MODULO SCHEDULING WITH ISOMORPHIC CONTROL TRANSFORMATIONS

BY

NANCY JEANNE WARTER

B.S., Cornell University, 1985
M.S., University of Illinois at Urbana-Champaign, 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, 1994

Urbana, Illinois

MODULO SCHEDULING WITH ISOMORPHIC CONTROL TRANSFORMATIONS

Nancy Jeanne Warter, Ph.D.
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign, 1994
Wen-mei W. Hwu, Advisor

This dissertation addresses the complexities involved with scheduling in the presence of conditional branches. This is a particularly important problem for processors that execute multiple operations per cycle and are not fully utilized by local scheduling techniques. Since conditional branches introduce multiple execution paths, it is difficult for a global scheduler to keep track of the various paths and to select the appropriate operations to schedule. A new approach to global instruction scheduling is presented that uses Isomorphic Control Transformations (ICTs). If-conversion is used to convert an acyclic control flow graph into a large basic block or hyperblock. Local scheduling techniques which are well-known and widely supported can then be applied to schedule operations. After scheduling, the control flow graph is regenerated using Reverse If-Conversion.

One well-known local scheduling based technique is Modulo Scheduling. Modulo Scheduling is a software pipelining technique that effectively schedules loops for high-performance processors. This dissertation highlights the benefits of Modulo Scheduling over other software pipelining techniques based on global scheduling. The ICTs are applied to Modulo Scheduling to schedule loops with conditional branches. Experimental results show that this approach allows more flexible scheduling and thus better performance than Modulo Scheduling with Hierarchical Reduction. Modulo Scheduling with ICTs targets processors with no or limited support for conditional execution such as superscalar processors. However, in processors that do not require instruction set compatibility, support for Predicated Execution can be used. This dissertation shows that Modulo Scheduling with Predicated Execution has better performance and lower code expansion than Modulo Scheduling with ICTs on processors without special hardware support.

DEDICATION

To my family, new and old.

ACKNOWLEDGMENTS

First and foremost, I would like to acknowledge my advisor, Professor Wen-mei W. Hwu, for his intellectual, financial, professional, and emotional support. I value his advice greatly and look forward to applying what I have learned from him as I start my career as a professor.

Next, I would like to acknowledge the support of my Ph.D. committee members. Throughout my years in graduate school, Professor Michael Loui has provided me with valuable guidance. Furthermore, I have a deep respect for his commitment to excellence in education. I would also like to acknowledge Professors Janak Patel and David Kuck for helping me to focus my research. Finally, I would like to acknowledge Dr. Bob Ramakrishna Rau, who while not officially on my committee, has provided me with invaluable insight into my research. I have thoroughly enjoyed our collaborations.

This research would not have been possible without the support of the IMPACT research group. I would like to acknowledge their support in the form of coding, practice talks, research discussions, and friends. Many thanks to Sadun Anik, Roger Bringmann, John Bockhaus, Pohua Chang, William Chen, Tom Conte, Dave Gallagher, John Gyllenhaal, Grant Haab, Rick Hank, Sabrina Hwu, Tokuzo Kiyohara, Dan Lavery, Scott Mahlke, Krishna Subramanian, and Yoji Yamada. Finally, since a group is lost without a secretary, I would like to acknowledge Vicki McDaniel and thank her for being a special friend.

Next, I would like to acknowledge the friends who have made graduate school a very enjoyable experience. I would like to thank John Fu for his friendship and the late night discussions over hot chocolate at Espresso Royale. Furthermore, I would like to thank Madhu Sharma for being a good friend and helping me through Black Thursday. I would like to thank the TKC for the fun: frisbee golf at Kranert, all-night poker games, and the Art and the Esquire. Thanks to Jeff Baxter, Paul Chen, Bob Dimpsey, John Fu, Kumar Goswami, John Holm, Bob Janssens, Suzanne Kuo, Antoine Mourad, Mike Peercy, Paul Ryan, Dale Schouten, Jonathan Simonson, and Tracy Tilton. For keeping me happy and healthy, I would like to thank the Illinois Woman's Rugby team, the Illinois Woman's Ultimate Frisbee team, and the Triathalon die hards. From these friends on the field, I would especially like to thank Betsy Andrews, Laura Drumm, Liz Higgins, Margi Pollack, Diane Snyder, and Margaret Ziegler. I would like to acknowledge the support of my Cornell friends, in particular, Linda Schadler Fiest and Betsy Troutt. Together we faced the rigors of graduate school. Finally, I would like to acknowledge my old roommate and close friend Linda Miller for helping me over the final hurdles.

Always save the best for last. I would like to acknowledge the support of my family "new and old." I would like to thank my sister, Lyn Goff, for always being there and my brother, Jamie Warter, for his love and guidance. And also thanks to their wonderful families whose photos brighten my office. I would like to acknowledge my parents, Jane and Peter Warter, for their love, support, and late night phone calls. I finally got it written, and hopefully it is also right. Last, I want to thank my husband of one month, Larry Perez. He has been my beacon through the stormy years of graduate school. I love him with all my heart and look forward to our life together (in the same state!).

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Compilers for high-performance processors must expose sufficient amounts of instruction-level parallelism (ILP) to efficiently utilize the processor resources. This becomes increasingly important as the pipelines in processors become deeper and the number of concurrently executing instructions increases. Traditionally, the scheduling scope has been limited to one basic block. For high-performance processors, there is typically insufficient ILP within a basic block to fully utilize the processor resources [1], [2]. Thus, compilers for high-performance processors must look beyond basic block boundaries to schedule instructions.

There are two problems associated with scheduling in the presence of conditional branches. First, to achieve a good schedule the compiler must take into account the resource and dependence constraints along multiple execution paths. Second, the compiler must identify the frequently executed portions of the code to ensure that the compiler's efforts yield improved program performance.

Scheduling beyond basic block boundaries is commonly referred to as global scheduling [3], [4]. There are two phases to global scheduling: 1) moving instructions between basic blocks, and 2) local scheduling. The proper ordering of these phases is difficult to determine, particularly in the presence of resource constraints. A different approach is to reduce the task of global scheduling to one of local scheduling. Local scheduling is the basic scheduling technique used in most existing compilers, and, thus is well-defined and supported.

Frequency-based scheduling techniques reduce the task of global scheduling to one of local scheduling by determining the most frequently executed path and scheduling the operations along that path [5], [6], [7]. In some cases there is more than one frequently executed path in a program region. To apply local scheduling techniques, these multiple execution paths must be merged into one by applying transformations to remove conditional branches. After merging, the constraints along all paths can be considered simultaneously.

Previously, there have been two techniques proposed to merge multiple execution paths. If-conversion has been used for processors that have special hardware support for conditional execution [8], [9], [10]. For example, this technique has been used in the Cydrome compiler for processors with hardware support for Predicated Execution [11], [12]. Since full predicated execution support requires a modification to the instruction set, this approach cannot be used in superscalar processors that want to maintain instruction set compatibility. Hierarchical Reduction is a technique that can be used to merge multiple execution paths in processors without special hardware support [13], [14], [15]. We have implemented both of these techniques in the IMPACT-C compiler and found that If-conversion allows more scheduling freedom and thus has the potential for achieving a tighter schedule. Based on this knowledge, we have developed a technique, Reverse If-conversion, that allows us to use If-conversion for processors without special hardware support.

The Isomorphic Control Transformation (ICT) approach presented in this dissertation consists of two transformations. If-conversion is used to transform the control flow graph into a predicated intermediate representation. Reverse If-conversion is used to regenerate the control flow graph after scheduling. The transformations are isomorphic since the condition for execution of the operation is preserved. As a result, local scheduling actions applied to the predicated intermediate representation correspond to global code motions in the resulting control flow graph.

The second problem with scheduling beyond basic block boundaries is to identify the frequently executed portions of a program to maximize the efforts of the compiler in improving the performance of the program. Otherwise, the compiler's scheduling efforts may improve the performance of infrequently executed code while actually degrading the performance of frequently executed code and thus, degrading the overall performance. Loops are ideal since they often execute a large number of times, and hence, the successor of the loop back branch is predictable. Software pipelining is an instruction scheduling technique that exposes large amounts of ILP by systematically overlapping different iterations of the loop.

Modulo Scheduling is a software pipelining technique [16], [14], [12] based on local scheduling. This dissertation demonstrates how ICTs can be applied to Modulo Scheduling and compares the performance against Hierarchical Reduction and If-conversion [8], [9] with Predicated Execution (PE) [11].

1.1 Contributions

The four major contributions in this dissertation are discussed below.

- The Isomorphic Control Transformation (ICT) approach proposed in this dissertation can reduce the task of global scheduling to one of local scheduling. There are two major benefits of ICTs over similar control transformations. First, they do not require special hardware support and, thus, can be used in existing processors and their future superscalar implementations. Second, unlike Hierarchical Reduction [15], no scheduling is performed during the control transformations. Since scheduling during the transformation phase will create more complicated resource usage requirements, avoiding scheduling reduces the potential for resource conflict. Ultimately, it is often possible to find a tighter schedule with ICTs than with Hierarchical Reduction [15]. To illustrate the use and benefits of ICTs, we present an application of ICTs to Modulo Scheduling, a software pipelining technique based on local scheduling.

- In this dissertation, we attempt to define the fundamental differences between Modulo Scheduling and the software pipelining techniques based on global scheduling. The primary benefit of a local scheduling based technique such as Modulo Scheduling, is that the processor resource constraints can be accounted for during scheduling. Thus, when an instruction is scheduled, the resources required are reserved. The global scheduling based techniques that have been previously proposed, only partially consider the resource constraints [17], [18], [19]. To illustrate the hazards of ignoring resource constraints, we compare the performance Modulo Scheduling against one global scheduling based approach, GURPR* [18].

- Predicated Execution (PE) is a form of hardware support that has been proposed for conditionally executing operations [11]. In this dissertation, we show that PE significantly reduces the code expansion due to overlapping conditional constructs and eliminates the bounds on performance due to the one branch per cycle constraint of most conventional processors. We also propose a profile-based Modulo Scheduling technique that uses PE support to allow the program execution to leave and re-enter the software pipeline code with ease.

- The techniques analyzed in this dissertation have been implemented in a prototype compiler. Algorithms and implementation details are provided for new techniques, including: modified RK If-conversion [10], [20], Reverse If-conversion, and Enhanced Modulo Scheduling.

1.2 Overview

The fundamental contribution of this dissertation is combining If-conversion and Reverse If-conversion to form an Isomorphic Control Transform (ICT) pair for reducing the task of global scheduling to one of local scheduling. The ICT approach is outlined in Chapter 2 with emphasis given to the new Reverse If-conversion technique.

Chapter 3 presents an overview of software pipelining and provides a comparison of the two basic approaches, one based on global scheduling and the other based on local scheduling. Chapter 4 provides a more in-depth description of the local scheduling based technique, Modulo Scheduling, and addresses how ICTs are incorporated.

Chapter 5 presents the experimental results which first motivate the use of Modulo Scheduling by comparing Modulo Scheduling against a global scheduling based technique, GURPR*. In addition, the benefit of Induction Variable Reversal, a proposed optimization for software pipelining, is shown. Finally, the performance benefits and scheduling characteristics of Modulo Scheduling with ICTs are presented and compared against Modulo Scheduling with Hierarchical Reduction.

Neither ICTs nor Hierarchical Reduction requires hardware support. In Chapter 6 the benefits of Predicated Execution hardware support for Modulo Scheduling are shown. Furthermore, a technique for using profiling information assuming PE support is presented.

Chapter 7 discusses conclusions and future work with respect to ICTs in general and with regards to their application to Modulo Scheduling.

A recurring theme in this dissertation is that Modulo Scheduling is a powerful technique since it is a technique based on local scheduling in which the exact resource constraints can be accounted for during scheduling. For this reason, Modulo Scheduling is an effective software pipelining technique for scheduling processors with unconventional resource usage patterns such as the manual pipeline of the Intel i860 [21]. A technique for applying Modulo Scheduling for manual pipelines is presented in Appendix A.

1.3 Processor Terminology

The scheduling techniques presented in this dissertation can be applied to any high-performance processors, RISC, CISC, superpipelined, superscalar, or VLIW. For clarity, the VLIW processor model will be used to illustrate examples and VLIW terminology will be used throughout. Thus, before scheduling, a program consists of operations that are equivalent to RISC instructions. After scheduling, an instruction refers to a set of operations that are scheduled in the same cycle.

CHAPTER 2

ISOMORPHIC CONTROL TRANSFORMATIONS

The goal of the compiler is to find a sufficient number of independent operations to fill each instruction for every cycle in the program. While this goal is seldom reached due to resource constraints and program dependences, it is also hindered by conditional branches which introduce multiple execution paths for the scheduler to consider. In the past, compilers have limited their scheduling scope to basic blocks, blocks of code without conditional branches. Since basic blocks tend to have only a few operations, global scheduling techniques have been proposed to schedule operations across basic block boundaries. Global scheduling consists of two phases, inter-block code motion and local (basic block) scheduling. The engineering problem of global scheduling is to determine how to properly order these phases to generate the best schedule.

In this chapter we present a set of isomorphic control transformations (ICTs) that simplify the task of global scheduling to one that looks like local scheduling [22]. This is achieved by defining a predicate intermediate representation (predicate IR) that embodies the code motion properties and thus, eliminates the need for an explicit code motion phase during scheduling. The ICTs convert from the control flow graph representation to the predicate IR and vice-versa. If-conversion, a well-known technique [9], [12], is used to convert an acyclic control flow graph into an enlarged basic block of predicated operations called a hyperblock [23], [20]. After scheduling, a new technique, Reverse If-conversion (RIC), is used to convert from the

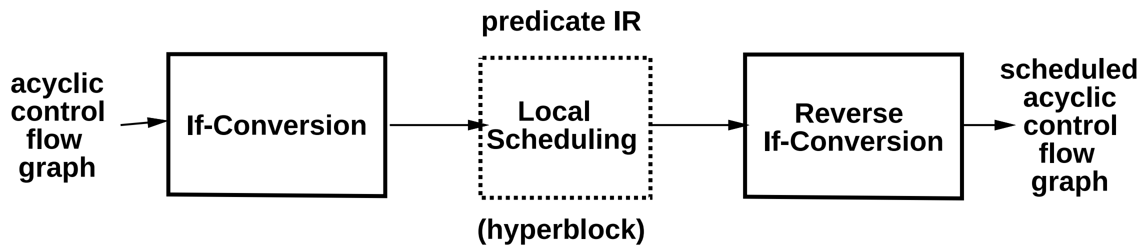


Figure 2.1 Overview of the Isomorphic Control Transformation (ICT) approach for global scheduling.

scheduled hyperblock to the scheduled control flow graph. Figure 2.1 shows an overview of the ICT approach to global scheduling.

If-conversion is considered an isomorphic control transformation because an operation in the original acyclic control flow graph will have the same condition for execution in the hyperblock. Likewise, RIC is isomorphic because an operation in the scheduled hyperblock will have the same condition for execution in the regenerated acyclic control flow graph. Furthermore, we assume that no operations are inserted during RIC. If an operation is inserted during RIC, then it may violate the schedule. This assumption is particularly important for VLIW processors that rely on precise timing relationships.

This chapter is organized as follows. In Section 2.1, we derive the predicate IR. If-conversion, the scheduling issues, and Reverse If-conversion are presented in Sections 2.2 through 2.4. In Section 2.5 and 2.6 we prove the correctness of the ICTs and present the complexity, respectively. Section 2.7 discusses how ICTs build upon previous work and how it differs from alternate approaches which simplify the task of global scheduling.

2.1 Intermediate Representation

The primary program representation is the control flow graph (CFG). The nodes of the CFG are basic blocks. Basic blocks correspond to a sequence of operations. The edges in the CFG represent the control flow between basic blocks, where control flow only enters the top of the basic block and only exits from the bottom. A basic block whose operation sequence ends with a conditional branch has two successors. A basic block with two successors is referred to as a split node. A split node has a true (false) successor which corresponds to the condition of the branch being true (false). The false successor is placed adjacent to the split node in the code layout. Thus, the false successor is on the *fall-through* path of the conditional branch. For the CFGs shown in this dissertation, the destination of the left (right) edge of a split node corresponds to the false (true) successor.

A basic block that has multiple predecessors is referred to as a merge node. Note that it is possible for a basic block to be both a merge and a split node. All but one of the predecessors of a merge node must end with a conditional branch or jump operation. The remaining predecessor is placed adjacent to the merge node in the code layout. Thus, the merge node is on the fall-through path of the remaining predecessor.

The problem addressed in this section is to define an intermediate representation for acyclic subgraphs of the CFG that allows the compiler to generate a globally-scheduled CFG by applying local scheduling techniques. Other researchers have noted the inadequacies of the CFG for applying compiler transformations and have proposed more powerful intermediate representations such as the Program Dependence Graph (PDG) [24], [25], [26], [3], [4]. Our intermediate representation builds on PDG concepts but is designed specifically to assist global instruction-level scheduling.

Before presenting the intermediate representation, we want to analyze the difficulty of scheduling an acyclic CFG.

Definition 1: An *acyclic CFG* is a directed acyclic graph G with a unique entry node START and a set of exit (EXIT) nodes such that each node in the graph has at most two successors. For any node N in G there exists a directed path from START to N and a directed path from N to one of the EXIT nodes. START has in-degree 0; each EXIT node has out-degree 0.

Applying global scheduling to an acyclic CFG involves two phases: code motion between basic blocks, and scheduling within basic blocks. Figure 2.2 illustrates the rules of code motion for global scheduling [5], [6]. These rules can also be viewed as the basic steps needed for moving operations in the CFG. For example, an operation in basic block **F** can be moved into basic blocks **C**, **D**, and **E** by applying rule 5. The operations can then be scheduled in these basic blocks. Identical operations in **D** and **E** can be merged into **B** by applying rule 1. Again, the operations can be scheduled in these basic blocks, or the identical operations in **B** and **C** can be merged into **A**. This simple example illustrates the phase ordering problem between the code motion and scheduling phases in global scheduling. It is difficult to determine when to stop code motion and when to schedule the operations to generate the best schedule.

In some code motion cases, it is necessary to create a basic block into which an operation may be copied. Consider the CFG in Figure 2.3(a). It is not possible to simply apply rule 2 to copy an operation from **B** into **D** and **E**. Because **E** has multiple predecessors, the operation from **B** cannot be placed in basic block **E**. Instead, a basic block (**D1**) must be created for the copy of the operation. Figure 2.3(b) shows an augmented CFG with all of the possible basic blocks (**D1**, **D2**, and **D3**) that could be created as a result of applying the code motion rules [5], [6].

