

© Copyright by Daniel Michael Lavery, 1997

MODULO SCHEDULING FOR CONTROL-INTENSIVE
GENERAL-PURPOSE PROGRAMS

BY

DANIEL MICHAEL LAVERY

B.S., University of Illinois, 1986

M.S., University of Illinois, 1989

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1997

Urbana, Illinois

MODULO SCHEDULING FOR CONTROL-INTENSIVE GENERAL-PURPOSE PROGRAMS

Daniel Michael Lavery, Ph.D.

Department of Electrical and Computer Engineering

University of Illinois at Urbana-Champaign, 1997

Wen-mei W. Hwu, Advisor

It is increasingly necessary for the compiler to overlap successive loop iterations in order to find sufficient instruction-level parallelism to effectively utilize the resources of high-performance processors. Two competing methods have been developed for moving instructions across iteration boundaries: unrolling followed by global acyclic scheduling and software pipelining. This dissertation investigates modulo scheduling, a software pipelining technique. Much of the previous work on modulo scheduling has targeted the relatively well-behaved loops in numeric programs. This dissertation develops new techniques that allow modulo scheduling to be effectively applied to control-intensive non-numeric programs. These techniques overcome the restrictions imposed by problematic control flow and loop exits.

This dissertation also demonstrates that unrolling-based optimization prior to scheduling improves the performance of modulo scheduled loops and is, in fact, necessary to allow modulo scheduling to surpass the performance of acyclic scheduling for control-intensive general-purpose programs. Modulo scheduling has the following advantages over the acyclic scheduling approach for control-intensive general-purpose programs. First, modulo scheduling increases performance by maintaining the overlap of loop iterations throughout the execution of the loop. Second, modulo scheduling reduces register pressure by initiating iterations at a consistent rate that is sustainable for the given resources and dependence structure. Third, with the appropriate architectural support, modulo scheduling results in less code expansion because unrolling is required only for optimization, but not to amortize the loss of overlap across the back edge.

DEDICATION

To my wife, Tzuping, and my parents, Robert and Mary Lou.

ACKNOWLEDGMENTS

First, I would like to thank my advisor, Professor Wen-mei W. Hwu, for providing an excellent environment in which to learn and carry out research, for the opportunities for exposure to other industry and university researchers in the field, and for his insight and guidance during my studies. He always found the time to give me the advice I needed. My future career will benefit greatly from the lessons I have learned.

This research would not have been possible without the support of the members of the IMPACT research group, both past and present. The group members provided an enjoyable work atmosphere and considerable assistance including research discussions, practice talks, and the IMPACT compilation environment itself. Scott Mahlke, Pohua Chang, William Chen, and Roger Bringmann answered my many questions in the early days. Rick Hank, John Gyllenhaal, and Grant Haab filled that role more recently. Grant Haab, Teresa Johnson, and Ben-Chung Cheng provided help with Pcode. Dave Gallagher introduced me to sync arcs and was my role model for management of the IMPACT/x86 project. The IMPACT superblock formation and ILP optimizations used in this thesis are the work of Scott Mahlke, John Gyllenhaal, and David August. Thanks to Nancy Warter for her friendship, introducing me to modulo scheduling and, together with Noubar Partamian, collaborating on the support for backtracking and hyperblock code. Rick Hank wrote the HP PA-RISC code generator and register allocator used in this dissertation and provided needed breaks via his frequent sojourns into my office. John Gyllenhaal and Roger Bringmann developed the machine description and acyclic scheduling capabilities used in this work. Thanks to Sabrina Hwu for providing technical support and an

always smiling face in the group. Many thanks to Jim McCormick, Matt Merten, Derek Cho, Andrew Hsieh, Liang-Chuan Hsu, and Sabrina Hwu for their hard work on the IMPACT/x86 project. Special thanks to John Gyllenhaal for providing tool support and for volunteering to write the Pentium Mdes for that project. I would like to thank Rick Hank, David August, and John Gyllenhaal for their extensive workstation and PC administration efforts.

Thanks to Bob Rau, Mike Schlansker, Scott Mahlke, and the others at Hewlett-Packard Laboratories for giving me the opportunity to make presentations there and for their valuable discussions on my research work.

I would like to thank my parents, Robert and Mary Lou, for their love and encouragement throughout my life, especially during graduate school. They provided a firm foundation for me at home and in my early education, and have always offered assistance when I needed it. Thanks especially for the annual family vacations that provide refreshing breaks, sitting through more than one of my presentations, and their patience when I was very busy. I would also like to thank my sister, Ann, and brother-in-law, Jim, for their love and patience during my studies.

Finally, I would especially like to thank my wife, Tzuping, for her love, caring, and companionship. She has been a constant source of joy in my life and has helped me through the difficult times. While pursuing her own graduate studies, she often did much more than her share in maintaining our household so that I could meet the various deadlines in my research.

TABLE OF CONTENTS

CHAPTER		PAGE
1	INTRODUCTION	1
1.1	Contributions	3
1.2	Overview	4
2	COMPILATION ENVIRONMENT	6
2.1	Pcode	8
2.2	Lcode	9
3	OVERVIEW OF INSTRUCTION SCHEDULING FOR LOOPS	12
3.1	Loop Unrolling and Superblock Scheduling	13
3.2	Software Pipelining	22
3.2.1	Enhanced pipeline scheduling	24
3.2.2	Perfect pipelining	26
3.2.3	Petri net software pipelining	27
3.2.4	Modulo scheduling	28
4	THE IMPACT MODULO SCHEDULER	35
4.1	Loop Selection and Preparation	37
4.2	Dependence Graph Construction	39
4.3	Calculation of the MII	41
4.4	Modulo Scheduling Engine	42
4.5	Extended Modulo Variable Expansion	44
5	MODULO SCHEDULING OF LOOPS IN CONTROL-INTENSIVE NON-NUMERIC PROGRAMS	48
5.1	Case Study and Methods	50
5.1.1	Overcoming control dependence using speculative code motion	54
5.1.2	Overcoming anti-dependence using modulo variable expansion	57
5.1.3	Review of a code generation scheme for single exit loops	63
5.1.4	A code generation scheme for multiple exit loops	67
5.1.5	Insertion of moves for live-out values	72
5.2	Experimental Results	75
5.3	Summary	79
6	UNROLLING-BASED OPTIMIZATION FOR MODULO SCHEDULED LOOPS	80
6.1	Case Study	81
6.2	Related Work	84
6.3	Unrolling-Based Optimization	85
6.3.1	Loop unrolling	85

6.3.2	IMPACT unrolling-based optimizations	88
6.4	Experimental Results	92
6.5	Summary	94
7	COMPARISON OF MODULO SCHEDULING AND ACYCLIC SCHEDULING	96
8	CONCLUSION	107
8.1	Future Work	108
	REFERENCES	110
	VITA	115

LIST OF TABLES

Table	Page
5.1 Percentage of Dynamic Instructions in Single Basic Block and Superblock Loops. . .	54
5.2 Processor Characteristics for Modulo Scheduling Experiments.	76
6.1 Processor Characteristics for Unrolling Experiments.	92
7.1 Summary of Distribution Statistics.	105

LIST OF FIGURES

Figure	Page
2.1 The IMPACT Compiler.	7
3.1 An Example of Superblock Formation.	14
3.2 Example Optimized Vector-Matrix Product Loop.	18
3.3 Dependence Graph for Example Loop.	19
3.4 Effect of Loop Unrolling on the Example Loop.	20
3.5 Effect of Loop Unrolling on the Schedule for the Example Loop.	21
3.6 Conceptual View of Software Pipelining.	23
3.7 Derivation of the Recurrence-Constrained MII.	31
3.8 Overlapped Iterations and Basic Code Structure.	33
3.9 Modulo Scheduled Loop Structure with Kernel Unrolling.	33
4.1 The IMPACT Modulo Scheduler.	36
5.1 Source Code for Example Loop from <i>lex</i>	50
5.2 Superblock Formation for Example Loop.	51
5.3 Assembly Code for Superblock Loop.	53
5.4 Dependence Graph for Example Loop.	56
5.5 Modulo Resource Table after Modulo Scheduling.	57
5.6 Assembly Code with Renaming of r34.	58
5.7 Relaxation of Cross-Iteration Anti-Dependence.	60
5.8 Relaxation of Intra-Iteration Anti-Dependence.	61
5.9 Execution Record and Lifetimes for Two Iterations.	63
5.10 Unrolled Kernel for Superscalar Processor.	64
5.11 Relationship Between Loop Back Branch Placement and Speculative Initiation of Iterations.	65
5.12 Code Generation Scheme for Single Exit Loops.	66
5.13 Structure of a Single Iteration of a Multiple Exit Loop.	68
5.14 Code Generation Scheme for Multiple Exit Loops.	69
5.15 Epilogue Generation Algorithm.	71
5.16 Final Assembly Code for the Example Loop.	74
5.17 Speedup over Single-Issue Processor with and without Modulo Scheduling.	77
6.1 Example Vector-Matrix Product Loop.	82
6.2 Dependence Graph for Example Loop.	83
6.3 Example Loop after Unrolling Three Times.	87
6.4 Example Loop after Induction Variable Optimization.	88
6.5 Example Loop after Accumulator Expansion and Renaming.	90
6.6 Speedup over Single-Issue Processor with and without Unrolling.	94

7.1	Speedup over Single-Issue Processor for Acyclic and Modulo Scheduling.	98
7.2	Distribution of Per Loop Speedups over Acyclic Scheduling.	99
7.3	Improvement in Register Usage over Acyclic Scheduling.	100
7.4	Improvement in MaxLive over Acyclic Scheduling.	102
7.5	Code Size Compared to Acyclic Scheduling.	103
7.6	Improvement in Code Size with Kernel-Only Code.	104

CHAPTER 1

INTRODUCTION

Superscalar and VLIW processors achieve high performance by exploiting instruction-level parallelism (ILP). The compiler's responsibilities are to translate and optimize the source code program so that the highest performance is achieved on the target processor. One of the crucial tasks in this process is to expose sufficient ILP to keep the processor's functional units busy. This task of exposing parallelism requires aggressive low-level code optimization and scheduling.

It is well-known that, for most non-numeric programs, the ILP available within individual basic blocks is extremely limited [1], [2], [3]. An ILP compiler must be able to optimize and schedule instructions across basic block boundaries to find sufficient parallelism. The optimization and scheduling of loops are of great interest because most programs spend the majority of their execution time in loops. There is often insufficient ILP within a single loop iteration, even after the parallelism across basic blocks within a single iteration has been exploited. Thus, it is necessary for the compiler to optimize across iteration boundaries and for the scheduler to overlap successive iterations of a loop in order to find sufficient ILP.

Two classes of loop scheduling schemes have been developed that allow the overlap of iterations. The first approach is to unroll the loop body some number of times and then apply a global acyclic scheduling algorithm to the unrolled loop body [4], [5], [6]. This allows the scheduler to overlap the iterations in the unrolled loop body. An advantage of this technique is that multiple iterations of the loop are directly exposed to the compiler, enabling optimizations

which are not possible without unrolling. The disadvantage is that all overlap is lost when the loop-back branch is taken, leaving a long start-up penalty for each iteration of the unrolled body. The second approach, software pipelining [7], [8], [9], generates code that maintains the overlap of the original loop iterations throughout the execution of the loop. An advantage of this approach is that there may be less need to unroll the loop, offering the potential for smaller code size. This dissertation focuses on a class of software pipelining methods called *modulo scheduling* [10]. These methods have been shown to be very effective for exposing the ILP in loops to the processor.

There are a few vague myths surrounding modulo scheduling, and to a lesser extent, software pipelining in general. These myths are due partly to the fact that while much research has been done on software pipelining, many un-investigated avenues and open questions remain. The first myth is that modulo scheduling is applicable only to numeric programs. Most of the previous work on modulo scheduling has been aimed at numeric programs, which is reflected in production compilers. While several production compilers have targeted numeric programs for modulo scheduling, the control-intensive integer code has been dealt with using the global acyclic scheduling approach. This dissertation dispels this myth by developing techniques that allow modulo scheduling to be applied to control-intensive general purpose programs and by presenting experimental evidence that these programs benefit from modulo scheduling.

The second myth is that modulo scheduling and unrolling followed by acyclic scheduling are completely competing technologies, that is, one does not unroll the loop prior to modulo scheduling. The picture is actually not quite so black and white. The unrolling done to expose multiple iterations to the acyclic scheduler also creates new opportunities for optimization. These opportunities are missed if unrolling is not done prior to modulo scheduling. This

dissertation investigates the benefits of unrolling prior to modulo scheduling and performing optimizations that reduce resource usage and dependence height.

Both loop scheduling approaches have been implemented in the IMPACT compiler. They have been designed and tuned for good performance on control-intensive code. A quantitative analysis of both techniques is performed to better understand the advantages and disadvantages of each technique. The results show that modulo scheduling has the following advantages over the acyclic scheduling approach for control-intensive code. First, modulo scheduling increases performance by maintaining the overlap of loop iterations throughout the execution of the loop. Second, modulo scheduling reduces register pressure by initiating iterations at a consistent rate that is sustainable for the given resources and dependence structure. Third, with the appropriate architectural support, modulo scheduling results in less code expansion because unrolling is required only for optimization, but not to amortize the loss of overlap across the back edge.

1.1 Contributions

The three major contributions of this dissertation are discussed below.

- Techniques for modulo scheduling of loops in control-intensive programs are developed.

These techniques effectively overcome the restrictions imposed by problematic control flow and loop exits. A state-of-the-art modulo scheduler incorporating these techniques has been implemented in the IMPACT compiler and is described in detail. The benefit of modulo scheduling for control-intensive programs is quantitatively evaluated by compiling and executing a set of control-intensive benchmarks. These are the first reported

performance results for modulo scheduling on control-intensive non-numeric programs, and they demonstrate the applicability of modulo scheduling to this class of programs.

- Traditionally, loop unrolling is done prior to acyclic scheduling to allow the overlap of iterations. However, there are also new optimization opportunities created by unrolling. The motivations for performing loop unrolling prior to modulo scheduling are explored. Heuristics to control the amount of unrolling have been implemented. The benefit of unrolling prior to modulo scheduling and performing optimizations to reduce resource usage and dependence height is quantitatively evaluated.
- Global acyclic scheduling of an unrolled loop body and modulo scheduling are alternative technologies for instruction scheduling in loops. However, they have never been compared within the same compiler framework. The techniques for unrolling-based optimization and modulo scheduling of control-intensive loops explored and developed in this dissertation are shown to be necessary to allow modulo scheduling to compete with and surpass the performance of acyclic scheduling for control-intensive programs. The two scheduling technologies are quantitatively compared with respect to performance, register pressure, and code size. This is the first time that a direct quantitative comparison of the two has been made.

1.2 Overview

This dissertation is composed of eight chapters. Chapter 2 presents an overview of the organization and operation of the IMPACT compiler. All of the compiler techniques described in this dissertation have been implemented within the framework of the IMPACT compiler.

An overview of instruction scheduling for loops is presented in Chapter 3. Chapter 4 describes the modulo scheduler that has been implemented to support the work done for this dissertation. The techniques for modulo scheduling of control-intensive loops are built on top of this state-of-the-art implementation.

Chapter 5 describes the methods developed for control-intensive loops. A case study is presented to show how these methods enable modulo scheduling to be effectively applied to control-intensive loops. Performance results demonstrate the correctness of the methods and the applicability of modulo scheduling to control-intensive general-purpose programs.

An overview of the optimization of loops is presented in Chapter 6, which also describes the benefits of unrolling for modulo scheduled loops and presents a case study to illustrate the benefits of unrolling-based optimization. Performance results are presented to quantify the benefit.

Chapter 7 quantitatively compares modulo scheduling and acyclic scheduling within the same framework, with full support for unrolling and optimization applied in both cases. Complete benchmark results are shown to compare the performance on the loops which contribute significantly to the benchmark execution time. Results are also presented for the individual loops across all the benchmarks to demonstrate the performance across all the various loop types and provide more insight into the differences between the approaches. The conclusions and directions for future work are presented in Chapter 8.

CHAPTER 2

COMPILATION ENVIRONMENT

A state-of-the-art modulo scheduler incorporating the techniques proposed in this dissertation has been implemented within the IMPACT compiler. The IMPACT compiler is a retargetable, optimizing C compiler being developed at the University of Illinois. It is used to study compilation techniques, architecture features, and compiler/architecture tradeoffs for ILP processors. Figure 2.1 shows a block diagram of the IMPACT compiler. IMPACT is a C compiler, but can accept Fortran code that is translated using *f2c* [11].

Two different intermediate representations (IR) are used: a high-level IR called *Pcode* and a low-level IR called *Lcode*. Pcode is a hierarchical representation of the C source with source-level constructs such as loops and if-statements visible. Memory dependence analysis [12], [13], statement-level profiling, and function inline expansion [14] are performed on Pcode. Pcode is further discussed in Section 2.1. Lcode is a generalized register transfer language that is a superset of most RISC processor instruction sets. Most of the machine-independent optimizations are performed at the Lcode level. Section 2.2 describes these code transformations.

Six architectures are currently supported by the IMPACT compiler. Four of these are commercial: the HP PA-RISC, Sun SPARC, TI DSP, and Intel x86 [15], [16]. The other two supported architectures, IMPACT and HP PlayDoh [17], are experimental ILP architectures and provide a framework for compiler and architecture research. The IMPACT architecture specifies a parameterized processor that executes the Lcode instruction set. After machine specific

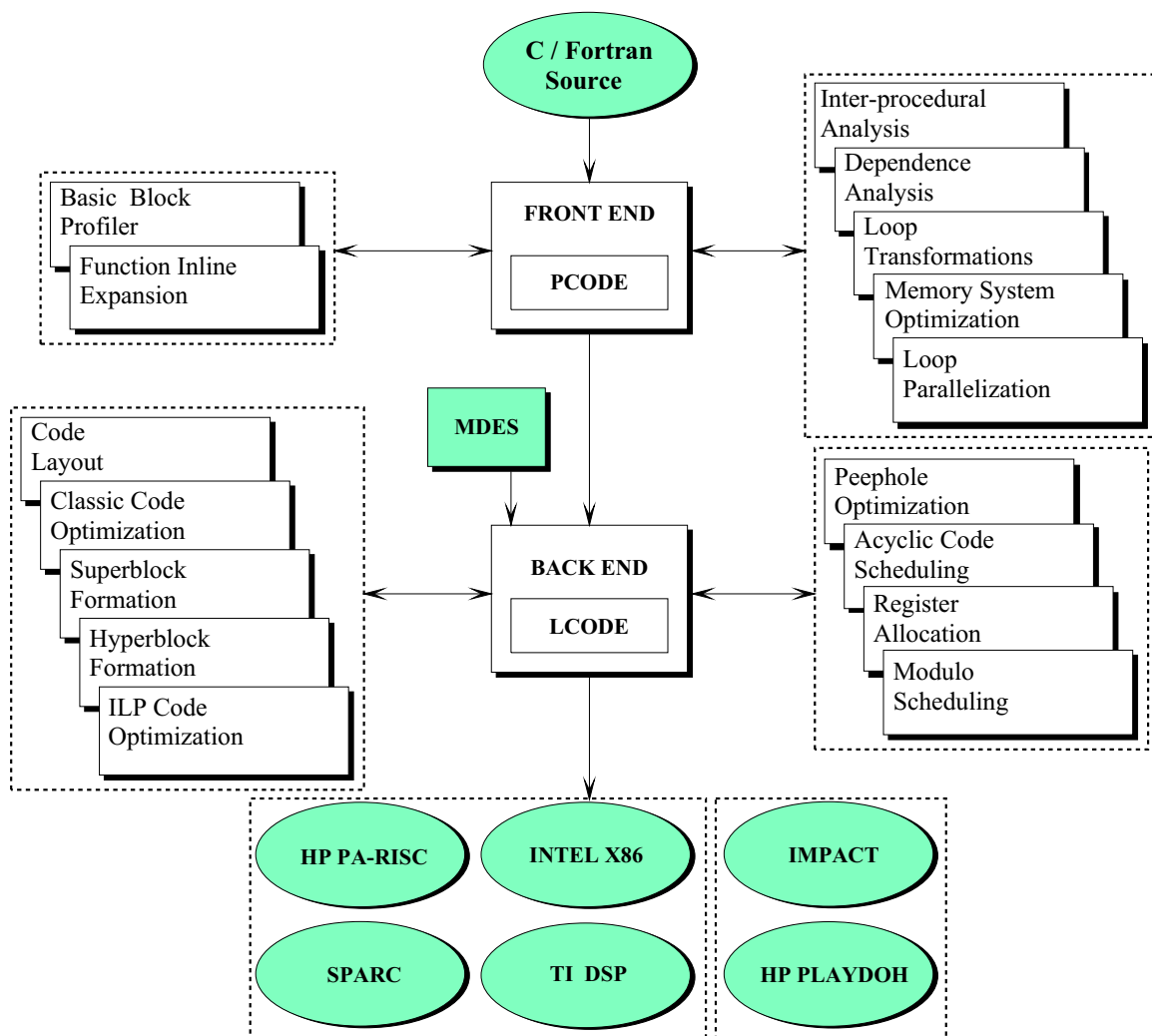


Figure 2.1 The IMPACT Compiler.

annotation of the Lcode, the IMPACT code generator can produce code for extended versions of the HP PA-RISC (IMPACT-HPPA) and the SPARC (IMPACT-SPARC) architectures. For this dissertation, all experiments are based upon the IMPACT-HPPA architecture.

The remainder of this chapter describes portions of the IMPACT compiler that are relevant to this dissertation. Sections 2.1 and 2.2 discuss the Pcode and Lcode levels of compilation, respectively.

2.1 Pcode

High-level analyses, profiling, inlining, and other optimizations that benefit from the availability of source-level information are performed at the Pcode level. In Pcode, the program is represented as an abstract syntax tree containing hierarchical statement and expression nodes. This hierarchical representation facilitates the manipulation of program structures such as loops and blocks of statements.

To support optimizations, Pcode performs data dependence analysis [12], which calculates the dependence relationships between each pair of accesses in the function. The Omega Test [18], developed by William Pugh at the University of Maryland, is used to compute dependences between array elements and generate distance and direction vectors. Inter-procedural analysis [13] determines the alias relationships between variables that are not apparent from analysis of each function individually, and removes the need to make conservative assumptions about aliasing due to pointers. It also analyzes the side effects of function calls, allowing better optimization and scheduling in the presence of function calls.

Once this memory dependence information is computed, it is propagated to the Lcode level in the form of explicit memory dependence arcs, called sync arcs [13]. The information is then used and maintained by the transformations at the Lcode level. Accurate information about memory dependences is crucial for modulo scheduling. Accurate information about cross-iteration dependences allows aggressive overlap of the loop iterations for both vectorizable and non-vectorizable types of loops. Accurate intra-iteration dependence information allows aggressive scheduling within an iteration, reducing the startup overhead, which is important for short trip count loops.

2.2 Lcode

Low-level code optimization and scheduling are applied to Lcode. There are two types of Lcode: machine-independent and machine-dependent. Although the internal and external representations of these two types of Lcode are identical, the machine-dependent version of the Lcode is sometimes referred to as *Mcode*. The difference between Mcode and Lcode is that for Mcode there is a one-to-one mapping between Mcode instructions and the target machine's assembly language. Lcode is converted to Mcode during the first phase of code generation by a process called *annotation*. For example, when generating code for the Intel x86 architecture, the Lcode is in three-operand format during machine-independent optimization, and is converted to two-operand format before the machine-dependent phases. Lcode instructions are also broken up for a variety of other reasons, including differences in addressing modes, branch instructions, and the ability to specify a literal operand. The machine-independent optimizations are performed prior to code generation on Lcode and machine-dependent optimizations are performed during code generation on Mcode.

The machine-independent optimizations consist of the classic local, global, and loop optimizations [19], [20], superblock formation, and ILP optimizations. The classic optimizations include constant propagation, forward copy propagation, backward copy propagation, common subexpression elimination, redundant load and store elimination, strength reduction, constant folding, constant combining, operation folding, operation cancellation, code reordering, dead code removal, jump optimization, unreachable code elimination, loop invariant code removal, loop global variable migration, loop induction variable strength reduction, loop induction variable elimination, and loop induction variable reassociation.

Following classic code optimization, superblock formation is performed. Superblocks [21] enlarge the scope of optimization and scheduling and remove the constraints associated with control flow paths that merge into the block. The superblock compilation structure is explained in detail in Chapter 3. Hyperblocks [22] can also be formed to allow the simultaneous optimization and scheduling of multiple paths and the removal of branches for architectures that support predicated execution. In this dissertation, only superblock formation was used for the modulo scheduling experiments.

After superblock formation, the profile-based classic optimizations are reaccomplished in the superblock framework to take advantage of the new opportunities created by removing the constraints associated with the side entrances [23]. Next, optimizations are performed that increase the available ILP of the intermediate code [24] including loop unrolling, register renaming, and height reduction optimizations.

After ILP optimization, machine-dependent code generation is performed for one of the five architectures shown in Figure 2.1. The machine-dependent optimizations include acyclic scheduling, cyclic scheduling, register allocation, and peephole optimization. The schedulers and register allocators are common modules shared by all code generators. Scheduling is performed via either acyclic superblock scheduling [25], [26] or modulo scheduling. Acyclic superblock scheduling is applied both before register allocation (prepass scheduling) and after (postpass scheduling) to generate an efficient schedule. Loops targeted for modulo scheduling are identified and marked prior to ILP optimization. These loops are modulo scheduled prior to register allocation and the remaining code is scheduled using the global acyclic scheduler. Both the cyclic and acyclic scheduling techniques are capable of exploiting architectural support for

control speculation to achieve more aggressive schedules. The modulo scheduler is further described in Chapter 4.

Both schedulers are driven by a machine description system (*Mdes*) [27]. The *Mdes* is used to obtain the latencies needed to construct the dependence graph, and the resources required by the instructions. The machine description for a processor is written at a high level and then compiled to a low level form for access by the compiler. The *Mdes* is optimized to support efficient scheduling [28].

The IMPACT global register allocator [29] is based on the graph-coloring algorithm described in [30]. When possible, the register allocator tries to minimize the number of registers used. Since register allocation is performed after scheduling, this does not affect performance, and it reduces the number of registers that need to be saved and restored at procedure call boundaries. At various points in the code generation process, a set of machine-dependent peephole optimizations is performed. These peephole optimizations are designed to remove inefficiencies introduced during Lcode to Mcode conversion, to take advantage of specialized opcodes available in the architecture, and to exploit new optimization opportunities after spill code has been added by the register allocator.

