded in the program. The user may be able to make decisions based on information not available to the compiler. Consider, for example indirect references like A[B[i]] where array A's index depends on the input data B[i]. Since the compiler cannot expect any specific input data pattern, the automatic algorithm will *always* assume that there might be a conflict. However, based on non-encoded knowledge,

the user may realize that A[B[1]] and A[B[3]] can *never* refer to the same location. Since precise disambiguation is critical to substantial parallelism such an assertion mechanism can enhance the compiler's performance.

Code explosion and compilation-time are two issues that still need careful attention in our compiler. Currently, although compilation-time and code explosion are not restricted by any heuristics we obtain very reasonable time and space when running the benchmarks. However, we expect that both measures will grow considerably as the input grows. To solve this problem we intend to tackle these problems in two different approaches. First, we are going to implement several code optimization techniques on the input data which should reduce the input code size. Second, we want to implement different heuristics to limit the parallelization process and estimate how much parallelism is lost by these limitations.

Another future extension that we intend to perform is that of better compaction in the presence of procedure calls. Currently, all procedure calls are prevented from being compacted into nodes with other operations. In other words, calls form barriers for compaction. Better compaction involves more analysis, but it is feasible.

In the context of resource constraints scheduling we plan to add a *path* priority to operations' weighted priority function such that operations on most probable paths will have priority over those on paths less probable.

Bibliography

[Ab91] A. Abnous. "Architectural Design and Analysis of a VLIW Processor". MS thesis, University of California, Irvine , 1991.

[APBN91] A. Abnous, R. Potasman, N. Bagherzadeh and A. Nicolau. "A Percolation Based VLIW Architecture". *Proceedings of the 1991 International Conference on Parallel Processing*, St. Charles, IL, August 1991.

[Ai88] A. S. Aiken. "Compaction-Based Parallelization". PhD thesis, Cornell University, August 1988.

[AiNi88a] A. Aiken and A. Nicolau. "A Development Environment for Horizontal Microcode". *IEEE Transactions on Software Engineering*, Vol. 14, No. 5, May 1988.

[AiNi88b] A. Aiken and A. Nicolau. "Optimal Loop Parallelization". *Proceedings SIGPLAN 88, Conference on Programming Language Design and Implementation*, Atlanta, GA, June 22-24, 1988.

[AiNi88c] A. Aiken and A. Nicolau. "Perfect Pipelining: A new loop parallelization technique". In *Proceedings of the 1988 European Symposium on Programming.* Springer Verlag Lecture Notes in Computer Science no. 300, March 1988.

[AiNi89] A. Aiken and A. Nicolau. "Fine-Grain Parallelization and the Wavefront Method". *Proceedings of the 2nd Workshop on Programming Languages and Compilers for Parallel Computing*, Urbana, IL, August 1989.

[ASU86] A. Aho, R. Sethi, J.D. Ullman. "Compilers: Principles, Techniques and Tools". Addison-Wesley, Reading, MA, 1986.

[Ba78] J. Barth. "A Practical Interprocedural Data Flow Analysis Algorithm". *Communication of the ACM*, Vol. 21, No. 9, pp. 724-736, 1978.

[Ba79] U. Banerjee. "Speed-up of Ordinary Programs". Technical Report UIUCDS-R-79-989, University of Illinois, Department of Computer Science, 1979.

[BEH91] D. Bradlee, S. Eggers and R. Henry. "Integrating Register Allocation and Instruction Scheduling for RISCs". *Proceedings of 4th International Conference on ASPLOS*, Santa Clara, CA, April 1991.

[Br91] M. Breternitz Jr. "Architecture Synthesis of High-Performance Application-Specific Processors". PhD thesis, Carnegie Mellon University, April 1991.

[CCK87] D. Callahan, J. Cocke, K. Kennedy. "Estimating interlock and improving balance for pipelined architectures". *Proceedings of the 1987 International Conference on Parallel Processing*, pp. 295-304, 1987.

[CF87] R. Cytron and J. Ferrante. "What's in a name? or The value of renaming for parallelism detection and storage allocation". *Proceedings of the 1987 International Conference on Parallel Processing*, 1987.

[CFRWZ89] R. Cytron, J. Ferrante, B. K. Rosen, M.N. Wegman and F.K. Zadeck. "An efficient method of computing static single assignment form". *16th Annual ACM Symposium on Principles of Programming Languages*, Austin, TX, January 1989.

[CKV85] R. Cytron, D.J. Kuck and A.V. Veidenbaum. "The effect of restructuring compilers on program performance for high-speed computers". *Computer Physics Communications*, 37:37-48, 1985.

[Co88] R. P. Colwell, R. P. Nix, J. J.O'Donnell,D. B. Papworth,P. K. Rodman. "A VLIW architecture for a Trace Scheduling Compiler". *IEEE Transactions on Computers*, Vol. 37, No. 8,1988.

[Eb87] K.Ebcioglu. "A Compilation Technique for Software Pipelining of Loops with Conditional Jumps". *Proceedings of the 20th Annual Workshop on Microprogramming*, pp. 69-79, ACM Press, 1987.

[Eb88] K.Ebcioglu. "Some Design Ideas for a VLIW Architecture for Sequential-Natured Software". *Proceedings IFIP*, 1988.

[EbNa89] K.Ebcioglu, and T. Nakatani. "A New Compilation Technique for Parallelizing Loops with Unpredictable Branches on a VLIW Architecture". *Proceedings of the 2nd Workshop on Programming Languages and Compilers for Parallel Computing*, Urbana, IL, 1989.