Rules of Code Motion

1) identical operations from B and C merged into A (hoisting)
2) operation from A copied to both B and C
3) operation moves up from B to A (speculative)
4) operation from A copied only to B (destination is not in live-in set of C)
5) operation from F copied into C, D, and E
6) identical operations from C, D, and E merged into F
7) operation from D copied into F (predicated)

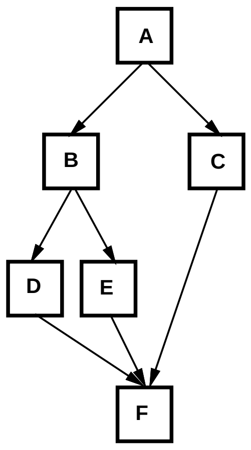
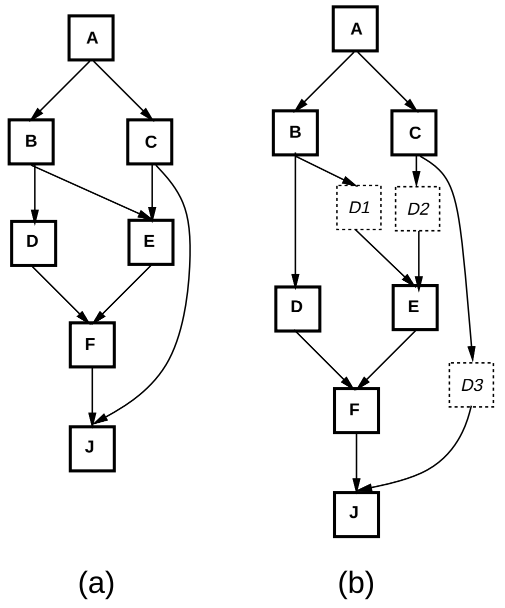


Figure 2.2 Rules of code motion for global scheduling.



operation from	Range	Bounds	
		after branch of	before merge at
A	all	-	-
B	B, D, D1	A	E, F
C	C, D2, D3	A	E, J
D	D	B	F
E	D1, D2, E	B, C	F
F	B, D, D1, D2, E, F	E, C	J
J	all	-	-

(c)

Figure 2.3 Bounds on code motion for example CFG. (a) Example CFG. (b) Augmented CFG. (c) Code motion bounds and ranges assuming no speculative or predicated execution.

By applying the code motion rules to operations from every basic block in the CFG, the range of possible destination blocks can be determined for each operation. The table in Figure 2.3(c) presents the range assuming no speculative or predicated code motion.[1] For example, consider an operation **b** in basic block **B**. Since we do not consider speculative execution, **b** cannot be moved to **A**. Under rule 2, **b** can be copied into **D** and **D1**. Since we assume no predicated execution, **b** cannot be copied from **D** to **F** or from **D1** to **E**. Thus, the range of allowable basic blocks for operation **b** is **B**, **D**, and **D1**. From this range, we can see that **b** is bounded from above by the conditional branch in **A** and bounded from below by the merges at **E** and **F**.

During local scheduling, only the precedence relations between operations are considered. Thus, if the bounds on code motion can be represented as precedence relations, then the task of global scheduling can be simplified to local scheduling. The *upper bound* on code motion defines the bound for upward code motion and corresponds to *control dependence* [9]. An operation x that is bounded by a conditional branch operation y is said to be *control dependent* on y. From the scheduler's viewpoint, x cannot be scheduled until after y has been scheduled. Given the following definition of postdominance, control dependence can be defined [9], [24].

Definition 2: A node X is *postdominated* by a node Y in G if every directed path from X to an EXIT node (not including X) contains Y.

Definition 3: All the operations in node Y are *control dependent* upon the conditional branch operation in node X if and only if (1) there exists a directed path P from X to Y such that every node Z in P (excluding X and Y) is postdominated by Y and (2) Y does not postdominate X.

[1]We also assume that there are no identical operations along different control flow paths in the original CFG that can be hoisted or merged. Thus, we assume that rule 1 is applied only in conjunction with rule 5, and that rule 6 is applied only in conjunction with rule 2.

The *lower bound* on code motion defines the bound for downward code motion. There is no existing precedence relation that can be used to define the lower bound on code motion. One problem in defining such a precedence relation is that precedence relations are between two operations, and there is no explicit merge operation. We can define an *implicit merge operation* to be an implied operation which precedes the first explicit operation in every merge node. Now, the lower bound on code motion can be represented by a *control anti dependence* relation. From a scheduler's viewpoint, a merge operation y cannot be scheduled until every operation x upon which it is control anti dependent has been scheduled. Using the following definition for dominance [27], control anti dependence can be defined.

Definition 4: A node X *dominates* a node Y in G if every directed path from START to Y contains X.

Note that the *immediate dominator* of a node X is the last dominator of X along any path from START to X. Lengauer and Tarjan present a fast algorithm for finding dominators in a CFG [28].

Definition 5: The implied merge operation in node Y is *control anti dependent* upon all the operations in node X if and only if (1) there exists a directed path P from X to Y such that X dominates every intermediate node of P, and (2) X does not dominate Y.

The matrices in Figure 2.4 show the non redundant control dependences and control anti dependences for the example CFG in Figure 2.3(b). A redundant control dependence corresponds secondary control dependences that are preserved by a combination of primary control dependences. For example, operations from **D** are control dependent upon the branch of **A**. This dependence is not shown since it is preserved by the combination of the control depensrc (branch of)

dest (operation from)

	A	B	C
A			
B	X		
C	X		
D		X	
E		X	X
F	X		X
J			
D1		X	
D2			X
D3			X

(a)

src (operation from)

dest (implied merge of)

	A	B	C	D	E	F	J	D1	D2	D3
E		X	X					X	X	
F		X		X	X					
J			X			X				X

(b)

Figure 2.4 Control precedence relations for operations in the example CFG. (a) Control dependence. (b) Control anti dependence.

dence between the branch of **A** and operations from **B** and the control dependence between the branch of **B** and operations from **D**.

Given the control dependences and control anti dependences, we need to define an IR that preserves these dependences. Since our goal is to apply local scheduling techniques, an operation is the basic unit of the IR. A predicate, which represents an operation's condition for execution, can be assigned to each operation. Every operation in the acyclic control flow subgraph becomes a *predicated operation*. To preserve control dependences, conditional branches become operations that define the predicates. These operations are referred to as *predicate defining operations* (pd). To preserve control anti dependences, implied merge operations become operations that merge (or kill) predicates, referred to as *predicate merging operations* (pm). As we will define later, the pm operation is used to indicate when a predicate can be removed from the active set of predicates.

Table 2.1 Predicate Intermediate Representation.

OPERATION	SYNTAX
predicated operation	$< p >$ op
predicate defining	$< p >$ pd [cond] {false}{true}
predicate merging	pm {no_jump}{jump}

It is sufficient to represent the control dependences and control anti dependences using one set of predicates per pd operation and one set of predicates per pm operation. To regenerate the CFG, the RIC algorithm must be able to determine which predicates were defined along the two paths of the conditional branch. Thus, two sets of predicates are needed for the pd operations, one for predicates defined along the true path and one for predicates defined along the false path of the conditional branch. Likewise, there are two types of predicates paths entering a merge node, those that have jump operations and those that do not. The pm operation has two sets of predicates, those corresponding to paths being merged that require a jump, and those that do not. Table 2.1 shows the syntax of the predicate IR. Note that since the implicit merge operation is not an actual operation in the original CFG, the pm operation is not predicated. The predicates in the jump and no_jump sets indicate the condition for execution of the pm operation.

Before discussing the If-conversion transformation, we return to the idea of augmenting the CFG with empty basic blocks. As discussed above, during code motion, basic blocks may need to be generated to hold copies of moved operations. Since local scheduling is applied to the predicate IR, these basic blocks are generated during RIC. When a basic block is generated, control operations need to be inserted to branch into and out of the new basic block. Since inserting new operations after scheduling may lengthen the schedule, inserting operations during

Algorithm DUM: Given a rooted directed graph G, (N, E, START), insert dummy nodes where necessary.

```
Let $\forall$ X $\in$ N
    num_succ(X) = the number of successors of X
    num_pred(X) = the number of predecessors of X

$\forall$[X,Y] $\in$ E
    if (num_succ(X) > 1) and (num_pred(Y) > 1)
        insert dummy node on edge [X,Y]
        if(Y is not the fall-through path of X)
            insert jump operation in dummy node
```

Figure 2.5 Algorithm DUM inserts dummy nodes into CFG to account for bookkeeping.

RIC is undesirable. To avoid inserting operations during RIC, the CFG is augmented with dummy blocks before If-conversion. For every edge $X \rightarrow Y$ in the CFG, if X has multiple successors and Y has multiple predecessors, then a dummy node is inserted on $X \rightarrow Y$. If Y is not on the fall-through path of X, then insert a jump operation in the dummy node. Algorithm DUM in Figure 2.5 is used to insert dummy nodes.

2.2 If-conversion

The If-conversion algorithm presented in this dissertation is based on the RK algorithm [10]. The basic RK algorithm decomposes the control dependences using two functions R and K. The function R(X) assigns a predicate to basic block X such that any basic block that is control equivalent [4] to X is assigned the same predicate. Two nodes X and Y in the CFG are control equivalent if X dominates Y and Y postdominates X. The function K(p) specifies the conditions under which a predicate p is defined. Predicate defining operations are placed into nodes according to K. For example, when $K(p) = \bar{A}$, a pd operation that defines p under the false condition of the branch of A is placed in node A. Note that in the following examples,

Algorithm S: Given (1) a rooted directed graph G, (N, E, START), (2) an ordered set of predicates, P, determined by R(X) $\forall$ X $\in$ N, and (3) the mapping K(p) $\forall$ p $\in$ P, compute S. For any split node X, S_{true}(X) (S_{false}(X)) specifies the set of predicates defined under the true (false) condition of the branch of X referred to as condition(X).

```
$\forall$ p $\in$ P
    $\forall$ X $\in$ K(p)
        if condition(X) is TRUE
            $S_{\text{true}}(X) = S_{\text{true}}(X) \cup p$
        else
            $S_{\text{false}}(X) = S_{\text{false}}(X) \cup p$
```

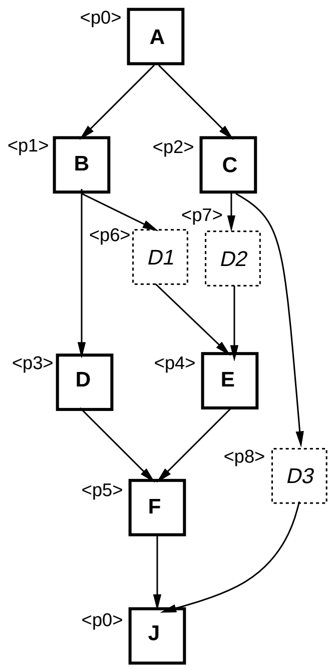
Figure 2.6 Algorithm S determines the predicates along true (false) paths of each conditional branch.

we use the name ($\overline{\text{name}}$) of the split node to indicate the true (false) conditions of the node's conditional branch.

To generate the true and false predicate sets of the pd operation, we have added two functions, S_{true}(X) and S_{false}(X), to combine all the predicates defined under the true and false conditions of conditional branch of X. The S algorithm is presented in Figure 2.6.

Figure 2.7 shows the predicated example CFG and corresponding R, K, S_{true}, and S_{false} functions. Note that basic blocks **A** and **J** are control equivalent; thus, they have the same predicate **p0**. Since **A** and **J** always execute, K(**p0**) is the empty set since no conditions define their execution.

Consider the conditional branch in basic block **A**. From the control dependence matrix in Figure 2.4(a), we know that basic blocks **B**, **C**, and **F** are control dependent on the branch in **A**. Using the R function, the predicates of basic block **B**, **C**, and **F** are **p1**, **p2**, and **p5**, respectively. The value of the K function is $\{\bar{\mathbf{A}}\}$ for **p1**, $\{\bar{\mathbf{A}}, \bar{\mathbf{C}}\}$ for **p5**, and $\{\mathbf{A}\}$ for **p2**. Thus, S_{false}(**A**) is {**p1**,**p5**} and S_{true}(**A**) is {**p2**}.



block x:	A	B	C	D	E	F	J	*D1*	*D2*	*D3*
predicate R(x):	p0	p1	p2	p3	p4	p5	p0	p6	p7	p8

predicate p:	p0	p1	p2	p3	p4	p5	p6	p7	p8
K(p):	Φ	{$\overline{A}$}	{A}	{$\overline{B}$}	{B,$\overline{C}$}	{$\overline{A}$,$\overline{C}$}	{B}	{$\overline{C}$}	{C}

block x:	A	B	C	D - *D3*
S_true(x):	{p2}	{p4,p6}	{p8}	Φ
S_false(x):	{p1,p5}	{p3}	{p4,p5,p7}	Φ

block x:	E	F	J	A-D, *D1-D3*
M_no_jump(x):	{p1,p2,p7}	{p1,p3}	{p2,p5}	Φ
M_jump(x):	{p6}	{p4}	{p8}	Φ

Figure 2.7 Values of the functions R, K, S, and M for example CFG.

Another way of viewing control anti dependences is to say that an operation is reverse control dependent on an implied merge operation.

Definition 6: All the operations in node X are *reverse control dependent* upon the implied merge operation in node Y if and only if (1) there exists a directed path P from X to Y such that X dominates every intermediate node in P, and (2) X does not dominate Y.

Reverse control dependence can be calculated by modifying the algorithm for calculating control dependence presented in [10]. The algorithm RCD in Figure 2.8 calculates the reverse control dependences of each node in the directed graph. RCD(X) calculates the set of implied merge operations which every operation in node X is reverse control dependent upon. The algorithm M in Figure 2.9 uses RCD(X) to determine which predicates are being merged at each pm operation. Once it has been determined which predicates to merge, the predicates are divided into the no_jump and jump sets for each pm operation. Given a merge node t in

Algorithm RCD: Given a rooted directed graph G, (N, E, START), compute the reverse control dependence. Assume that dominators of G have been calculated.

```
Let $\forall$ X $\in$ N
    dom(X) = {Y $\in$ N: Y dominates X}
    idom(X) = the immediate dominator of X

$\forall$ [X,Y] $\in$ E such that X $\notin$ dom(Y)
    LUB = idom(Y)
    t = X
    while (t $\neq$ LUB)
        RCD(t) = RCD(t) $\cup$ Y
        t = idom(t)
```

Figure 2.8 Algorithm RCD computes reverse control dependence.

RCD(X) and the predicate, p, of X, p is placed in the jump set of the the pm operation of t if (1) t is an immediate successor of X, and (2) t is not on the fall-through path of X. Otherwise, p is placed in the no_jump set of the pm operation of t.

Figure 2.7 shows the merge functions M_{jump} and M_{no_jump} of the implicit pm operations at nodes **E**, **F**, and **J**. Consider the implied merge operation at node **F**. From the control anti dependence matrix in Figure 2.4(b), we know that operations in basic blocks **B**, **D**, and **E** are control anti dependent on the implied merge operation of **F**. Node **E** is an immediate predecessor of **F** and is assumed to have a jump operation (e.g., **F** is not on the fall-through path) and thus its predicate is in the jump predicate set. Thus, M_{jump}(**F**) is {**p4**} and M_{no_jump}(**F**) is {**p1**,**p3**}.

After If-conversion, each operation is predicated, conditional branch operations are replaced by pd operations, implied merge operations are replaced by pm operations, and jump operations are deleted.

Figure 2.10(a) shows the code after If-conversion has been applied to the example CFG. Note that the R function defines the predicate of each operation; the S_{true} and S_{false} functions

Algorithm M: Given a rooted directed graph G, (N, E, START) and RCD of G, compute M. M_{jump} specifies the set of predicates to be merged whose corresponding blocks require an explicit jump to be inserted. $M_{\text{no\_jump}}$ specifies those that do not.

```
Let $\forall$ X $\in$ N
    P(X) = predicate that X is control dependent upon

$\forall$ X $\in$ N
    $\forall$ t $\in$ RCD(X)
        if t $\neq \emptyset$
            if (t fall-through path of X) or
              (t not immediate successor of X)
                $M_{\text{no\_jump}}(t) = M_{\text{no\_jump}}(t) \cup P(X)$
            else
                $M_{\text{jump}}(t) = M_{\text{jump}}(t) \cup P(X)$
```

Figure 2.9 Algorithm M computes the set of predicates for each merge point.

define the true and false sets of the pd operations; and the $M_{\text{no\_jump}}$ and M_{jump} functions define the no_jump and jump sets of the pm operations. To illustrate the scheduling and regeneration complexities, Figure 2.10(a) shows predicated operations for each basic block. In this example, generic operations are used. The lower case letter is used to indicate which basic blocks the operations were from in the original CFG. For example, **op_a1**, ..., **op_a4** are the operations from basic block **A** in the example CFG. Furthermore, the basic block identifier is also used to specify the condition of the pd operation. For example, <**p0**>**pd[A]**{**p1,p5**}{**p2**} is a pd operation defined under predicate **p0**. The pd operation defines predicates **p1** and **p5** (**p2**) under the false (true) condition of the conditional branch of **A**. Also, the pm operations indicate from which basic block they originate. For example, **pm_F** is the pm operation associated with the implied merge operation of basic block **F**. Note that since dummy nodes, **D1**, **D2**, and **D3** do not have any operations before scheduling, there are no predicated operations corresponding to these blocks. Note, however, that their predicates are placed in the appropriate sets of the pd and pm operations.

2.3 Scheduling Issues

By using the set of isomorphic control transformations, local (basic block) scheduling techniques can be used to perform global scheduling. The control dependences and control anti dependences are used to preserve the control flow properties during scheduling. Control dependence prevents operations from being scheduled before the corresponding pd operation. In terms of the CFG, control dependences prevent the operations from being moved above a conditional branch. Control dependences may be removed if the processor has hardware support for speculative execution. In this case, the predicate of an operation can be promoted to or replaced by the predicate of the pd operation and thus, the operation can be scheduled before the pd operation [23]. Control anti dependence ensures that the pm operation is scheduled after any operations that it is control anti dependent upon. Thus, with respect to the resultant CFG, the control anti dependence prevents any operations from being incorrectly scheduled after a merge point. Control anti dependences may be removed if the processor has hardware support for predicated execution [11], [12].

The disadvantage of speculative and predicated execution is that the processor will fetch the operation even when the result of the operation is not used. The hyperblock formation can be used to ensure that operations are moved only along the most frequently executed paths. A hyperblock is defined as follows.