CHAPTER 3

OVERVIEW OF INSTRUCTION SCHEDULING FOR LOOPS

Loops are a potentially large source of instruction-level parallelism because separate iterations of the loop are often completely or mostly independent. Two classes of loop scheduling methods have been developed. One approach is to unroll the loop some number of times and then schedule the instructions from the unrolled iterations using a global acyclic scheduling technique. In acyclic scheduling, there is no knowledge of cross-iteration dependences and no knowledge that another iteration of the loop even exists. Thus the loop must be unrolled to explicitly expose multiple iterations to the scheduler. Even with the unrolling, the instructions at the end of the unrolled loop body are scheduled without regard to the instructions at the beginning of the loop body. Thus, at the beginning of each iteration of the unrolled loop, a delay may occur if the results of the instructions from the previous iteration are not yet available. Also at the beginning and end of the schedule for the unrolled loop body, there may be available instruction slots, but no available instructions to fill them.

Alternatively, to avoid this delay and to more fully utilize the processor resources, cyclic scheduling, otherwise known as software pipelining, can be applied to the loop. In software pipelining, knowledge of cross-iteration dependences and the cyclic nature of the scheduling region do exist. Software pipelining generates code that maintains the overlap of iterations throughout the execution of the loop. There are no gaps in the resource utilization, and dependences from one iteration to the next are taken into account. The following sections

describe the acyclic and cyclic scheduling approaches relevant to this dissertation. Section 3.1 describes superblock scheduling, the global acyclic scheduling technique implemented in the IMPACT compiler. This is combined with superblock loop unrolling to effectively overlap loop iterations in both numeric and control-intensive non-numeric code. Section 3.2 discusses several software pipelining techniques including modulo scheduling.

3.1 Loop Unrolling and Superblock Scheduling

This section describes the IMPACT global acyclic code scheduler, which is based on a variation of trace scheduling [4, 31] called *superblock scheduling* [26]. The idea is to select frequently executed paths through the code and optimize them, perhaps at the expense of the less frequently executed paths. In this approach, the optimization and scheduling region is a block of code called a *superblock* [6], [22]. A superblock is a block of instructions for which the flow of control may only enter from the top, but may leave at one or more exit points. It is formed by identifying sets of basic blocks which tend to execute in sequence (called a *trace*) [4]. These blocks are coalesced to form the superblock. Tail duplication is then performed to eliminate any side entrances into the superblock [23]. Effective superblock formation can be done using profile information [23] and/or static analysis of the structure and hazards in the program [32].

The formation of superblocks is illustrated in Figure 3.1, taken from [22]. Part (a) of the figure shows the control flow graph for an example loop. The nodes in the graph correspond to basic blocks and the arcs represent the possible control transfers. Each node is labeled with the execution count of the basic block and each arc is labeled with the execution count for that control transfer path. The execution counts are obtained by profiling. The most

As described in Chapter 2, classic (again) and ILP optimizations are applied after superblock formation. One of the ILP optimizations is *loop unrolling*. If a loop is unrolled N times, $N - 1$ copies of the loop body are appended to the original loop. The resulting loop contains multiple copies of the loop back branch. For all the copies except the last, the target and fall-through path are reversed so that the loop is exited when the branch is taken rather than when it falls through. If the iteration count is known on loop entry, it is possible to remove all the copies except the last by using a preconditioning (postconditioning) loop to execute the first (last) modulo N iterations. For simplicity, the loop examples used in this section assume that the branch copies are not removed. After loop unrolling, the new loop body contains N iterations of the loop, which can then be overlapped by the scheduler. Using the terminology of [33], the iterations of the new unrolled loop are defined as the *major iterations* and each of the N iterations of the original loop as the *minor iterations*. In the general case, the body of the loop is a superblock with multiple exits; so there are additional control transfers out of the loop beside those associated with the loop back branch. Loop unrolling and ILP optimization in the context of modulo scheduling are described in detail in Chapter 6.

Superblock scheduling is applied to each superblock independently. The first step is to build a dependence graph that represents all the data and control dependences between instructions within a superblock. There are three types of data dependences: flow, anti-, and output. The data dependences may exist between accesses to registers or memory. Control dependences enforce the ordering between a branch instruction and other instructions before and after a branch. There is a control dependence between a branch and a subsequent instruction **I** if the branch must execute before instruction **I**. This will be described in more detail below. There is also a control dependence between an instruction **I** and a subsequent branch if the branch must

execute after instruction **I**.¹ The arcs in the dependence graph are annotated with the *delay* of the dependence, which is the number of cycles that must separate the two instructions in the schedule. The delay is derived from the latencies and operand read/write times of the target processor.

The second step in superblock scheduling is to perform list scheduling using the dependence graph and the resource constraints of the processor. The general idea of the list scheduling algorithm is to pick, from the set of instructions that are *ready* to be scheduled, the best combination of instructions to issue in a cycle. An instruction is *ready* if all of its parents have been scheduled and the result produced by each parent is available (i.e., since the time that the parent was scheduled, enough cycles have passed to cover its latency). The best combination of instructions is determined by using heuristics to assign priorities to the ready nodes [25].

For control-intensive code, the presence of control dependences can severely restrict the ability of the scheduler to produce efficient code. As stated earlier, the compiler must be able to move instructions across branches (basic block boundaries) to find sufficient parallelism. The code motion may be either upward or downward across the branch. Moving instructions upward across branches is called *speculative code motion* because the instruction will be executed before the branch that determines whether or not it should be executed. There are three major restrictions on speculative code motion across a branch **B**:

- (1) The instruction must not write a virtual register that is in *live_out*(**B**).
- (2) The instruction must not cause an exception that terminates the program execution.
- (3) The instruction must not write to memory.

¹Note that this does not correspond to the traditional definition of control dependence. Rather, any constraint associated with code motion across branches that does not involve an explicit data dependence is enforced by inserting a control dependence arc.

The set $live_out(\mathbf{B})$ is defined as the set of virtual registers that may be used before they are defined when $\mathbf{B}$ is taken. The first restriction can usually be eliminated with sufficient compiler variable renaming support. The implications of this for modulo scheduling will be discussed in Chapter 5.

As an example of the second restriction, it is not safe to move a division or floating-point instruction above a branch because of the possibilities of a division by zero or a floating-point exception, respectively. It is also not safe to move a memory load instruction above a branch because of the possibility of a memory access violation. Page faults are not a problem, because they do not cause the execution to terminate. However, moving loads from below to above branches may increase the number of page faults.

To enable speculative code motion of loads and other instructions that can cause exceptions, either the architecture must contain support for speculative execution [34], [35] or the compiler must be able to prove via program analysis that the speculatively executed instruction will not cause an exception [25]. In this dissertation, it is assumed that the instruction set architecture contains silent (non-trapping) versions of the instructions that can cause exceptions [34].

In order to remove restriction 3, the architecture must contain support for speculative execution of stores. This support involves modification of the store buffer to delay the write to memory until it is confirmed that the store should execute and to nullify the store if the superblock is exited before the store should execute. In this dissertation, it is assumed that there is no architectural support for speculative execution of stores. It is also assumed that branches are not reordered. Speculative code motion of an instruction is enabled by removing the control dependence between the instruction and the preceding branches.

The following is a code example to show the effect of loop unrolling and acyclic superblock scheduling. Figure 3.2 shows the source and assembly code for the loop. The loop nest computes the product of a vector and a matrix. The loop nest has been optimized by interchanging the two loops. This avoids a reduction in the inner loop, making each loop iteration independent of the others. The assembly code shown assumes that classic loop optimizations such as induction variable elimination and global variable migration have been performed. Registers r2-r8 and f1-f5 are integer and floating-point registers, respectively.

Source Code

```

initialize C to 0.0
for (j=0; j<m; j++) {
    for (i=0; i<n; i++) {
        C[i] = C[i] + A[j] * B[j][i]
    }
}

```

Assembly Code for Inner Loop

Inst.	Assembly	Register Contents
1	L1: f1 = MEM(r8+r4)	f1 = C[i]
2	f5 = MEM(r2+r4)	f3 = A[j]
3	f6 = f3 * f5	f5 = B[j][i]
4	f1 = f1 + f6	r8 = &C[0]
5	MEM(r8+r4) = f1	r2 = &B[j][0]
6	r4 = r4 + 4	r4 = 4*i
7	bgt ((--r5) 0) L1	r5 = n

Figure 3.2 Example Optimized Vector-Matrix Product Loop.

Figure 3.3 shows the dependence graph for the inner loop. Each node is numbered with the ID (from Figure 3.2) of the instruction it represents. The branch nodes are shaded. The data and control dependences are shown with solid and dashed lines, respectively. Some of the transitive dependences are not shown.

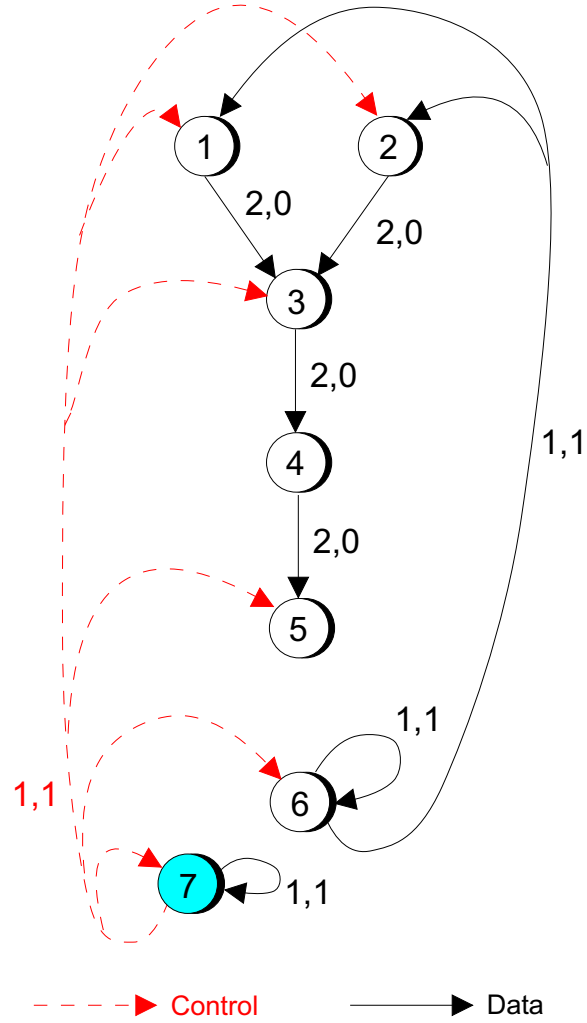


Figure 3.3 Dependence Graph for Example Loop.

Each arc is labeled with two numbers. The first is the minimum delay in cycles required between the starts of the two instructions. The second number is the distance, which is the number of iterations between the two dependent instructions. Arcs with a distance of zero are intra-iteration dependences and those with a distance greater than zero are cross-iteration dependences. The instruction set assumed is similar to HP's PA-RISC 1.1 but has no branch delay slots. Except for the branches, the delays shown are those of the PA7100. It is assumed that the instructions in the fall-through path of a branch can potentially be executed in the

same cycle as the branch and that instructions in the taken path are executed in the cycle following the branch.

Assuming a realistic eight-issue processor that contains three load/store units, two floating-point units, and one branch unit, a single iteration of this loop can be executed in seven cycles. The time to complete the iteration is limited by the dependences and not by the resource constraints.

Figure 3.4 shows the effect of loop unrolling on the assembly code for the loop. The loop has been unrolled twice. The registers for each iteration have been renamed to remove the anti-dependences that would prohibit the scheduler from overlapping the iterations.

Original Assembly Code		Unrolled Assembly Code	
Inst.	Assembly	Inst.	Assembly
1	L1: f1 = MEM(r8+r4)	1 ₁	L1: f11 = MEM(r8+r41)
2	f5 = MEM(r2+r4)	2 ₁	f51 = MEM(r2+r41)
3	f6 = f3 * f5	3 ₁	f61 = f3 * f51
4	f1 = f1 + f6	4 ₁	f11 = f11 + f61
5	MEM(r8+r4) = f1	5 ₁	MEM(r8+r41) = f11
6	r4 = r4 + 4	6 ₁	r42 = r41 + 4
7	bgt ((--r5) 0) L1	7 ₁	ble ((--r5) 0) exit
		1 ₂	f12 = MEM(r8+r42)
		2 ₂	f52 = MEM(r2+r42)
		3 ₂	f62 = f3 * f52
		4 ₂	f12 = f12 + f62
		5 ₂	MEM(r8+r42) = f12
		6 ₂	r41 = r42 + 4
		7 ₂	bgt ((--r5) 0) L1

Figure 3.4 Effect of Loop Unrolling on the Example Loop.

Figure 3.5(a) shows the effect of scheduling on the unrolled loop. The instructions from the two iterations are differentiated by the subscripts. The first iteration is also shaded. The two iterations are executed in eight cycles, almost twice the performance of the original loop. However, it is evident that the resources of the eight-issue processor are far from fully utilized. No more than three instructions are issued in any cycle.

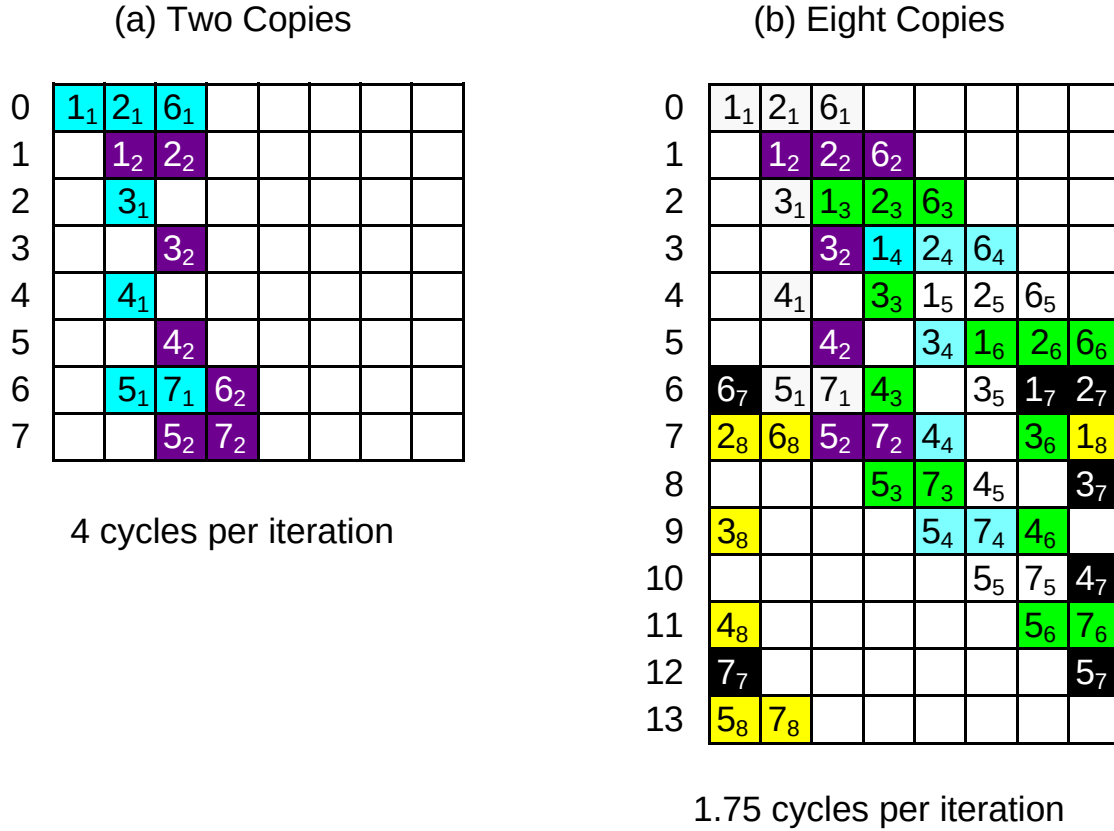


Figure 3.5 Effect of Loop Unrolling on the Schedule for the Example Loop.

Figure 3.5(b) shows the schedule if the loop is unrolled eight times instead of twice. The resources are now much more fully utilized. However, at the beginning and end of the schedule, there are still many empty slots. In particular, the peak utilization of seven instructions per cycle is achieved for only two cycles (cycles 6 and 7). Dependences and the limited number of

functional units force the scheduler to stagger the execution of the iterations in time, resulting in a ramp-up phase at the beginning of the schedule and a ramp-down phase at the end of every iteration of the new loop body. Performance can be improved at the cost of increased code expansion and scheduling effort. If the loops is unrolled a very large number of times, the performance asymptotically approaches the maximum obtainable.

3.2 Software Pipelining

Software pipelining [7, 36, 37, 38], is a loop scheduling scheme that allows motion of instructions from one iteration to another and maintains the overlap of loop iterations throughout the execution of the loop. A description of the various approaches to software pipelining is given in [39].

Figure 3.6 illustrates the concept of software pipelining. Imagine that the loop has been unrolled completely, exposing all of the iterations to the compiler. The scheduler initiates each iteration at some time interval after the previous one. After a sufficient number of iterations have been started (five in Figure 3.6), a steady state is reached. After the last iteration has been initiated, the steady state terminates and there is a phase in which the remaining portions of the iterations in progress are completed. The time interval between the start of successive iterations is called the *initiation interval* or II. Depending on the software pipelining algorithm used, the II can be a single fixed value, a periodic sequence of values, or a sequence of fixed values which depend upon the control flow within the loop body. The schedule for each iteration can be the same or can vary from iteration to iteration.

As shown in the figure, the steady-state portion of the execution can be re-rolled, producing a new loop called the *kernel*. The peak processor utilization occurs during this phase. The

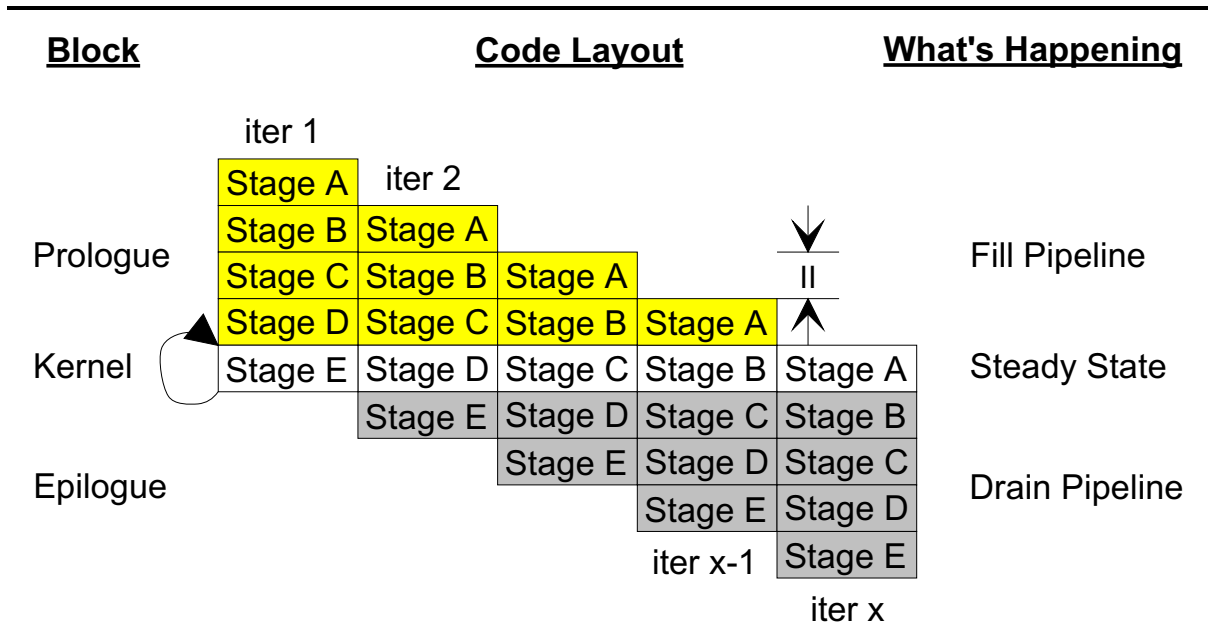


Figure 3.6 Conceptual View of Software Pipelining.

ramp-up phase before the kernel is called the *prologue* and the ramp-down phase is called the *epilogue*.

If the II is a fixed value and all the iterations have identical schedules, the schedule for each iteration of the loop can be divided into *stages* of II cycles each. As each new iteration is initiated, the previous iteration moves on to the next stage. Multiple iterations are simultaneously in execution, each in a different stage of the schedule, hence, the term software pipelining. In analogy to a hardware pipeline, the prologue code sequence corresponds to the filling of the pipeline. New iterations are initiated, but none have yet completed. The epilogue code corresponds to the draining of the pipeline. No new iterations are started, and the ones in progress complete. The *throughput* of the software pipeline is the rate at which iterations complete. The throughput is one iteration per II cycles. The *pipeline latency* is the number of cycles required to finish the first iteration. This is equal to the II multiplied by the number of

stages in the schedule for an iteration. Thus, like a hardware pipeline, after the initial latency, the iterations complete at a rate of one every II cycles.

The intuition behind the benefit of software pipelining is as follows. With loop unrolling and acyclic scheduling, the example loop from the last section achieved peak performance for only a fraction of the cycles. The more the loop is unrolled, the larger that fraction becomes, but at some point the code expansion becomes prohibitive. The ramp-up and ramp-down penalties occur for every iteration of the unrolled loop body. In contrast, software pipelining maintains the peak steady-state utilization continuously. The ramp-up and ramp-down phase occurs once per loop invocation. The peak performance is maintained without the need to unroll the loop many times.

Software pipelining is a powerful scheduling technique. It can be applied to loops with cross-iteration dependences and with arbitrary control flow including loops for which the number of iterations cannot be determined at the time the loop is invoked (e.g., while loops). It is most often used to schedule inner loops, but can also be hierarchically applied to outer loops [40], [41]. The remainder of this section discusses several well-known software pipelining techniques with their strengths and weaknesses. Modulo scheduling, which is the focus of this dissertation, is described in more depth than the others.

3.2.1 Enhanced pipeline scheduling

Ebcioğlu and Nakatani’s enhanced pipeline scheduling technique [8, 36, 41] divides the software pipelining problem into a sequence of acyclic scheduling problems. This technique has been implemented as part of the IBM VLIW project. At each step, a barrier is placed across one or more control flow edges of the loop to break the cyclic nature of the scheduling region

and define an acyclic graph for the next phase of scheduling. In the first step, the barrier is placed prior to the first instruction in the loop. Selected instructions are moved up in the graph to fill fence regions residing just after the barrier. A fence region can be thought of as a VLIW instruction to be filled with operations, or a cycle in which multiple superscalar instructions can be concurrently issued. As many instructions as possible are scheduled in the fence region. Then the barrier is moved so that it follows the fence regions. This defines new fence regions and a new acyclic graph, and scheduling begins again. If desirable, the instructions in the old fence regions can now be moved across the back edge and into the new fence region creating overlap of loop iterations. Instructions that are moved across the back edge are also inserted into the prologue. There are no epilogues, and no pipeline code generation phase is necessary after scheduling.

The primary advantage of this technique is in the handling of loops with multiple control flow paths. There can be more than one back edge and more than one fence region. This allows different cycles in the control flow graph to be scheduled with a different total delay, resulting in a variable II. This can be beneficial for loops where the number of cycles to execute one path to a back edge is much longer (due to either resource or dependence constraints) than for another. Another advantage is the potential to handle register constraints by performing the software pipelining after register allocation and using dynamic renaming (renaming while scheduling) when a register is available. Finally, the recent acyclic scheduling algorithms used with the technique replicate and unify instructions as they are moved past merge and fork points in the control flow graph. This can reduce the amount of speculative and redundant computation compared to methods that parallelize only selected paths.

One of the disadvantages of the technique has been that code is moved upward (perhaps speculatively) across branches, but not downward. This eliminates the epilogues, but results in more speculation than necessary in loops for which the condition for the loop back branch can be computed early (a good example is counted loops). Recently, techniques have been developed to allow upward movement of the branches [42]. Other unresolved issues with the method are the lack of a goal to determine when to stop scheduling and the complexity of the recomputation of required information for the new graph of each step. Jones and Allan empirically showed that modulo scheduling can achieve a smaller II than enhanced pipeline scheduling [43].

3.2.2 Perfect pipelining

Aiken and Nicolau’s perfect pipelining algorithm unrolls and compacts the loop body until a repeating pattern is formed [9, 37]. It performs software pipelining in two phases. In the first phase, global code motion is applied to move instructions up as early as possible in the loop body. The second phase consists of iteratively unrolling the compacted iterations, scheduling instructions in a greedy manner, and examining the execution history for a repeating pattern. This repeating pattern becomes the kernel.

The advantage of this technique is that there are no arbitrary barriers or constraints on the scheduling. There are no fence regions. The II is not constrained to have any particular properties such as being an integer or representing the rate at which an a-priori fixed group of iterations executes. Each copy of the original loop iteration is not constrained to have the same schedule. In summary, both the value of the II and the set of instructions comprising the repeating pattern are determined during scheduling. In contrast, for modulo scheduling, both

are determined prior to scheduling. In enhanced pipeline scheduling, the II is not fixed a priori, but the set of instructions is.

The disadvantage of this technique is the computational complexity involved in recognizing the repeating pattern (kernel recognition). First, the amount of information that must be checked to verify a repeating pattern is very large. It includes identifying two cycles of the execution record in which the exact same instructions are issued, the exact same resources are committed into the future, and the exact same operands will be produced at the same time in the future. This state of the scheduling process must be checked after each cycle is scheduled. Without a target II, a large number of iterations may be scheduled before a repeating pattern emerges, further increasing the computational complexity.

3.2.3 Petri net software pipelining

The Petri net software pipelining method [38, 44] attempts to solve the problem of kernel recognition. The state of the scheduling process is represented by the combination of a Petri net and a behavior table with attached resource state. A Petri net is a graph consisting of two types of nodes: places and transitions. The transitions represent instructions, and a combination of arcs and places between transitions represent dependences. Associated with each transition is a set of input (output) places, one for each incoming (outgoing) dependence. Each place has an incoming arc from the instruction at the source of the dependence and an outgoing arc to the instruction at the sink of the dependence. When a place contains a token, the dependence between the two instructions is satisfied. A transition fires (instruction is issued) when all of its input places contain tokens. Thus the Petri net represents both the dependence graph and the current set of ready instructions. If the Petri net is cyclic, a series of firings will eventually

take the Petri net to a state through which it has already passed. When the Petri net reaches a previously encountered state with an identical resource state, the repeating pattern has been found.