[EbNi89] K.Ebcioglu, and A.Nicolau. "A *global* resource-constrained parallelization technique". *Proceedings of ACM SIGARCH ICS-89: International Conference on Supercomputing*, Crete, Greece June 2-9 1989.

[El86] J. R. Ellis. "Bulldog—A Compiler for VLIW Architectures". MIT Press, 1986.

[Fl66] M. J. Flynn. "Very high speed computing systems". *Proceedings of the IEEE*, 54(12) ,pp. 1901-1909, 1966.

[Fi81] J. A. Fisher. "Trace Scheduling: A technique for global microcode compaction". *IEEE Transactions on Computers*, No. 7,pp. 478-490, 1981.

[Fi83] J. A. Fisher. "Very Long Instruction Word architectures and the ELI-512". *Proceedings of the 10th Annual Internation Architecture Conference*, Stockholm, June 1983.

[GrLa86] T. Gross, M. S. Lam. "Compilation for high-performance systolic array". *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, July 1986.

[He77] M. S. Hecht. "Flow Analysis of Computer Programs". elsevier North-Holland, New York, 1977.

[He85] J. L. Hennessy. "VLSI RISC processors". *VLSI Systems Design*, VI:10, pp. 22-32, October 1985.

[JoMu81] N. D. Jones, S. S. Muchnick. "Program Flow Analysis: Theory and Applications". Prantice-Hall, Englewood Cliffs, NJ, 1981.

[Ku78] D. J. Kuck. "The Structure of Computers and Computations". Vol I, New York: Wiley, 1978.

[Ku87] M. Kumar. "Effect of Storage Allocation/Reclamation Methods on Parallelism and Storage Requirements". *Proceedings of the 14th Annual International Symposium on Computer Architecture*, Pittsburgh, PA, June 1987.

[Ku88] M. Kumar. "Measuring Parallelism in Computation-Intensive Scientific/Engineering Applications". *IEEE Trans. on Computers*, Vol 37, No. 9, pp. 1088-1098, September 1988.

[KuMuCh72] D. J. Kuck, Y. Muraoka and S. C. Chen. "On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup". *IEEE Trans. on Computers*, C-21, 12, December 1972.

[McPC88] M. C. McFarland, A. C. Parker and R. Camposano. "Tutorial on High-Level Synthesis". *Proceedings of the ACM IEEE 25th Design Automation Conference*, June 1988.

[NaEb90] T. Nakatani, K.Ebcioglu. "Using a lookahead window in a compaction based parallelizing compiler". *Proceedings of the 23rd Annual Workshop on Microprogramming*, Orlando, FA, November 1990.

[Ni84] A. Nicolau. "Percolation Scheduling: A parallel compilation technique". Technical Report 85-678, Cornell University, 1984.

[Ni85a] A. Nicolau. "Uniform Parallelism Exploitation in Ordinary Programs". *Proceedings of the 1985 International Conference on Parallel Processing*, 1985.

[Ni85b] A. Nicolau. "Parallelism, Memory Anti-aliasing and Correctness for Trace Scheduling Compilers". PhD thesis, Yale University, March 1985.

[Ni88] A. Nicolau. "Loop Quantization: A Generalized Loop Unwinding Technique." *Journal of Parallel and Distributed Computing*, 5, pp. 568-586, 1988.

[NiFi84] A. Nicolau, J. Fisher. "Measuring the parallelism available for VLIW architectures". *IEEE Trans. on Computers*, C-33, pp. 968-976, November 1984.

[NiPo90] A. Nicolau, R. Potasman. "Realistic Scheduling: Compaction for Pipelined Architectures". *Proceedings of the 23rd Annual Workshop on Microprogramming*, Orlando, FA, November 1990.

[NiPo91] A. Nicolau, R. Potasman. "Incremental Tree Height Reduction for High Level Synthesis". *Proceedings of the ACM IEEE 28th Design Automation Conference*, San Francisco, CA, June 1991.

[NPW91] A. Nicolau, R. Potasman and H. Wang. "Register allocation, renaming and their impact on parallelism". Technical Report, University of California, Irvine, April 1991.

[PaGa86] B. M. Pangrle and D. D. Gajski. "States Synthesis and Connectivity Binding for Microarchitecture compilation". *Proceedings of ICCAD* ,Nov. 1986, pp. 210-213.

[PaKn89] P. G. Paulin and J. P. Knight. "Force-Directed scheduling for the Behavioral Synthesis of ASIC's". *IEEE trans. on CAD*, Vol. 8, No. 6, June 1989.

[PaPa88] N. Park and A. C. Parker. "Sehwa: A Software Package for Synthesis of Pipelines from Behavioral Specifications". *IEEE Trans. on CAD*, Vol. 7, No. 3, March 1988.

[PLNG90] R. Potasman, J. Lis, A. Nicolau, D. Gajski. "Percolation Based Synthesis". *Proceedings of the ACM IEEE 27th Design Automation Conference*, Orlando, FA, June 1990.

[RaGl82] B. R Rau, C. D. Glaeser. "Efficient Code Generation for Horizontal Architectures: Compiler Techniques and Architectural Support". *Proceedings of the 9th Symposium on Computer Architecture*, April 1982.

[RiFo72] E. Riseman, C. Foster. "The inhibition of potential parallelism by conditional jumps". *IEEE Trans. on Computers*, Vol. 21, No. 12, December 1972.

[TjFl70] G. S. Tjaden and M. J. Flynn. "Detection and parallel execution of independent instructions". *IEEE Trans. on Computers*, Vol. 19, No. 10, October 1970.

[Tr87] H. Trickey. "Flamel: A High-Level Hardware Compiler". *IEEE Trans. on CAD*, Vol. 6, No. 2, March 1987.