Definition 7: A *hyperblock* is an predicated region formed from an acyclic subgraph of the CFG which has only one entry (START) block and one or more exit (EXIT) blocks.

```
<p0>  op_a1
<p0>  op_a2
<p0>  op_a3
<p0>  op_a4
<p0>  pd [A] {p1,p5}{p2}
<p1>  op_b1
<p1>  op_b2
<p1>  op_b3
<p1>  pd [B] {p3}{p4,p6}
<p2>  op_c1
<p2>  op_c2
<p2>  op_c3
<p2>  pd [C] {p4,p5,p7}{p8}
<p3>  op_d1
<p3>  op_d2
<p3>  op_d3
pm_E {p1,p2,p7}{p6}
<p4>  op_e1
<p4>  op_e2
<p4>  op_e3
pm_F {p1,p3}{p4}
<p5>  op_f1
<p5>  op_f2
<p5>  op_f3
pm_J  {p2,p5}{p8}
<p0>  op_j1
<p0>  op_j2
<p0>  op_j3
<p0>  op_j4
```

(a)

```
I1:     <p0>  op_a1
I2:     <p0>  op_a2
I3:     <p0>  op_j1
I4:     <p0>  pd [A] {p1,p5}{p2}
I5:     <p1>  op_b1
I6:     <p0>  op_j2
I7:     <p2>  op_c1 /  <p1>  op_b2
I8:     <p1>  pd [B] {p3}{p4,p6}
I9:     <p3>  op_d1 /  <p2>  pd [C] {p4,p5,p7}{p8}
I10:    <p4>  op_e1
I11:    <p5>  op_f1
I12:    <p2>  op_c2 /  <p3>  op_d2
I13:    <p4>  op_e2
I14:    <p1>  op_b3 /  <p2>  op_c3
I15:    pm_E {p1,p2,p7}{p6}
I16:    <p3>  op_d3
I17:    <p5>  op_f2
I18:    <p4>  op_e3
I19:    pm_F {p1,p3}{p4}
I20:    <p0>  op_a3
I21:    <p0>  op_j3
I22:    <p5>  op_f3 / pm_J {p2,p5}{p8}
I23:    <p0>  op_a4
I24:    <p0>  op_j4
```

(b)

Figure 2.10 (a) Result of applying If-conversion to example CFG. (b) Example code after scheduling but before RIC.

Thus, if some basic blocks are known to be infrequently executed, then they can be excluded from the hyperblock. This allows the compiler to more effectively optimize and schedule the operations from the more frequently executed basic blocks that form the hyperblock. To exclude basic blocks, a technique called tail duplication is used to remove additional entry points into the hyperblock. The details of selecting basic blocks and using tail duplication to form hyperblocks are provided in [23], [20].

In the scheduled hyperblock, operations with mutually exclusive predicates can be scheduled to the same processor resource in the same cycle. This is allowed since operations with mutually exclusive predicates will be placed in different basic blocks in the scheduled CFG. In Figure 2.10, a '/' is used to indicate that two operations are scheduled to the same resource. We use the Predicate Hierarchy Graph (PHG) [23], [20] to determine whether two predicates are mutually exclusive, or equivalently, do not have a control flow path between them. The PHG is an acyclic graph with two types of nodes: predicate and condition. Figure 2.11(b) shows the PHG for the example predicated CFG shown in Figure 2.11(a).

The levels of the graph alternate between predicate and condition nodes, where the root node is the predicate **p0**, which is always defined. The Boolean expression that defines a predicate can be determined by traversing the PHG from the predicate node to the root node. Each conjunction in the graph corresponds to a Boolean AND, and each disjunction corresponds to a Boolean OR. For example the Boolean expression of predicate **p4** is $(\mathbf{a\_bar} \cdot \mathbf{b}) + (\mathbf{a} \cdot \mathbf{c\_bar})$. With respect to the CFG, the Boolean expression translates into the following. Basic block **E** will execute if the condition of the branch of basic block **A** is false and the condition of the branch of basic block **B** is true, or if the condition of **A** is true and the condition of **C** is false.

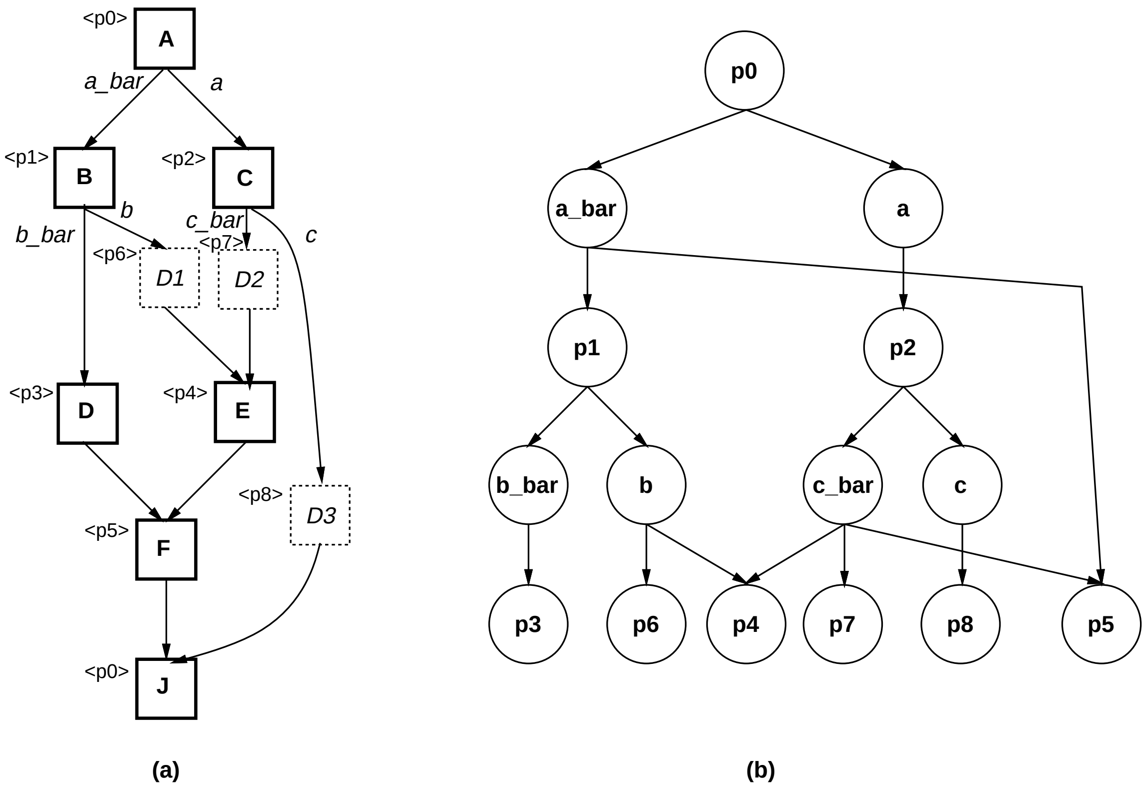


Figure 2.11 (a) Example CFG with branch conditions. (b) Predicate Hierarchy Graph.

Determining whether there is a control path between two predicates is equivalent to determining whether the two predicates can be ever both be true at the same time. Thus, there is not a control flow path between two predicates if the AND of the two corresponding Boolean expressions can be simplified to 0.

The pm operation also has special scheduling characteristics. A pm operation is scheduled as a jump operation under the jump set predicates and as a null operation under the no_jump set predicates. Since there may be multiple predicates in the jump set, multiple jump operations can be scheduled per pm operation. All predicates in the jump set are mutually exclusive, and thus, their operations can be scheduled to the same processor resource in the same cycle. A null operation does not require any resources or cycles. To schedule a pm operation at a given time to a given resource, only the predicates in the jump set must be mutually exclusive to the predicates of other operations already scheduled to the resource.

Figure 2.10(b) shows the example hyperblock after scheduling. The schedule assumes a single-issue processor without interlocking. We present the most restrictive processor model to illustrate how *no-op* operations are inserted if needed. Note that the control dependences (control anti dependences) between pd (pm) operations and their respective predicated operations are preserved. We assume that all other data dependences between operations are preserved. Also note that the pm operation **pm_J** is scheduled in instruction **I22**. It can be scheduled in instruction **I22** since all operations that are predicated on **p2**, **p5**, and **p8** are scheduled at or earlier than **I22**. Also, **p8** and **p5** are mutually exclusive; thus, both **op_f3** and **pm_J** can be scheduled in the same cycle. Note, that although a pm operation is scheduled as a jump operation, it is not converted to a jump operation until RIC. Similarly, a pd operation is scheduled as a conditional branch but is not converted to a conditional branch operation until RIC.

Before If-conversion was applied to the CFG, the CFG was augmented with dummy nodes. The scheduler can determine whether a dummy node is needed. If a dummy node is not needed, then the scheduler should apply the following jump optimization to remove unnecessary jump operations when pm operations are scheduled. When a pm operation is scheduled, a jump is required if any operations have been scheduled along the control path between the pd operation that defines a predicate p and the pm operation that contains p in its jump set. The scheduler can determine whether a jump is needed by checking all scheduled operations between *pd* and the cycle in which *pm* is being scheduled. If any operations have a predicate that is not mutually exclusive with the predicate p, then a jump is needed. For VLIW processors, a further condition is needed. To remove a jump, the *pm* operation must be scheduled in the same cycle as the *pd* operation . If a jump is not needed, then jump optimization is performed by deleting the predicate p from the jump set and inserting it into the no_jump set of the pm operation.

2.4 Reverse If-conversion

After scheduling, the hyperblock represents the merged schedule for all paths of the CFG. The task of RIC is to generate the correct control flow paths and to place operations into the appropriate basic blocks along each path. Whereas a basic block in the original CFG was assigned one predicate, each basic block in the regenerated CFG has a set of predicates associated with it. The *allowable predicate set* specifies the predicates of the operations that can possibly be placed in the corresponding basic block. Intuitively, the allowable predicate set is the mechanism that accounts for the code motion phase of global scheduling.

The RIC algorithm is presented in Figure 2.12. The RIC algorithm processes the operations in the hyperblock in a sequential manner. As the CFG is generated, there is a continually changing set of leaf nodes which at a given cycle contains one node from every possible execution path. The algorithm maintains the leaf node set L to determine the basic blocks in which operations can be placed. Initially, L consists of the root node of the regenerated CFG. Each node X in L has an allowable predicate set $\rho(X)$. When an operation *op* in the hyperblock is processed, it is placed in every node X in L for which $P(op) \in \rho(X)$. For a VLIW processor, it is necessary to insert *no-op* operations into empty slots. Since each operation slot may have multiple operations with mutually exclusive predicates, *no-op* operations are not inserted until after the entire instruction has been processed.

When a pd operation is encountered, it is inserted into every node X in L for which $P(pd) \in \rho(X)$. For each such X, two successor nodes $Succ_t$ and $Succ_f$ corresponding to the true and false path of *pd* are created and inserted into L and X is deleted from L. The allowable predicate sets of $Succ_t$ and $Succ_f$ are,

$$\rho(Succ_t) = \rho(X) \cup true\_predicates\ of\ pd, \text{ and}$$

$$\rho(Succ_f) = \rho(X) \cup false\_predicates\ of\ pd.$$

When a pm operation is encountered, the following is done for each node X in L. If a predicate in $\rho(X)$ is in the jump or no_jump sets, then the predicates specified in the jump and no_jump sets are deleted from $\rho(X)$ to create $\rho_{new}(X)$. If it is in the jump set, then a jump operation is inserted into X. L is searched for a node Y with the same allowable set as $\rho(X)$. If one is found, then Y becomes the successor of X. Otherwise, a successor is created for X with the allowable set $\rho_{new}(X)$.

Algorithm RIC: Given hyperblock H, regenerate a correct CFG. For VLIW processors, insert_no-op fills each empty operation slot in an instruction with a no-op operation.

```
Let $\forall$ X $\in$ H
    P(X) = predicate that X is control dependent upon

create root node
L = {root}
$\rho$(root) = {p0}
$\forall$ op $\in$ H in scheduled order
    $\forall$ X $\in$ L
        if op is predicate defining (pd) operation
            if P(op) $\in \rho$(X)
                insert a conditional branch operation into X
                create successor nodes $\mathrm{Succ_t}$ and $\mathrm{Succ_f}$
                $\rho(\mathrm{Succ_t}) = \rho(X) \cup$ (true predicate set of op)
                $\rho(\mathrm{Succ_f}) = \rho(X) \cup$ (false predicate set of op)
                $L = (L - X) \cup \mathrm{Succ_t} \cup \mathrm{Succ_f}$

        else if op is predicate merging (pm) operation
            $\rho_{\mathrm{new}}(X) = \rho(X)$ - (no_jump predicate set of op) - (jump predicate set of op)
            if $\rho_{new}(X) \neq \rho(X)$
                if (jump predicate set of op) $\cap \rho(X) \neq \emptyset$
                    insert jump operation in X
                $\forall$ Y $\in$ L
                    if $\rho_{\mathrm{new}}(X) \equiv \rho(Y)$
                        successor of X = Y
                if successor found
                    $L = L - X$
                else
                    create successor node Succ
                    $\rho(\mathrm{Succ}) = \rho_{\mathrm{new}}(X)$
                    $L = (L - X) \cup \mathrm{Succ}$

        else if op is predicated operation
            if P(op) $\in \rho$(X)
                insert op in X

    if target processor is VLIW and last op of instruction
        insert_no-op
```

Figure 2.12 Algorithm RIC performs Reverse If-conversion.

Figure 2.13 shows the CFG generated from the scheduled hyperblock in Figure 2.10(b). The target machine in this example is a single-issue processor without interlocking. Thus, *no-op* operations are required and are represented by a dash. The allowable predicate set is indicated on top of each basic block.

2.5 Correctness of ICT approach

In this section we prove the correctness of the isomorphic control transform approach. First, we show that DUM preserves the control dependences of the original CFG G.

Lemma 2.5.1 *Given a directed graph G, the augmented CFG created by DUM preserves the control dependence and reverse control dependence of G.*

Proof: Let P be a directed path between node X and node Y in G. First consider control dependence, where Y is control dependent on X. Assume that DUM inserts a node Z in P. Since Z only has one successor and Z is in P, every path from Z to an EXIT node contains Y. Thus, Z is postdominated by Y, and hence, Y remains control dependent on X. A similar argument can be made for reverse control dependence. □

Lemma 2.5.2 *Assuming no speculative or predicated execution, an operation in the regenerated CFG is executed if and only if it would be executed in the original CFG.*

Proof: During If-conversion, every operation is assigned a condition for execution (predicate). The control dependences and control anti dependences are preserved using pd and pm operations, respectively. We assume that the scheduler does not violate these dependences.

There are three cases in which an operation can execute in the original CFG but not in the regenerated CFG: 1) the operation is moved from above to below a branch but only placed on

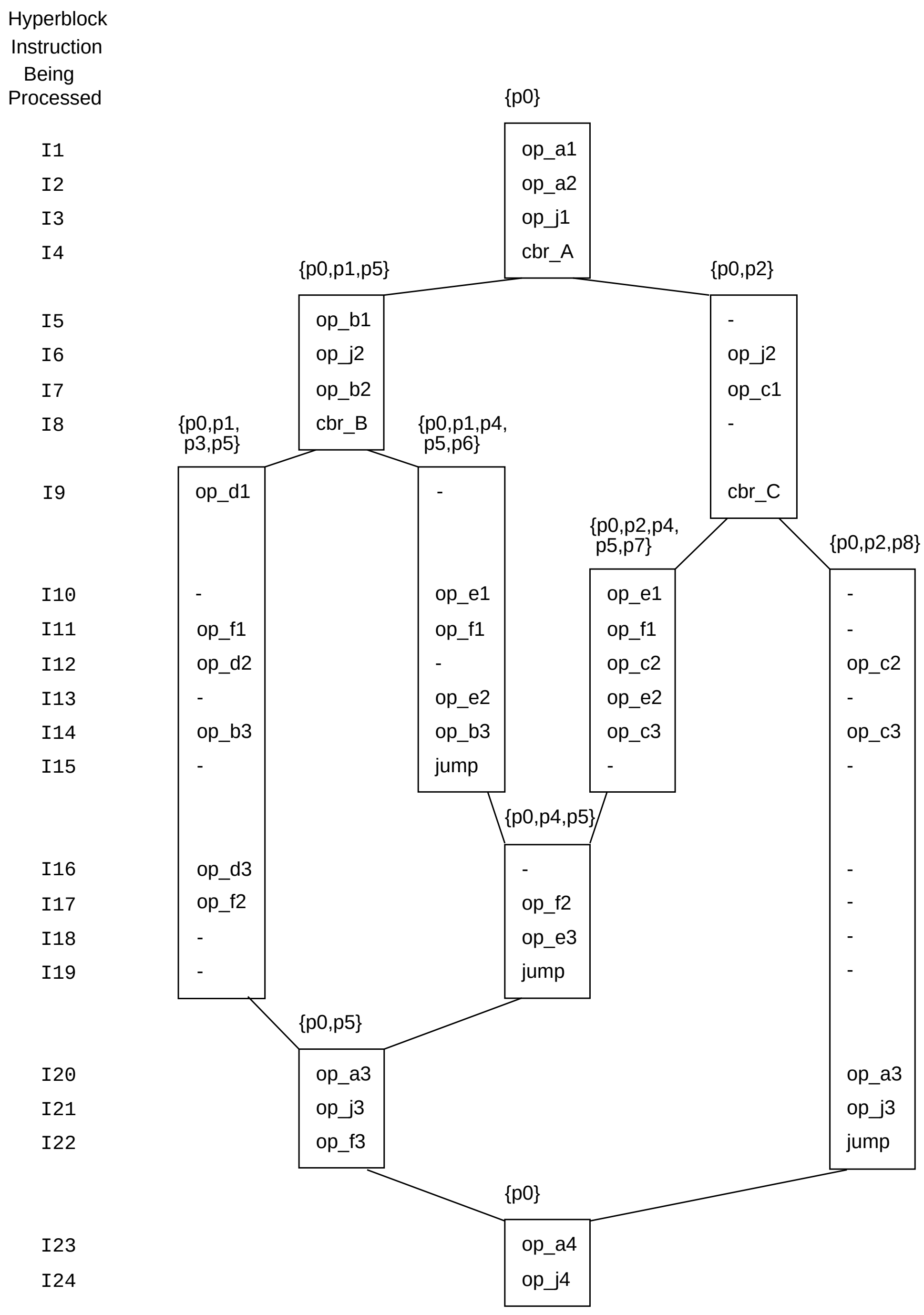


Figure 2.13 CFG of scheduled hyperblock after RIC.

one path of the branch, 2) the operation is moved from below to above a merge but not along every path, and 3) the operation is deleted from a path. Since the scheduler does not allow these cases, all three cases can occur only if RIC places an operation on the wrong path or fails to place an operation.

There are three cases in which an operation can execute in the regenerated CFG but not in the original CFG: 1) the operation is moved above a branch it is control dependent upon, 2) the operation is moved below a merge it is control anti dependent upon, and 3) the operation is executed along a mutually exclusive path. The first two cases can occur only if RIC violates the dependences. The last case can occur only if RIC places an operation on the wrong path.