The advantage of this approach is that it defines a systematic method for kernel recognition. The Petri net can also be used to speed the formation of a pattern by adding arcs and places to the graph to make it strongly connected. However, there are still several unresolved issues. It is not clear how the previous states of the Petri net and the resources are stored, how the current state is compared with the previous states, and what the computational complexity of the process is. Also, there is no notion of delay for an arc in the Petri net. Thus instructions with latencies longer than one are represented by adding dummy transitions to the graph. This increases the size of the representation and the number of nodes that must be scheduled. Making the graph strongly connected increases the number of recurrence circuits and the chance that the Petri net fails to fire at the optimal rate in the presence of resource conflicts.

3.2.4 Modulo scheduling

Modulo scheduling [45], [10] was originally proposed by Rau and Glaeser [7]. It overcomes many of the complexities and practical problems associated with the approaches described in the earlier sections by taking a less ad hoc approach to forming a kernel. Modulo scheduling simplifies the generation of overlapped schedules by initiating iterations at a constant rate and by requiring all iterations of the loop to have identical schedules. A fixed II is determined prior to scheduling based on the loop requirements and the machine constraints. Then an attempt is made to engineer a valid schedule at that II . One of the advantages of modulo scheduling is that the II gives the scheduler a goal and offers the potential for lower register pressure because

the iterations are started at a consistent rate that is sustainable for the given resources and dependence structure.

Modulo scheduling is the most well-developed cyclic scheduling method. It has been implemented in production compilers at Cydrome [46], HP [47], and SGI [48]. It is capable of pipelining loops for superscalar and VLIW processors, with complex resource constraints, recurrences, and control flow. It can pipeline both loop-counter-based loops and, using the techniques presented in this dissertation, non-loop-counter-based loops and loops with multiple exits. For these reasons, modulo scheduling was the approach taken in the IMPACT compiler.

In modulo scheduling, it is necessary to choose an initial candidate $\mathbf{II}$ before scheduling the instructions. Two lower bounds on $\mathbf{II}$ have been developed in the modulo scheduling theory [10]. The maximum of the two is the minimum initiation interval ($\mathbf{MII}$). Scheduling at an $\mathbf{II}$ below the $\mathbf{MII}$ can be attempted, but will almost surely fail² due to lack of resources or failure to meet dependence constraints. In this case, the $\mathbf{II}$ is increased and scheduling is attempted again. Choosing the initial $\mathbf{II}$ equal to the $\mathbf{MII}$ saves scheduling effort. In the presence of recurrences or complex resource constraints, it is possible for the modulo scheduling algorithm to fail to schedule the loop even at the $\mathbf{MII}$.

The first lower bound is derived from the resource requirements of the loop. The periodic initiation of iterations with the same schedule results in a rule known as the *modulo constraint*. If an instruction in an iteration uses a resource at time $\mathbf{x}$, then the same resource is also used at time $\mathbf{x} + n * \mathbf{II}$ by subsequent iterations. Therefore, the schedule for a single iteration may not use any resource more than once at the same time modulo $\mathbf{II}$. This constraint is enforced using a Modulo Resource Table (MRT) [40]. The MRT contains $\mathbf{II}$ rows and one column for each

²*Almost* because in some cases one of the lower bounds is approximate rather than exact.

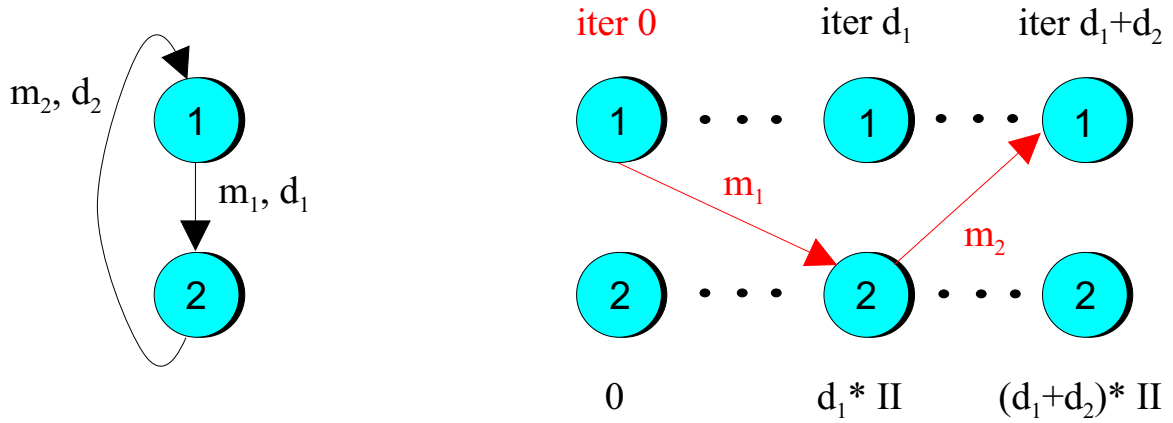
resource. Using this table, modulo scheduling generates a schedule for a single iteration of the loop. If that schedule can be wrapped around the MRT without any resource conflicts, then successive iterations of the loop can be started every Π cycles without resource conflicts.

The MRT must contain a sufficient amount of each resource to support the resource requirements of the loop. From the resource point-of-view, the throughput of a software pipeline is maximized when one of the processor resources is fully utilized. Thus the resource that is the most heavily used by the loop body determines the resource-constrained lower bound on the Π (ResMII). ResMII is equal to the number of cycles that this resource is used. One possible algorithm for computing the ResMII is given in [10]. This algorithm performs a bin packing of the resources required by all the instructions in the loop. In the presence of complex resource usage and alternative reservation tables for an instruction, the problem is NP complete. Thus, the algorithm computes an approximate lower bound. The details of how this algorithm is implemented in the IMPACT compiler are given in Chapter 4.

An alternative method to compute the ResMII (and the method described in most of the early modulo scheduling papers) is to use the following equation:

$$ResMII = \max_i \left\lceil \frac{N_i}{R_i} \right\rceil,$$

where N_i is the number of cycles that resource i is used by a single iteration and R_i is the number of copies of the resource. This method for the computation of the ResMII can only be used if the functional units can be partitioned into equivalence classes and if each instruction can be executed by only one equivalence class [10]. Unfortunately, most real processors do not have this structure.



$$\text{Delay}(c) = m_1 + m_2$$

$$\text{Distance}(c) = d_1 + d_2$$

$$\text{Delay}(c) \leq \text{Distance}(c) * II$$

Figure 3.7 Derivation of the Recurrence-Constrained MII.

The second constraint on the II comes from cycles in the dependence graph. Figure 3.7 shows a dependence cycle or recurrence consisting of two instructions. To the right, several copies of the two instructions from different iterations of the loop are shown, along with the dependence chain starting at instruction 1 of iteration 0. The first dependence has delay m_1 and distance d_1 , so it goes from iteration 0 to iteration d_1 . The second dependence has distance d_2 , so it goes from iteration d_1 to iteration d_1+d_2 . Assuming iteration 0 starts at time 0, iteration d_1+d_2 starts at time $(d_1+d_2)*II$. Define $\text{Delay}(c)$ to be the sum of the delays around the cycle, that is, m_1+m_2 , and define $\text{Distance}(c)$ to be the sum of the distances around the cycle, that is, d_1+d_2 . Then the difference in start time between iteration 0 and iteration d_1+d_2 becomes $\text{Distance}(c)*II$. This difference in time must be at least as large as m_1+m_2 , or $\text{Delay}(c)$, leading to the final equation in the figure and the equation below.

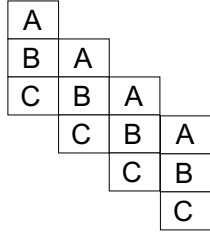
$$RecMII = \max_c \left\lceil \frac{Delay_i}{Distance_i} \right\rceil$$

The largest such constraint over all the dependence cycles for the loop determines the RecMII. One approach to compute the RecMII is to enumerate all the recurrence circuits in the dependence graph and apply the above equation [49]. An alternative method, proposed by Huff [50], is to pose the problem as a minimum cost-to-time ratio problem, where a dependence arc is viewed as a cost of $0 - Delay$ and a time of $Distance$. If M is the minimum cost-to-time ratio, then $RecMII = \lceil -M \rceil$. Using this method, the RecMII can be computed for each strongly connected component (SCC) of the dependence graph in $O(n^3)$ time [10]. The maximum over all the SCCs is the RecMII.

The actual scheduling can be done using a variety of algorithms. Possibilities include backtracking algorithms [10] and algorithms that attempt to reduce the register usage of the modulo schedule [51, 52]. Once a valid schedule is found, the code for the software pipelined loop is generated. Figure 3.8(a) shows four overlapped loop iterations after modulo scheduling. In this dissertation, the abstract code representation of [53] is used to reduce the complexity of the examples. In the figure, each square represents one stage worth of instructions for a single iteration. The number of stages in the schedule is called the *stage count*. Once enough iterations have been started, a steady state is reached. This steady state can be rolled into a loop, producing the basic code structure in Figure 3.8(b).

Overlap of the loop iterations often results in overlap of the lifetimes of the loop variants (a variant is a variable that may be redefined every iteration). That is, the variant may be defined again by a subsequent iteration before the value defined in the current iteration has been used.

(a) Overlapped Iterations



(b) Basic Code Structure

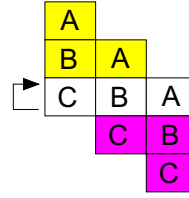


Figure 3.8 Overlapped Iterations and Basic Code Structure.

Overlapping lifetimes can be renamed using a technique called modulo variable expansion [40].

This technique unrolls the kernel and renames the lifetimes so that there is no overlap.

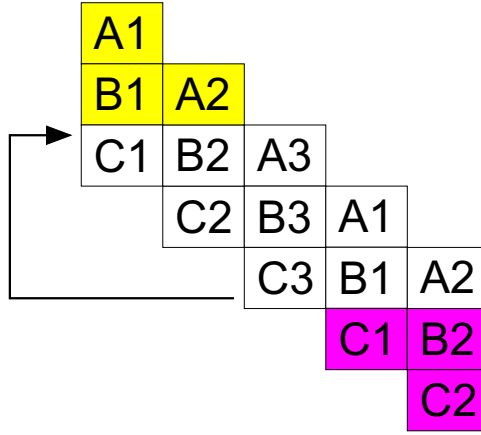


Figure 3.9 Modulo Scheduled Loop Structure with Kernel Unrolling.

Figure 3.9 shows the code structure with kernel unrolling for modulo variable expansion. Each stage is now numbered to show the version of the code used. Each version uses different register names. The figure assumes that the longest register lifetime spans three pipeline stages. Thus the kernel is unrolled three times. This code structure is simplistic and does not allow an arbitrary number of iterations to be correctly executed [53]. Complete code generation schemes will be discussed in Chapter 5. Renaming can be done in hardware using rotating registers [46, 53].

This eliminates the need for kernel unrolling. If the architecture contains rotating registers and predication, code can be generated without kernel unrolling and without a prologue or epilogue, resulting in no code expansion. This is referred to as *kernel-only* code [53].

CHAPTER 4

THE IMPACT MODULO SCHEDULER

This chapter describes the modulo scheduler that was implemented as part of this thesis. Figure 4.1 shows the overall structure of the modulo scheduler. The scheduler receives as input the Mcode for the target architecture after the prepass portion of the machine-dependent peephole optimizations. The first steps are to identify and prepare the appropriate loops for modulo scheduling. Next, the dependence graph is built using latencies obtained from the machine description. The initial candidate II is computed using the dependence graph and knowledge of the resources used by each instruction. The resource information is obtained through an interface to the resource manager.

An attempt is then made to schedule the loop at the MII using the dependence graph and the resource manager. If the scheduler fails to find a valid schedule at the candidate II, the II is incremented and scheduling is attempted again. Scheduling is aborted if the II or the number of tries reach predetermined values. If scheduling is aborted, the loop is scheduled by the acyclic scheduler. Once a valid schedule is found, kernel unrolling and modulo variable expansion are performed. Finally, the prologue and epilogues are generated.

Following the modulo scheduling phase, code that has not been software pipelined is scheduled by the acyclic scheduler. Then global register allocation is applied to the entire function. If spill code is introduced into the kernel of a modulo scheduled loop, the kernel with spill code is rescheduled using postpass acyclic scheduling. This is not the optimal way to handle spill

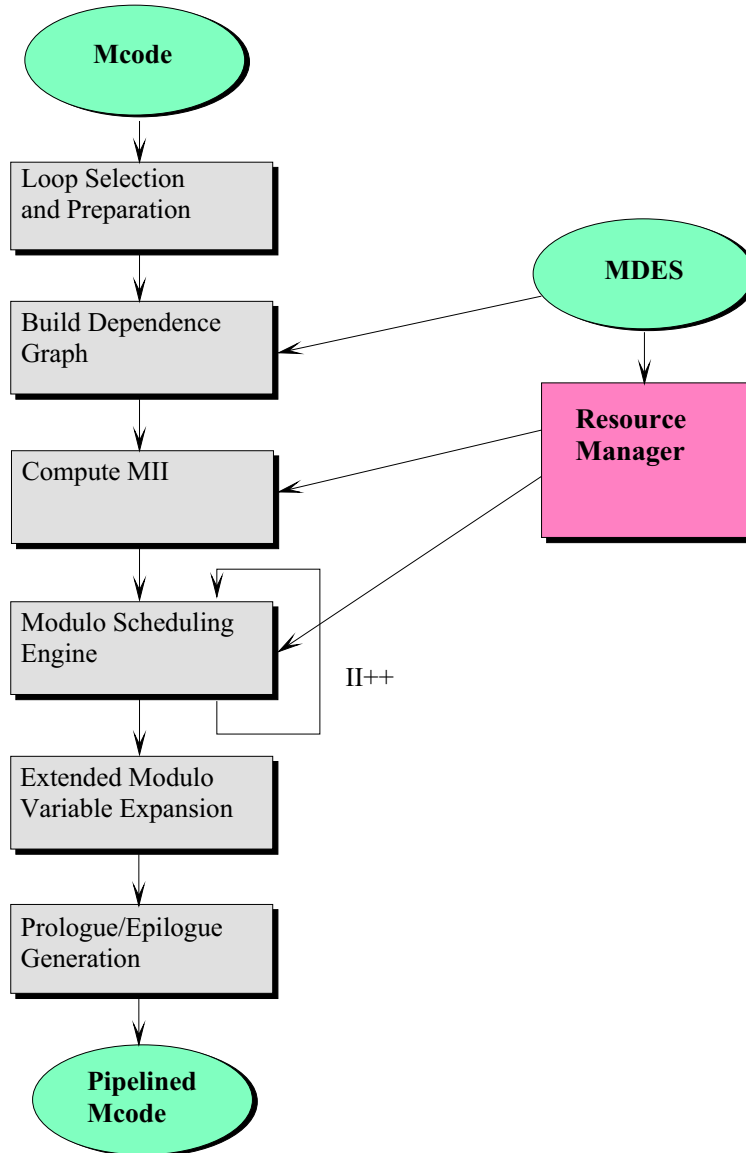


Figure 4.1 The IMPACT Modulo Scheduler.

code. To make sure that this did not affect the results of the studies done in later chapters, the modulo scheduled loops were checked for spill code after compilation was complete. Only a handful of infrequently executed loops contained spill code. The following sections describe each of the steps of the implementation in detail, with the exception of the generation of the prologue and the epilogues. This step is covered in detail in Chapter 5.

4.1 Loop Selection and Preparation

Loops are targeted for modulo scheduling early in the compilation process. This allows them to be appropriately optimized. During the first step of modulo scheduling, the targeted loops are re-examined to determine if they are still eligible for modulo scheduling. This is necessary because the translation from Lcode to Mcode could introduce hazards that prohibit modulo scheduling of the loop. For example, if the target processor supports integer divides via a function call, the Lcode divide instruction would be translated into a function call during annotation, prohibiting modulo scheduling of the loop.

The same criteria are used to select eligible loops both early in the compilation process, and later during the first step of modulo scheduling. The point at which eligible loops are first identified depends upon the compilation path used. If superblock formation and hyperblock formation are not applied, the eligible loops are identified immediately following machine-independent classic optimization. Otherwise, the eligible loops are identified after superblock formation, hyperblock formation, and optimization. Hyperblock formation and superblock formation can increase the number of loops eligible for modulo scheduling (see Chapter 5). In either case, the loops are selected prior to loop unrolling so that the appropriate ILP optimizations can be applied. The basic block, superblock, or hyperblock that makes up the body of the loop is marked to indicate that the loop is eligible for modulo scheduling.

The criteria for selecting loops are as follows. The total number of executions of the loop body and the average number of iterations per invocation must be above a pre-defined threshold. Modulo scheduling of loops containing function calls is currently not supported. Loops containing floating-point to integer or integer to floating-point conversions are also not eligible, because, in most architectures, the converted data is transferred between units via a memory

location on the stack. The memory reads and writes are inserted during annotation. The same memory location is overwritten every iteration creating a recurrence. However, the sync arcs necessary for the modulo scheduler to detect the recurrence are not added during annotation. Finally, the loop body must end with a conditional branch back to the top of the loop rather than a conditional branch to the exit followed by a jump to the top of the loop. All of the above restrictions will yield to further implementation effort.

Parameters can be set by the user to control whether or not the modulo scheduling is performed for loops with non-trivial recurrences, while (non-counted) loops, multiple-exit superblock loops, and hyperblock loops. Another parameter enables the printing of statistics that report information on which loops were pipelined, which were not, and why.

After the eligible loops have been verified, the disjoint live ranges within the loop body are renamed. If the live range is disjoint within the loop body, but not outside, a compensation code block is added at the exits for which the renamed register is live-out. A move is inserted in this block if the current name for the virtual register is not the same as that expected outside the loop. After modulo scheduling, the compensation blocks are merged with the epilogues and copy propagation is performed to remove any unnecessary moves. Renaming the disjoint live ranges ensures that for any elementary cycle in the dependence graph, there is only one definition of a particular live range. This in turn ensures that any remaining anti-dependences can be removed using modulo variable expansion, effectively allowing the loop to be put into dynamic single assignment form [54].

A data structure is built for each instruction to hold the information needed for scheduling and for the code generation scheme. Some of the more important items contained in the structure are a pointer to the alternative chosen for the instruction (if scheduled), a pointer

to the machine description information for the instruction, the priority, the issue time, stage, and slot in which the instruction is scheduled (if any), the copy of the kernel (due to kernel unrolling) in which the instruction resides, the ID of the basic block in which the instruction resided prior to scheduling, and pointers to information about the lifetimes of the source and destination registers (for use by the modulo variable expansion algorithm).

4.2 Dependence Graph Construction

The dependence graph builder is shared with the acyclic scheduler. The graph can be built in two modes. The acyclic mode builds a graph containing only the intra-iteration dependences and is used by the acyclic scheduler. The modulo scheduler invokes the dependence graph builder in cyclic mode, which adds the cross-iteration dependences. In both cases, the graph contains register, memory, and control dependences and is built for a single superblock or hyperblock (in cyclic mode, that block is always the body of a loop). The cyclic mode was added to the dependence graph builder as part of the work for this thesis and supports modulo scheduling of superblock loops. The data structure for a dependence arc contains the type (register, memory, control, flow, anti-, output) and the delay and distance of the dependence.

The cross-iteration register dependences are added in four linear passes over the superblock. In the first top-down pass, the lexically last definition of each register is stored in a hash table. In the second top-down pass, the cross-iteration flow and output dependences are added, and register definitions are removed from the table when they are killed by the lexically first definition of the same register (i.e., the computation of dependences does not rely on the superblock being in SSA form). In the third bottom-up pass, the lexically first definition of each register is stored in the hash table. In the final bottom-up pass, anti-dependences are

added, and register definitions are removed from the table when they are killed (in the sense that they cannot be anti-dependent upon any further uses in the pass) by the lexically last definition of the same register. For general virtual registers, the extended modulo variable expansion described in Chapter 5 allows the anti-dependences associated with overwriting the same register every loop iteration to be removed. Thus the only register anti-dependences added are those associated with the special-purpose non re-nameable registers.

The memory dependences are computed by examining the sync arcs between memory instructions. If there is an inner-loop-carried dependence between two memory instructions, then a memory dependence arc is added.

Cross-iteration control dependences are added from the loop back branch to the appropriate instructions which precede the first exit branch in the superblock. Subsequent instructions in the superblock (if any) are guarded by the control dependences on the first exit branch. Instructions for which speculative execution is allowed¹ do not have a control dependence added. The rules for adding control dependences (both cross-iteration and intra-iteration) are relaxed in cyclic mode compared to acyclic mode. For example, instructions with a destination register that is live-out can be moved upward across the branch during modulo scheduling (because of the extended modulo variable expansion described in Chapter 5) but not during acyclic scheduling.² Thus in such a case, a control dependence is not added in cyclic mode. Control dependences are also used to prevent the motion of instructions downward across a branch. In acyclic mode, stores and instructions with a destination register that is live out cannot be moved downward

¹The set of instructions which are eligible for speculative code motion depends on the architecture. The dependence graph builder queries the machine description to find out if a particular instruction can be speculatively executed.

²For acyclic scheduling, a renaming-with-copy optimization similar to dynamic renaming [41] is performed during ILP optimization to reduce the effect of this restriction. This optimization increases the number of instructions in the loop and is not needed for modulo scheduling, so it is not applied to loops targeted for modulo scheduling.

across a branch. The IMPACT acyclic scheduler does not generate the compensation code required to allow such motion. The epilogues in the code generation scheme for a modulo scheduled loop provide the necessary compensation code, so this restriction does not exist in cyclic mode. Only control and synchronization instructions are prevented from moving downward across branches in cyclic mode.

START and STOP pseudo-instructions are added to the dependence graph to facilitate some of the graph algorithms such as computing the height-based priority. They are made the predecessor and successor respectively of all the nodes in the graph. All the dependences between these pseudo-instructions and the rest of the instructions have both delay and distance 0.

4.3 Calculation of the MII

The ResMII is computed using the bin-packing algorithm of [10]. This is done through an interface to the resource manager. The modulo scheduler maintains an array containing the usage count of each resource. The array and an instruction are passed to the resource manager, which updates the array with the resource usages of the best alternative for the instruction. The best alternative is that which yields the smallest partial ResMII. The instructions are processed in order of the number of alternatives starting with the instruction with the fewest alternatives.

The RecMII is computed using the MinDist algorithm described in [10] and [50]. For simplicity, the algorithm is run on the entire superblock rather than on each strongly connected component (SCC), making the calculation more expensive. The initial trial value of II is 1 rather than the ResMII so that statistics can be gathered on the RecMII values.

4.4 Modulo Scheduling Engine

The modulo scheduler itself is an implementation of Rau’s iterative modulo scheduling [10]. Limited backtracking (unscheduling) is performed when the scheduler fails to meet dependence or resource constraints. The scheduler is given a budget that is equal to some multiple of the number of instructions in the superblock loop body. If the scheduler has not found a valid schedule by the time that it has scheduled a number of instructions equal to the budget, it aborts the scheduling attempt and increments the II. The multiple is called the *budget_ratio* and is a parameter. In effect, the scheduler can schedule each instruction an average of *budget_ratio* times before giving up on the candidate II.

The instructions are scheduled in priority order. A simple height-based priority was shown to work well in [10] and is used here. The loop body can be scheduled either from the top down or from the bottom up. In the former case, the height-based priority of an instruction $\mathbf{x}$ is equal to $MinDist[\mathbf{x}, STOP]$, and the earliest start time of an instruction is computed based on its scheduled predecessors. In the latter case, the height-based priority of an instruction $\mathbf{x}$ is equal to $MinDist[START, \mathbf{x}]$, and the latest start time of an instruction is computed based on its scheduled successors. The scheduled instructions that are not currently scheduled are kept in a priority queue. When an instruction is unscheduled, it is inserted into this queue in priority order.

During scheduling, a special schedule cycle numbering scheme is used to keep the cycle numbers from becoming negative during bottom-up scheduling. The initial latest start time for the STOP node is set to a large multiple of the II minus 1 (1000 times the II in the current implementation on the assumption that the schedule for an iteration will never be stretched over 1000 stages). This corresponds to the last row of the MRT. For consistency, in top-down

scheduling, the initial earliest start time for the START node is set to a large multiple of the II , corresponding to the first row of the MRT. After scheduling, the cycle numbers are shifted toward 0 by subtracting off the largest multiple of II such that all the instruction issue times remain positive.

When scheduling an instruction, up to $II + 1$ cycles are checked rather than II . The reason for this is that in the first cycle checked, some of the slots may not have been tried due to a dependence with a zero-cycle delay on an instruction already scheduled in that cycle. Examples of zero-cycle delays depend on the machine description, but typically include a branch followed by a control dependent instruction in the fall-through path, or a store followed by a load of the same address.

The loop back branch is freely scheduled (subject to dependence and resource constraints) in any cycle of the MRT. The placement of the branch determines the boundaries of the stages. After scheduling, to form the kernel, the MRT is rotated until the loop back branch is in the last row. In this dissertation, no branch delay slots are assumed, so the loop back branch must be scheduled in the rightmost slot of its row in the MRT. This assumes either a superscalar processor for which the instructions are laid out in sequential order in memory by scanning the MRT from left to right and top to bottom or a VLIW processor for which a taken branch operation must occupy the rightmost slot in the instruction word.

After an instruction is scheduled, the dependences going to its scheduled successors (for top-down scheduling) or coming from its scheduled predecessors (bottom-up scheduling) are examined. If any dependence is violated, the successor or predecessor is unscheduled and placed back in the priority queue.

After scheduling, register lifetimes are optimized by moving instructions that have no predecessors as late as possible and instructions that have no successors as early as possible by multiples of II cycles. This has the same effect as the Sink/Source (SS) heuristic in [51]. Scheduling from the bottom up can reduce register pressure compared to scheduling from the top down. This is because dependence graphs often have more merge points than fork points, especially in numeric programs.

4.5 Extended Modulo Variable Expansion

The extended modulo variable expansion (MVE) implemented for the IMPACT modulo scheduler is capable of renaming lifetimes that cross iterations and are live out of loop exits. The theory behind this is described in Chapter 5. This section describes the implementation.

The modulo variable expansion is broken into an analysis phase and a transformation phase. During the analysis phase, information is collected about the lifetimes of the loop variants. For each lifetime a structure is built that contains the length of the lifetime, the issue time of the first definition, and a flag indicating whether or not the lifetime is live out of any exit. There is also a flag to indicate if the last use is scheduled in a slot farther to the right than the first definition. In some cases, for a superscalar processor model, this is needed to determine if a lifetime with a given amount of renaming will overlap with itself or not. If the last use of the current lifetime is in the same cycle as the first definition of a subsequent instance of the lifetime, the ordering of the first definition and the last use in the slots determines whether or not the two lifetimes will overlap when the instructions residing in the same cycle are placed in sequential order.

The analysis proceeds by examining each instruction in the original program order.³ In a first pass over the instructions, the outgoing register flow dependences are examined to find the uses of each register defined by the current instruction. Since cross-iteration register dependences are included in the dependence graph, cross-iteration lifetimes can be analyzed. Whenever a new virtual register destination is encountered, a new lifetime structure is created. There is also an MVE info structure associated with each source operand. This structure contains a pointer to the lifetime structure associated with the source operand and the length of the lifetime from the first definition to this use. When each flow dependence is examined, this structure is filled in. The total length of the lifetime is updated by taking the maximum of the current total length and the length from the first definition to this use. A list of the branch instructions is also built up during the first pass.

During a second pass over the instructions, each destination register is checked to see if it is live out of any of the branches. For each branch for which the destination register is live out, the lifetime length is updated by taking the maximum of the current total length and the length from the first definition to the branch.

The amount of renaming required for each lifetime is computed in one pass over the lifetime structures. During the same pass, the amount of kernel unrolling is computed as the maximum amount of renaming required by any one lifetime. Define **kmin** as this amount of kernel unrolling. A second pass is then made over the lifetime structures to make the amount of renaming for each lifetime a factor of **kmin** and to allocate an array of register numbers for each lifetime. One of the register numbers allocated is the original virtual register number for

³During modulo scheduling, the issue times and slots of the instructions are determined, but the instructions are not actually reordered until after the MVE analysis phase.

the lifetime. As described later, this can reduce the number of moves inserted in the prologue and the epilogues for live in/out variants.

After the analysis phase but before the transformation phase, the instructions in the loop body are reordered to form the kernel. During the transformation phase, the kernel is unrolled and the registers are renamed. The renaming is done in such a way that the first use of a live-in variant (a variant that is used before defined in the prologue) in the prologue uses the original virtual register number for the variant. A variant defined in the first copy of the unrolled kernel is assigned a register number as follows. The array of allocated register numbers is indexed starting from 0. The register number at index 0 is the original virtual register number. Assume there are N register numbers allocated for the variant and that the first definition of the variant occurs in stage D of the schedule for a single iteration. The live-in value can be viewed as being defined by an imaginary iteration during stage $D - 1$ of the first iteration. This definition is assigned the register number at index 0. The definition of the variant in the first copy of the unrolled kernel occurs during stage $SC - 1$ of the first iteration, where SC is the stage count. This definition is assigned the register number at index:

$$(SC - 1 - (D - 1)) \text{ modulo } N = (SC - D) \text{ modulo } N$$

Definitions of the variant in subsequent copies of the kernel are assigned the register number at index:

$$((SC - D) + C) \text{ modulo } N,$$

where C is the copy of the kernel in which the definition resides. The kernel copies are numbered starting from 0.

Each use of a variant is renamed by finding the kernel copy in which the variant was defined (using the stored information about the length of the lifetime from the first definition to the current use) and then assigning the same register number that was assigned to the definition in that copy.

CHAPTER 5

MODULO SCHEDULING OF LOOPS IN CONTROL-INTENSIVE NON-NUMERIC PROGRAMS

Most of the previous work on modulo scheduling has targeted numeric programs, in which, often, the majority of the loops are well-behaved "DO" loops (loop-counter-based loops) without early exits. All of the more extensive performance evaluations of modulo scheduling techniques have been for such loops. In control-intensive non-numeric programs, the loops frequently have characteristics that make it more difficult to apply modulo scheduling and to obtain significant speedup. These characteristics include multiple control flow paths, loops that are not based on a loop counter, and multiple exits. Several techniques have been developed to allow modulo scheduling of loops with intra-iteration control flow, such as hierarchical reduction [40], predicated execution [46], and reverse if-conversion [55]. The above work has assumed that all of the paths through the loop body are included for scheduling. Unfortunately, including all of the paths can be detrimental to overall loop performance. The presence of unimportant paths with high resource usage or long dependence chains can result in a schedule that penalizes the important paths. A path that contains a hazard, such as another nested loop or a function call, can prohibit modulo scheduling of the loop.

Previous work has also been done on modulo scheduling of loops that are not based on a loop counter [53], [56]. The key difficulty with this type of loop is that it may take many cycles to determine whether or not to start the next iteration, limiting the overlap of the iterations. This difficulty is overcome by speculatively initiating the next iteration. The work in [56]

also mentions a source-to-source transformation to convert a loop with multiple exits into a single-exit loop. The resulting loop contains multiple paths of control and is dealt with using one of the methods for modulo scheduling of loops with intra-iteration control flow. However, this method adds extra instructions and delays the early exits until the end of the loop body. More work is needed to evaluate the performance of this approach, especially for architectures without predicated execution.

This chapter describes a new set of methods that allows effective modulo scheduling of loops with multiple paths of control and multiple exits. Superblock [6] techniques are used to exclude the unimportant and detrimental paths from the loop. Loops with multiple exits often occur naturally in control-intensive programs and the beneficial exclusion of paths via the formation of superblock loops creates many more of them. Thus, an effective method for handling multiple exits is essential.

Rather than transform the loop into a single exit loop, the proposed methods schedule the loop as is, with the multiple exits present. A new code generation scheme is described that creates correct epilogues for the early exits. Speculation is used to increase both the overlap of the basic blocks within each iteration and the overlap of successive iterations. Modulo variable expansion is extended to allow the speculation of instructions that write to variables that are live at the loop exits. Altogether, the methods described in this chapter allow effective modulo scheduling of the selected paths of loops with arbitrary control flow.

The chapter is organized as follows: Section 5.1 describes the methods developed and presents a case study to show how these methods, when combined with superblock techniques, enable modulo scheduling to be effectively applied to control-intensive loops. Section 5.2 reports speedup results for several SPEC CINT92 benchmarks and Unix utilities. These are the

first reported performance results for modulo scheduling on control-intensive non-numeric programs, and they demonstrate the applicability of modulo scheduling to this class of programs and validate the correctness of the proposed methods.

5.1 Case Study and Methods

A detailed example is used to illustrate the difficulties caused by control-intensive loops and the benefits of the techniques developed. The loop chosen for this case study is one of the frequently executed loops in *lex*, the lexical analyzer generator. The source code for the loop is shown in Figure 5.1.

```
for (i = n; i >= 0; i--) {  
    j = state[i];  
S1: if (count == *j++) {  
        for (k = 0; k < count; k++)  
            if (!temp[*j++]) break;  
        if (k >= count)  
            return (i);  
    }  
}
```

Figure 5.1 Source Code for Example Loop from *lex*.

Loops in general-purpose non-numeric programs frequently have complex control flow, which is evident in the example loop. The outer loop contains an if-statement, an inner loop, and an early exit via a return statement. The inner loop contains an if-statement and an early exit via a break statement.

Obviously, this loop contains a number of hazards for modulo scheduling. Modulo scheduling would ordinarily target the inner loop. However, profile information indicates that the inner

loop is infrequently invoked and usually has few iterations. The condition for the if-statement S1 evaluates to false more than 90% of the time. Figure 5.2(a) shows a simplified version of the control flow graph for the loop. Block X contains the code to load `state[i]` and `*j` and do the comparison for statement S1. Block Y consists of the post-increment of the pointer `j` and all the code in the body of the if-statement S1. The control flow within block Y has been omitted for clarity. Block Z contains the code to update `i` and to test the exit condition. The branch preceding block X to test the exit condition for the initial value of `j` has also been omitted for clarity.

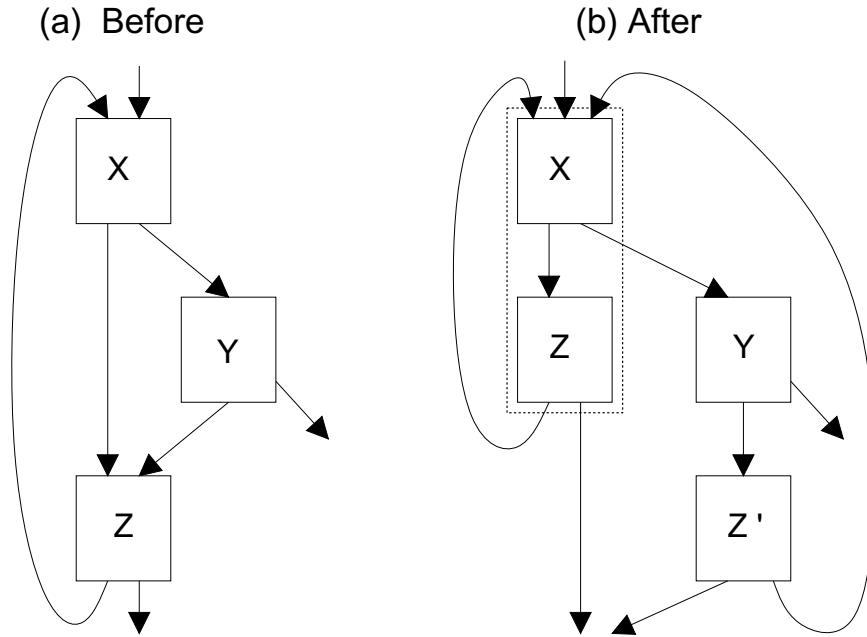


Figure 5.2 Superblock Formation for Example Loop.

The detrimental path containing the inner loop can be excluded from the loop via superblock formation. A superblock loop consisting of the most frequent path through the outer loop (blocks X and Z) is formed as shown in Figure 5.2(b). The path through block Y has been excluded via tail duplication of block Z. A superblock loop consists of a single path through a

loop, with a single entrance and one or more exits. The loop consisting of blocks X and Z now appears to be an inner loop with multiple exits and can be targeted for modulo scheduling.

It has been shown that superblock optimization and acyclic scheduling techniques provide substantial speedup [6]. For modulo scheduling, the ability of superblocks to exclude undesirable paths of execution can provide the following benefits:

- Decrease ResMII by excluding unimportant paths with high resource usage.
- Decrease RecMII by excluding unimportant paths that contribute to long dependence cycles.
- Increase the number of loops that can be modulo scheduled by excluding paths containing hazards such as nested loops and function calls.

Although the modulo scheduling methods developed in this chapter are described using superblock examples, they are equally applicable to hyperblock code.

Figure 5.3 shows the assembly code for the example superblock loop. Each instruction is numbered for later reference. Block X in the control flow graph consists of instructions 1 through 3. Instructions 4 through 6 are in block Z. The assembly code shown is that produced by the IMPACT compiler after classic optimizations, such as loop invariant code removal, global variable migration, and strength reduction, have been applied. The elements of the array **state** are four bytes in size. The registers shown are virtual registers. Recall that register allocation is done after modulo scheduling.

Control exits the superblock loop if instruction 3 is taken, or if instruction 6 is not taken. In this dissertation, the exit associated with the fall-through path of the loop back branch is

Inst.	Assembly	Register Contents
1	L1: r12 = MEM(r34+r8)	r8 = state
2	r13 = MEM(r12+0)	r34 = i*4
3	beq (r6 r13) L2	r12 = j
4	r4 = r4 - 1	r13 = *j
5	r34 = r34 - 4	r6 = count
6	ble (0 r4) L1	r4 = i

Figure 5.3 Assembly Code for Superblock Loop.

termed the *final exit*. Any other exits from a superblock loop are via taken branches and are termed *early exits*.

The virtual registers r34, r4, and r12 are live out when the early exit to **L2** (block Y) is taken. The values in r34 and r4 are decremented in block Z'. The value in r12 is incremented in block Y. Note that there is no increment of the pointer **j** in the superblock loop. The incremented value of **j** is dead when the condition for if-statement S1 evaluates to false, so the increment shown in the condition for the if-statement S1 is done after control is transferred to block Y. No virtual registers are live out when the loop exits via the final exit (instruction 6).

Loops with complex control flow occur frequently in general-purpose non-numeric programs. Table 5.1 shows statistics on the percentage of dynamic instructions that are in single basic block loops (*Basic Block*) and multiple exit superblock loops (*Superblock*) for the SPEC CINT92 benchmarks and several Unix utility programs. The column labeled *Total* is the sum of the other two columns. The time not spent in these two types of loops is spent in the excluded paths of inner and outer loops and in acyclic code.

For all the programs except *gcc* and *tbl*, little or no time is spent in single basic block loops. For all the programs except *tbl*, more time (usually much more) is spent in multiple

Table 5.1 Percentage of Dynamic Instructions in Single Basic Block and Superblock Loops.

Benchmark	Basic Block	Superblock	Total
008.espresso	5.6	57.8	63.4
022.li	0.7	21.4	22.1
023.eqntott	1.8	70.5	72.3
026.compress	0.6	49.8	50.4
072.sc	4.4	34.6	39.0
085.gcc	14.1	28.5	42.6
cmp	0.0	94.5	94.5
eqn	2.6	20.9	23.5
lex	2.0	86.2	88.2
tbl	17.4	9.6	27
yacc	3.2	45.5	48.7

exit superblock loops than in single basic block loops. From this table, it is clear that modulo scheduling must be able to effectively handle loops with control flow to be applicable to these programs. The remainder of this chapter describes how the proposed techniques overcome the control dependences and register anti-dependences associated with loops that have multiple exits and live-out virtual registers. A code generation scheme for loops with multiple exits is also presented.

5.1.1 Overcoming control dependence using speculative code motion

Control dependences are a major impediment to the exploitation of ILP in the loops of general-purpose non-numeric programs. Cross-iteration control dependences restrict the overlap of loop iterations by delaying the start of subsequent iterations until all the branches from the current iteration have been executed. Frequently the branches are dependent on earlier

computations in the loop body and cannot be executed until late in the iteration, severely limiting any overlap.

Intra-iteration control dependences combined with cross-iteration data dependences create recurrences which limit the throughput of the modulo scheduled loop. They also increase the length of the critical paths through a single iteration, resulting in a longer schedule for each iteration, an important consideration for short trip count loops.

As described in [53] and [56], the cross-iteration control dependences from the loop back branch to the instructions in the next iteration can be relaxed, allowing speculative code motion and overlap of the iterations. For loops with multiple exits, this concept must be extended to the early exit branches. It is often necessary to remove the cross-iteration control dependences from an early exit branch to the instructions in subsequent iterations to achieve the desired level of overlap. It is also often necessary to remove intra-iteration control dependences to allow overlap of the blocks within an iteration and to achieve good performance for short trip count loops.

The effect of control dependences on the example superblock loop is now shown. Figure 5.4(a) shows the dependence graph. Each node is numbered with the ID (from Figure 5.3) of the instruction it represents. The branch nodes are shaded. The data and control dependences are shown with solid and dashed lines, respectively. Some of the transitive dependences are not shown. None of the register anti-dependences are shown, assuming that they can be removed. Removal of anti-dependences is discussed in Section 5.1.2.

There are several non-trivial recurrences apparent in the graph. The longest recurrence circuit runs through instructions 1, 2, 3, 4, 6, and back to 1. It has a total delay of six and spans one iteration, resulting in a RecMII of six. If the loop is scheduled using this dependence

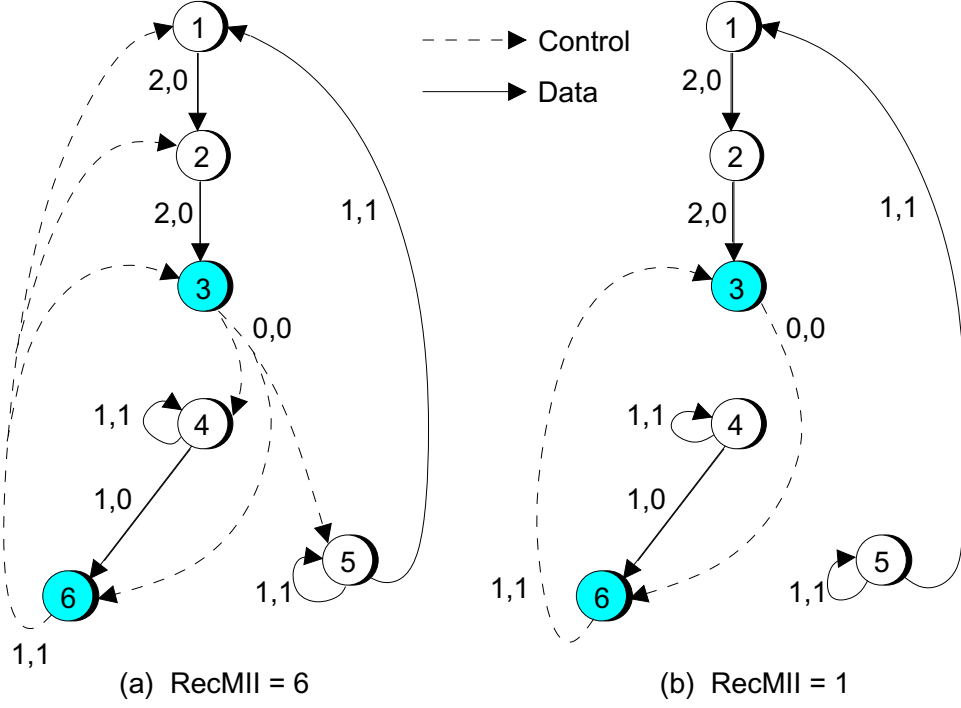


Figure 5.4 Dependence Graph for Example Loop.

graph, there is no overlap of the iterations. The loop back branch is dependent on almost all of the instructions in the loop and the next iteration is dependent on the loop back branch. The cross-iteration control dependences from the loop back branch to the instructions in the next iteration (except instruction 3) can be removed, allowing speculative code motion and overlap of the iterations. However, there are still limiting control dependences present. The recurrence circuit consisting of instructions 1, 2, 3, and 5 limits the RecMII to five. To break this recurrence, the intra-iteration control dependence between instructions 3 and 5 must be removed, enabling speculative execution of instruction 5. The control dependence from instruction 3 to instruction 4 must also be removed to break the remaining limiting recurrence. Figure 5.4(b) shows the dependence graph after all of the limiting control dependences have been removed, reducing the RecMII to one.

An instruction can legally be moved during modulo scheduling from above to below a branch if the branch is not data dependent on the instruction. For example, instruction 5 could legally be scheduled after instruction 6. When an instruction is moved from above to below a branch, it is automatically moved into both paths of the branch during the generation of epilogues following the actual modulo scheduling process. In Sections 5.1.2 and 5.1.4, it is shown that special attention must be paid to this type of code motion for correct code generation in multiple exit loops.

Figure 5.5 shows the modulo resource table (MRT) [10] after modulo scheduling by the IMPACT compiler using the dependence graph of Figure 5.4(b). Assuming a four-issue processor that can execute one branch per cycle, the ResMII for the example loop is two. The RecMII was one, resulting in an II of two. This is a speedup of three over modulo scheduling using the dependence graph of Figure 5.4(a).

Cycle Mod II	Issue Slot			
	0	1	2	3
0	instr. 3 issue 4	instr. 4 issue 4	instr. 2 issue 2	instr. 1 issue 0
	instr. 5 issue 1			instr. 6 issue 5

Figure 5.5 Modulo Resource Table after Modulo Scheduling.

5.1.2 Overcoming anti-dependence using modulo variable expansion

Thus far, nothing has been said about anti-dependences and the constraints imposed by the virtual registers that are live out of the loop exits. In its original form, an instruction **I** that writes a virtual register that is live out of an exit branch **B** cannot be moved from below to above

B because it overwrites a value that is used when the exit is taken. This constraint on upward code motion is exactly the same as if the virtual register were one of the operands of **B** (i.e., it is an anti-dependence constraint) but is represented differently in many compilers. Instead of adding an explicit anti-dependence arc, many compilers, including IMPACT, overload the control dependence arc to represent both the control dependence and the anti-dependence.

There are several examples of anti-dependence in the case study loop. Instruction 1 uses r34, which is later defined by instruction 5. Virtual register r34 is live out when the branch to **L2** (instruction 3) is taken, so there is an anti-dependence between instruction 3 and instruction 5.

Anti-dependences can be removed by renaming [36], [57], allowing the lifetimes of two different values, which formerly were assigned to the same virtual register, to overlap in time. For example, the destination of instruction 5 can be locally renamed as shown in Figure 5.6. This allows instruction 5 to be moved up in the schedule past instruction 3 and instruction 1. A move instruction is added to preserve correctness because r34 is used outside the superblock.

Inst.	Assembly
1	L1: r12 = MEM(r34+r8)
2	r13 = MEM(r12+0)
3	beq (r6 r13) L2
4	r4 = r4 - 1
5	r100 = r34 - 4
	r34 = r100
6	ble (0 r4) L1

Figure 5.6 Assembly Code with Renaming of r34.

Unfortunately, the move is now an extra resource used in the loop, and there still exists an anti-dependence, from instruction 3 to the move, which is on a critical recurrence circuit.

Furthermore, all the resources required by the loop must be known before modulo scheduling begins, so that the ResMII can be computed, resulting in a phase ordering problem. The decision to rename the destination of an instruction so that it can be speculatively executed must be made before modulo scheduling, but at that time it may be unclear whether the instruction has to be speculatively executed.

Modulo variable expansion (MVE) [40], [53] unrolls the kernel and renames the successive lifetimes corresponding to the same variant so that they no longer overlap in time. This allows register anti-dependences to be removed before scheduling, knowing that modulo variable expansion will correct the overlap of lifetimes that the lack of these dependences allows. Modulo variable expansion does not require the addition of moves within the loop body and therefore, does not present a phase-ordering problem. The modulo variable expansion algorithm, as originally described [40], allows the removal of cross-iteration anti-dependences. However, intra-iteration anti-dependences can also be removed if the lifetime analysis and the renaming algorithms are extended to include lifetimes that cross iterations. It is assumed that this can be done in [53]. In this dissertation, the changes necessary are described.

Figure 5.7 illustrates the relaxation of a cross-iteration anti-dependence using modulo variable expansion, as described in [40]. Three iterations of an abstract loop body containing a definition and use of a virtual register `r1` are shown in Figure 5.7(a). There is an intra-iteration flow dependence (marked with an `f`) and a cross-iteration anti-dependence (marked with an `a`). The cycle in which each instruction is issued is shown in square brackets to the right of the abstract instruction, assuming the delay for the flow dependence is two and the anti-dependence is zero. In its original form as shown on the left, the minimum `II` that can be achieved is two.

Using modulo variable expansion, the anti-dependence can be removed prior to scheduling, reducing the II to one. Two virtual registers are now used as shown in Figure 5.7(b).

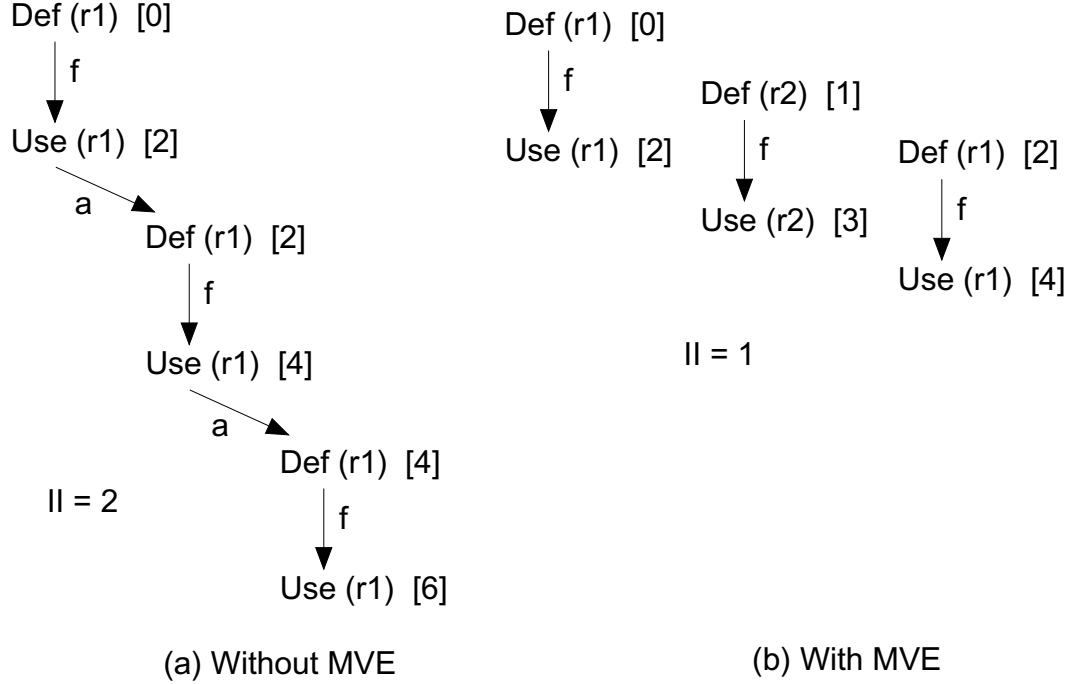


Figure 5.7 Relaxation of Cross-Iteration Anti-Dependence.

Figure 5.8 shows the relaxation of an intra-iteration anti-dependence. In this case, the use appears before the definition in the original iteration, and the lifetime of `r1` now crosses the iterations. Removal of the intra-iteration anti-dependence prior to scheduling allows the definition to be moved above the use as shown in Figure 5.8(b). As in the previous case, two registers are used and the II is reduced from 2 to 1.

The lifetime of a virtual register extends from its first definition to its last use. The lifetime of a loop-variant virtual register **V** from a definition **D** to a use **U** is computed using the following equation, assuming that the lifetime starts when **D** is issued and ends when **U** is issued.