The control dependences and control anti dependences are preserved by RIC in the following manner. During RIC, a pd operation is converted into a conditional branch. The predicates defined along one path of the branch are inserted into the allowable predicate set of that path. An operation is inserted only along the path that has its predicate in the allowable predicate set. Thus, an operation will not be placed before a branch upon which it is control dependent or along the wrong path of the branch. The pm operation is scheduled after every operation upon which it is control anti dependent. During RIC, paths are not merged until a pm operation is encountered. Thus, an operation that the pm is control anti dependent upon will not be placed after a merge. Since RIC preserves the control dependences and control anti dependences, the predicate of an operation will be in the allowable predicate set of a block when the operation is processed. Thus, an operation cannot be deleted during RIC. □

Lemma 2.5.3 *Given a VLIW processor, the timing relationship between any two operations in a scheduled hyperblock is preserved in the regenerated CFG.*

Proof: During RIC, instructions are processed according to the schedule. Thus, instructions are generated with the proper number of cycles between them unless operations are added or deleted. Only jump operations would need to be added during RIC. Dummy node insertion and the pm operation ensure that no jump operations need to be added during RIC. After all VLIW instructions are processed, empty operation slots are filled with *no-op* operations. Thus, no operations will be inserted or deleted during RIC.□

Theorem 2.5.1 *Given a correct schedule, the ICT approach generates a correct globally scheduled CFG.*

Proof: This theorem follows from Lemmas 2.5.1, 2.5.2, and 2.5.3.

2.6 Complexity

The time complexity of the RIC algorithm is $O(H_p L_{max} + H_d L_{max}^2 + H_m 2 L_{max}^2)$, where H_p is the number of operations in the hyperblock excluding pd and pm operations, H_d is the number of pd operations, H_m is the number of pm operations, and L_{max} is the maximum size of set L. The first term arises from placing predicate operations in the appropriate leaves of L. The second term arises from placing pd operations in the appropriate leaves of L and then updating L accordingly. The time for one update to L is O(L). The third term arises from placing the pm operations in the appropriate leaves and for each leaf, searching L for leaves with the same allowable predicate set, and updating L.

Note that the operations performed on the allowable predicate set are ignored since the number of predicates in a hyperblock is typically small enough that they can be represented as bits and manipulated by O(1) logical operations. In Section 5.5.1, the maximum number of predicates in the hyperblock of the software pipelined loops studied is 8.

It is difficult to determine the complexity of RIC without actually measuring the parameters during RIC. An upper bound can be derived since L_{max} is bounded by $|N|$, the number of nodes in the resultant CFG. Thus, the time complexity of RIC in terms of $|N|$ is $O(H|N|^2)$. Note that $|N|$ is a function of the scheduler and does not necessarily have any correlation with the number of nodes in the original CFG. Since, L_{max} is typically considerably smaller than the number of nodes in the resultant graph, this is a very loose upper bound. Section 5.5.7 presents the parameters used to calculate the time complexity of RIC for software pipelined loops.

2.7 Related Work

The phase ordering problem of global scheduling is inherent in Percolation Scheduling [29]. Actually, Percolation Scheduling is not truly a scheduling technique but a set of transformations for global scheduling. The premise however, is to schedule each operation by repeatedly applying these transformations.

Recently, researchers have noted that global scheduling should have the same basic strategy as local scheduling: Namely, heuristics should be applied to identify which operation to schedule and then the operation should be moved accordingly. This approach was proposed by Ebcioğlu and Nicolau in their global resource constrained technique [30]. This technique calculates a set of *unifiable ops* at each instruction boundary which identifies the operations that can be moved into the instruction from the successor instructions. Note that this approach assume that an instruction can have multiple branch operations and thus an instruction can have multiple successors. After code motion, the set of unifiable ops must be updated to reflect the code motion. Moon and Ebcioğlu made the following improvements to this technique: 1) the set of unifiable ops is associated with a basic block rather than a VLIW instruction, and 2) dynamic renaming and combining are performed to increase the number of unifiable ops [31].

Global scheduling techniques have also been proposed using the Program Dependence Graph (PDG) [24]. The PDG represents control dependences as well as data dependences by assigning predicates to regions. Thus, operations from control equivalent basic blocks are placed in the same region. Simply applying local scheduling to the region is a form of global scheduling [24] since the operations may have originally existed in different basic blocks. There have been several more aggressive global scheduling techniques proposed based on the PDG. Region Scheduling, proposed by Gupta and Soffa [3], applies transformations to the PDG until a region has a predefined level of parallelism. After all transformations have been applied, local scheduling is applied to the regions. This approach also has a phase ordering problem in which the two phases are 1) applying the transformations, and 2) calculating the parallelism measure. Bernstein and Rodeh [4] also use the PDG, but they only calculate the ready operations and apply code motions once for each operation. Their current implementation limits instruction scheduling by disallowing code duplication, preventing any new basic blocks from being created during instruction scheduling, and allowing an instruction to be speculatively executed above only one branch. With these restrictions, the incremental update of dataflow information is relatively simple.

The primary benefits of using ICT to simplify the task of global instruction scheduling are 1) the scheduler does not need to keep track of bookkeeping information; 2) local scheduling techniques are well-defined and widely implemented; and 3) conditional branches can be scheduled in the same manner as any other type of operation and thus, the code motion complexity incurred by moving branch operations above other branch operations [6] is eliminated.

Furthermore, at this point, no data flow information is incorporated and thus, the expensive incremental updates to the dataflow information do not need to be performed during scheduling. As more aggressive scheduling techniques requiring data flow information are applied using the ICT technique, the complexity of the ICT approach will likely increase.

The concept of simplifying the task of global scheduling to one that looks like local scheduling is not new. Frequency-based scheduling techniques such as Trace Scheduling [5],[6] and Superblock Scheduling [32], [7] generate sequential portions of code to schedule by predicting the most frequently executed paths. But, branches do not always have a dominant direction or may not be predictable. In such cases, paths can be merged by using techniques such as the ICTs.

Two previous methods merge multiple execution paths. The first method uses If-conversion [8], [9] to convert control flow into *data* dependences and assumes hardware support for conditional execution. For instance, this approach was used for vector processors with mask registers [9]. The Cydra 5 [11] is a VLIW machine with hardware support for Predicated Execution (PE). Scheduling techniques such as Hyperblock Scheduling [23] and Modulo Scheduling [12] can then be applied to processors with PE support. Chapter 6 discusses the benefits of PE support for Modulo Scheduling.

The ICT approach presented in this chapter extends the concepts of the predicate IR to processors without special hardware support. Previously, Hierarchical Reduction [13], [14], [15] has been used to merge execution paths for processors without hardware support for conditional execution. To understand the differences between the ICT approach and the Hierarchical Reduction approach, the later is discussed in some detail in the following section.

2.7.1 Hierarchical Reduction

Hierarchical Reduction is a technique that converts code with conditional constructs into straight-line code by collapsing each conditional construct into a *reduct_op* [13], [14], [15]. It is a hierarchical technique since nested conditional constructs are reduced by collapsing from the innermost to the outermost. After Hierarchical Reduction, reduct_op's can be scheduled with other operations in the loop. Hierarchical Reduction assumes no special hardware support. Thus, the conditional constructs are regenerated after modulo scheduling, and all operations that have been scheduled with a reduct_op are duplicated to both paths of the conditional construct.

A reduct_op is formed by first list scheduling both paths of the conditional construct. The resource usage of the reduct_op is determined by the union of the resource usages of both paths after list scheduling. The dependences between operations within the conditional construct and those outside are replaced by dependences between the reduct_op and the outside operations. A dependence between two operations is characterized by the type (flow, anti, and output), distance, and latency. The distance is the number of loop iterations the dependence spans. The latency is the minimum number of cycles between the operations to guarantee the dependence is met. While the type and distance of the dependence do not change, the latency for a given dependence is modified to take into account when the original operation is scheduled with respect to the reduct_op. Consider two operations op_i and op_j, where op_j is an operation from the conditional construct that has been list scheduled at time t_j. A dependence arc with latency d, source op_i, and destination op_j, is replaced by a dependence arc between op_i and the

```
for (i = 0; i < MAX; i++) {
  if (i == 0) {
    A[ i ] = (A[ i ] + c1) * c2 + c3;
  } else {
    A[ i ] = (A[ i ] + c4);
  }
}
```

(a)

```
     op1:   r1 <- 0
     op2:   r2 <- label_A
     op3:   r3 <- MAX * 8
L1:  op4:   r4 <- r2 (r1)
     op5:   beq  r1, 0, L2
     op6:   r7 <- r4 + c4
     op7:   jump  L3
L2:  op8:   r5 <- r4 + c1
     op9:   r6 <- r5 * c2
     op10:  r7 <- r6 + c3
L3:  op11:  r2 (r1) <- r7
     op12:  r1 <- r1 + 8
     op13:  bne  r1, r3, L1
```

(b)

Figure 2.14 Example C loop segment with corresponding assembly code.

reduct_op, op_r, with latency $d' = d - t_j$.[2] If instead, op_j is the source and op_i is the destination, the latency for the dependence between op_r and op_j is $d + t_j$.

Figure 2.14 shows a C code segment of a loop and the corresponding assembly code. Note that each element of array **A** is assumed to occupy 8 bytes; thus, the loop is incremented by 8 every iteration to simplify the array address calculations within the body of the loop. The loop has a simple if-then-else construct.[3] The machine being scheduled in this example is a VLIW with two uniform function units with the exception that only one branch can be scheduled per cycle. The multiply operation has a two cycle latency. All other operations have a one cycle latency. Figure 2.15 shows the dependence graph for the loop body. The dependence arcs are marked with the type, distance and latency. The types are abbreviated as follows: flow (f) and anti (a).

[2]It is possible to have a dependence with a negative latency. After op_i is scheduled, op_r can be scheduled d' cycles earlier if there are no resource conflicts and all other dependences are satisfied.

[3]Whereas, ICTs can be applied to unstructured CFGs that have arbitrary control flow generated by goto statements, Hierarchical Reduction is limited to structured CFGs.

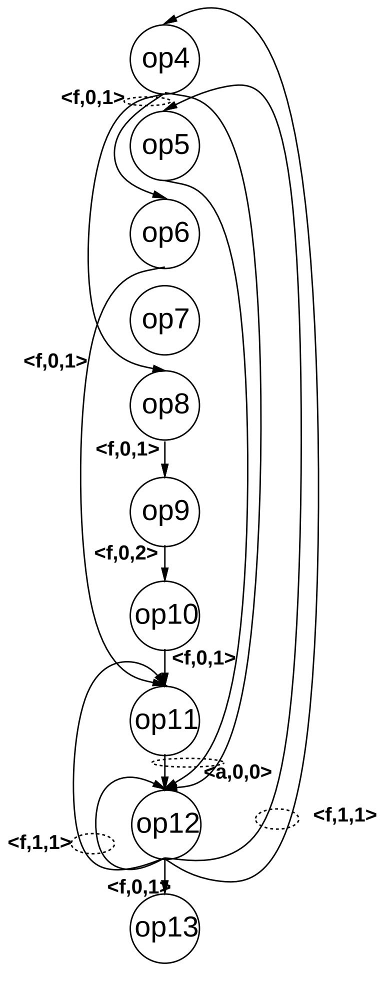


Figure 2.15 Dependence graph of example code segment.

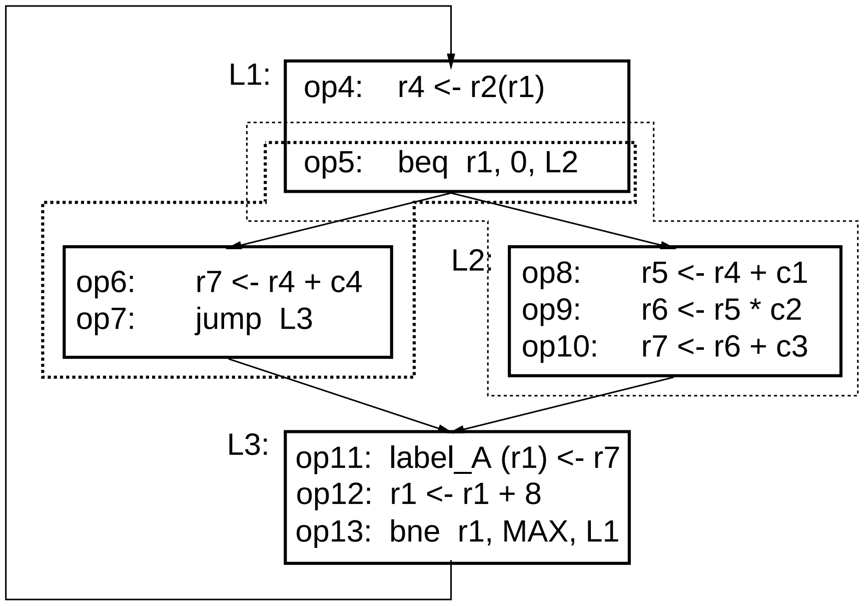
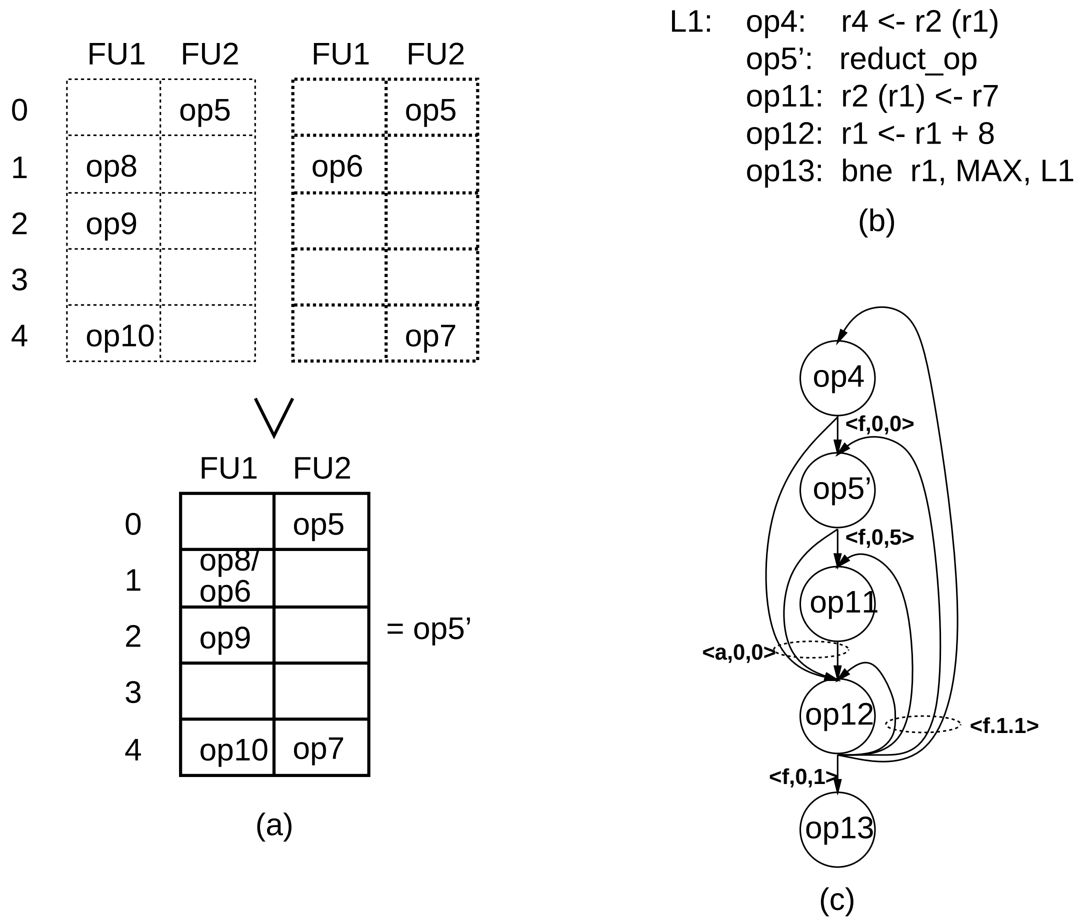


Figure 2.16 Loop CFG with paths to list schedule highlighted.

Figures 2.16 and 2.17 show how the Hierarchical Reduction technique reduces the control construct in this code segment. Figure 2.16 illustrates that the reduction operation is formed by list scheduling the two paths { **op5, op8, op9, op10**} and { **op5, op6, op7** }. Figure 2.17(a) shows the list schedule of each side of the control construct. Note that **op 7** must be scheduled in the last cycle of the reduct_op such that operations that are scheduled with the reduct_op can be copied appropriately to both paths after regeneration. The resultant resource usage of the reduct_op **op5′** formed by taking the union of the resource constraints along both paths.

Figure 2.17(b) shows the resultant loop assembly code segment after Hierarchical Reduction. Since conditional branches have been removed, local scheduling techniques can be applied. Figure 2.17(c) shows the modified loop dependence graph. Note that the latency for the flow dependence between operations **op4** and **op6** (and also between **op4** and **op8**) is originally one. These dependences are replaced by a flow dependence between **op4** and **op5′** with zero latency since **op6** and **op8** are scheduled one cycle later than **op5′**. After scheduling, the reduct_op's



```
L1:   op4:    r4 <- r2 (r1)
      op5':   reduct_op
      op11:   r2 (r1) <- r7
      op12:   r1 <- r1 + 8
      op13:   bne  r1, MAX, L1
```

Figure 2.17 Applying Hierarchical Reduction to example code segment. (a) Reduct_op generation. (b) Assembly language segment after Hierarchical Reduction. (c) Data dependence graph after Hierarchical Reduction.

must be expanded. Any operations scheduled within a reduct_op must be copied to both paths of the regenerated conditional construct.

While Hierarchical Reduction allows a code region with conditional constructs to be locally scheduled, it places some artificial scheduling constraints on the loop by first list scheduling the operations of the conditional construct. The list schedule causes the reduct_op to have a complex resource usage which may conflict with already scheduled operations and hence result in a longer schedule.

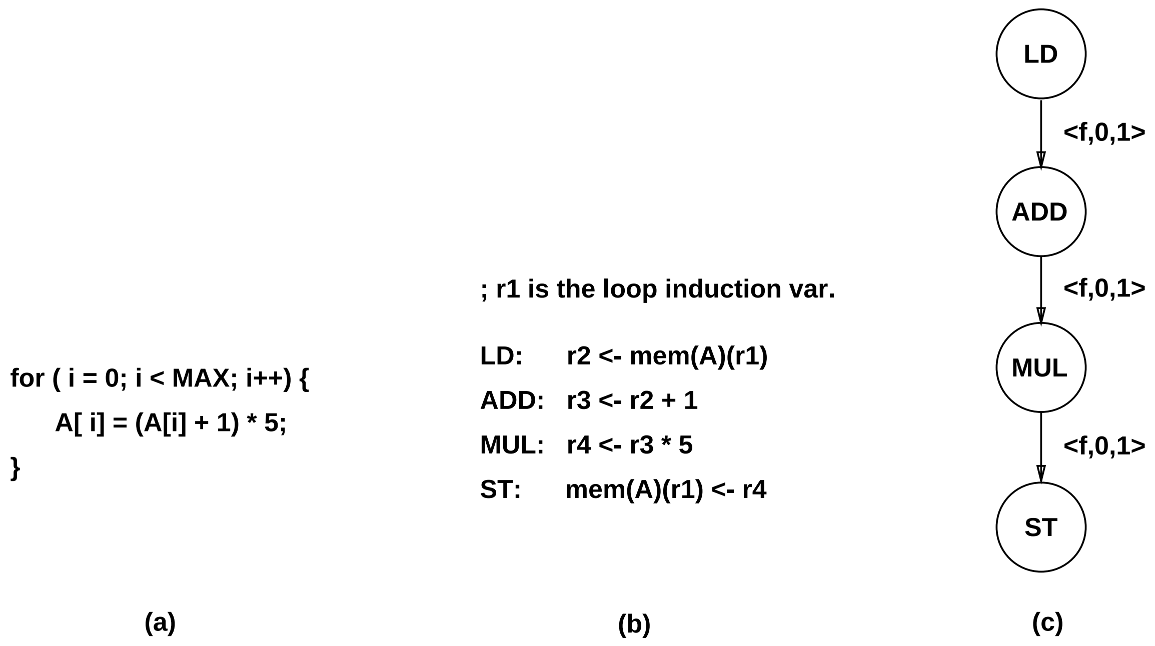
CHAPTER 3

SOFTWARE PIPELINING

Loops are an ideal source of instruction-level parallelism since operations from different iterations can be scheduled together. One approach is to unroll the loop a certain number of times and then schedule the operations from the unrolled iterations together. The operations at the end of the unrolled loop are scheduled without regard to the operations at the beginning of the loop. Thus, at the end of each iteration of the unrolled loop, some delay may occur waiting for operations to finish executing. To avoid this delay, the loop can be software pipelined [33].

Figures 3.1 and 3.2 illustrate the concept of software pipelining. Figure 3.1(a) shows the C code for a simple loop, Figure 3.1(b) shows the corresponding assembly code without the loop control code, and Figure 3.1(c) shows the data dependence graph. The loop has four operations, **LD**, **ADD**, **MUL**, and **ST**. Since there is a dependence chain involving every operation in the loop, there is no instruction-level parallelism within the loop body. Assuming that it takes one cycle to execute each operation, it would take 4 cycles to execute one iteration of the loop. Since there are no cross-iteration dependences, a better schedule can be found by overlapping the different iterations.

Figure 3.2(a) illustrates the effect of systematically overlapping different iterations of the simple loop. In this example, the first iteration is initiated in cycle 1, the second in cycle 2, and so on. By cycle 4 a steady state has been reached. The code executed in cycle 4 is the same as that executed in cycle 5. This steady state is the new loop body of the software pipelined



```
for ( i = 0; i < MAX; i++) {
     A[ i] = (A[i] + 1) * 5;
}
```

(a)

```
; r1 is the loop induction var.

LD:     r2 <- mem(A)(r1)
ADD:    r3 <- r2 + 1
MUL:    r4 <- r3 * 5
ST:     mem(A)(r1) <- r4
```

(b)

$$\begin{tikzcd}
\text{LD} \arrow[d, "{<f,0,1>}"] \\
\text{ADD} \arrow[d, "{<f,0,1>}"] \\
\text{MUL} \arrow[d, "{<f,0,1>}"] \\
\text{ST}
\end{tikzcd}$$

(c)

Figure 3.1 Simple example loop. (a) C code. (b) Assembly language without loop control. (c) Data dependence graph.

code referred to as the *kernel* as shown in Figure 3.2(b).[1] In the software pipeline, a new loop body is initiated every II cycles, where II is the *Initiation Interval.* In this example, $II = 1$. In the kernel, one iteration completes every II cycles. Note that it takes 3 cycles before the kernel code is reached. This code corresponds to the *prologue.* Likewise, after the execution leaves the kernel, an *epilogue* is needed to complete the execution of the remaining iterations.

As with its hardware counterpart, the software pipeline can be viewed as having *stages*, where each stage is II cycles long. The *throughput* of the pipeline is the rate at which iterations complete. Thus, the throughput is one iteration per II cycles.[2] Given that the prologue has N stages, the pipeline becomes full after $N + 1$ stages. The *pipeline latency*, the number of cycles until the first iteration completes, is thus, $II(N + 1)$. After the last iteration is initiated,

[1]For simplicity, the explicit loop back branch operation is not shown here but the loop back edge is.

[2]It is possible to schedule more than one iteration per II cycles. In general, if n iterations are scheduled, then the throughput is n iterations per II cycles.

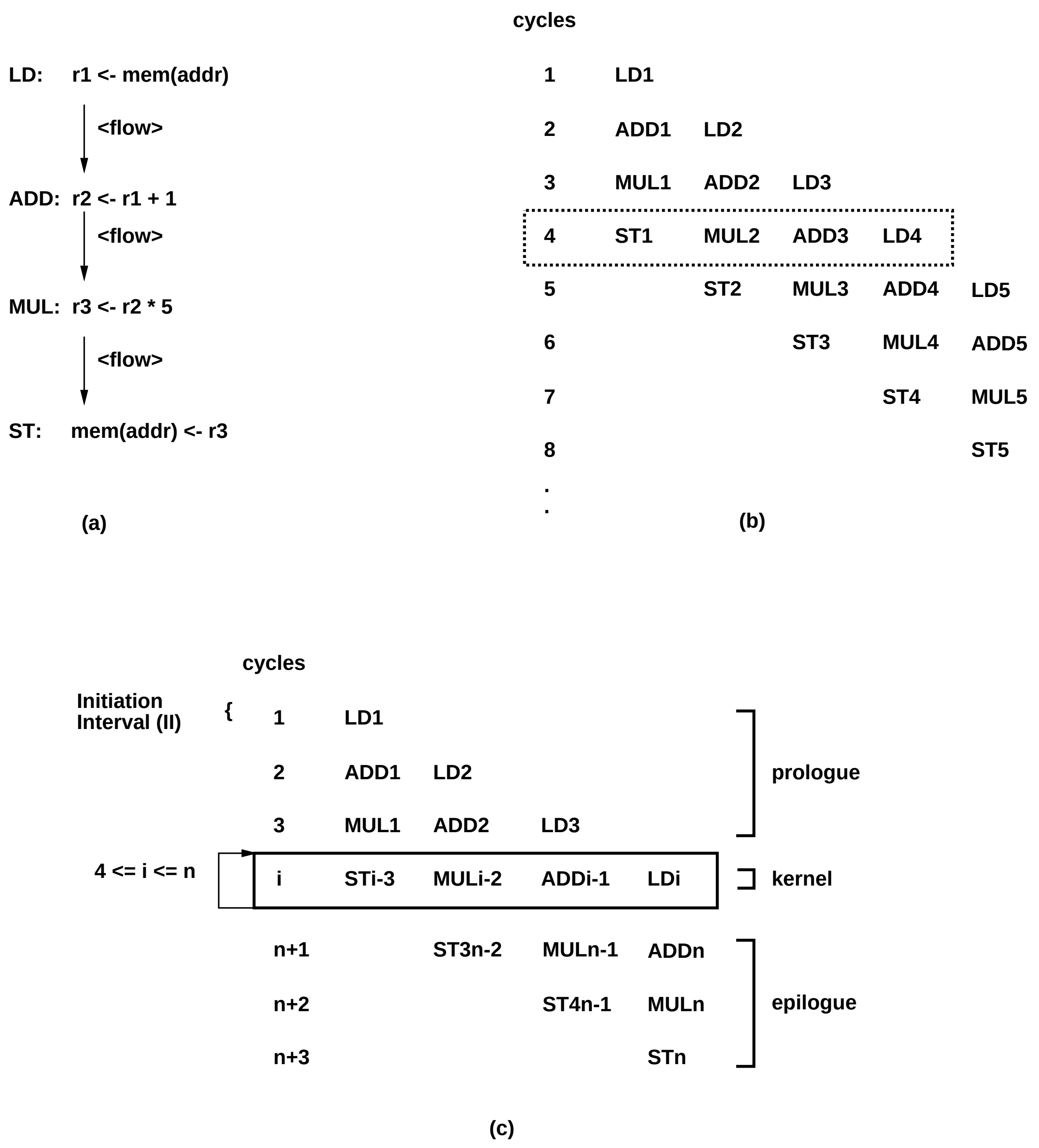


Figure 3.2 Basic concept of software pipelining. (a) Example loop data dependence graph. (b) Overlapping loop iterations. (c) Software pipeline of example loop.

control exits the kernel and there are N stages in the epilogue to empty the pipeline. Note that this is the most simplistic view of a software pipeline. As discussed later, it is possible to not have any epilogue code if the prologue code is speculatively executed [34], [35]. Also, the number of stages in the prologue and epilogue may depend on the register allocation scheme. Furthermore, it is possible to have multiple epilogues [36], [37].

The benefit of software pipelining can be understood by comparing the time to execute 100 iterations of the loop of Figure 3.1 before and after software pipelining. Without any overlapping, the loop in Figure 3.1(b) takes 4 cycles to execute and thus 100 iterations would take 400 cycles. After software pipelining, the loop takes 3 cycles to fill the pipeline, executes the kernel 97 times, and takes 3 cycles to empty the pipeline. In total, the loop takes 103 cycles to execute. Thus, a 3.88 speedup is achieved. The super-linear speedup illustrates that software pipelining can be used to utilize parallel resources and hide long operation latencies.

Software pipelining is a powerful scheduling technique. It can be applied to loops with cross-iteration dependences and with arbitrary control flow including loops with unknown loop bounds (e.g., while loops). It is most often used to schedule inner loops, but can also be hierarchically applied to outer loops [15], [31].

Software pipelining was originally used for hand coding microprogrammed pipelined machines [38] and was applied to multiprocessors in the form of *doacross* loops [39], [40], [41]. More recently, it has been applied to a wide set of architectures [42], [43], [44]. In this dissertation we focus on the approaches developed for high-performance RISC and CISC processors such as superscalar and VLIW processors.

3.1 Two Approaches

Two approaches to software pipelining are based on local and global scheduling techniques, respectively.

3.1.1 Local scheduling approach

Originally, only loops without conditional constructs were considered for software pipelining [16], [45], [46]. These techniques take the same tack to software pipelining by first determining the minimum II based on the loop requirements and the machine constraints.

3.1.1.1 Modulo Scheduling

The most general approach, originally proposed by Rau and Glaeser [16], is Modulo Scheduling [16], [47], [14], [48]. Modulo Scheduling determines a lower bound for II based on the resource and recurrence constraints. After a lower bound is found, local scheduling techniques can be used to schedule the loop.

The lower bound on II depends on the loop resource usage and recurrence constraints. Patel has shown that an acyclic dependence graph can be scheduled to fully utilize at least one resource [49]. Thus, if an iteration uses a resource r for c_r cycles and there are n_r copies of this resource, then the minimum II due to resource constraints, RII, is

$$RII = \max_{r \in R} \left\lceil \frac{c_r}{n_r} \right\rceil ,$$

where R is the set of all resources.

II is also bounded by the worst case recurrence circuit. A recurrence circuit exists if there is a dependence between an operation and an instance of itself d iterations later. Call d the distance of the dependence. If a dependence edge e in a recurrence circuit has latency l_e

and connects operations that are d_e iterations apart, then the minimum II due to recurrence circuits, CII, is

$$CII = \max_{c \in C} \left\lceil \frac{\sum\limits_{e \in E_c} l_e}{\sum\limits_{e \in E_c} d_e} \right\rceil ,$$

where C is the set of all recurrence circuits and E_c is the set of edges in recurrence circuit c.

To calculate CII, Huff [50] proposed an efficient algorithm to find a circuit with the minimum cost-to-time ratio, where a dependence arc is viewed as a cost of -l and a time of d. Given that M is the minimum cost-to-time ratio, $CII = \lceil -M \rceil$.

Once RII and CII have been calculated, the initial II, II_{init}, is determined,

$$II_{init} = max(RII, CII).$$

After the initial value of II is found, the loop can be software pipelined by applying local scheduling techniques and using a Modulo Resource Table (MRT). The MRT is used to indicate that once a resource is scheduled in cycle t, it will use the same resource in cycles $t+II$, $t+2II$, $t+3II$, etc. [47]. The MRT has II rows and N columns, where N is the number of resources available to schedule each cycle. Figure 3.3 shows the MRT for the example loop shown in Figure 3.1(b) assuming two uniform pipelined function units. Uniform is used to indicate that they can execute any type of operation. Since there are no recurrences, $II = RII = \lceil \frac{4}{2} \rceil = 2$. Thus, there are two rows and two columns in the MRT.

If a schedule is not found for a given II, then II is incremented and the loop body is rescheduled. This process repeats until an II is found that satisfies the resource and recurrence constraints. If the loop does not have any recurrences, then a schedule can usually be found for the minimum II [16]. In the presence of recurrences, heuristics were developed in the Cydra 5

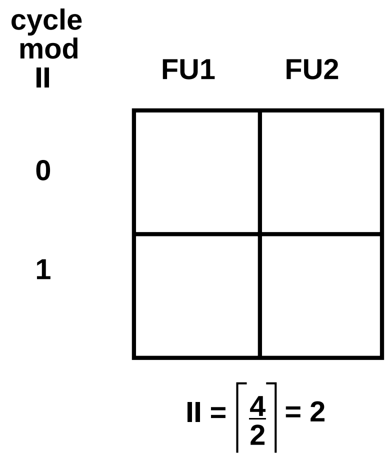


Figure 3.3 Modulo Resource Table (MRT) for example loop.

compiler to generate near-optimal schedules. The basic principle is to schedule the recurrence node first and then the nodes not constrained by recurrences [51], [52], [53].

After II is scheduled, the code for the software pipelined loop is generated. The software pipeline consists of the prologue, kernel, and epilogue. The number of stages in the prologue is $S = \left(\left\lceil\frac{latest\ issue\ time}{II}\right\rceil - 1\right)$. The number of stages in the kernel depends on the overlapping register lifetimes. Since the loop body spans multiple II's, a register lifetime may overlap itself. If this happens, then the registers for each lifetime must be renamed. This can be done in hardware using a rotating register file as in the Cydra 5 [11]. With hardware support for register renaming, the kernel has one stage. Without special hardware support, Modulo Variable Expansion can be used to unroll the kernel and rename the registers [15]. The number of times the kernel is unrolled, U, is determined by the maximum register lifetime modulo II. The last part of the pipeline is the *epilogue* which consists of S stages needed to complete executing the last S iterations of the loop. In total, the software pipeline has $2S + U$ stages.

This is actually a very simplistic view of the issues involved in generating the software pipeline. Rau, Schlansker, and Tirumalai provide a good overview of the possible code generation schemata for modulo scheduled loops [37].

To summarize, the essence of Modulo Scheduling is to use the resource and recurrence constraints to define an lower bound on the schedule length. Then local scheduling techniques can be applied to try to achieve this lower bound. These techniques take both the resource and dependence constraints into account during scheduling. The details of our Modulo Scheduling implementation are provided in the next chapter.

3.1.2 Global scheduling approach

Three well-known software pipelining techniques are based on global scheduling. Aiken and Nicolau's Perfect Pipelining technique compacts and unrolls the loop body until a pattern is reached [17]. Su and Wang have developed a similar technique, GURPR*, that does not have the computational complexity associated with determining a repeating pattern and that reduces the code expansion overhead [18]. Ebcioğlu and Nakatani's Enhanced Pipeline scheduling technique also compacts the code by moving operations upward until an instruction is filled. Once an instruction is filled, it is moved across the loop backedge and is again available for scheduling. In this fashion, software pipelining is achieved since operations from different iterations are scheduled together [19]. A brief description of these approaches is provided below.

3.1.2.1 Perfect Pipelining

The Perfect Pipelining algorithm proposed by Aiken and Nicolau [17], [29] performs software pipelining in two phases. In the first phase, global code motion is applied to move operations up as early as possible in the loop body. To avoid unnecessary race conditions during this phase, no resource constraints are applied. A race condition refers to the case when moving one operation up blocks another operation. The second phase consists of unrolling the compacted iterations and scheduling operations. This is an iterative process. An iteration of the loop is peeled and then scheduled as early as possible using both resource and recurrence constraints. Then the schedule is checked for a repeating pattern. Iterations are peeled and scheduled until a pattern appears.

Bodin and Charot also present a similar unroll and schedule technique [54].

3.1.2.2 GURPR*

One problem with Perfect Pipelining is that identifying a pattern is a complex task [55]. The GURPR* algorithm avoids the need for pattern matching [18]. In this approach, the loop is again compacted and then unrolled and scheduled. But, the loop is unrolled only until the last operation of the first iteration is scheduled. Then II is determined by searching for the smallest number of consecutive instructions that contain all the operations in the loop. Once II is determined, the software pipeline schedule is generated.

Scheduling after unrolling is restricted to overlapping loop iterations and delaying instructions if necessary. If an instruction cannot be scheduled due to resource constraints, then it is delayed one cycle. If it still cannot be scheduled, then a new cycle is inserted into the loop schedule to hold the instruction. Notice that since the loop has already been compacted and this ordering is preserved during scheduling, intra-iteration data dependences will be preserved. Furthermore, the iterations are overlapped according the worst case recurrence circuit. Thus, cross-iteration dependences are also preserved. Therefore, during scheduling, only the resource constraints are considered. After scheduling, however, when an interval is selected, it may contain multiple copies of an operation. When these copies are deleted, they create gaps in the schedule.

3.1.2.3 Enhanced Pipeline Scheduling

The previous two methods software pipeline by using the loop body as the scheduling unit. Enhanced Pipeline Scheduling (EPS) achieves software pipelining by globally scheduling loop operations [19]. To schedule, a fence instruction is placed at the beginning of the loop. Global code compaction techniques such as Percolation Scheduling [56], [29], Selective Scheduling [31], or Region Scheduling [35], [57] are used to move operations upward to fill the fence instruction. Resource restrictions are enforced only at the fence instruction, not within the loop during compaction. Thus, it is possible to over schedule an instruction within the loop body.

Once a fence instruction is filled, the instruction is moved across the loop back edge and copied into the prologue. This activates software pipelining since the operations of the previous fence instruction originate in iteration i and once moved across the loop back edge, can be scheduled with operations of iteration $i+1$. The scheduling process continues until all operations from the original iteration have been scheduled. Notice that a valid loop exists during every phase of scheduling. Thus, no pipeline must be created after scheduling.

3.1.3 Discussion

There are advantages and disadvantages to every approach. The tradeoffs are analyzed below.

Local versus global scheduling.

The fundamental difference between the approaches based on local scheduling and those based on global scheduling is simply that they are based on different scheduling techniques.