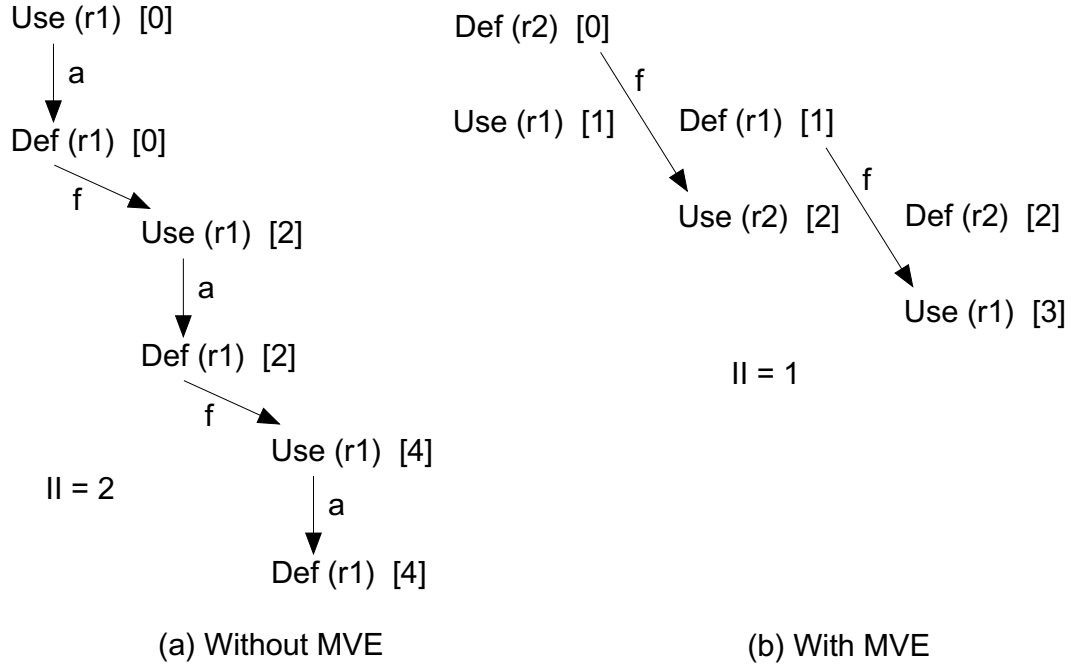


Figure 5.8 Relaxation of Intra-Iteration Anti-Dependence.

$$Lifetime(\mathbf{V}) = Issue(\mathbf{U}) - Issue(\mathbf{D}) + II * Dist(\mathbf{V}) \quad (5.1)$$

$Issue(\mathbf{D})$ and $Issue(\mathbf{U})$ are the issue times of the instances of $\mathbf{D}$ and $\mathbf{U}$ from the same original iteration.¹ $Dist(\mathbf{V})$ is the number of iterations separating $\mathbf{D}$ and the instance of $\mathbf{U}$ that uses the value defined by $\mathbf{D}$ in the original loop. The distance is either zero or one.²

Note that in Equation (5.1), use $\mathbf{U}$ could be a branch for which $\mathbf{V}$ is live out. For correct renaming, the lifetime analysis must be extended to include such uses. There is an additional consideration for live out virtual registers. Instruction $\mathbf{D}$ can be moved downward across the branch $\mathbf{B}$. If such code motion occurs, the definition is moved into both paths of the branch

¹Recall that modulo scheduling generates a schedule for a single iteration of the original loop. It is this schedule that the compiler is working with when analyzing the lifetimes for modulo variable expansion.

²The distance can be greater than one if the compiler's intermediate representation supports expanded virtual registers (EVRs) [54].

during epilogue generation, $\mathbf{V}$ is no longer live-out, and $\text{Lifetime}(\mathbf{V})$ as computed by Equation (5.1) becomes less than or equal to 0. Thus, the lifetime of $\mathbf{V}$ is computed for all the uses except those associated with the exits that $\mathbf{D}$ has been moved down across.

Figure 5.9 shows the execution of two iterations of the case study loop after modulo scheduling. The first iteration starts at time 0 and its instructions are denoted with the subscript 1. The second iteration starts at time 2 and its instructions are denoted with subscript 2. The second iteration's instructions are also shaded to further distinguish between the two iterations. The lifetimes of all the virtual registers written in the loop are shown to the right of the execution record. Each virtual register's lifetime begins with its definition in the first iteration. Each of the subsequent tic marks denotes either an explicit use of the virtual register as a source operand, or a branch for which the register is live-out. The lifetime extends until the last use of the register.

The lifetime of r13 is entirely contained within one iteration. It is defined by instruction 2 and used by instruction 3. $\text{Issue}(2)$ is 2, $\text{Issue}(3)$ is 4, $\text{Dist}(r13)$ is 0, and II is 2. Using Equation (5.1), the length of the lifetime is 2. The lifetime of r34 crosses iterations. It is defined by instruction 5, used by instructions 1 and 5 of the next iteration, and live-out of instruction 3 of the next iteration. $\text{Issue}(5)$ is 1, $\text{Issue}(3)$ is 4, and $\text{Dist}(r34)$ is 1. Using Equation (5.1), the total length of the lifetime is 5.

The longest lifetime, that of r34, is five cycles so the loop must be unrolled three times for modulo variable expansion. Figure 5.10 shows the unrolled kernel of the modulo scheduled loop after modulo variable expansion. The instructions have been renumbered. When renaming, one of the names used is the original virtual register name. The set of registers used for r34 is r34, r342, and r343. The set of registers used for r12 is r12, r122, and r123. The instructions

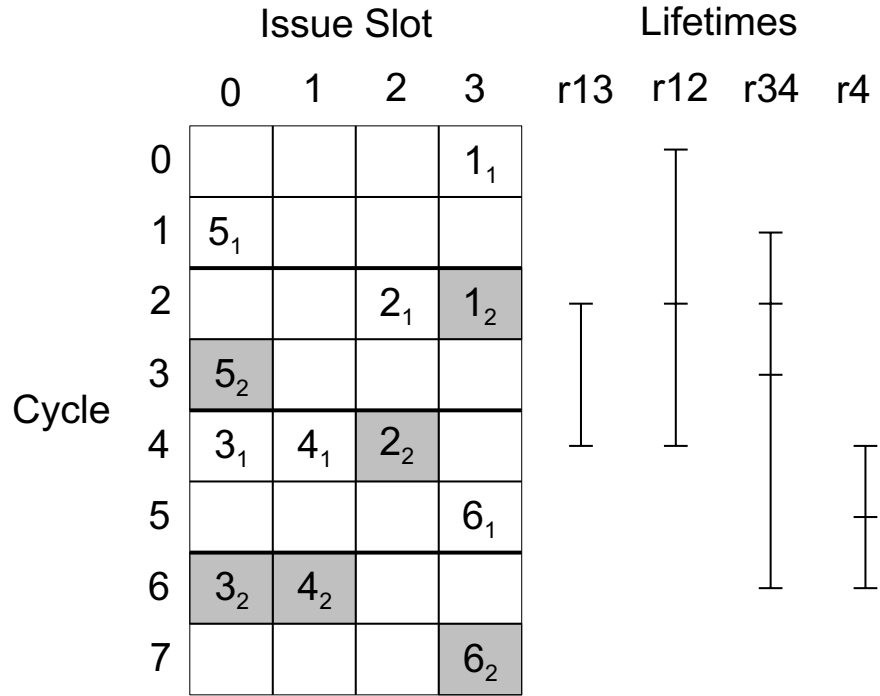


Figure 5.9 Execution Record and Lifetimes for Two Iterations.

have been put into sequential order, as would be done when generating code for a superscalar processor. The target and fall-through path for the first two copies of the loop back branch (instructions 6 and 12) have been reversed in preparation for epilogue generation. Block **L3** is the original fall-through path of the loop.

5.1.3 Review of a code generation scheme for single exit loops

This subsection reviews an existing code generation scheme for single exit loops in preparation for introducing a modified scheme for multiple exit loops. For a complete discussion of this and other possible code schemes for single exit loops, see [53].

After modulo scheduling and kernel unrolling, there are multiple copies of the loop back branch from the original loop. For all the copies except the one that becomes the loop back

Inst.	Assembly	Cycle
1	L1: beq (r6 r13) L2	0
2	r4 = r4 - 1	0
3	r13 = MEM(r123+0)	0
4	r12 = MEM(r343+r8)	0
5	r34 = r343 - 4	1
6	bgt (0 r4) L3	1
7	beq (r6 r13) L2	2
8	r4 = r4 - 1	2
9	r13 = MEM(r12+0)	2
10	r122 = MEM(r34+r8)	2
11	r342 = r34 - 4	3
12	bgt (0 r4) L3	3
13	beq (r6 r13) L2	4
14	r4 = r4 - 1	4
15	r13 = MEM(r122+0)	4
16	r123 = MEM(r342+r8)	4
17	r343 = r342 - 4	5
18	ble (0 r4) L1	5

L3:

Figure 5.10 Unrolled Kernel for Superscalar Processor.

branch of the kernel, the target and fall-through path are reversed, as was shown in Figure 5.10, so that the loop is exited when the branch is taken rather than when it falls through. Extending the terminology of Section 5.1, all the exits associated with the copies of the loop back branch are called *final exits*. When speaking of a single iteration, the exit associated with the fall through path of the original loop back branch is still referred to as the final exit. Also in this chapter, the *last* iteration refers to the last iteration that would have been executed in the original non-pipelined loop.

A few words must be said about the effect of speculation on the basic code structure of Figure 3.8. In the schedule for a single iteration, the final exit must be placed at the end

Assume the stages of an iteration are numbered such that stage A corresponds to 0, stage B corresponds to 1, and so on. Using the terminology of [53], if the final exit is scheduled in stage θ , then there are θ speculatively executed stages for each iteration after the first.

Figure 5.12 shows the complete structure of the code that is generated for each of the possible stages in which the final exit branch could be placed for a three-stage schedule. In Figures 5.12(a), (b), and (c), the final exit is scheduled at the end of stages A, B, and C, respectively ($\theta = 0, 1$, and 2).

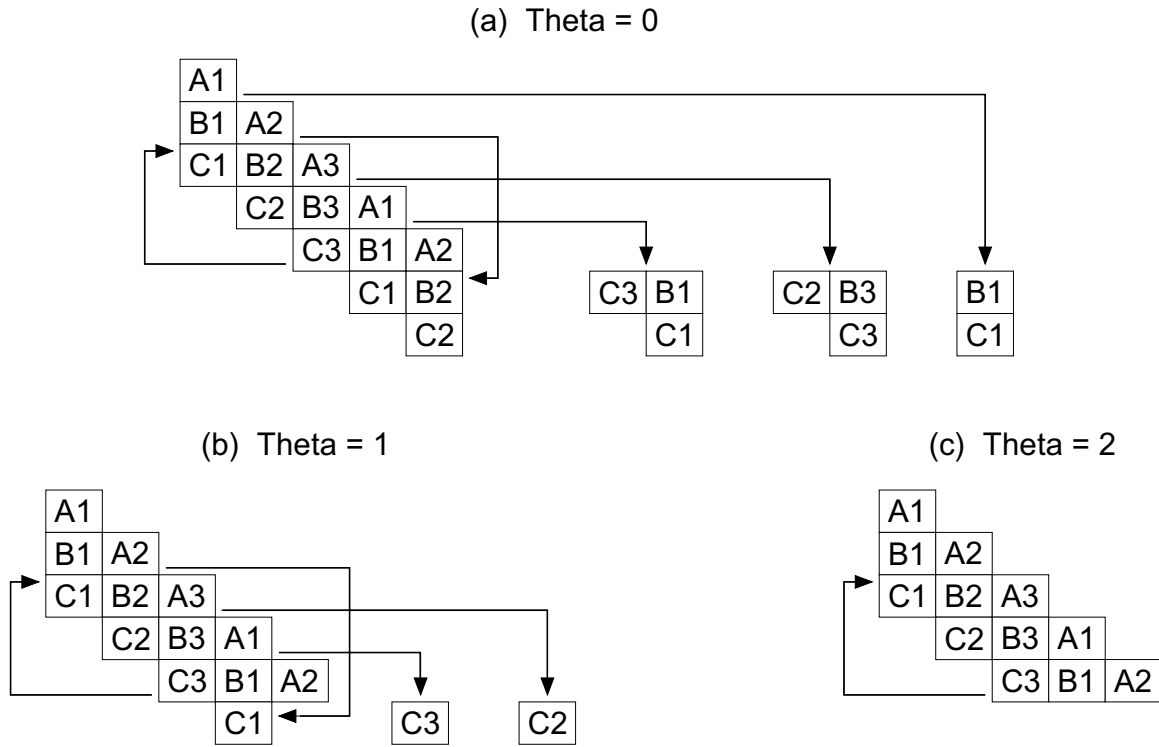


Figure 5.12 Code Generation Scheme for Single Exit Loops.

The arrows (except the back-edge) represent control transfers from the prologue and kernel to the epilogues shown. Because the final exits are scheduled at the end of the stage, the arrows originate very close to the bottom of each row of squares. There is now a separate epilogue for each code version plus an epilogue reached from the prologue. The latter epilogue has fewer

columns because for this exit, there is no second last iteration. The epilogue for the last exit from the prologue contains exactly the same code as the epilogue for the exit associated with the fall-through path of the loop back branch at the end of the kernel. These two epilogues have been merged into one. There are other examples of redundant code in the epilogues, which can be removed in practice. This redundancy has been left in the figure for clarity. Although it is not explicitly shown, at the end of each epilogue there exists code to move any live-out values to the registers in which the code outside the loop expects to find them and to jump to the original target block of the exit.

By comparing Figures 5.12(a) and (b), one can see how the structure of the generated code changes when the loop back branch is scheduled at the end of stage B instead of stage A. Because the loop back branch is executed one stage later, there are fewer stages left to execute in the epilogues for the last iteration and its predecessors. Thus, the epilogues all have one fewer rows. The one speculative iteration that is in progress when the loop exits is aborted; therefore, there are fewer columns in each epilogue. The number of exits from the prologue is reduced by one, so one of the epilogues has disappeared altogether. In general, when the loop back branch is placed in stage θ instead of stage 0, the θ rightmost columns of each epilogue are removed, corresponding to the θ aborted speculative iterations [53]. The resulting epilogues have $\mathbf{SC} - \theta - 1$ rows where $\mathbf{SC}$ is the stage count.

5.1.4 A code generation scheme for multiple exit loops

Figure 5.13 shows the structure of a single iteration of a multiple exit loop before and after modulo scheduling and illustrates the two key differences between a final exit and an early exit with respect to code generation. Before modulo scheduling, the final exit is always at the end

of the loop body and the early exit is somewhere in the middle. The grey box between the exits represents the instructions that follow the early exit and that should not be executed when the early exit is taken. After modulo scheduling, the final exit is always placed at the end of a stage as described earlier. The early exit is scheduled somewhere in the middle of the stage. In Figure 5.13(b), some grey instructions have been moved upward across the early exit and some white instructions have been moved downward across both exits. When instructions are moved upward across an exit, they are executed speculatively. When they are moved downward across an exit, they are also copied to an epilogue associated with the exit.

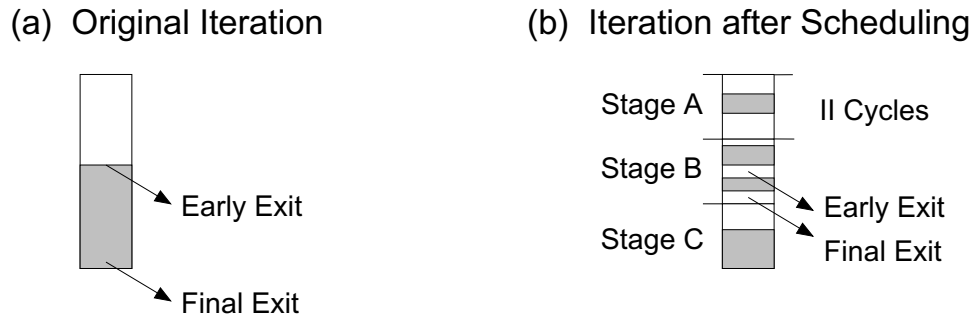


Figure 5.13 Structure of a Single Iteration of a Multiple Exit Loop.

Figure 5.14 illustrates the changes to the code generation scheme for multiple exit loops. The figure assumes a loop with two exits, where both the early exit and the loop back branch are scheduled in the same stage as in Figure 5.13. In Figures 5.14(a), (b), and (c), the final exit is scheduled in stages A, B, and C, respectively. There are now more exits from the modulo scheduled loop and thus more epilogues. The arrows associated with the early exits originate very close to the top of each row and have dashed lines to distinguish them from the final exits.

The epilogues for the final exits are all the same as in Figure 5.12. The epilogues for the early exits contain one more row than the epilogues for the final exits, because the early exit is

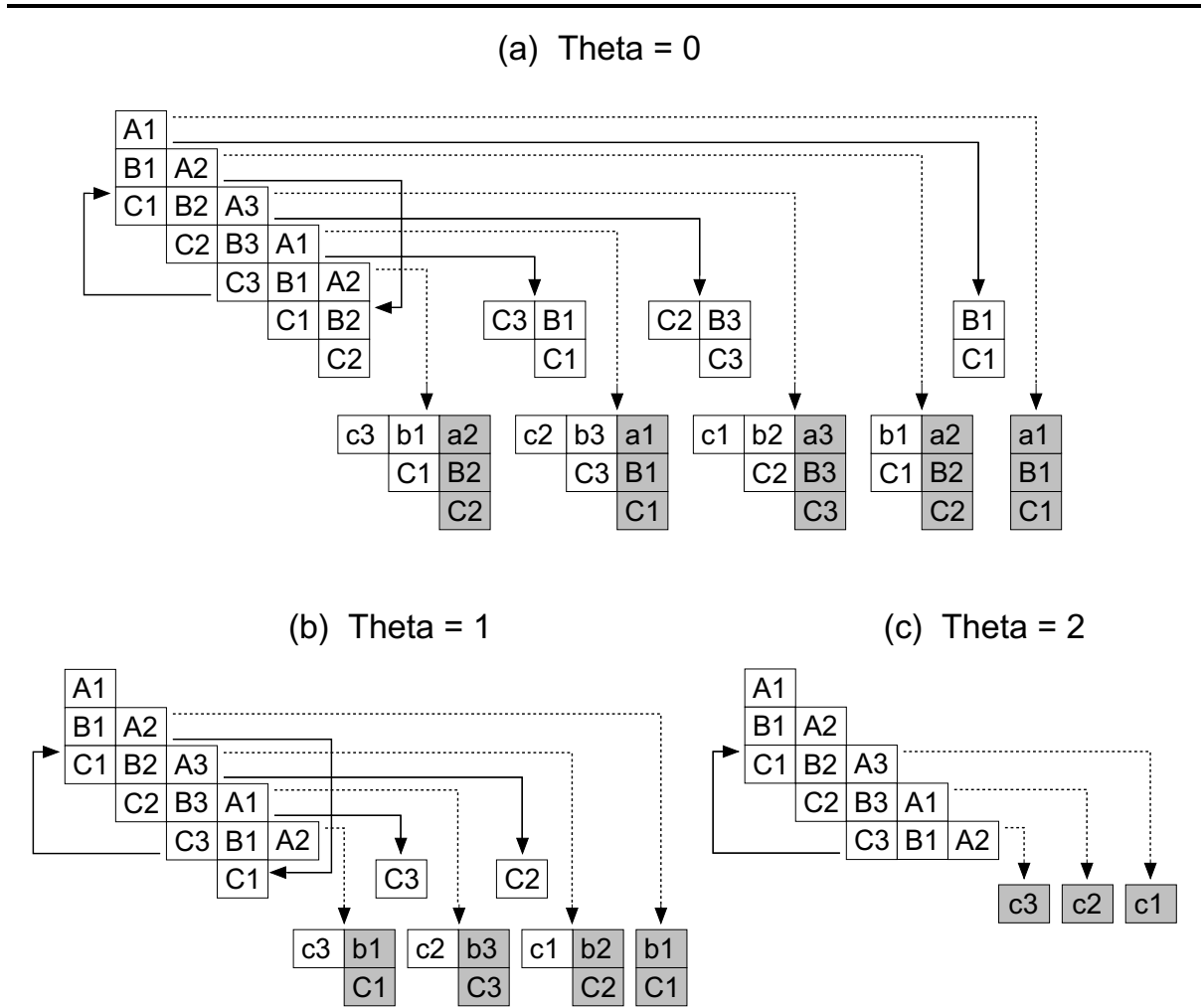


Figure 5.14 Code Generation Scheme for Multiple Exit Loops.

from the middle of a stage, i.e., it is from the middle of a row in the kernel or prologue. When the early exit is taken, the remaining portion of the row must be executed in the epilogue. Thus, the remainder of the row containing the exit branch is copied to the epilogue. In Figure 5.14, small letters are used to denote a partial stage resulting from an exit branch from the middle of a stage.

Special consideration must be given to the last iteration. The columns of the epilogues corresponding to the last iteration are highlighted in grey. Only the instructions from the last

iteration that appeared before the exit branch in the original loop body should be executed. Assume the basic blocks in a superblock are assigned numerical IDs starting sequentially from zero. Define the *home block* for an instruction to be the basic block in which the instruction resides in the original loop body. For an early exit, an instruction from the remaining stages of the last iteration is copied to the epilogue only if the ID of its home block is less than or equal to the home block ID of the exit branch.

Figure 5.15 shows the algorithm for generating an epilogue for an exit branch. The algorithm starts with the instructions following the exit branch and copies rows of instructions from the unrolled kernel to the epilogue, wrapping around the kernel until the last row of the epilogue is complete. Instructions are not copied if they are from iterations after the last or from the last iteration and appeared after the exit branch in the original loop body. The algorithm as shown assumes a processor that does not have branch delay slots. The following paragraphs describe the data structures and concepts needed to understand the algorithm. The term *stage* refers to the stage in which an instruction is placed in the schedule for a single iteration.

The unrolled kernel is divided into sections of Π cycles each called *kernel rows*. There are **kmin** rows where **kmin** is the degree of unrolling of the kernel. Each row contains a linked list of the instructions contained in that row. The data structure for each instruction contains a pointer to an information structure which contains among other items: the stage in which the instruction is scheduled, the instruction's home block ID, and the row of the kernel that contains the instruction.

There are $\mathbf{SC} - \theta$ rows in each epilogue (numbered zero to $\mathbf{SC} - \theta - 1$) where θ is the stage in which the exit branch is scheduled. Row zero is the partial row and is empty for a final exit (the linked list for each kernel row ends with a final exit). In row zero of the epilogue, the

```

Algorithm gen_epi(exit branch) {

    create epilogue block
    theta = exit->info->stage
    row = exit->info->row
    exit_home_block = exit->info->home_block

    /* Generate all rows of epilogue */
    for (epi_row = 0 to stage count - theta - 1) do {

        /* Determine where to start copying */
        if (epi_row == 0)
            /* Potentially a partial row. If exit is a final exit, exit->next_op
               is NULL and no instructions will be copied to the partial row. */
            oper = exit->next_op
        else /* Full row */
            oper = kernel[row]->first_op

        /* Generate one full or partial row */
        while (oper != NULL) {
            oper_stage = oper->info->stage
            oper_home_block = oper->info->home_block

            /* Copy instruction if it is from an iteration previous to the last
               iteration, or if it is from the last iteration and appears before
               the exit branch in the original loop body */
            if ( (oper_stage > epi_row + theta) or
                (oper_stage == epi_row + theta and
                 oper_home_block <= exit_home_block) ) {
                new_op = copy_operation(oper)
                insert_op_after(epilogue->last_op, new_op)
            }
            oper = oper->next_op
        }
        row = (row + 1) mod kmin /* Rotate through the rows of the kernel */
    }

    /* insert moves at end of epilogue for variants that are live out of exit */
    insert_moves_for_live_variants(epilogue, exit)

    /* Last exit branch falls through to epilogue */
    if (exit is not last exit in unrolled kernel) {
        jump = create_jump_to_target_of_exit_branch
        insert_op_after(epilogue->last_op, jump)
        make_epilogue_block_the_target_of_exit_branch
    }
}

```

Figure 5.15 Epilogue Generation Algorithm.

last iteration is executing in stage θ , since that is the stage containing the exit branch. In row **epi_row** of the epilogue, the last iteration is executing in stage **epi_row** + θ . Instructions from stages less than **epi_row** + θ must be from iterations after the last, and thus are not copied.

For simplicity, the algorithm shown generates correct epilogues for exits from the kernel, but not for exits from the prologue. In practice, the algorithm contains additional code to map an exit in the prologue to the corresponding exit in the kernel. The prologue is generated in a similar manner to the epilogues, by copying selected instructions from the rows of the unrolled kernel. Mapping a prologue exit to a corresponding kernel exit facilitates the copying of rows for the epilogue. Also in practice, if the epilogue is for an exit from the prologue, the algorithm does not copy an instruction that is from a later stage than the stage that the very first iteration is executing. Such instructions correspond to the non-existent iterations prior to the first one.

The code generation scheme is now applied to the example loop. The schedule for a single iteration of the example loop contains three stages. Stage A consists of instructions 1 and 5 from the original loop (see Figure 5.9). Stage B contains instruction 2. Instructions 3 (early exit), 4, and 6 (final exit) are in stage C. The code scheme in Figure 5.14(c) is similar to what would be generated for the example loop. Because of the dependence structure of the loop, there is no opportunity for downward code motion across the early exit branch. Thus, when the early exit branch is taken there are no remaining instructions from the last iteration that appeared before the exit branch in the original loop body and the shaded epilogues are empty.

5.1.5 Insertion of moves for live-out values

As mentioned earlier, code must be appended to the end of each epilogue to move the values that are live-out of the corresponding exit into the register in which the code outside the loop

expects to find them. For single exit superblock loops, without EVRs, the value used outside the loop must have been defined in the last iteration. Thus, for each final exit, the instructions from the last iteration are examined in the corresponding epilogue and in the kernel. If the value produced by the instruction is live-out and the destination register is not the one expected outside the loop, a move instruction is inserted at the end of the epilogue.

For multiple exit loops, the procedure is the same for the final exits. However, for the early exits there is an additional consideration. The live-out value could be defined in the last iteration by one of the instructions that preceded the exit branch in the original loop body, or it could be defined in the second-to-last iteration by one of the instructions that followed the exit branch in the original loop body. Thus, the last iteration is examined for instructions that originally resided in the same or earlier home block as the early exit branch, and the second-to-last iteration is examined for instructions that originally resided in a later home block than the exit.

In the example loop, when the early exit (instruction 3 from Figure 5.3) is taken, the live-out values of r34 and r4 are from the second-to-last iteration and the live-out value of r12 is from the last iteration. There are no values live-out of the final exit. Figure 5.16 shows the code generated for the example loop using the multiple epilogue code scheme of Figure 5.14(c). The instructions have again been renumbered. The moves for the live-out values (instructions 25, 27, 28, and 30) are also shown.

The code is laid out as it would be by the IMPACT compiler. The blocks labeled **Pro** and **L1** are the prologue and the unrolled kernel, respectively. The blocks labeled **LE1**, **LE3** and **LE5** are epilogues. The block immediately following the kernel is the epilogue reached by falling through the loop back branch. Block **L3** is the original fall-through path of the loop.

Inst.	Assembly	Cycle
1	Pro: r122 = MEM(r34+r8)	0
2	r342 = r34 - 4	1
3	r13 = MEM(r122+0)	2
4	r123 = MEM(r342+r8)	2
5	r343 = r342 - 4	3
6	L1: beq (r6 r13) LE1	0
7	r4 = r4 - 1	0
8	r13 = MEM(r123+0)	0
9	r12 = MEM(r343+r8)	0
10	r34 = r343 - 4	1
11	bgt (0 r4) L3	1
12	beq (r6 r13) LE3	2
13	r4 = r4 - 1	2
14	r13 = MEM(r12+0)	2
15	r122 = MEM(r34+r8)	2
16	r342 = r34 - 4	3
17	bgt (0 r4) L3	3
18	beq (r6 r13) LE5	4
19	r4 = r4 - 1	4
20	r13 = MEM(r122+0)	4
21	r123 = MEM(r342+r8)	4
22	r343 = r342 - 4	5
23	ble (0 r4) L1	5
24	jump L3	0
25	LE1: r12 = r122	0
26	jump L2	0
27	LE3: r12 = r123	0
28	r34 = r342	0
29	jump L2	0
30	LE5: r34 = r343	0
31	jump L2	0

L3:

Figure 5.16 Final Assembly Code for the Example Loop.

Label **L2** is the start of block Y. The epilogues for the final exits (instructions 11, 17, and 23) are all empty because no code was moved downward across the loop back branch and there are no virtual registers live-out of the final exits. Rather than branching to empty epilogues, the final exits branch directly to **L3**. The exception is the loop back branch, which falls through into its epilogue and then jumps to **L3**.

The early exits (instructions 6, 12, and 18) all require moves for one or more of the live virtual registers, so all branch to epilogues. As discussed in Chapter 4, when renaming, one of the names used is the original virtual register name. Thus, if the live-out value is already in the correct register, a move is not necessary. This is the case for r34 in epilogue **LE1** and r12 in epilogue **LE5**. A jump is placed at the end of each early exit epilogue to transfer control to Block Y.

Also as discussed earlier, the virtual registers are renamed during modulo variable expansion such that the uses of a live-in virtual register in the first iteration refer to the original virtual register name. Thus, no moves are required for live-in values. For example, virtual register r34 is live-in and the first iteration in the prologue uses r34 (instructions 1 and 2) rather than one of the renamed versions (r342 and r343).

5.2 Experimental Results

This section reports experimental results on the applicability of modulo scheduling to control-intensive non-numeric programs. The results were obtained using the IMPACT compiler and the modulo scheduler implemented as part of the work for this dissertation. The modulo scheduler has been used to pipeline loops for high issue rate versions of the PA-RISC (in this dissertation) and SPARC architectures. Loops are eligible for modulo scheduling if they are inner loops (outer loops may become inner loops after superblock formation), are single basic block or superblock loops, and do not contain function calls on the included path (function calls may be excluded from the loop by superblock formation, enabling modulo scheduling).

The target processors for these experiments are multiple issue processors with issue rates between 4 and 8 with varying resource constraints and load latencies. Table 5.2 shows the

functional unit mix for each processor. All processors are assumed to have 64 integer registers and 64 double-precision floating-point registers. The latencies used for processor A and the base processor are those of the HP PA7100 processor. For processors B, C and D, the load latency is increased to reflect the higher clock speeds of future high-performance designs. Processor A is similar to the microprocessors available today. Processor B is a more aggressive version of A with a higher clock frequency and a longer load latency. Processor C is the IMPACT group's projection of what a typical ILP processor might look like in the future and processor D is a more aggressive version of that.

Table 5.2 Processor Characteristics for Modulo Scheduling Experiments.

Name	Number of					Load Latency
	Issue Slots	Integer ALUs	Memory Ports	Branch Units	FP ALUs	
Base	1	1	1	1	1	2
A	4	2	2	1	1	2
B	4	2	2	1	1	4
C	8	4	3	2	2	3
D	8	4	3	2	2	6

All speedups are reported over the single-issue pipelined base processor. For the base processor, ILP optimizations and modulo scheduling are not applied. For the multiple issue processors, code is generated three ways, once without modulo scheduling, once with modulo scheduling of only the single basic block loops, and once with modulo scheduling of the superblock loops using the techniques described in this chapter. All the code that is not software pipelined is scheduled using acyclic superblock scheduling [6]. None of the loops are unrolled before acyclic scheduling or modulo scheduling. The effects of unrolling prior to scheduling and performance

comparisons of modulo scheduling versus acyclic scheduling of unrolled loops are the subject of later chapters.

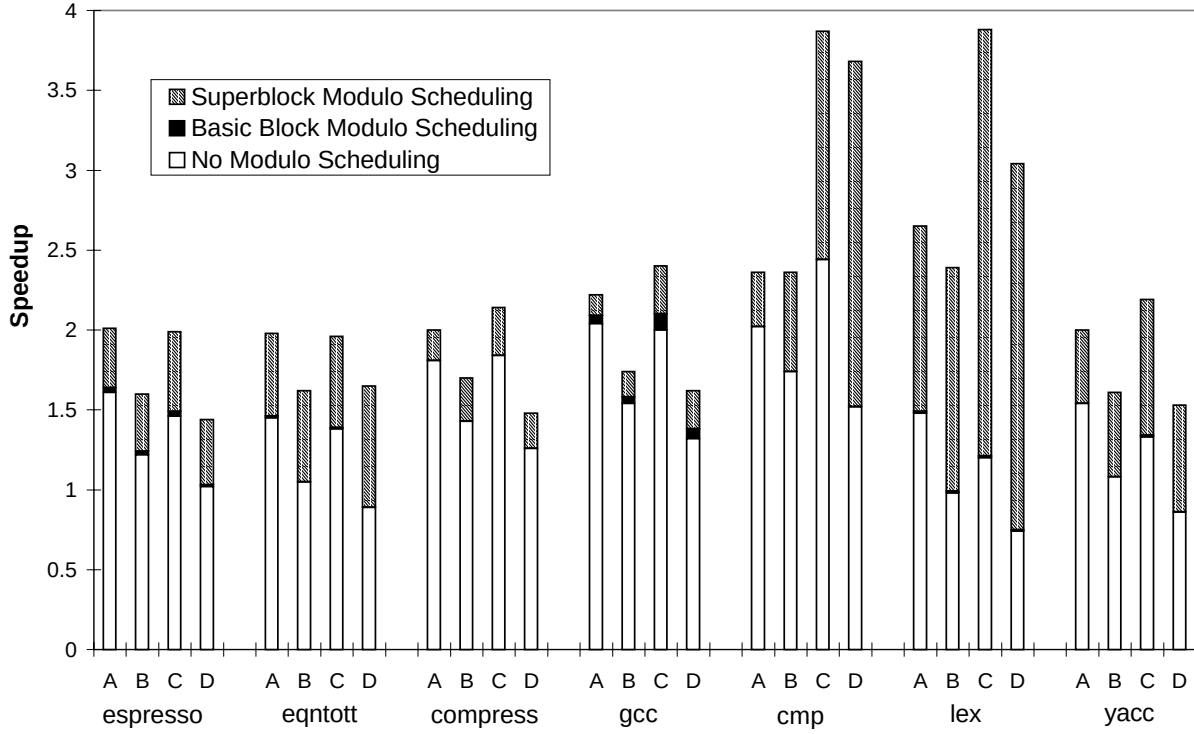


Figure 5.17 Speedup over Single-Issue Processor with and without Modulo Scheduling.

The execution times of the whole programs are calculated using scheduler cycle counts for each basic block and profile information. A 100% cache hit rate is assumed. To insure accuracy, the benchmarks are profiled after all transformations. The profiling is done by instrumenting the target (virtual) processor's assembly code and then emulating it on an HP PA-RISC workstation. This execution produces benchmark output which is used to verify the correctness of the target processor's assembly code.

The benchmarks chosen for the experiments are the seven SPEC CINT92 and Unix programs from Table 5.1 (espresso, eqntott, compress, gcc, cmp, lex, and yacc) that spend the most time in basic block and superblock loops, loops to which modulo scheduling is applied. For all of

the chosen programs, over 40% of the dynamic instructions were in such loops. A total of 368 loops were modulo scheduled.

Figure 5.17 shows the speedup results. The white part of the bars shows the speedup over the base processor when acyclic scheduling is applied to all of the code. For all of the benchmarks, the performance declines (in terms of cycles - when the faster clock cycle is factored in, the performance would be higher) for the more aggressive processors because of the longer load latency. Without overlapping the iterations, the ILP that can be exploited is limited. The black part of the bars shows the slightly increased performance when modulo scheduling is applied to the single basic block loops. For all of the benchmarks except *gcc*, less than 6% of the dynamic instructions are in basic block loops. Thus only a slight performance improvement can be expected. The benchmark *gcc* spends about half as much time (14%) in single basic block loops as it does in superblock loops and shows speedups of about 5%.

The cross-hatched part of the bars shows the increased performance when superblock modulo scheduling is applied to the eligible loops. Modulo scheduling almost doubles the performance of *lex* for the four-issue processors and more than triples performance for the eight-issue processors. As was shown in the case study, there is very limited ILP within a single iteration of the loops in that program. Modulo scheduling provides good speedup across all the benchmarks and processors. In particular, speedups of 25% or more are obtained across all the processors for *espresso*, *eqntott*, *cmp*, *lex*, and *yacc*. For the most aggressive processor, performance is improved by 30% or more for all the benchmarks except *compress* and *gcc*.

With superblock modulo scheduling, the performance of *cmp*, *lex*, and *yacc* no longer declines for the more aggressive processors. More ILP is being exploited by overlapping the loop

iterations. The results clearly show that modulo scheduling, using the techniques described in this chapter, is applicable to control-intensive, non-numeric programs.

5.3 Summary

This chapter has described a set of methods that allows effective modulo scheduling of loops with multiple exits. These methods can be used to allow modulo scheduling of the selected paths of loops with arbitrary control flow. A case study was presented to show how these methods enable modulo scheduling to be effectively applied to control-intensive non-numeric programs. Performance results for several SPEC CINT92 benchmarks and Unix utility programs demonstrated that modulo scheduling can significantly accelerate loops in this class of programs.

Previous work by this author has shown that unrolling prior to modulo scheduling improves performance for numeric programs [58]. Unrolling enables additional optimization and an effective II that is not an integer. The next chapter investigates the benefits of unrolling prior to modulo scheduling and presents performance results on the benefit of unrolling for control-intensive integer programs.

CHAPTER 6

UNROLLING-BASED OPTIMIZATION FOR MODULO SCHEDULED LOOPS

The previous two chapters have concentrated on the design of loop scheduling algorithms that produce a high quality schedule for the given input code and a target processor. In this chapter, the characteristics of the code given to the scheduler are examined. The instruction types, resources required, and the dependence pattern of the computation in the loop body can all be modified by program transformations. Specifically, this chapter describes the advantages of unrolling the loop before modulo scheduling and of performing optimizations that reduce the resource requirements and the height of critical paths in loops.

There are two sources of motivation for this work. First, the II for the loop is restricted to be an integer. If the lower bound on the II computed before scheduling is not an integer, the performance degradation caused by rounding it up to an integer can be reduced by unrolling the loop [10]. A related restriction is that the minimum possible value for II is one. This limits the performance of a modulo scheduled loop to one iteration per cycle. By unrolling the loop and applying optimizations, it is possible to complete multiple iterations per cycle given sufficient execution resources. These restrictions have been known for some time, but the benefits of unrolling prior to modulo scheduling have never been quantified.

The second source of motivation comes from comparisons of modulo scheduling with global acyclic scheduling of an unrolled loop body. Without unrolling prior to modulo scheduling, it is possible for the modulo scheduled loop to perform worse than the acyclicly scheduled loop.

The acyclic scheduler is not required to initiate the iterations within the unrolled body at a constant rate nor to generate the same schedule for each of those iterations. Thus, it can achieve an effective Π which is not an integer. Additionally, unrolling exposes new opportunities for optimizations that reduce the resource requirements and dependence height. The benefits of unrolling and these unrolling-based optimizations have been quantified for acyclic scheduling [24], but have not been measured for modulo scheduling.

This chapter describes the benefits of unrolling and a set of optimizations for unrolled loops that have been implemented in the IMPACT compiler. Unrolling and unrolling-based optimizations are applied to the loops in seven SPEC CINT92 and Unix and the achieved speedup is measured.

This chapter is organized as follows: Section 6.1 introduces the case study that is used in this chapter and Section 6.2 reviews the related work. Section 6.3 describes the benefits of unrolling and the unrolling-based optimizations. The performance results are reported in Section 6.4.

6.1 Case Study

There is another important use for the lower bounds on the Π that was not described in Chapter 3. They can be used to guide the optimization process [49]. Optimizations change the bounds and can target the bound that represents the performance bottleneck. For example, there is no point in doing optimizations to reduce dependence constraints if that bound is already lower than the resource-constrained lower bound. The bounds can be used to estimate what the final performance of the software pipeline will be after an optimization.

Figure 6.1 shows the source and assembly code to compute a vector-matrix product. The inner loop produces one element of the result vector C and is used as an example throughout this chapter. The assembly code shown assumes that classic loop optimizations, such as induction variable elimination and global variable migration, have been performed. Registers r4-r7 and f1-f6 are integer and floating-point registers, respectively. For this example, an eight-issue processor is assumed with no restrictions on the combination of instructions that may be issued. The issue slots are thus the most heavily used resources resulting in a ResMII of one.

(a) Original Loop

do i = 1,n
C(i) = 0.0
do j = 1,m
C(i) = C(i) + A(j) * B(j,i)
end do
end do

**(b) Assembly Code
For Inner Loop**

Instr.	Assembly	Register Contents:
L1: 1	f3 = MEM(r8+r4)	f1 = C(i)
2	f5 = MEM(r2+r4)	f3 = A(j)
3	f6 = f3 * f5	f5 = B(j,i)
4	f1 = f1 + f6	r8 = &A(1)
5	r4 = r4 + 4	r2 = &B(1,i)
6	ble (r4 r7) L1	r4 = 4*j
		r7 = 4*m
		r9 = 4*i

Figure 6.1 Example Vector-Matrix Product Loop.

Figure 6.2 shows the dependence graph for the example loop. The data and control dependences are shown with solid and dashed lines, respectively. All anti- and output dependences have been removed assuming that modulo variable expansion will be performed after scheduling. Each node is numbered with the id (from Figure 6.1(b)) of the instruction it represents. Each arc is labeled delay and distance of the dependence, respectively. The delays shown are those of the HP PA-RISC PA7100.

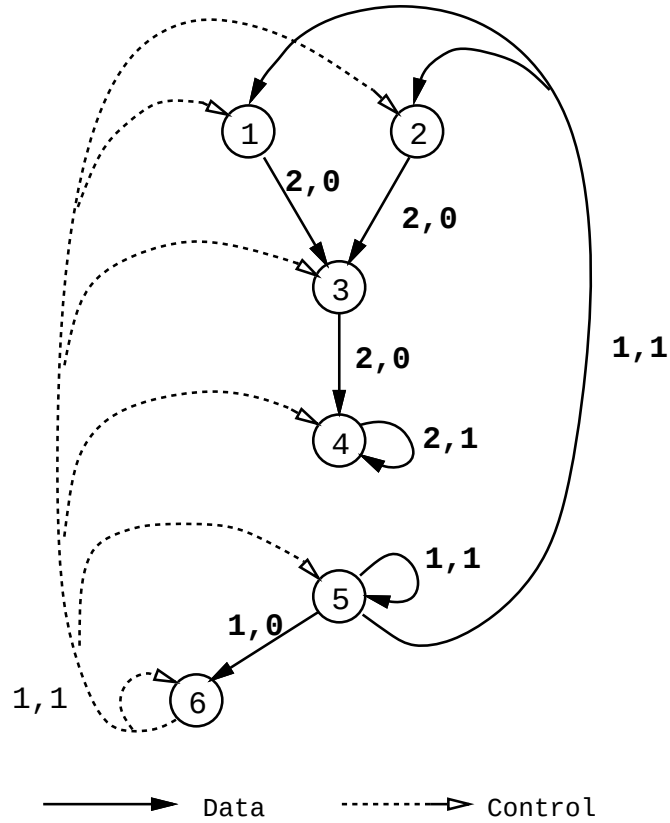


Figure 6.2 Dependence Graph for Example Loop.

The longest cycle in the example graph is from instruction 4 to itself. The cycle has a delay of 2 and spans from one iteration to the next, resulting in a RecMII of 2. It may appear that there is another recurrence with a delay of two involving instructions 5 and 6, the update of

the loop counter and the loop back branch. Such a recurrence would create a RecMII of 2 for every DO loop. As described in the last chapter, the cross-iteration control dependence from instruction 6 to instruction 5 is removed, allowing instruction 5 to be speculatively executed.

6.2 Related Work

There has been extensive prior work on the optimization of loops, some of it targeted directly at software pipelined loops. The work has focused either on reducing the number of instructions (the resource requirement) in the loop body or on changing the dependence pattern to create more ILP. Traditional loop optimizations try to reduce the number and complexity of the instructions in the loop body [19]. These optimizations indirectly reduce the dependence height of a critical path by reducing the number of instructions along the path.

The compiler for the Cydra-5 performed redundant load and store elimination across loop iterations [49]. This optimization reduces the number of memory ports used and reduces the dependence height when a load is on a critical recurrence path. The Cydra-5 compiler also performed *symmetric back-substitution* of data recurrences to reduce dependence height [49]. The compiler for the RS/6000 architecture performs an optimization called *predictive commoning* [59]. This optimization achieves an effect similar to redundant load elimination and common subexpression elimination across loop iterations for loops in which a sequence of values is computed and the value of each member of the sequence, except the first, is computed again in the next iteration. Unrolling is not required for this optimization but can be used to eliminate the copy instructions inserted by the optimization.

Recently, transformations have been proposed which require that the loop be unrolled. *Blocked back-substitution* [60] unrolls the loop b times and reduces the RecMII by a factor of

b. Control recurrences within loops can also be accelerated by a factor of b by unrolling the loop b times and applying techniques similar to blocked back-substitution [33]. Loop unrolling is required for these techniques because the code is optimized asymmetrically such that all iterations in the unrolled loop body do not execute the same code.

Optimizations that unroll loops and then reduce the height of dependence chains associated with induction and accumulator variables have been implemented in the IMPACT compiler [24]. These can be viewed as special cases of symmetric back-substitution. These techniques have been evaluated in the context of global acyclic scheduling of the unrolled loop body, but not modulo scheduling.

Unrolling can also enable optimizations that reduce the number of instructions executed per iteration. Compilers that unroll loops before applying a global acyclic scheduling algorithm take advantage of these optimization opportunities [5, 24]. However, the potential benefits of these optimizations for loops that are modulo scheduled have not been fully explored.

6.3 Unrolling-Based Optimization

6.3.1 Loop unrolling

The requirement that the II be an integer can result in less than full utilization of processor resources, or the allowance of more cycles than necessary for the completion of a recurrence. For example, in Figure 6.1(b), there are six instructions in the loop body and it was assumed that the processor has eight issue slots. The $ResMII$ is one (rounded up from 0.75) and two issue slots are wasted every cycle. Unrolling allows a smaller non-integral effective $ResMII$ to be achieved. For example, if the loop in Figure 6.1(b) is unrolled four times, the $ResMII$ for the unrolled loop body is three and the effective $ResMII$ for each iteration within the loop body

is 0.75. Unrolling helps to reduce the degradation by creating larger loop bodies that require more resources and a larger ResMII. The larger the ResMII, the smaller the degradation caused by rounding it up to the next integer.

The RecMII will not be an integer if the delay of the limiting cycle is not a multiple of the distance of the cycle. Unrolling helps because it reduces the distance of the recurrence. This makes the RecMII larger, decreasing the degradation caused by rounding it up to the next integer. If the distance of the cycle becomes one, the RecMII becomes an integer. For example, a recurrence with a distance of three becomes a recurrence with a distance of one if the loop is unrolled three or more times.

Figure 6.3 shows the way the IMPACT compiler unrolls Fortran-style DO-loops. The example loop of Figure 6.1 has been unrolled three times. Using the terminology of [33], iterations of the unrolled loop are defined as the *major iterations* and the iterations of the original loop as the *minor iterations*.

An optimization has already been applied to remove the loop exit branches from the unrolled loop body. A simple check is done before the loop (and for each major iteration) to ensure that there are at least three more minor iterations to be executed. Two extra copies of the original loop body are inserted after the loop at label L2. These copies are executed when the trip count is not a multiple of three. If the loop is unrolled u times, there are $u - 1$ copies of the original loop body inserted after the unrolled loop. If the code expansion is too great, the remaining iterations can be re-rolled into a loop at the cost of lower performance for those iterations.

This type of unrolling has two benefits. First, the number of branch unit resources required for each major iteration is reduced from three to one. Second, the control dependences associated with the exit branches are removed, allowing the possibility of executing more than one

	r3 = r7 - 8
	bgt (r4 r3) L2
L1:	f3 = MEM(r8+r4)
	f5 = MEM(r2+r4)
	f6 = f3 * f5
	f1 = f1 + f6
	r4 = r4 + 4
	f3 = MEM(r8+r4)
	f5 = MEM(r2+r4)
	f6 = f3 * f5
	f1 = f1 + f6
	r4 = r4 + 4
	f3 = MEM(r8+r4)
	f5 = MEM(r2+r4)
	f6 = f3 * f5
	f1 = f1 + f6
	r4 = r4 + 4
	ble (r4 r3) L1
	bgt (r4 r7) L3
L2:	f3 = MEM(r8+r4)
	f5 = MEM(r2+r4)
	f6 = f3 * f5
	f1 = f1 + f6
	r4 = r4 + 4
	bgt (r4 r7) L3
	f3 = MEM(r8+r4)
	f5 = MEM(r2+r4)
	f6 = f3 * f5
	f1 = f1 + f6
	r4 = r4 + 4
L3:	

Figure 6.3 Example Loop after Unrolling Three Times.

minor iteration per cycle (resource permitting) without speculation. This can also be viewed as control height reduction [33] where the conditions under which the minor iterations execute have been collapsed into the single check to see if there are at least u minor iterations remaining.

II now refers to the initiation interval for the major iterations. Define II_{eff} , the effective initiation interval for the minor iterations, to be II/u . For the example loop, the $ResMII_{eff}$ falls from 1 to 0.66 as a result of unrolling and exit branch removal.

6.3.2 IMPACT unrolling-based optimizations

This section describes the remaining unrolling-based optimizations done by the IMPACT compiler [24] and their effect on the MII. Figure 6.4 shows the effect of induction variable optimizations applied to the unrolled loop body. In Figure 6.4(a), the unrolled loop body without exit branches has been extracted from the code in Figure 6.3. The induction variable increment instructions have been highlighted.

(a) Unrolled Loop Body	(b) After Induction Rewriting	(c) After Induction Expansion	(d) After Induction Elimination
L1: <pre> f3 = MEM(r8+r4) f5 = MEM(r2+r4) f6 = f3 * f5 f1 = f1 + f6 r4 = r4 + 4 f3 = MEM(r8+r4) f5 = MEM(r2+r4) f6 = f3 * f5 f1 = f1 + f6 r4 = r4 + 4 f3 = MEM(r8+r4) f5 = MEM(r2+r4) f6 = f3 * f5 f1 = f1 + f6 r4 = r4 + 4 ble (r4 r3) L1 </pre>	L1: <pre> r80 = r8 r20 = r2 f3 = MEM(r80+0) f5 = MEM(r20+0) f6 = f3 * f5 f1 = f1 + f6 r80 = r80 + 4 r20 = r20 + 4 f3 = MEM(r80+0) f5 = MEM(r20+0) f6 = f3 * f5 f1 = f1 + f6 r80 = r80 + 4 r20 = r20 + 4 f3 = MEM(r80+0) f5 = MEM(r20+0) f6 = f3 * f5 f1 = f1 + f6 r80 = r80 + 4 r20 = r20 + 4 ble (r80 r31) L1 r4 = r80 - r8 </pre>	L1: <pre> r81 = r8 r82 = r8 + 4 r83 = r8 + 8 r21 = r2 r22 = r2 + 4 r23 = r2 + 8 f3 = MEM(r81+0) f5 = MEM(r21+0) f6 = f3 * f5 f1 = f1 + f6 f3 = MEM(r82+0) f5 = MEM(r22+0) f6 = f3 * f5 f1 = f1 + f6 f3 = MEM(r83+0) f5 = MEM(r23+0) f6 = f3 * f5 f1 = f1 + f6 r81 = r81 + 12 r82 = r82 + 12 r83 = r83 + 12 r21 = r21 + 12 r22 = r22 + 12 r23 = r23 + 12 ble (r83 r33) L1 r4 = r81 - r8 </pre>	L1: <pre> r83 = r8 + 8 r23 = r2 + 8 f3 = MEM(r83-8) f5 = MEM(r23-8) f6 = f3 * f5 f1 = f1 + f6 f3 = MEM(r83-4) f5 = MEM(r23-4) f6 = f3 * f5 f1 = f1 + f6 f3 = MEM(r83+0) f5 = MEM(r23+0) f6 = f3 * f5 f1 = f1 + f6 r83 = r83 + 12 r23 = r23 + 12 ble (r83 r33) L1 r4 = r83 - r8 - 8 </pre>

Figure 6.4 Example Loop after Induction Variable Optimization.

In the original loop body, the vector and matrix are addressed using two different bases (r8 and r2, respectively) and a common offset (r4). This reduces the number of induction instructions for good performance in the original loop. In Figure 6.4(b), the address calculations have been rewritten in preparation for later induction variable elimination (described below).

The objective of the rewriting is to specify each memory address using only one operand. This frees up the other address source operand in each load for use by induction variable elimination. In the unrolled loop, after the rewriting, the vector and the matrix are each addressed using a separate induction variable (r80 and r20 respectively).

In Figure 6.4(c), induction variable expansion [24] has been applied to the loop. For the each induction variable (r80 and r20), three temporary induction variables have been created, one for each definition of the original induction variables. The new induction variables are now incremented by three times the original increment. They are initialized to the initial series of values in the preheader.

In the example, induction variable expansion reduces the delay for the cycle involving the three increments of r80 from three to one. Induction variable expansion is a special case of symmetric back-substitution [60] where the reduction is a simple addition of a loop invariant.

Figure 6.4(d) shows the loop after the application of induction variable elimination. Elimination of induction variable r22 is done as follows. First, r22 is rewritten in terms of r23: $r22 = r23 - 4$. Then this definition of r22 is combined with the load which uses r22 in the next iteration [61]. After combining, there are no further uses of r22 and its defining instruction can be removed.

Induction variable elimination significantly reduces the number of integer ALU and issue slot resources required by making use of the separate effective address addition available in most load/store units and the two address operands of the load/store instructions. For the example loop, the number of induction instructions has been reduced from three (in Figure 6.4(a)) to two (in Figure 6.4(d)). If the loop is unrolled eight times, the number of induction instructions

is reduced from eight to two. Loop unrolling is required for this type of induction variable elimination because the minor loop iterations are no longer identical.