Modulo Scheduling can use simple local scheduling techniques as list scheduling [51], [15] or more complicated backtracking algorithms [53]. Both of these local scheduling techniques require that the scheduler keep track of only data dependences and resource constraints. On the other hand, global scheduling techniques must also take into account data flow information. Global scheduling techniques such as Selective Scheduling [31] and Region Scheduling [4], [57] must update the data flow information every time an operation is moved across basic block or region boundaries.

Since the current Modulo Scheduling techniques do not use data flow information, operations are not scheduled speculatively. Disallowing speculative execution does not affect the throughput of the pipeline. As Rau and Glaeser showed, it is possible to achieve the lower bound on II for loops with simple resources and acyclic dependences [16]. Furthermore, Jones and Allan empirically showed that Modulo Scheduling can achieve a tighter II than EPS [55]. In fact, it is possible that speculatively executing instructions may decrease the throughput (increase II) since additional copy operations may be needed to ensure program correctness [35].

One benefit of speculative execution is that it may decrease the pipeline latency. Also, in EPS, all of the prologue operations can be viewed as speculative on the loop back branch. Thus, software pipelines generated by the EPS technique do not have epilogues. The disadvantage of speculative execution for conditional branches within the loop (i.e., not the loop back branch), is that the throughput of the pipeline may decrease. This is because operations moved above the branch will be executed along both paths of control. If an operation is moved into an operation slot that would be empty otherwise, there is no penalty in executing the operation speculatively. If however a slot is filled by an operation from a less frequently executed path when it could be filled by an operation from a more frequently executed path, then executing the operation will penalize the frequently executed code and likely decrease the performance of the loop. This is particularly a problem for fixed II techniques (described below) such as GURPR* [58].

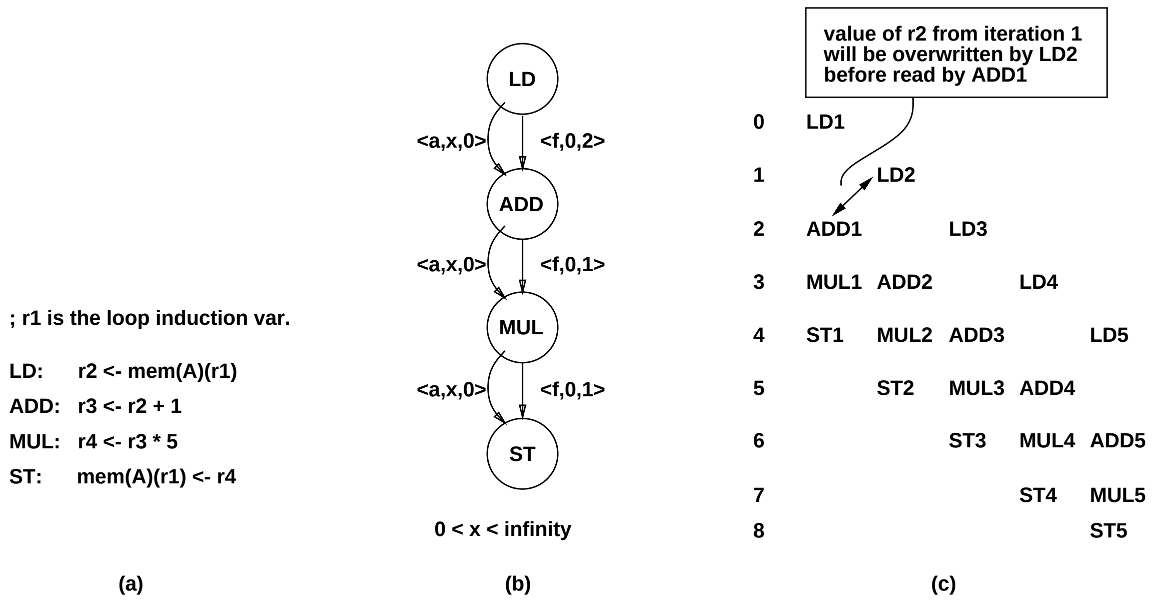
Register renaming.

Register renaming is needed to remove anti dependences to allow scheduling operations from different iterations that use the same register. Figure 3.4 illustrates how anti dependences can prevent software pipelining. Figure 3.4(a) shows the simple loop assembly code and Figure 3.4(b) shows the data dependence graph. Note that for this example we assume that the load takes 2 cycles. Also note that there are actually cross iteration anti dependences due to register re-use. Figure 3.4(c) shows an example software pipeline. In the pipeline, the load of the second iteration (**LD2**) is scheduled before the add of the first iteration (**ADD1**). Thus, to preserve the result of the load from the first iteration (**LD1**), the destination of **LD2** must be renamed.

Two techniques have been proposed for renaming register lifetimes with Modulo Scheduling. Lam proposed Modulo Variable Expansion (MVE) to rename overlapping register lifetimes after the kernel is scheduled [14], [15]. MVE is similar to the scalar expansion technique proposed for vector processors [59]. MVE requires that the kernel code be unrolled enough times to rename the longest register lifetime. To avoid unrolling, the overlapping register lifetimes can be dynamically renamed using hardware support such as the rotating register file of the Cydra 5 supercomputer [11], [12].

The EPS technique uses “dynamic”[3] renaming during scheduling to remove anti dependences that prevent upward code motion [31], [35]. The disadvantage of dynamic renaming is that

[3]The authors use the term dynamic renaming since the apply the technique during scheduling rather than after scheduling. Dynamic renaming typically refers to run-time renaming methods.



```
; r1 is the loop induction var.

LD:    r2 <- mem(A)(r1)
ADD:   r3 <- r2 + 1
MUL:   r4 <- r3 * 5
ST:    mem(A)(r1) <- r4
```

```
0   LD1
1         LD2
2   ADD1        LD3
3   MUL1  ADD2        LD4
4   ST1   MUL2  ADD3        LD5
5         ST2   MUL3  ADD4
6               ST3   MUL4  ADD5
7                     ST4   MUL5
8                           ST5
```

Figure 3.4 Register renaming for software pipelined loops. (a) Assembly language without loop control. (b) Data dependence graph. (c) Example software pipeline highlighting need for register renaming.

it involves inserting copy operations which can increase the resource requirements and thus possibly increase *RII*.

Bounding II.

The *II* of the resulting software pipeline is bounded by the resource and recurrence constraints. Only Modulo Scheduling takes both constraints into account before scheduling. Ignoring these constraints to bound *II* during scheduling results in an *overpipelined* schedule. Overpipelining means that more iterations are pipelined than necessary [35]. The result of overpipelining is that the pipeline latency will increase and the code expansion will increase. Most researchers acknowledge that *CII* should be calculated before scheduling and used as a lower bound for scheduling. The GURPR* technique overlaps iterations according to *CII* [18]. Lee proposed an improvement to EPS which increases the fence region from one instruction to *CII* instructions [35] to avoid overpipelining. Both GURPR* and EPS ignore *RII*. Bockhaus showed the benefit of considering RII for GURPR* [58]. For the EPS technique, RII is not a tight lower bound for EPS since dynamic renaming continually increases the resource constraints and thus continually increases RII. Nevertheless, using the initial value of RII to determine the fence region size is better than ignoring it altogether.

Integrating resource constraints.

RII is the lower bound on II calculated using the loop's resource usage and the machine's resource constraints. Only Modulo Scheduling accounts for the exact resource constraints during all phases of scheduling by using the Modulo Resource Table [47]. While Perfect Pipelining compacts the loop body, it ignores the resource constraints to avoid race conditions. Here a race condition refers to the case where moving one operation blocks other operations from moving. The GURPR* technique also ignores resource constraints when compacting the loop body. It does however, use the resource constraints when overlapping different iterations. Since the final II selected by the GURPR* technique may have redundant operations which are deleted, II may be larger than RII. The EPS technique applies the resource constraints only to the fence region. Thus, code motion through the loop body may cause instructions to be overscheduled due to the insertion of bookkeeping operations. If this occurs, then the overscheduled instructions must be broken up after the software pipeline is determined. This may result in a pipeline with a lower throughput (larger II) than necessary. A more critical problem is that since the resource constraints are applied only to the fence region, it is not clear that the algorithm will work for machines with more complicated resource usage patterns. For example, such patterns arise when scheduling the result bus.

Fixed II versus variable II.

One drawback of Modulo Scheduling and GURPR* is that II remains constant for every iteration [60]. Thus, a loop that has a conditional construct with an infrequently executed path much longer than the frequently executed path will take longer to execute than techniques which have a variable II such as EPS [34], [19].

Iterations per II.

Another consideration is how many iterations are software pipelined together. Perfect Pipelining is able to schedule operations from multiple iterations within one II. Other techniques can also schedule multiple iterations per II by unrolling the loop body before scheduling [50]. This may be desirable to achieve a tighter schedule. For example, consider a loop with $RII = \lceil \frac{3}{2} \rceil = 2$. If the loop is unrolled once, then $RII = \lceil \frac{6}{2} \rceil = 3$. Thus, the throughput is increased from one iteration per 2 cycles to one iteration per 1.5 cycles.

Hardware support.

While no software pipelining techniques *require* special hardware support, several were designed with special hardware support in mind. For example, EPS was originally proposed for the IBM VLIW processor [34]. This architecture supports multi-way branching and conditional execution via the *tree* instruction. Recently, EPS has also been shown to be effective for RISC and superscalar processors [31]. Likewise, Perfect Pipelining uses a multi-way branching instruction. For Modulo Scheduling, Predicated Execution has been proposed for conditionally executing operations. The benefit of Predicated Execution over multi-way branching is that it reduces the code expansion. As we will show in Chapter 6, all of these special architectures prevent the branch unit from becoming a limiting resource.

For superscalar processors which must remain object code compatible with their scalar predecessors, these proposed features cannot be used since they require changing the instruction set. For such architectures, techniques are needed that do not assume any special hardware support.

CHAPTER 4

ENHANCED MODULO SCHEDULING

This chapter presents the Enhanced Modulo Scheduling (EMS) technique [61]. As Figure 4.1 shows, EMS is Modulo Scheduling with ICTs. This approach to Modulo Scheduling is referred to as "enhanced" for loops with conditional branches because, EMS

(1) allows operations to be scheduled independently,

(2) uses the most constrained resource along *any* path to form the lower bound on the schedule length, and

(3) requires no special hardware support.

While the previous techniques using either Hierarchical Reduction or hardware support for Predicated Execution have some of these benefits, neither has all.

Unlike Hierarchical Reduction, ICTs do not perform any prescheduling and thus, they do not limit the scheduling freedom of operations within a conditional construct with operations outside the construct. Whereas Predicated Execution requires that all operations be fetched, by regenerating the control flow graph (CFG), the II for EMS is bounded by the most constrained resource along any path rather than the sum of all resource constraints along any path. Finally, EMS does not require any special hardware support. Thus, it can be used with existing architectures and any future ones that are limited due to object code compatibility. Note that the use of ICTs does not preclude using hardware support to improve program performance.

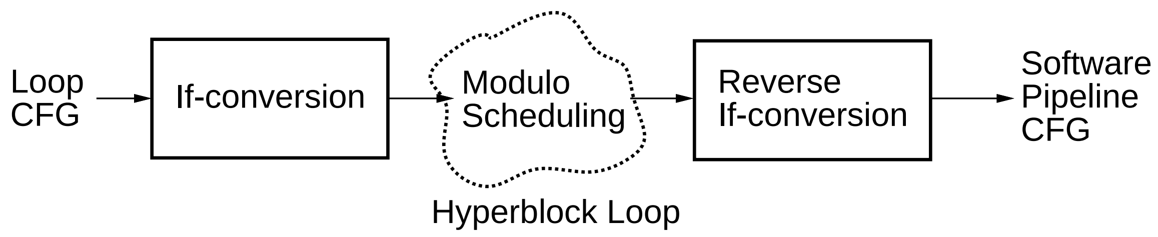


Figure 4.1 Enhanced Modulo Scheduling (EMS).

```
for (i = 0; i < MAX; i++) {
  if (i == 0) {
    A[ i ] = (A[ i ] + c1) * c2 + c3;
  } else {
    A[ i ] = (A[ i ] + c4);
  }
}
```

(a)

```
     op1:   r1 <- 0
     op2:   r2 <- label_A
     op3:   r3 <- MAX * 8
L1:  op4:   r4 <- r2 (r1)
     op5:   beq  r1, 0, L2
     op6:   r7 <- r4 + c4
     op7:   jump  L3
L2:  op8:   r5 <- r4 + c1
     op9:   r6 <- r5 * c2
     op10:  r7 <- r6 + c3
L3:  op11:  r2 (r1) <- r7
     op12:  r1 <- r1 + 8
     op13:  bne  r1, r3, L1
```

(b)

Figure 4.2 Example loop segment with conditional branches. (a) C code segment. (b) Assembly language segment.

An outline of the EMS algorithm is presented in Figure 4.3. The remainder of this chapter explains the steps of this algorithm. In addition to presenting the EMS algorithm, we will provide some of the details of our implementation of the EMS algorithm in the IMPACT prototype compiler.

To illustrate the EMS algorithm, the example loop in Figure 4.2 is used. Note that the code in Figure 4.2(b) has been optimized by the classical optimizations induction variable strength reduction and induction variable elimination [27]. Since each element of array **A** is assumed to occupy 8 bytes, the loop is incremented by 8 every iteration to simplify the array address calculations within the body of the loop.

Algorithm EMS: Given a loop, apply Enhanced Modulo Scheduling to software pipeline the loop.

```
Step 1: determine whether loop is appropriate for software pipelining
Step 2: build data dependence graph
Step 3: apply Induction Variable Reversal
Step 4: determine lower bound on $II$
Step 5: apply If-conversion
Step 6: rebuild data dependence graph
Step 7: while $II < MAX\_II$ {
            modulo schedule loop using MRT(II)
            if success
                break
            else
                increment $II$
            }
            if not success
                done (no software pipeline schedule can be found)
Step 8: apply Modulo Variable Expansion
Step 9: generate software pipeline
Step 10: apply Reverse If-conversion
```

Figure 4.3 The EMS algorithm modulo schedules loops with conditional branches using ICTs.

4.1 Step 1. Selecting Loops to Software Pipeline

One of the strengths of software pipelining as a loop scheduling technique is the fact that it can be applied to a diverse set of loops. It can be applied to loops with arbitrary conditional branches. It can be applied to loops with known upper bounds (e.g., *for* loops) and to loops with unknown upper bounds (e.g., *while* loops, loops with exits, etc.) [52]. It can be applied to loops with no cross-iteration dependences (e.g., doall loops) as well as to loops with recurrences (e.g., doacross loops). It is usually applied to inner loops but can also be applied hierarchically to outer loops [14], [31].

Acknowledging the versatility of the approach, we have restricted the loops that we consider in our implementation to those meeting the following criteria:

- inner loop
- no function calls
- no multi-node recurrences
- normalized *for* loops
- no early exits

The reasons for these restrictions are practical rather than a limitation of the EMS approach.

We consider only inner loops since inner loops typically have a sufficient number of operations to fully utilize the machine resources. We do not allow functions calls. This is a standard restriction. Note that loops with function calls can be software pipelined by first applying function-inline expansion. We do not support multi-node recurrences, recurrence circuits involving more than one operation, because we do not have detailed memory dependence analysis at the register transfer level of the IMPACT compiler.[1] At present we consider only *for* loops without early exits because we do not generate the multiple epilogues needed for loops with early exits. This simplifies the application of some optimizations. The experiments in this dissertation are performed on numerical benchmarks which contain mostly for loops. To software pipeline non numerical code, it will be important to allow loops with early exits.

4.2 Step 2. Build Data Dependence Graph

After the loop to be software pipelined has been selected, its data dependence graph is generated. Figure 4.4 shows the dependence graph of the example loop segment. To reiterate, the following terminology is used. The arcs are labeled with the tuple $< type,\ distance,\ latency >$. The type is either flow (f), anti (a), or output (o). The distance is the number of iterations the dependence spans. The latency is the minimum number of cycles between the operations to guarantee the dependence is met. Note that cross iteration anti dependences are not shown since they will be removed by Modulo Variable Expansion as described in Step 8.

A *recurrence circuit* is a cycle in the data dependence graph. A recurrence circuit will only exist if the loop has cross-iteration dependences. These are dependences with distances greater than zero. A multi-node recurrence is a recurrence circuit containing more than one operation. Most recurrences are multi-node recurrences. Those that are not multi-node are referred to as self-recurrences. A self-recurrence involves one operation. An induction variable, $i = i + 1$, causes a self-recurrence in the dependence graph since it has a flow dependence of distance one to itself.

[1]At this time, memory analysis has been implemented at the front-end of the IMPACT compiler. Techniques are being explored to migrate the memory dependence information down to the register transfer level.

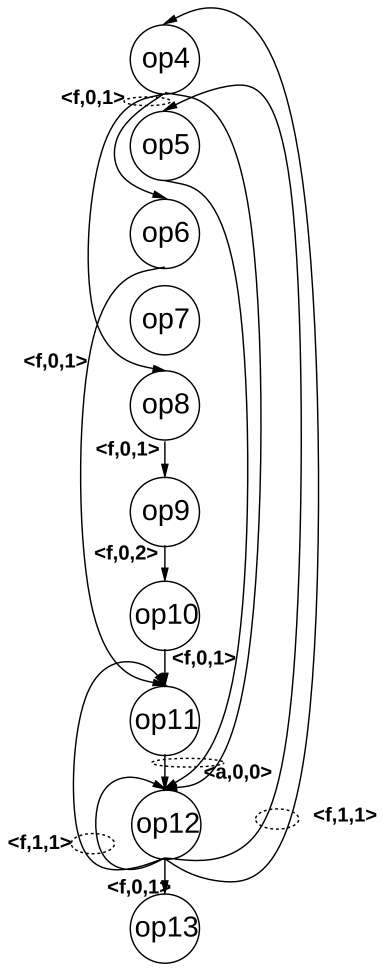


Figure 4.4 Dependence graph of example code segment.

4.3 Step 3. Apply Induction Variable Reversal

In the presence of recurrences, Dehnert et al, discuss the importance of compiler optimizations such as Back-Substitution and Load-Store Removal to reduce CII [12]. Another important optimization is needed to reduce multi-node recurrences involving induction variables to self-recurrences. Most compilers generate post increment induction variables such that the induction variable is incremented at the end of the loop body. Thus, within one iteration an operation that has the induction variable as a source operand is not flow dependent upon the induction variable. There is however, a flow dependence from the induction variable increment operation to the operation that uses the induction variable in the next iteration. For schedulers that schedule one iteration of the loop, post increment induction variables reduce the critical path length of paths containing the induction variable. When software pipelining, post incremented induction variables cause multi-node recurrences as shown in Figure 4.4. The loop increment *r1* of the assembly code segment in Figure 4.2(b) is a post increment loop induction variable since **op12** is placed at the end of the loop.