Figure 6.5 shows the effect of two more optimizations. In Figure 6.5(b), accumulator variable expansion has been applied [24]. In the original loop, f1 is an accumulator variable. Three temporary accumulators (f11, f12, f13) have been created, one for each definition of the original accumulator. The temporary accumulators are summed after the loop.

(a) After Induction Optimization

	r83 = r8 + 8 r23 = r2 + 8
L1:	f3 = MEM(r83-8) f5 = MEM(r23-8) f6 = f3 * f5 f1 = f1 + f6 f3 = MEM(r83-4) f5 = MEM(r23-4) f6 = f3 * f5 f1 = f1 + f6 f3 = MEM(r83+0) f5 = MEM(r23+0) f6 = f3 * f5 f1 = f1 + f6 r83 = r83 + 12 r23 = r23 + 12 ble (r83 r33) L1
	r4 = r83 - r8 - 8

(b) After Accumulator Expansion

	f11 = 0.0 f12 = 0.0 f13 = 0.0 r83 = r8 + 8 r23 = r2 + 8
L1:	f3 = MEM(r83-8) f5 = MEM(r23-8) f6 = f3 * f5 f11 = f11 + f6 f3 = MEM(r83-4) f5 = MEM(r23-4) f6 = f3 * f5 f12 = f12 + f6 f3 = MEM(r83+0) f5 = MEM(r23+0) f6 = f3 * f5 f13 = f13 + f6 r83 = r83 + 12 r23 = r23 + 12 ble (r83 r33) L1
	f1 = f11 + f12 f1 = f1 + f13 r4 = r83 - r8 - 8

(c) After Renaming

	f11 = 0.0 f12 = 0.0 f13 = 0.0 r83 = r8 + 8 r23 = r2 + 8
L1:	1 f31 = MEM(r83-8) 2 f51 = MEM(r23-8) 3 f61 = f31 * f51 4 f11 = f11 + f61 5 f32 = MEM(r83-4) 6 f52 = MEM(r23-4) 7 f62 = f32 * f52 8 f12 = f12 + f62 9 f33 = MEM(r83+0) 10 f53 = MEM(r23+0) 11 f63 = f33 * f53 12 f13 = f13 + f63 13 r83 = r83 + 12 14 r23 = r23 + 12 15 ble (r83 r33) L1
	f1 = f11 + f12 f1 = f1 + f13 r4 = r83 - r8 - 8

Figure 6.5 Example Loop after Accumulator Expansion and Renaming.

In the example, accumulator variable expansion reduces the delay for the cycles involving f1 by a factor of three. Accumulator variable expansion is also called interleaved reduction [5] and riffing of reductions [49]. Since accumulator variable expansion reassociates the terms in the

accumulation, which may change the results of floating-point accumulations, its use is under user control.

In Figure 6.5(c), variable renaming has been applied to produce the final optimized version of the loop. The three iterations of the loop are now independent. Overall, the $RecMII_{eff}$ for the example loop was reduced by a factor of three: from 2 to 0.66 for the eight-issue processor. The $ResMII_{eff}$ was reduced from 1 to 0.66 due to the removal of the loop exit branches described earlier. Removal of both data and control dependences and reduction of resources were all necessary to achieve these improvements in the MII_{eff} . For example, without the removal of control dependences and the reduction in resources, the MII_{eff} is limited to 1 for this loop. If this example loop is unrolled eight times, an MII_{eff} of 0.25 can be achieved given sufficient resources. This is eight times the performance of the original loop in Figure 6.1!

Although not shown in the running example, unrolling also allows redundant load and store elimination, common subexpression elimination, and copy propagation across minor iterations. These optimizations, along with accumulator and induction variable expansion, can be done without unrolling if the compiler representation has support for *expanded virtual registers* (EVRs) [54]. If the compiler supports EVRs, but the architecture does not have support for rotating registers [46], the loop must still be unrolled to allow modulo variable expansion. Even in this case, EVRs allow the optimizations to be performed without having to first decide how much to unroll the loop. For compilers that do not support EVRs, unrolling is the only way to perform load and store elimination and common subexpression elimination across iterations without introducing copy instructions and to perform accumulator and induction variable expansion. However, for any compiler, the exit branch removal and induction variable elimination described in this chapter require unrolling, as does blocked back-substitution.

6.4 Experimental Results

In this section, experimental results are reported on the importance of the unrolling-based optimizations for modulo scheduled loops. The results are obtained using the IMPACT compiler and the seven SPEC CINT92 and Unix benchmarks described earlier. Heuristics are used to guide the amount of unrolling. Loops are categorized into one of four classes based on the size of the loop body, the total execution count for the loop body, and the average iterations per invocation. For example, small loops with a large number of average iterations are unrolled the most.

The target processors for these experiments are multiple issue processors with issue rates between four and eight with varying resource constraints and load latencies. The functional unit mix for each processor is the same as for the earlier experiments. The table is shown here again (Table 6.1) for ease of reference. All processors are assumed to have 64 integer registers and 64 double-precision floating-point registers. The latencies used for processor A and the base processor are those of the HP PA7100 processor. For processors B, C and D, the load latency is increased to reflect the higher clock speeds of future high-performance designs.

Table 6.1 Processor Characteristics for Unrolling Experiments.

Name	Number of					Load Latency
	Issue Slots	Integer ALUs	Memory Ports	Branch Units	FP ALUs	
Base	1	1	1	1	1	2
A	4	2	2	1	1	2
B	4	2	2	1	1	4
C	8	4	3	2	2	3
D	8	4	3	2	2	6

All speedups are reported relative to the single-issue base processor. For the base processor, ILP optimizations and modulo scheduling are not applied. For the multiple issue processors, code is generated two ways, once with and once without the unrolling-based optimizations for the software pipelined loops.

The execution time of the programs is calculated using scheduler cycle counts for each basic block and profile information. The benchmarks are profiled after all transformations to insure accuracy. The profiling is done by instrumenting the target (virtual) processor's assembly code and then emulating it on an HP PA-RISC workstation. This execution produces benchmark output which is used to verify the correctness of the code transformations.

Figure 6.6 shows the speedup for each target processor with and without the unrolling-based optimizations for software pipelined loops over the single-issue base processor. The unrolling-based optimizations produce excellent performance improvement for *eqntott*, *cmp*, and *lex* and moderate improvement for *yacc*, especially for the eight-issue processors. The speedups for *eqntott*, *cmp*, and *lex* range from 15-22% for processor C. The loops in *cmp* contain many induction instructions. The number of dynamic induction instructions is decreased when the loop is unrolled. For the four-issue processors, the performance is limited by the lack of branch resources. The eight-issue processors are less limited by branch resources and show the benefit of eliminating induction instructions. For *lex* and *yacc* the reduction in resources and the achievement of an effective Π that is not an integer are important for all of the processors.

Unrolling has little effect on the performance of *espresso*, *compress*, and *gcc*. Unrolling does improve the performance of many individual loops in these benchmarks, but unfortunately, these loops do not account for a significant fraction of the execution time. Also many of the loops have a short trip count and the unrolling simply allows overlap, within a major iteration,

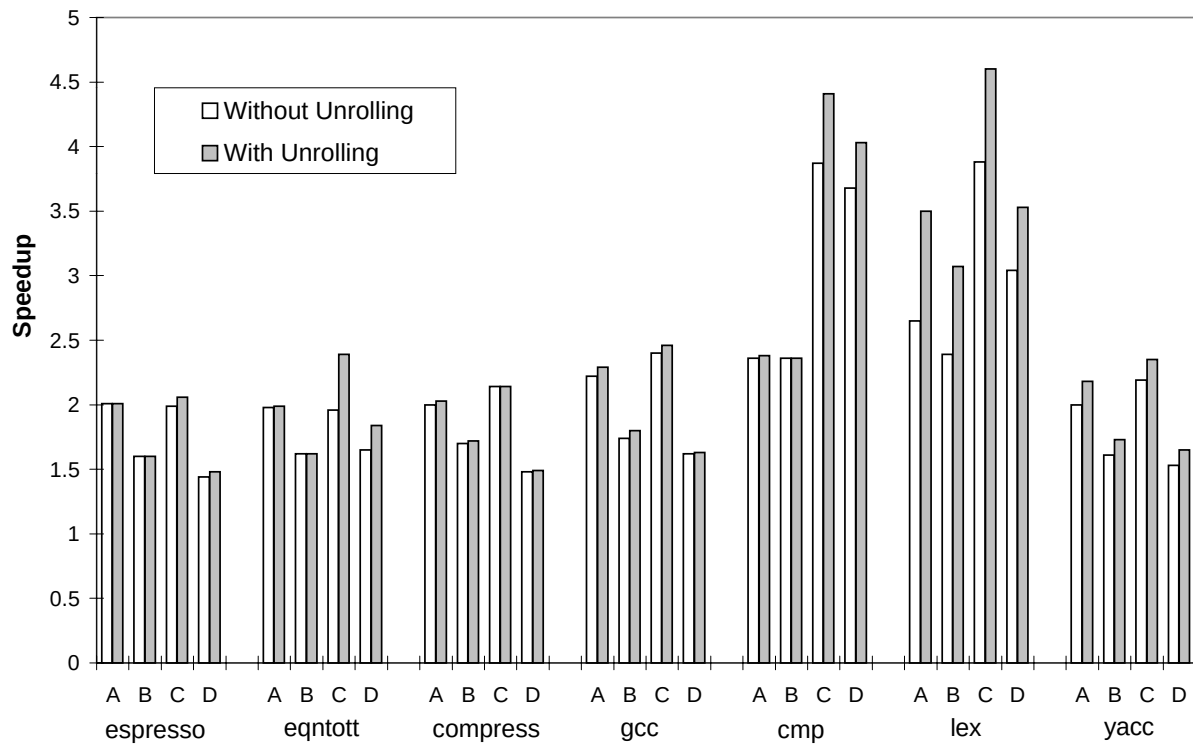


Figure 6.6 Speedup over Single-Issue Processor with and without Unrolling.

of minor iterations which would have been overlapped by modulo scheduling without unrolling anyway.

6.5 Summary

This chapter described a set of unrolling-based optimizations which reduce resource requirements and the height of critical paths in software pipelined loops. Unrolling is the only way to reduce the the effective number of loop-back branches executed per iteration and to allow optimizations which asymmetrically optimize the loop iterations. Unrolling also helps to reduce the degradation caused by rounding the MII up to the nearest integer. Experimental results

showed that unrolling prior to modulo scheduling improves the performance of the benchmarks used in this dissertation.

CHAPTER 7

COMPARISON OF MODULO SCHEDULING AND ACYCLIC SCHEDULING

This chapter compares modulo scheduling using the techniques developed in this thesis with the global acyclic scheduling approach with full support for unrolling-based optimization in both cases. This is the first such comparison of the two techniques within the same compiler framework.

Heuristics are used to guide the amount of unrolling. The modulo scheduled loops are usually unrolled half as much as the acyclicly scheduled loops. The rationale for this is that for modulo scheduling, unrolling is required only for optimization, but not to amortize the loss of overlap across the back edge. There are two exceptions to this. First, the modulo scheduled loops are unrolled the same amount as for acyclic scheduling for very small, very critical loops with high optimization potential. Second, the loops are unrolled one less time if the unroll amount is equal to the issue rate and doing so improves resource utilization. In a loop, there are some number $\mathbf{x}$ of instructions that are duplicated when the loop is unrolled and a few that are not (typically a few induction instructions and the loop back branch). When the unroll amount is equal to the issue rate, the duplicated instructions completely fill $\mathbf{x}$ cycles worth of issue slots. Then one cycle is filled with the few instructions that are not duplicated. For a wide issue processor, many slots in that cycle may be wasted. When $\mathbf{x}$ is less than the issue rate but greater than or equal to the number of non-duplicated instructions, higher performance

can be achieved by reducing the unroll amount by one and using the extra **x** issue slots to accommodate the instructions that are not duplicated. The result is less wasted issue slots.

For the experiments in this chapter, the benchmark set is compiled three different times. The first time, the loops are unrolled and superblock scheduling is applied. None of the loops are modulo scheduled. The second time, the eligible loops are modulo scheduled without prior unrolling. All the other loops are unrolled and acyclicly scheduled. The third time, the modulo scheduled loops are also unrolled prior to scheduling to gain the benefits of unrolling-based optimization. The four processor models are the same as those used for the results in the earlier chapters (see Table 6.1).

Figure 7.1 compares the results of these three compilations. Each bar shows the speedup over the base single-issue processor. The results show that for five of the seven benchmarks, *eqntott*, *compress*, *cmp*, *lex* and *yacc*, modulo scheduling achieves significantly higher performance than acyclic scheduling. For *cmp*, *lex*, and *yacc*, prior unrolling is necessary to allow modulo scheduling to surpass the performance of acyclic scheduling.

For *espresso* and *gcc*, acyclic scheduling and modulo scheduling provide equivalent performance (although modulo scheduling provides other benefits as will be shown shortly). The performance of many loops in these two benchmarks was improved by modulo scheduling, but these loops were not executed frequently enough to affect the overall benchmark execution time. Also, many of the loops in these two benchmarks have a small number of iterations on average, limiting the benefit of modulo scheduling.

Figure 7.2 shows the distribution of the per loop speedups for the 368 loops that were modulo scheduled. The results in Figure 7.2 and the rest of the individual loop results in this chapter are based on processor C (see Table 6.1). Loops for which the two techniques produced

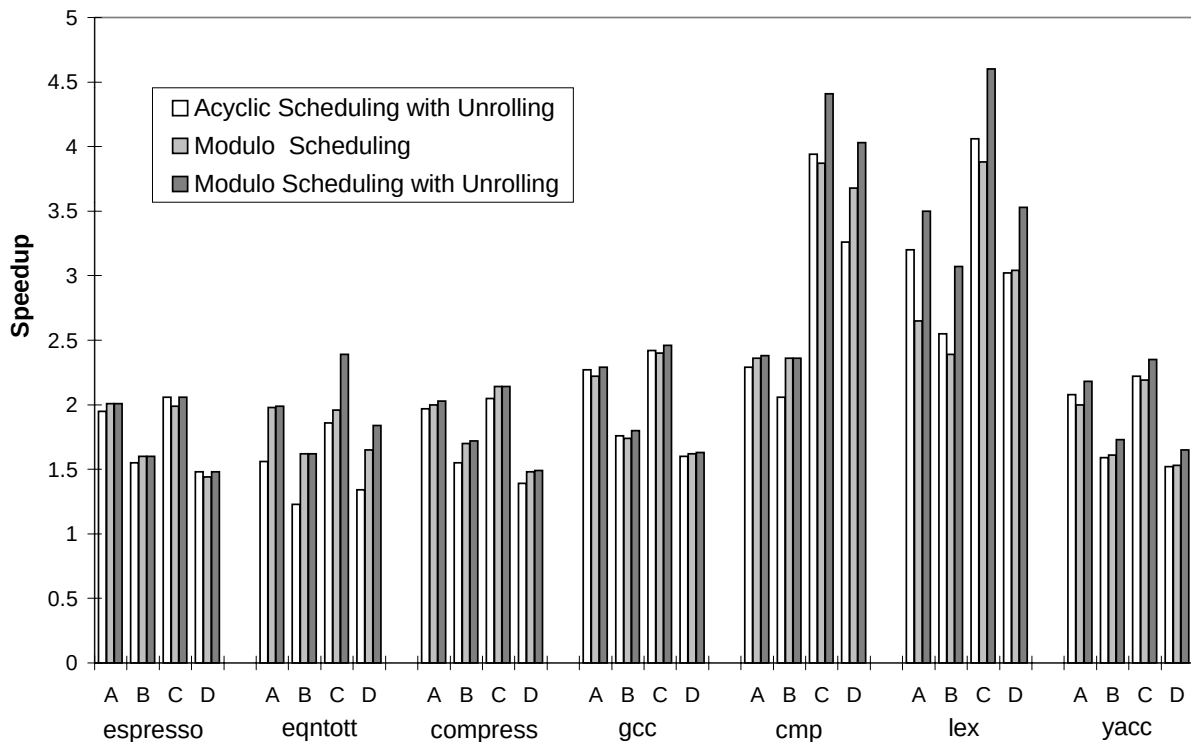


Figure 7.1 Speedup over Single-Issue Processor for Acyclic and Modulo Scheduling.

the same performance are not shown. The speedups are computed using the actual execution time for the individual loop. This takes into account the startup overhead in the prologue and the execution time of the epilogues. This provides an accurate execution time measurement even for short trip count loops. A loop that originally had a moderate to long trip count prior to optimization and scheduling may have a short to moderate trip count after unrolling for optimization and modulo variable expansion. Almost all previous studies of modulo scheduling and other software pipelining techniques have reported the speedup in terms of only the achieved II for the loop. This assumes that the trip count will be very long and is not appropriate for control-intensive non-numeric programs.

Each bar shows the number of loops that achieved the corresponding percentage speedup. The range of percentages is divided into segments of 5% each. For example, the bar labeled 30 corresponds to the loops which achieved between a 25 and 30% speedup over acyclic scheduling. A negative number means that the acyclic approach performed better than modulo scheduling by the corresponding percentage.

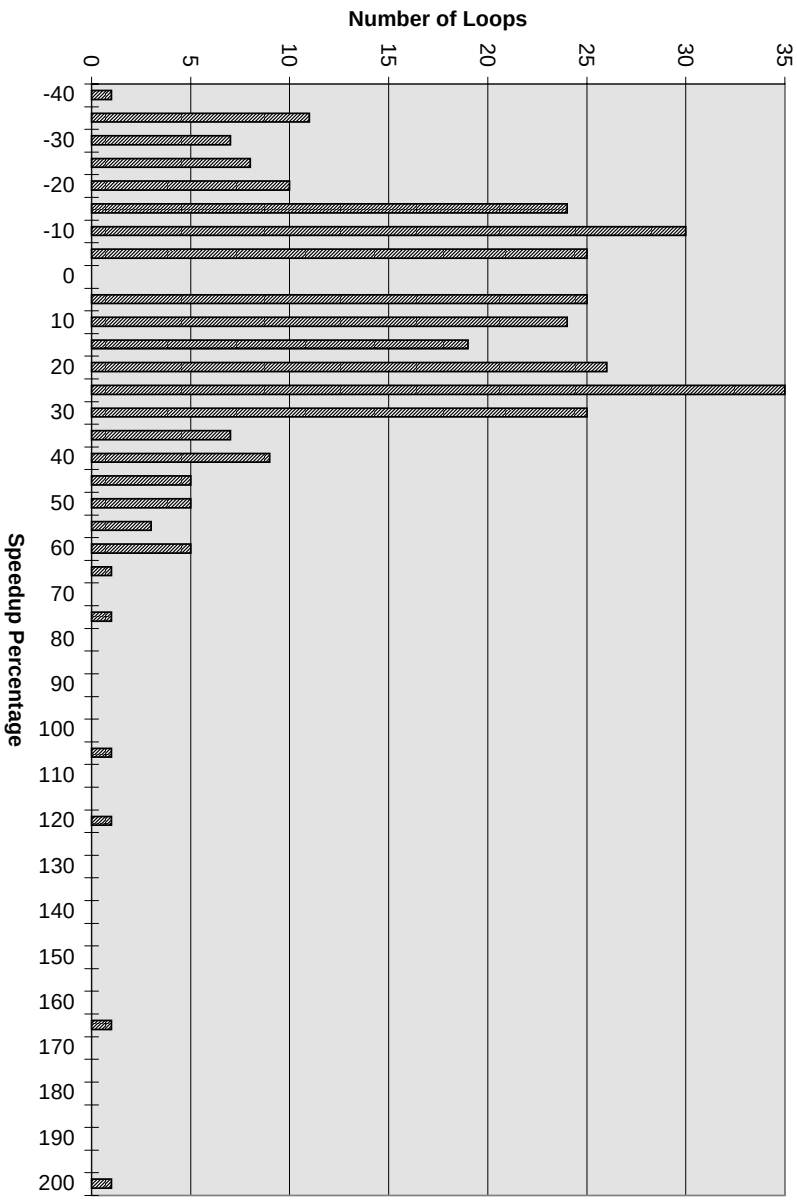


Figure 7.2 Distribution of Per Loop Speedups over Acyclic Scheduling.

Almost twice as many loops performed better with modulo scheduling than performed better with acyclic scheduling. Furthermore, the average percentage gain is higher for modulo scheduling. Most of the wins for acyclic scheduling are in the 5 to 15% range. For modulo scheduling there are also many gains in that range. In addition, there are a large number of

wins in the 20 to 30% range. The maximum loss is limited to less than 40%. By contrast, there are 19 loops for which modulo scheduling wins by 40% or more.

Figure 7.3 shows the distribution of the percentage of fewer registers used per loop. Each bar shows the number of loops for which the corresponding percent fewer registers were used. The range of percentages is again divided into segments of 5% each. Almost twice as many loops use fewer registers with modulo scheduling than use fewer registers with acyclic scheduling and the average decrease is more than the average increase.

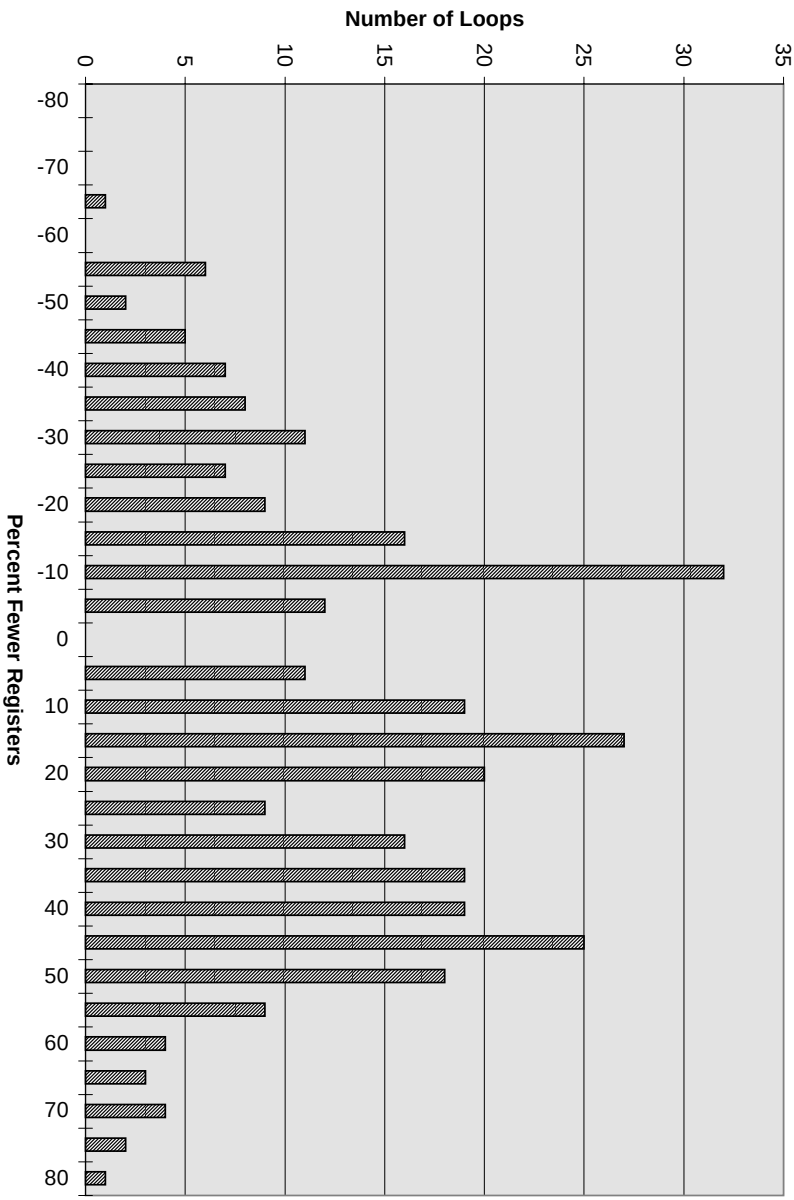


Figure 7.3 Improvement in Register Usage over Acyclic Scheduling.

Modulo scheduling decreases register usage for two partially overlapping reasons. First, the loops are unrolled only about one half as much. Thus the scheduler is working with a smaller number of instructions. Within the schedule for a single major iteration, an instruction

can only be moved among the minor iterations. For modulo scheduling, the number of minor iterations is smaller than for acyclic scheduling. Less code motion reduces the amount that a register lifetime can be stretched. Second, although modulo scheduling also overlaps the major iterations it cannot start a major iteration any sooner than II cycles after the previous one.

The intuition behind this can be seen in the following example. Assume a loop is unrolled eight times for acyclic scheduling, but only four times for modulo scheduling. If the computation chain for each minor iteration begins with a load, and each minor iteration is independent, the acyclic scheduler could start eight loads in the first cycle for a machine that could issue eight loads per cycle. Starting all eight iterations at once increases the register pressure, but may not yield any performance gain if the execution time of the major iteration is limited by a factor other than the issue time of the loads. For example, if each minor iteration also contains a branch and the machine can only execute two branches per cycle, a bottleneck exists through which only two iterations per cycle can pass. Under these circumstances, starting all eight iterations in the same cycle is pointless.

In contrast, the modulo scheduler would only start at most four loads in the first cycle. The next four loads would have to be scheduled II cycles later. In modulo scheduling, the maximum rate at which iterations can be executed (one per MII cycles) is computed up front. Iterations are scheduled at a consistent rate that is sustainable for the given resources and dependence structure.

For the results in Figure 7.3, the number of registers used is computed as follows. First, global register allocation is applied to the entire function assuming a very large set of physical registers so that there is no spilling. Then the set of registers actually referenced (read or

written) is computed. This represents an upper bound on the register pressure, since it is the actual number of registers that would be allocated for the variables in the loop.

This measurement is affected by the register reuse policy of the allocator and other influences on the allocation within the function but outside the loop. In the IMPACT compiler, the register allocator tries to reuse physical registers that it has already assigned to another lifetime. This reduces the total number of registers used, and can decrease the amount to code required for saving and restoring the registers. This tends to make the measurement more accurate. However, two lifetimes that do not interfere inside the loop may interfere outside the loop, requiring two separate registers to be used.

Figure 7.4 shows the result of another measurement of register pressure. To measure the

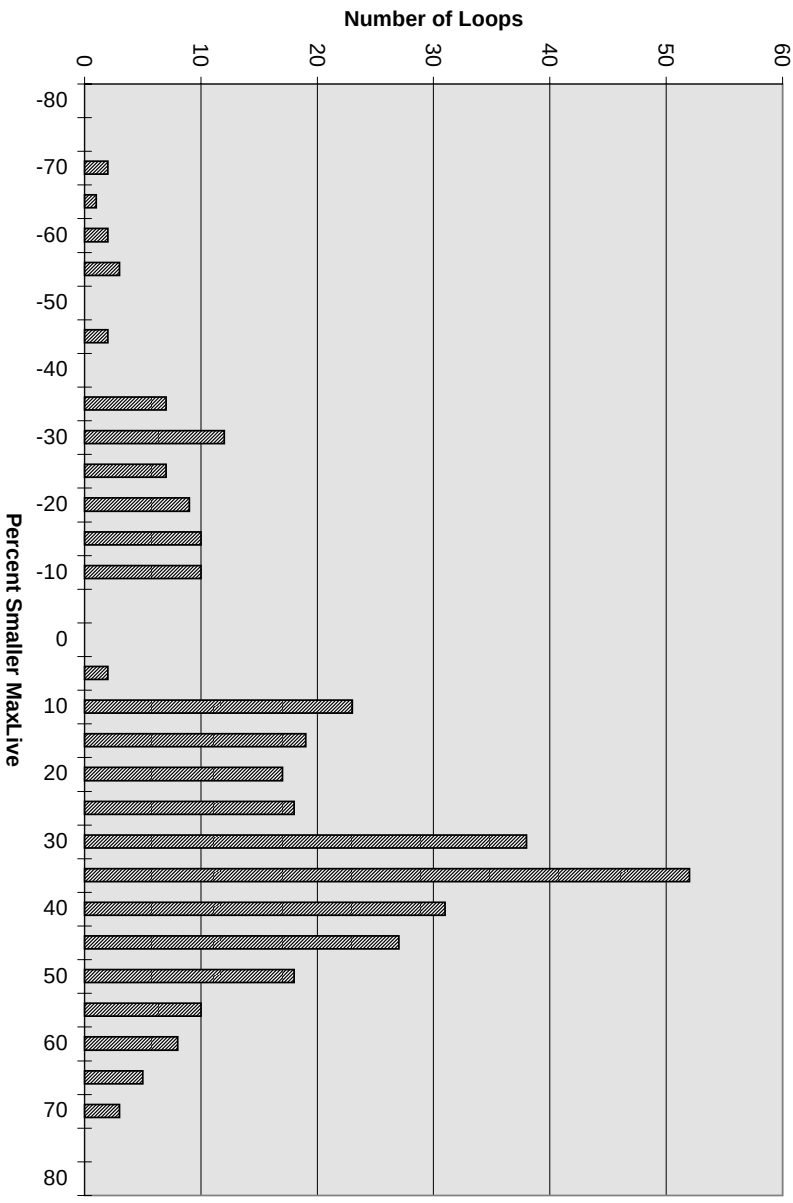


Figure 7.4 Improvement in MaxLive over Acyclic Scheduling.

register pressure within the loop alone, without any external influences, *MaxLive*, the maximum number of registers live at any cycle during the loop is computed. This is a lower bound on the number of registers that would be needed for the loop in the absence of external influences. The distribution for the percentage improvement in MaxLive is shown in Figure 7.4. The difference between modulo scheduling and acyclic scheduling is now more pronounced. Modulo scheduling produces a smaller MaxLive for 73% of the loops. It is expected that MaxLive is a more favorable measurement for modulo scheduling. It is possible for live ranges to interfere with each other in a cyclic manner such that the MaxLive lower bound is not achievable by any register allocator. Modulo scheduling increases the number of registers live across the back edge, increasing the chance of cyclic interference.

Figure 7.5 shows the distribution of the code size compared to that for acyclic scheduling.

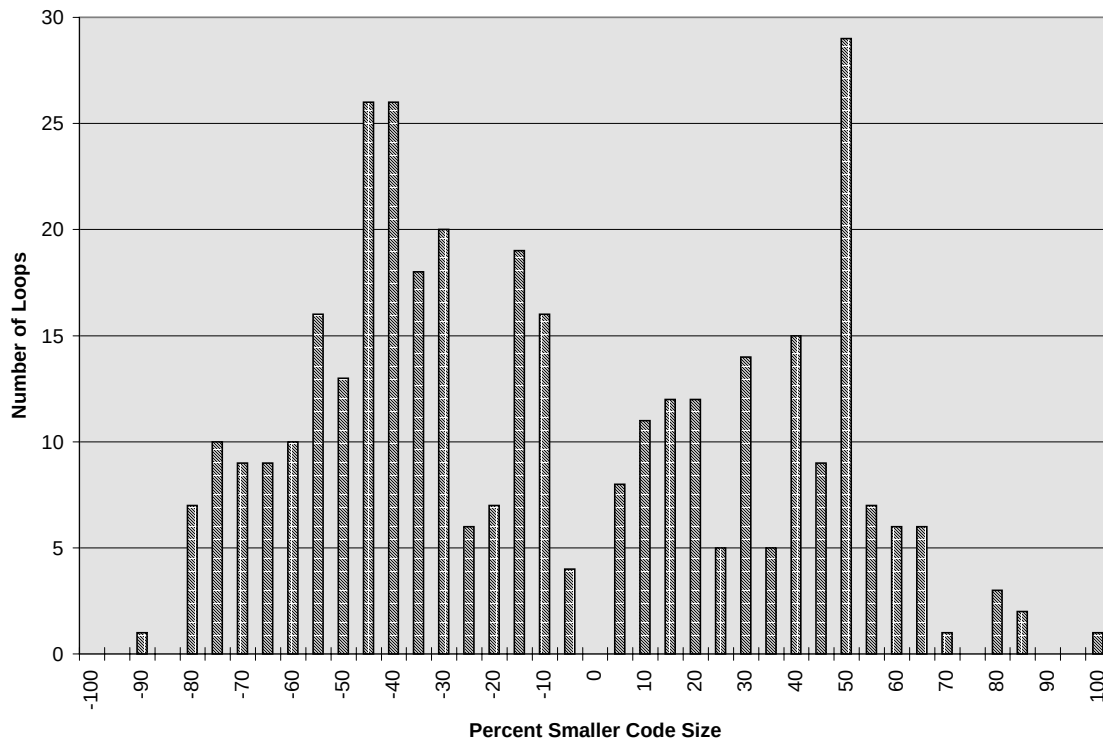


Figure 7.5 Code Size Compared to Acyclic Scheduling.

Modulo scheduling increases the code size for 59% of the loops. This is due to the kernel unrolling needed for modulo variable expansion, the prologue, and the multiple epilogues that are created for the code generation scheme used in the IMPACT compiler. Even though the loop body given to the modulo scheduler is usually smaller than that given to the acyclic scheduler, the code expansion due to the code generation scheme often (but not always) more than mitigates that advantage. However, for 40% of the loops this advantage is not mitigated and the code size is smaller with modulo scheduling.

Fortunately, hardware techniques have been developed that allow the creation of more space-efficient code generation schemes for modulo scheduled loops [53]. Figure 7.6 shows the distribution of the code size assuming that the processor contains hardware support to allow the

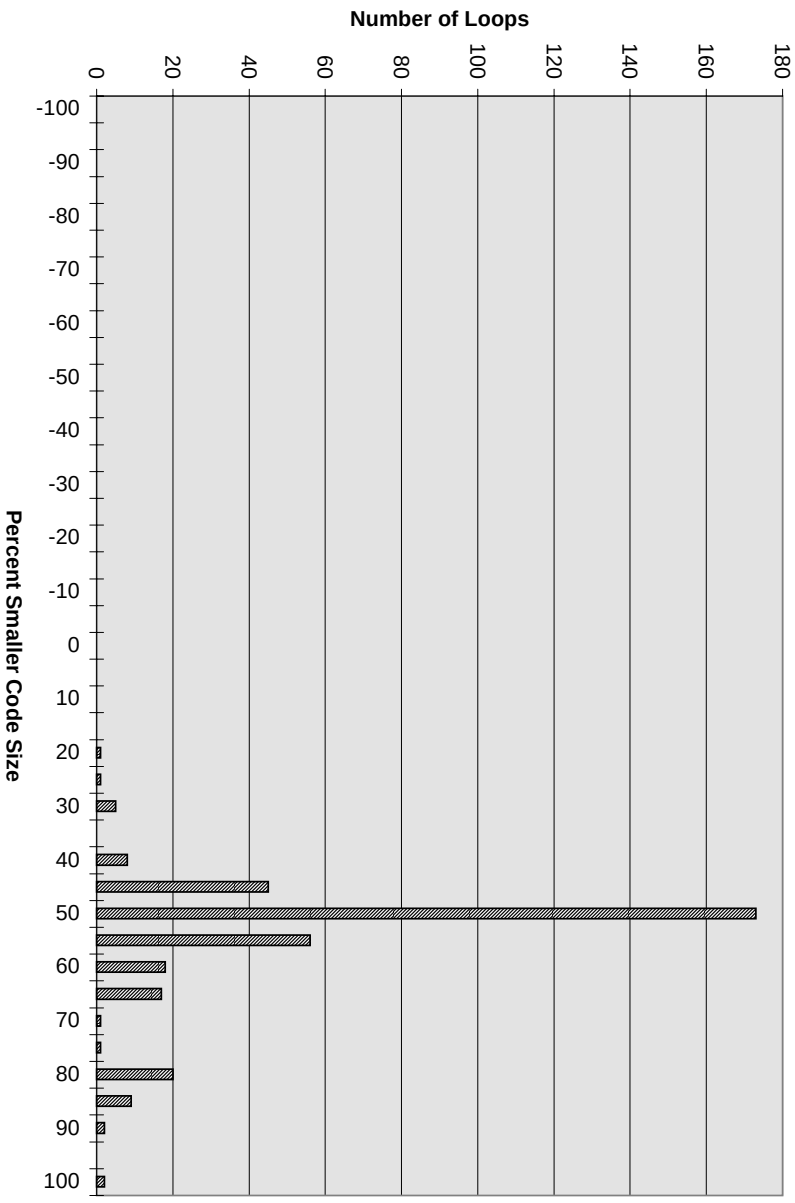


Figure 7.6 Improvement in Code Size with Kernel-Only Code.

generation of kernel-only code (see Chapter 3). In this case only the (non-unrolled) kernel contributes to the code size. With this support, the code size with modulo scheduling is always smaller. Note the very large number of loops where the code size is one-half that of acyclic scheduling. This occurs because loops are usually unrolled for modulo scheduling one-half the amount that they are for acyclic scheduling. The bars slightly greater than 50% represent the loops that are unrolled fewer times to allow better resource utilization.

Table 7.1 summarizes the statistics for the distributions presented in this chapter. The column labeled *Number Win* is the number of loops for which modulo scheduling performed better than acyclic scheduling for the metric of interest. For *Speedup*, this means a higher speedup. For the other metrics, it means reduced register pressure or code size. *Number Loss* means the number of loops for which modulo scheduling performed worse than acyclic scheduling. *Number Tie* means the two scheduling methods produced the same results. *Percent Win* is the percentage of loops for which modulo scheduling performed better than acyclic scheduling. *Max Win* is the maximum percentage by which modulo scheduling performed better than acyclic scheduling. *Max Loss* is the maximum percentage by which acyclic scheduling performed better than modulo scheduling. Note that the worst-case performance of modulo scheduling is always better than the worst-case performance for acyclic scheduling, indicating

Table 7.1 Summary of Distribution Statistics.

Metric	Number Win	Number Lose	Number Tie	Percent Win	Max Win	Max Loss	Average Win	Average Loss
Speedup	200	119	49	54%	196%	38%	23%	13%
Registers	206	116	46	56%	75%	62%	30%	19%
MaxLive	272	65	31	74%	84%	69%	31%	25%
Code Size	146	217	5	40%	95%	86%	35%	39%
KO Size	359	0	9	98%	95%	0%	51%	0%

a more stable performing loop scheduling method. For example, the maximum speedup with modulo scheduling is 196% while the maximum loss is only 38%. *Average Win* and *Average Loss* are the average gain (loss) for the loops for which modulo scheduling performed better (worse), respectively.

CHAPTER 8

CONCLUSION

This dissertation has proposed a set of methods that allow effective modulo scheduling of loops with multiple exits. These methods can be used to allow modulo scheduling of the selected paths of loops with arbitrary control flow. A case study was presented to show how these methods enable modulo scheduling to be effectively applied to control-intensive non-numeric programs. Performance results for several SPEC CINT92 benchmarks and Unix utility programs demonstrated that modulo scheduling can significantly accelerate loops in this class of programs.

This dissertation also described a set of unrolling-based optimizations which reduce resource requirements and the height of critical paths in software pipelined loops. Unrolling is the only way to reduce the effective number of loop-back branches executed per iteration and to allow optimizations which asymmetrically optimize the loop iterations. Unrolling also helps to reduce the degradation caused by rounding the MII up to the nearest integer. Experimental results showed that unrolling improves the performance of the benchmarks used in this dissertation.

Acyclic scheduling and modulo scheduling were also compared within in the same compiler framework, the first time that this has been done. The results show that modulo scheduling provides increased performance over acyclic scheduling with a reduction in register pressure. The code size with modulo scheduling is often, but not always, larger than for acyclic scheduling.

However, with appropriate hardware support, the code size is always less than with acyclic scheduling.

8.1 Future Work

There are four areas of potential future work on this topic. The first is modulo scheduling of short trip count loops. There are two steps that can be taken to attack this problem. First, the speculation should be controlled so that only the speculation necessary to achieve good long-trip count performance is done. Second, the number of iterations overlapped can be limited to trade off long-trip count performance for short-trip count performance. If there are still cases where acyclic scheduling performs better than modulo scheduling, work will be needed on sophisticated ways to combine or choose between the two techniques. For example, loop iterations could be peeled off to handle the short trip count cases and the software pipeline could be entered when the trip count is large enough.

The second area of future work is to make improvements in the methods used to determine the best number of times to unroll. Currently, the unroller uses heuristics and a very limited amount of information such as the number of instructions in the loop and the issue rate of the processor. This could be improved if the unroller used the detailed machine description information on the available resources and had knowledge of the effect of optimizations. Accurate decisions require that the loop body be in the form that the scheduler would see. This requires generation of the machine dependent code prior to loop unrolling. Also, the unrolling and optimization should take the ResMII and RecMII into account so that the correct tradeoff can be made between resource reducing and dependence height reducing optimizations.

Finally, modulo scheduling of hyperblock loops would allow the inclusion of more than one path for scheduling, but retain the ability to exclude undesirable paths. Inclusion of more paths would increase the trip count for loops which frequently exit the software pipeline at a branch to an excluded path.

REFERENCES

- [1] E. M. Riseman and C. C. Foster, "The inhibition of potential parallelism by conditional jumps," *IEEE Transactions on Computers*, vol. C-21, pp. 1405–1411, December 1972.
- [2] M. D. Smith, M. Johnson, and M. A. Horowitz, "Limits on multiple instruction issue," in *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 290–302, April 1989.
- [3] N. P. Jouppi and D. W. Wall, "Available instruction-level parallelism for superscalar and superpipelined machines," in *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 272–282, April 1989.
- [4] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Transactions on Computers*, vol. C-30, pp. 478–490, July 1981.
- [5] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donnell, and J. C. Ruttenberg, "The Multiflow Trace scheduling compiler," *The Journal of Supercomputing*, vol. 7, pp. 51–142, January 1993.
- [6] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The Superblock: An effective technique for VLIW and superscalar compilation," *The Journal of Supercomputing*, vol. 7, pp. 229–248, January 1993.
- [7] B. R. Rau and C. D. Glaeser, "Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing," in *Proceedings of the 20th Annual Workshop on Microprogramming and Microarchitecture*, pp. 183–198, October 1981.
- [8] K. Ebcioglu, "A compilation technique for software pipelining of loops with conditional jumps," in *Proceedings of the 20th Annual Workshop on Microprogramming and Microarchitecture*, pp. 69–79, December 1987.
- [9] A. Aiken and A. Nicolau, "Optimal loop parallelization," in *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pp. 308–317, June 1988.
- [10] B. R. Rau, "Iterative modulo scheduling: An algorithm for software pipelining loops," in *Proceedings of the 27th International Symposium on Microarchitecture*, pp. 63–74, December 1994.
- [11] S. I. Feldman, D. M. Gray, M. W. Maimore, and N. L. Schryer, "A Fortran-to-C converter," Computing Science Tech. Rep. 149, AT&T Bell Laboratories, Murray Hill, NJ, June 1990.

- [12] G. E. Haab, "Data dependence analysis for Fortran programs in the IMPACT compiler," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1995.
- [13] D. M. Gallagher, "Memory disambiguation to facilitate instruction-level parallelism compilation," Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1995.
- [14] B.-C. Cheng, "Pinline: A profile-driven automatic inliner for the IMPACT compiler," M.S. thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1997.
- [15] B. T. Sander, "Performance optimization and evaluation for the IMPACT X86 compiler," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1995.
- [16] W. F. Dugal, "Code scheduling and optimization for a superscalar x86 microprocessor," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1995.
- [17] V. Kathail, M. S. Schlansker, and B. R. Rau, "HPL PlayDoh architecture specification: Version 1.0," Tech. Rep. HPL-93-80, Hewlett-Packard Laboratories, Palo Alto, CA, February 1994.
- [18] W. Pugh, "A practical algorithm for exact array dependence analysis," *Communications of the ACM*, vol. 35, pp. 102–114, August 1992.
- [19] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*. Reading, MA: Addison-Wesley, 1986.
- [20] S. A. Mahlke, "Design and implementation of a portable global code optimizer," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1991.
- [21] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The superblock: An effective structure for VLIW and superscalar compilation," tech. rep., Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, February 1992.
- [22] S. A. Mahlke, "Exploiting instruction level parallelism in the presence of conditional branches," Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1996.
- [23] P. P. Chang, S. A. Mahlke, and W. W. Hwu, "Using profile information to assist classic code optimizations," *Software Practice and Experience*, vol. 21, pp. 1301–1321, December 1991.
- [24] S. A. Mahlke, W. Y. Chen, J. C. Gyllenhaal, W. W. Hwu, P. P. Chang, and T. Kiyohara, "Compiler code transformations for superscalar-based high-performance systems," in *Proceedings of Supercomputing '92*, pp. 808–817, November 1992.

- [25] R. A. Bringmann, "Compiler-controlled speculation," Ph.D. dissertation, Department of Computer Science, University of Illinois, Urbana, IL, 1995.
- [26] P. P. Chang, D. M. Lavery, S. A. Mahlke, W. Y. Chen, and W. W. Hwu, "The importance of prepass code scheduling for superscalar and superpipelined processors," *IEEE Transactions on Computers*, vol. 44, pp. 353–370, March 1995.
- [27] J. C. Gyllenhaal, "A machine description language for compilation," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1994.
- [28] J. C. Gyllenhaal, W. W. Hwu, and B. R. Rau, "Optimization of machine descriptions for efficient use," in *Proc. 29th Annual Conference on Microprogramming and Microarchitectures*, (Paris, France), pp. 349–358, Dec. 1996.
- [29] R. E. Hank, "Machine independent register allocation for the IMPACT-I C compiler," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1993.
- [30] G. J. Chaitin, "Register allocation and spilling via graph coloring," in *Proceedings of the ACM SIGPLAN 82 Symposium on Compiler Construction*, pp. 98–105, June 1982.
- [31] J. Ellis, *Bulldog: A Compiler for VLIW Architectures*. Cambridge, MA: The MIT Press, 1985.
- [32] R. E. Hank, S. A. Mahlke, R. A. Bringmann, J. C. Gyllenhaal, and W. W. Hwu, "Superblock formation using static program analysis," in *Proceedings of the 26th Annual International Symposium on Microarchitecture*, December 1993.
- [33] M. Schlansker, V. Kathail, and S. Anik, "Height reduction of control recurrences for ILP processors," in *Proceedings of the 27th International Symposium on Microarchitecture*, pp. 40–51, December 1994.
- [34] P. P. Chang, N. Warter, S. A. Mahlke, W. Y. Chen, and W. W. Hwu, "Three architectural models for compiler-controlled speculative execution," *IEEE Transactions on Computers*, vol. 44, pp. 481–494, April 1995.
- [35] S. A. Mahlke, W. Y. Chen, R. A. Bringmann, R. E. Hank, W. W. Hwu, B. R. Rau, and M. S. Schlansker, "Sentinel scheduling: A model for compiler-controlled speculative execution," *Transactions on Computer Systems*, vol. 11, November 1993.
- [36] K. Ebcioglu and T. Nakatani, "A new compilation technique for parallelizing loops with unpredictable branches on a VLIW architecture," in *Languages and Compilers for Parallel Computing*, pp. 213–229, 1989.
- [37] A. Aiken and A. Nicolau, "A realistic resource-constrained software pipelining algorithm," in *Advances in Languages and Compilers for Parallel Processing*, A. Nicolau, D. Galernter, T. Gross, and D. Padua, Eds., London: Pitman/The MIT Press, 1991, pp. 274–290.
- [38] M. Rajagopalan and V. H. Allan, "Efficient scheduling of fine grain parallelism in loops," in *Proceedings of the 26th International Symposium on Microarchitecture*, pp. 2–11, December 1993.

- [39] B. R. Rau and J. A. Fisher, "Instruction-level parallel processing: History, overview, and perspective," *The Journal of Supercomputing*, vol. 7, pp. 9–50, January 1993.
- [40] M. S. Lam, "Software pipelining: An effective scheduling technique for VLIW machines," in *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pp. 318–328, June 1988.
- [41] S.-M. Moon and K. Ebcioglu, "An efficient resource-constrained global scheduling technique for superscalar and VLIW processors," in *Proceedings of the 25th International Symposium on Microarchitecture*, pp. 55–71, December 1992.
- [42] K. Ebcioglu, R. D. Groves, K. Kim, G. M. Silberman, and I. Ziv, "VLIW compilation techniques in a superscalar environment," in *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pp. 36–48, June 1994.
- [43] R. B. Jones and V. H. Allan, "Software pipelining: An evaluation of enhanced pipelining," in *Proceedings of the 24th International Workshop on Microprogramming and Microarchitecture*, pp. 82–92, November 1991.
- [44] V. H. Allan, U. R. Shah, and K. M. Reddy, "Petri net versus modulo scheduling for software pipelining," in *Proceedings of the 28th International Symposium on Microarchitecture*, pp. 105–110, December 1995.
- [45] M. Lam, "A systolic array optimizing compiler," Ph.D. dissertation, Carnegie Mellon University, Pittsburgh, PA, 1987.
- [46] J. C. Dehnert, P. Y. Hsu, and J. P. Bratt, "Overlapped loop support in the Cydra 5," in *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 26–38, April 1989.
- [47] S. Ramakrishnan, "Software pipelining in PA-RISC compilers," *Hewlett-Packard Journal*, pp. 39–45, June 1992.
- [48] J. Ruttenberg, G. R. Gao, A. Stoutchinin, and W. Lichtenstein, "Software pipelining showdown: Optimal vs. heuristic methods in a production compiler," in *Proceedings of the ACM SIGPLAN 96 Conference on Programming Language Design and Implementation*, pp. 1–11, May 1996.
- [49] J. C. Dehnert and R. A. Towle, "Compiling for the Cydra 5," *The Journal of Supercomputing*, vol. 7, pp. 181–227, January 1993.
- [50] R. A. Huff, "Lifetime-sensitive modulo scheduling," in *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pp. 258–267, June 1993.
- [51] A. E. Eichenberger and E. S. Davidson, "Stage scheduling: A technique to reduce the register requirements of a modulo schedule," in *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pp. 338–349, December 1995.
- [52] J. Llosa, M. Valero, E. Ayguade, and A. Gonzalez, "Hypernode reduction modulo scheduling," in *Proceedings of the 28th International Symposium on Microarchitecture*, pp. 350–360, Nov. 1995.

- [53] B. R. Rau, M. S. Schlansker, and P. P. Tirumalai, "Code generation schema for modulo scheduled loops," in *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pp. 158–169, December 1992.
- [54] B. R. Rau, "Data flow and dependence analysis for instruction-level parallelism," in *Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing*, pp. 236–250, 1992.
- [55] N. J. Warter, G. E. Haab, K. Subramanian, and J. W. Bockhaus, "Enhanced modulo scheduling for loops with conditional branches," in *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pp. 170–179, December 1992.
- [56] P. Tirumalai, M. Lee, and M. Schlansker, "Parallelization of loops with exits on pipelined architectures," in *Supercomputing*, November 1990.
- [57] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe, "Dependence graphs and compiler optimizations," in *Proceedings of the 8th ACM Symposium on Principles of Programming Languages*, pp. 207–218, January 1981.
- [58] D. M. Lavery and W. W. Hwu, "Unrolling-based optimizations for modulo scheduling," in *Proceedings of the 28th International Symposium on Microarchitecture*, pp. 327–337, Nov. 1995.
- [59] K. O'Brien, B. Hay, J. Minish, H. Schaffer, B. Schloss, A. Shepherd, and M. Zaleski, "Advanced compiler technology for the RISC System/6000 architecture," in *IBM RISC System/6000 Technology*, 1990.
- [60] M. Schlansker and V. Kathail, "Acceleration of first and higher order recurrences on processors with instruction level parallelism," in *Proceedings of Languages and Compilers for Parallel Computing, 6th International Workshop*, August 1993.
- [61] T. Nakatani and K. Ebcioglu, "Combining as a compilation technique for VLIW architectures," in *Proceedings of the 22nd International Workshop on Microprogramming and Microarchitecture*, pp. 43–55, September 1989.

VITA

Daniel Michael Lavery was born in Evanston, Illinois, in 1963. He grew up in Chicago, Illinois, and attended Notre Dame High School in Niles, Illinois. He pursued his undergraduate studies at the University of Illinois at Urbana-Champaign, where he received the Bachelor of Science degree in Electrical Engineering in 1986. From 1982 to 1985, he was a cooperative engineering student at IBM in Burlington, Vermont. In 1986, he began his graduate studies at the University of Illinois at Urbana-Champaign and received the Master of Science degree in Electrical Engineering in 1989. While pursuing the M.S. degree, he was a research assistant at the Center for Supercomputing Research and Development. From 1988 to 1991, he held the position of computer systems engineer at the Center for Supercomputing Research and Development, where he worked on the development of the Cedar multiprocessor. In 1991, he joined the Center for Reliable and High-Performance Computing as a member of the IMPACT group to pursue the Ph.D. degree at the University of Illinois at Urbana-Champaign. He spent the summer of 1992 at Cray Research, Inc., in Chippewa Falls, Wisconsin. After completing his Ph.D. work, he will join Intel Corporation in Santa Clara, California, as a microprocessor architect.