If this loop is software pipelined, the CII for this loop is determined by the worst case recurrence circuit {**op4, op8, op9, op10, op11, op12**}. Thus, $CII = \lceil \frac{5}{1} \rceil = 5$, since the sum of the latencies is 5 and the dependence distance is one. It is possible to convert the multi-node recurrence into a self-recurrence by applying *Induction Variable Reversal.*

Induction Variable Reversal converts post increment induction variables into pre increment induction variables. Thus, all operations within one iteration will be flow dependent upon the loop induction variable. These flow dependent operations do not need to be scheduled within one II and thus will not bound II.

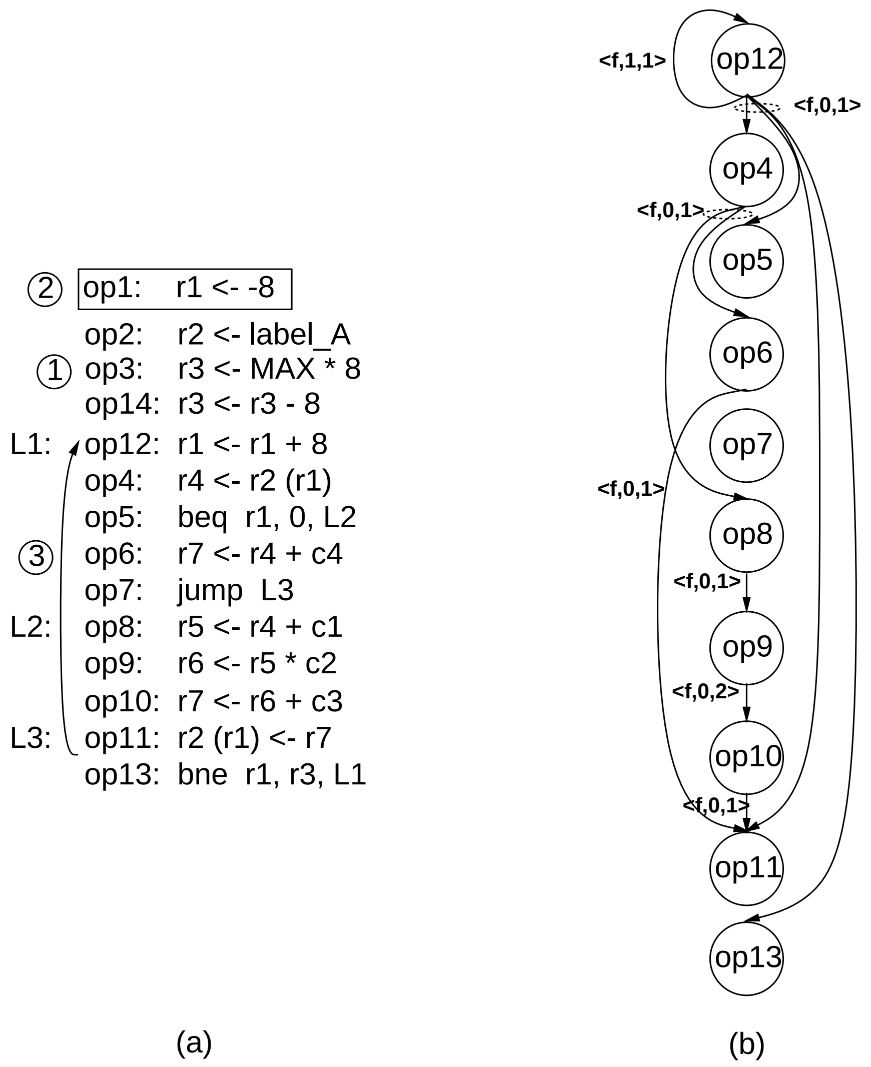
Algorithm Induction Variable Reversal: Given a loop, convert post increment induction variables into pre increment induction variables.

```
$\forall$ op $\in$ loop {
    /* check if induction variable */
    if((add operation) and (destination is a register) and (destination equal to source)
    and (unique definition in loop)) {
        /* check if post increment */
        if((no output flow dependences) or (only output flow dependences to loop back branch) {
            /* check if loop increment */
            if(output flow dependence to loop back branch) {
                decrement loop bound in preheader by loop increment
            }
            decrement loop induction variable in preheader by loop increment
            move operation to beginning of loop
        }
    }
}
```

Figure 4.5 Algorithm Induction Variable Reversal converts multi-node induction variable recurrences into self-recurrences.

To perform this optimization, the operation that increments the induction variable is moved to the beginning of the loop, and the initial value of the induction variable is decremented by the increment value. When the induction variable spans more than one II, overlapping lifetimes are renamed using Modulo Variable Expansion. Notice that Induction Variable Expansion combined with Modulo Variable Expansion will result in a transformation similar to Induction Variable Expansion, which is done for unrolled loops [62].

The algorithm for Induction Variable Reversal is presented in Figure 4.5. Note that the loop increment is a special case for Induction Variable Reversal since the loop increment is not a true post increment operation. Rather, there is a flow dependence between the loop increment and the loop back branch. Thus, to convert the loop increment to a pre increment induction variable, the loop increment must be subtracted from the loop bound in the loop preheader.



```
②      op1:   r1 <- -8
       op2:   r2 <- label_A
①      op3:   r3 <- MAX * 8
       op14:  r3 <- r3 - 8
L1:    op12:  r1 <- r1 + 8
       op4:   r4 <- r2 (r1)
       op5:   beq  r1, 0, L2
③      op6:   r7 <- r4 + c4
       op7:   jump  L3
L2:    op8:   r5 <- r4 + c1
       op9:   r6 <- r5 * c2
       op10:  r7 <- r6 + c3
L3:    op11:  r2 (r1) <- r7
       op13:  bne  r1, r3, L1
```

Figure 4.6 Applying Induction Variable Reversal to example loop. (a) Assembly language segment. (b) Loop data dependence graph.

Figure 4.6(a) shows the application of Induction Variable Reversal to the example code segment. First, **op14** is inserted to decrement the loop bound by the loop increment. Second, the loop increment, 8, is subtracted from the induction variable , **r1**, in the loop preheader. This ensures that the loop induction variable contains the proper initial value after **op12**. Third, **op12** is moved to the beginning of the loop. Notice that there are no longer any multi-node recurrences in the dependence graph shown in Figure 4.6(b).

4.4 Step 4. Determine Lower Bound on II

The lower bound on II is determined by the most constrained resource *along any path* and by the most constrained recurrence circuit. Since the CFG will be regenerated after modulo scheduling, the schedule will be constrained by the worst case resource constraint along any path. Thus, the lower bound on II due to resource constraints, RII is

$$RII = \max_{p \in P} \left(\max_{r \in R} \left\lceil \frac{c_{pr}}{n_r} \right\rceil \right),$$

where P is the set of all execution paths and R is the set of all resources.

The value of RII can be determined after applying If-conversion by using the Predicate Hierarchy Graph (PHG) [23], [20] to distinguish different control flow paths.

Note that since the modulo scheduling algorithm in Step 7 does not speculatively schedule operations, this will be a tight lower bound. When an operation is speculatively executed it increases the resource usage along some paths. For example, consider the example partial CFG in Figure 4.7(a). If the load operation is speculatively scheduled (moved from **BB2** to **BB1**), then two actions must be taken due to the fact that the value of **r1** is used in **BB3**. First, the destination of the load is changed to $r1'$ and a copy operation is inserted into **BB2** to

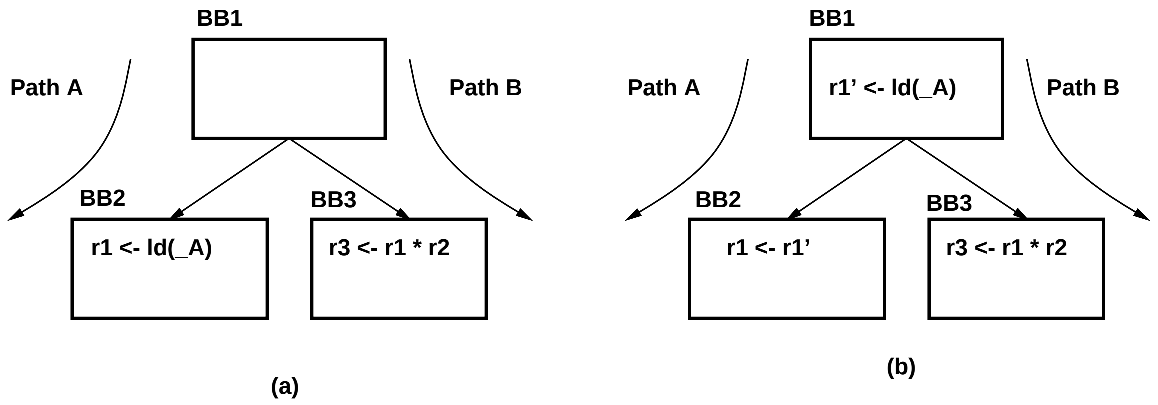


Figure 4.7 Effect of speculative scheduling on RII. (a) Partial CFG. (b) Partial CFG with speculatively scheduled load.

restore the value in **r1**. Note that the copy operation increases the resource usage along **Path A** and the speculative load operation increases the resource usage along **Path B**. Thus, while the action of scheduling by delaying operations, as done in modulo scheduling, may increase the pipeline latency, it is a deterministic scheduling algorithm since RII is predictable.

As discussed in Section 3.1.1.1, II is also bounded by the worst case recurrence. For loops without cross-iteration memory dependences, CII is determined by the recurrence circuits containing loop induction variables. If Induction Variable Reversal is applied, most induction variables become self-recurrences. The only possible exception is recurrences involving the loop increment and the loop back branch. If the loop increment and loop back branch is one operation, then the loop increment can always be converted into a self-recurrence. If this is not the case, then there is multi-node recurrence circuit containing the loop increment and the loop back branch. In this case, there are two possible scenarios, one for single epilogue software pipelines and the other for multiple epilogue software pipelines.

For loops with only one epilogue, it is assumed that the loop executes at least $S + U$ times where S is the number of stages in the prologue and U is the number of stages in the kernel.

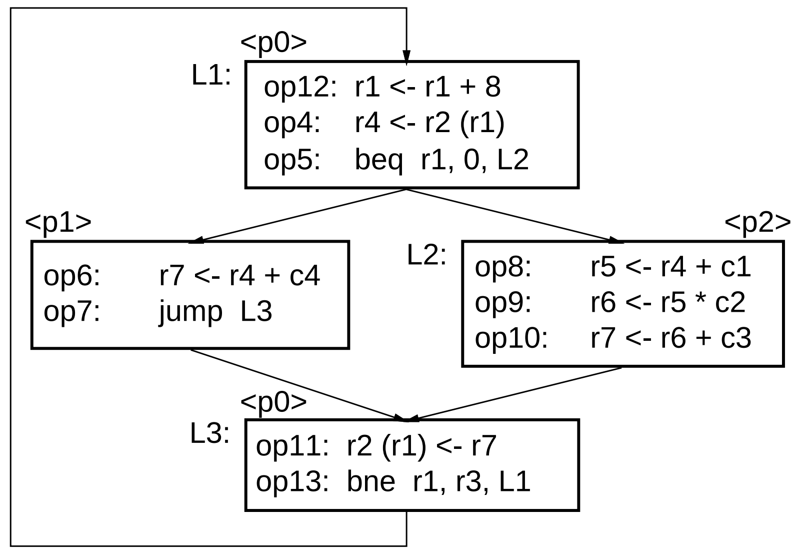
Furthermore, once the steady state is reached, the loop executes a multiple of U times. This can be guaranteed by inserting a preconditioning loop to execute the remaining iterations. Preconditioning is discussed in Step 9. If there is only one epilogue, then there is only one loop back branch and it is scheduled in the last stage of the kernel. Thus, the loop increment, which must be scheduled before the loop back branch, can be scheduled in any stage of the prologue or kernel. Note, that if it is scheduled in the kth stage, where $0 \leq k < S + U$, then the loop bound must be decremented by k in the preheader. Since the loop increment and loop back branch do not have to be scheduled in the same stage, the loop increment can be considered to be a self-recurrence and CII is equal to the latency of the loop increment operation.

A software pipelined loop with multiple epilogues has an epilogue for every stage of the prologue and kernel. Thus, there must be a loop back branch at every stage of the prologue and kernel. In this case, both the loop increment and loop back branch must be scheduled within II cycles. Thus, for software pipelined loops with multiple epilogues, the lower bound on II due to the recurrence circuit containing the loop increment is the sum of the latencies of the loop increment operation and the loop back branch operation.

After RII and CII are determined, the initial value of II, II_{init}, is set to the maximum of these two values.

4.5 Step 5. Apply If-Conversion

After the loop has been selected and Induction Variable Reversal is applied, the loop body is converted into a hyperblock by applying If-conversion. We use the modified RK If-conversion algorithm presented in Section 2.2. Figure 4.8(b) shows the loop hyperblock after If-conversion. Note that **p0** is the default predicate that is always defined. The conditional branch operation



(a)

```
L1: op12: <p0>  r1 <- r1 + 8
    op4:  <p0>  r4 <- r2 (r1)
    op5': <p0>  pd (r1==0){p1}{p2}
    op6:  <p1>  r7 <- r4 + c4
    op8:  <p2>  r5 <- r4 + c1
    op9:  <p2>  r6 <- r5 * c2
    op10: <p2>  r7 <- r6 + c3
    op7':   pm {p2}{p1}
    op11: <p0>  r2 (r1) <- r7
    op13: <p0>  bne r1, r3, L1
```

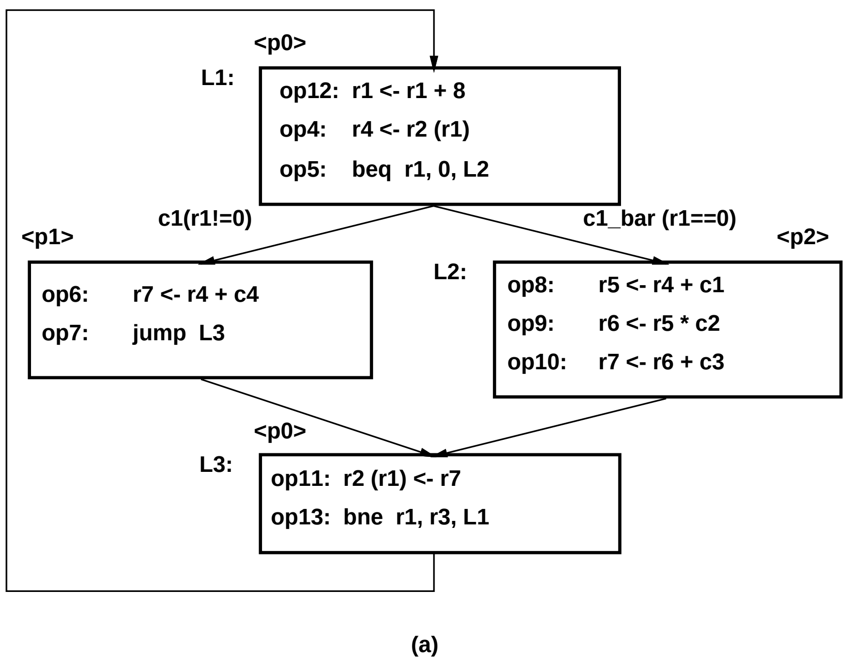
(b)

Figure 4.8 Example loop after If-conversion. (a) CFG with predicates assigned to basic blocks. (b) Loop hyperblock.

is converted into a predicate defining (pd) operation that defines predicate **p1** if the condition **(r1 == 0)** is false or defines predicate **p2** if the condition is true. The predicate merging (pm) operation merges predicates **p1** and **p2** where **p1** is placed in the jump set since the path corresponding to predicate **p1** originally had a jump operation.

4.6 Step 6. Rebuild Data Dependence Graph

Only operations which have a control path between them can be dependent on one another. In the original loop body, operations along different control paths are in different basic blocks with no path of control connecting them. After If-conversion, the loop body is reduced to one hyperblock. Thus, predicates need to be used to determine whether there is a control path between two operations in one iteration. There is always a control path between two operations from different iterations. The Predicate Hierarchy Graph (PHG) [23], [20] is used to determine



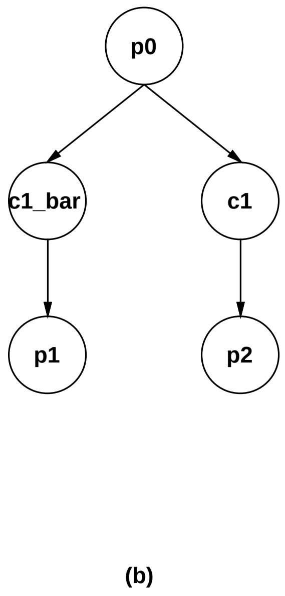


Figure 4.9 Predicate Hierarchy Graph of example loop. (a) Control flow graph of loop with predicates assigned to basic blocks and conditions assigned to arcs. (b) Predicate Hierarchy Graph.

whether there is a control flow path between predicates. The PHG of the example loop is shown in Figure 4.9(b). Note that there is no control flow path between predicates **p1** and **p2** since the Boolean expression $c1\_bar \cdot c1$ simplifies to zero.

After the PHG is generated, the dependence graph is built. To calculate the dependences, in addition to using the normal conditions for dependence, an additional condition is added to determine whether there is a control path between the operations. Furthermore, there are new operation types due to predication that must be considered when calculating dependences. There is a flow dependence between the pd operation that defines a predicate *pred* and operations that are predicated with *pred*. There is an anti dependence between the pm operation that merges a predicate *pred* and any operations predicated with *pred*.

Figure 4.10(b) shows the resultant dependence graph for our example. Note that **op6, op8, op9, and op10** are all flow dependent on the pd operation **op5′**. Likewise, the pm operation **op7′** is anti dependent on these operations. Notice that there is no output dependence between **op6** and **op10** since **p1** and **p2** are mutually exclusive and thus, there is no control flow path between these operations.

4.7 Step 7. Modulo Schedule Loop

Once II_{init} has been determined, the hyperblock formed, and the corresponding dependence graph built, the loop can be modulo scheduled. If a schedule cannot be found for a given II, then II is incremented and the scheduling process is repeated. This iterative scheduling process proceeds until II reaches a predetermined upper limit, at which time the loop is considered to be unfit for software pipelining.

The loop is "modulo" scheduled by applying a local scheduling algorithm to the Modulo Resource Table (MRT) [47]. The question is which local scheduling algorithm to use. The answer depends on the scheduling objective, and on the type of loop. For example, the objective can be to increase the throughput (e.g., minimize II), to decrease the pipeline latency, or both. Hsu provides a thorough analysis of algorithms designed for minimizing II and the pipeline latency. These are summarized below.

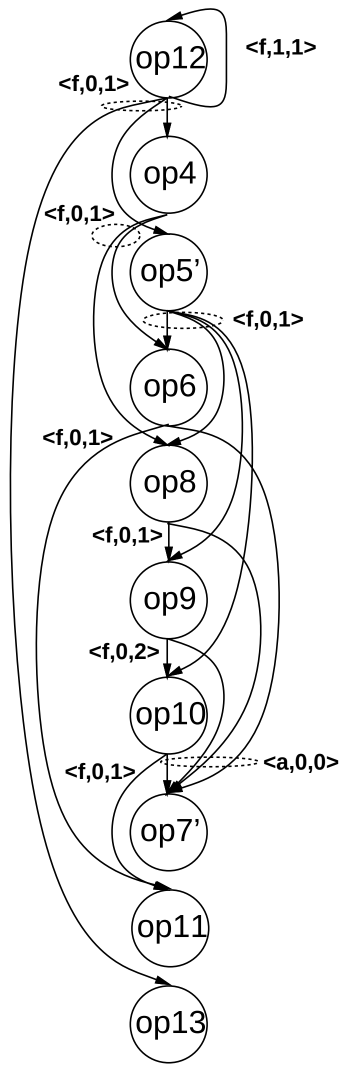
After the loop has been modulo scheduled, the pipeline latency can be calculated based on the latest issue time of an operation from the first iteration. The pipeline latency L is

$$L = \lceil \frac{latest\ issue\ time + 1}{II} \rceil,$$

assuming the schedule starts at cycle 0. Typically, a loop executes a large number of times and thus the suboptimal performance while filling the pipeline is amortized over the performance

```
L1: op12: <p0>  r1 <- r1 + 8
    op4:  <p0>  r4 <- r2 (r1)
    op5': <p0>  pd (r1==0){p1}{p2}
    op6:  <p1>  r7 <- r4 + c4
    op8:  <p2>  r5 <- r4 + c1
    op9:  <p2>  r6 <- r5 * c2
    op10: <p2>  r7 <- r6 + c3
    op7':   pm {p2}{p1}
    op11: <p0>  r2 (r1) <- r7
    op13: <p0>  bne r1, r3, L1
```

(a)



(b)

Figure 4.10 (a) Loop hyperblock. (b) Hyperblock dependence graph.

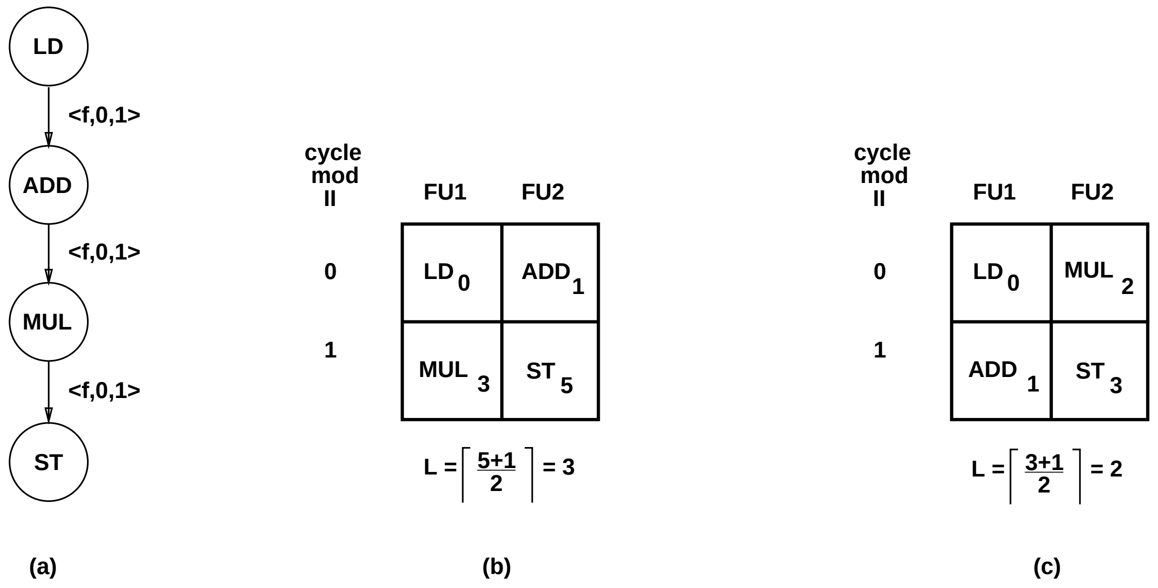


Figure 4.11 The effect of different Modulo Scheduling algorithms for scheduling the example loop. (a) Example data dependence graph. (b) Resultant MRT after applying technique that schedules first available resource with respect to top of MRT. (c) Resultant MRT after applying technique that schedules first available slot with respect to when an operation is ready.

of the entire loop execution. If minimizing L is critical, then Patel has presented an efficient branch-and-bound algorithm for finding an optimal schedule with minimum L [49].

Nevertheless, L should be ignored. Rau presents a minimum complexity (O(V), where V is the number of operations in the loop), optimal throughput scheduling algorithm [63]. This algorithm uses equations to fill the MRT from top to bottom. An alternative algorithm is to search the MRT for the first available slot after the cycle the operation is ready. Figure 4.11 illustrates the tradeoffs of the two approaches using the simple loop shown in Figure 4.11(a). Figure 4.11(b) shows the resultant example loop schedule when the operation is scheduled with respect to the top of the MRT. The ADD operation is ready in cycle 1, but is not scheduled until 2. Figure 4.11(c) shows that if the operations are scheduled in the first available slot with respect to the time it is ready, then a lower L is achieved (2 versus 3).

Algorithm List Schedule MRT: Given a hyperblock loop and *II*, schedule loop using an Modulo Resource Table (MRT) with *II* rows.

```
clear MRT
∀ op ∈ hyperblock loop {
    compute priority of op
    add op to set of unscheduled operations
}
    sort unscheduled set according to operation priority
issue_time = 0
while (unscheduled set of operations is not empty) {
    issue_time = issue_time + 1
    active_set = set of unscheduled operations that are ready
    ∀ op ∈ active_set {
        schedule op at earliest available resource
        if(op cannot be scheduled within II cycles)
            return schedule not found
        mark required resources of op busy in MRT
        delete op from set of unscheduled operations
        issue time of op = issue_time
}
```

Figure 4.12 Algorithm List Schedule MRT schedules loops without recurrences using Modulo Resource Table.

Figure 4.12 presents the O(V*II) list scheduling algorithm that is used in this dissertation, where V is the number of operations in the loop. The algorithm schedules operations in the first available slot after they are ready. Before any operations are scheduled, the loop-back branch is fixed in the last cycle of the schedule (assuming no branch delay slots). The list scheduling algorithm topologically sorts the operations according to their dependences and then schedules the ready operations whose dependences are resolved. The ready operations are sorted based on a latest issue time priority. Thus, operations with a higher predicted issue time are scheduled first. In this way, the schedule of an iteration is tighter which corresponds to a smaller pipeline latency.

There are three possible states for each slot in the MRT: empty, no-conflict, and full. A slot is empty if no operations have reserved that slot. A no-conflict slot occurs when there is no control path between the operation being scheduled and operations that have already reserved the slot. For instance, operations from one iteration that are from different paths of a conditional branch can be scheduled in the same slot. A slot is full with respect to the operation being scheduled, if there is a control path between the operation and the operations that have already reserved the slot.

Two operations do not have a control path between them if they are from the same iteration, and there is no control path between their predicates. With respect to one slot, two operations can be from the same iteration only if they reserve the resource at the same cycle (not modulo II). For fully pipelined functional units, an operation reserves a resource for one slot at its start time. To determine the no-conflict state, each slot in the MRT contains the predicates of the operations that use the resource and the start time.

To find the tightest schedule, with respect to throughput, first determine whether there are any no-conflict slots in the modulo resource reservation table for this operation. If there are, select the earliest available slot with respect to the earliest start time for that operation. Otherwise, schedule the operations in the earliest available empty slot.

Figure 4.13 shows the steps taken to modulo schedule the example loop of Figure 4.10. The target machine for this example is a VLIW with two function units. Function unit 1 (FU1) can execute any type of operation except a branch operation. Function unit 2 (FU2) can execute any type of operation including branch operations. Given these resource constraints, $RII = max(\lceil \frac{8}{2} \rceil, \lceil \frac{3}{1} \rceil) = 4$. This calculation determines which is more constrained, the general purpose function unit or the branch function unit. Assuming general purpose function units,

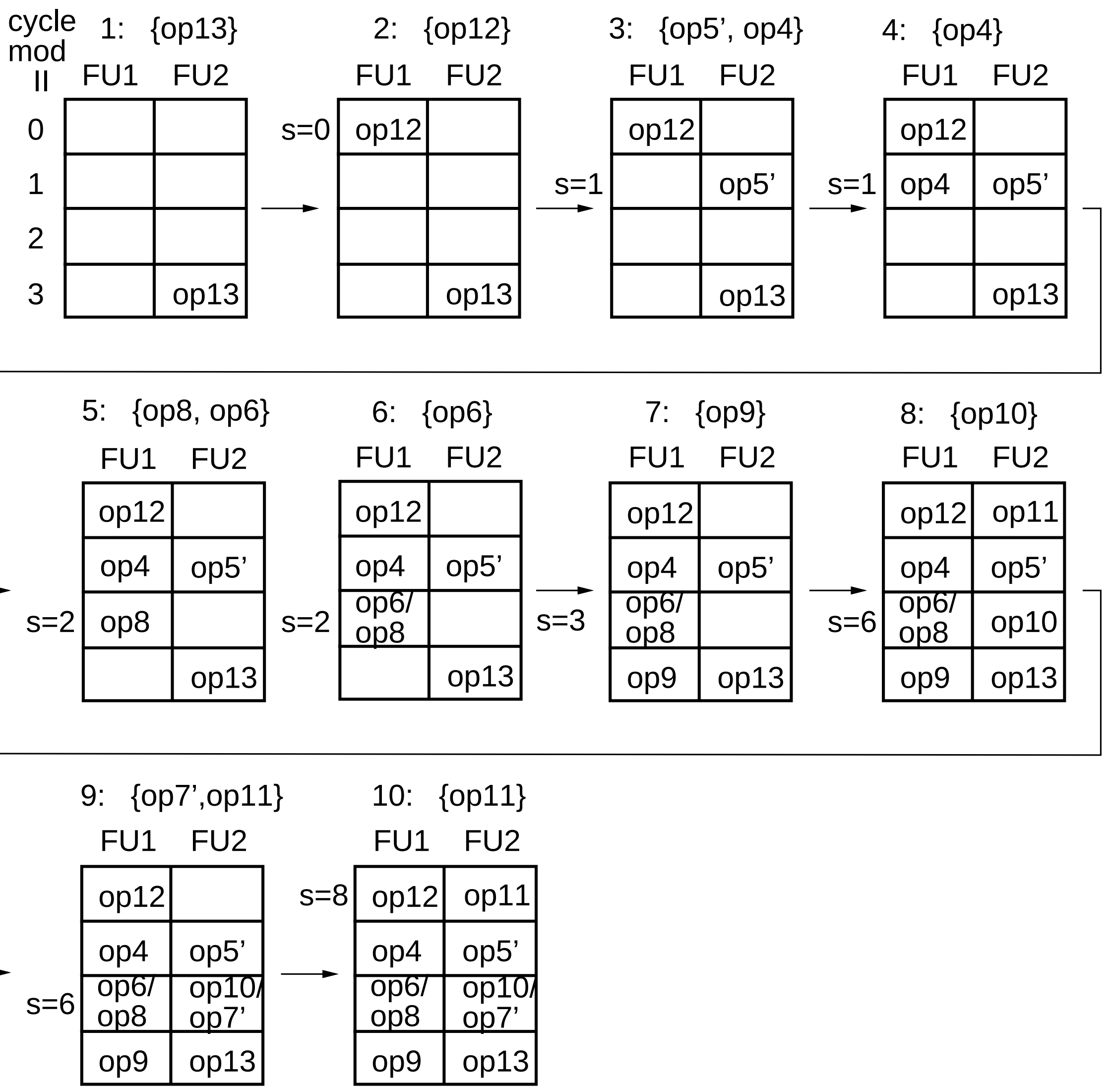


Figure 4.13 Modulo scheduling example loop with MRT.

the most constrained path has 8 operations. For the branch unit, the most constrained path has 3 branch operations. Note that pd operations are considered branch operations. The pm operations are considered branch operations for the predicates in the jump set.

Figure 4.13 shows the MRT at each step. The MRT has $II = 4$ rows and 2 columns. The active set at each step is shown above the MRT. The first step is to place the loop back branch **op13** in the last slot of the MRT (assuming a single cycle branch with no delay slot). The rest of the operations are then list scheduled using the last completion time estimate as the scheduling priority. The start time is marked next to the operation scheduled. Note that in step 6, **op8** is ready to schedule in cycle 2. Since the slot containing **op6** is a no-conflict slot (since **op6** is scheduled in cycle 2 and has a mutually exclusive predicate), **op8** is scheduled in the same slot as **op6**. Likewise, in step 9, **op7′** is scheduled in the same slot as **op10** since the predicate in the jump set of the pm operation, **p1**, is mutually exclusive with the predicate of **op10** and is issued at the same time as **op10**.

4.7.1 Scheduling loops with recurrences

Now lets consider loops with recurrences. There are two types of recurrences, self and multi-node. A self-recurrence contains only one operation. For example, loop induction variables are self-recurrences [27]. Since self-recurrences contain only one operation, the only constraint in scheduling them is that II is greater than the latency of the operation. Given II is large enough, the list scheduling algorithm used for loops without recurrences can be used for loops with only self-recurrences.

A straightforward list scheduling algorithm cannot be used to schedule loops with multi-node recurrences. Lam proposed a list scheduling method that first schedules all recurrence circuits and reduces them to pseudo operations [14], [15]. She proposes the following guidelines for scheduling recurrence circuits.

(1) Partial schedules constructed at each point in the scheduling process should not violate any of the dependence constraints.

(2) Heuristics must be sensitive to the value of II.

The first guideline is important since the scheduling process for a given II can be terminated as soon as it is determined that a schedule cannot be found. The second guideline is important for increases the likelihood of finding a schedule for a given II.

Another approach for scheduling loops with recurrences is to use a backtracking algorithm. The Cydrome compiler adopted such an approach [53]. It first attempts to schedule operations in recurrences and then backtracks if a schedule cannot be found. Thus, unlike the list scheduling approach, multiple attempts are made for each value of II. The Cydrome compiler uses static heuristics to schedule the recurrences. Huff proposes Slack Scheduling, a backtracking algorithm with dynamic heuristics [50]. The dynamic heuristics allow the scheduler to adjust priorities when operations are scheduled. Slack Scheduling also integrates recurrence constraints with critical path considerations for register allocation.

4.8 Step 8. Apply Modulo Variable Expansion

At this point, the steady state, or kernel, of the software pipeline has been scheduled. It consists of one II. Before generating the rest of the software pipeline, we have to determine whether the kernel needs to be unrolled to avoid overlapping register lifetimes. Since one loop iteration can span multiple stages, the lifetime of a register can overlap itself. To guarantee that a value in a register is not overwritten, the loop body must be unrolled enough times to satisfy the longest register lifetime and the overlapping register lifetimes are renamed. This optimization is called Modulo Variable Expansion [14], [15]. Register renaming can be done dynamically using hardware support such as the rotating register files in the Cydra 5. [11]. We assume no special hardware support. The lifetime of a predicate variable may also overlap itself.

Although these variables do not map to physical registers, they are also renamed to regenerate the code properly.

The number of times the loop must be unrolled is determined by the longest register lifetime. The lifetime l of a register is determined by the earliest time the value in that register is defined, and the latest time the value in that register is used before the value in that register is redefined. Given a variable v_i, $q_i = \left\lceil \frac{l_i}{II} \right\rceil$ is the minimum number of copies of v_i needed to avoid register conflicts. The minimum degree of unrolling, U, is $\max\limits_i q_i$. Once the degree of unrolling (i.e., the number of copies of the kernel) is determined, k_i is the number of copies of v_i that are needed, where k_i is the minimum integer, such that $k_i \geq q_i$ and $U \bmod k_i = 0$. Note that a variable can be either a predicate or a register.

After unrolling the kernel U times, register renaming can be performed for each stage s in the kernel, where a stage has II cycles and $0 \leq s < U$. For each variable in the original code v_i, reserve k_i variables with base offset b_i.[2] At stage s, a destination variable is renamed as variable $b_i + s \bmod k_i$. A variable can have multiple uses. Let v_{ij} be the j^{th} use of variable i. Source variable v_{ij} is renamed depending on its individual lifetime relative to II, $\lambda_{ij} = \left\lceil \frac{issue\_time(use(v_{ij})) - issue\_time(define(v_i)) + 1}{II} \right\rceil$. A source variable is renamed differently depending on whether it is used before defined with respect to the beginning of the stage s. If it is used before defined, then a source is renamed as variable $b_i + (s + k_i - \lambda_{ij}) \bmod k_i$. Otherwise, it is renamed as variable $b_i + (s + k_i - \lambda_{ij} + 1) \bmod k_i$.

[2]Note that at this point, register allocation has not been performed. Therefore, renaming is performed on virtual registers or predicates.

4.9 Step 9. Generate Software Pipeline

After renaming has been performed, the kernel has U stages. Next, the stages of the prologue and epilogue are generated. The number of stages in the prologue (and epilogue), S, is $\left\lceil \frac{latest\ issue\ time}{II} \right\rceil - 1$, where the latest issue time is defined by the modulo schedule over all operations in the loop.[3]

The prologue with S stages is created from S_u copies of the kernel appended to copies of the last S_r stages of the kernel, where $S_u = \left\lfloor \frac{S}{U} \right\rfloor$ and $S_r = S \bmod U$. This ordering ensures that the registers are aligned properly. Likewise, the epilogue is created from a copy of the first S_r stages of the kernel appended to S_u copies of the kernel. For each instruction i in the prologue, $0 \leq i < S * II$, an operation is nullified if its start time is greater than i. Conversely, for instruction i in the epilogue, where $0 \leq i < S * II$, an operation is nullified if its start time is less than or equal to i. Lastly, the loop-back branch is nullified in all but the last stage of the kernel.

Once the prologue has been formed, if any of the variables that are live at the beginning of the loop have been renamed, then copies must be inserted in the loop preheader. Likewise, any renamed variables that are live at the end of the loop must restore the initial value by inserting the appropriate copy operations at the end of the epilogue.

The above assumes that there is only one exit from the loop, which is at the end of the kernel. Allowing early exits from the loop requires special epilogues for each stage in the prologue and kernel, which increases the code generation complexity and code expansion considerably. With only one exit from the loop, the software pipelined loop must execute $S + k * U$ times, where k is an integer greater than or equal to one. A non software pipelined version of the loop

[3]The number of stages in the prologue and epilogue may depend on the register allocation scheme [36].

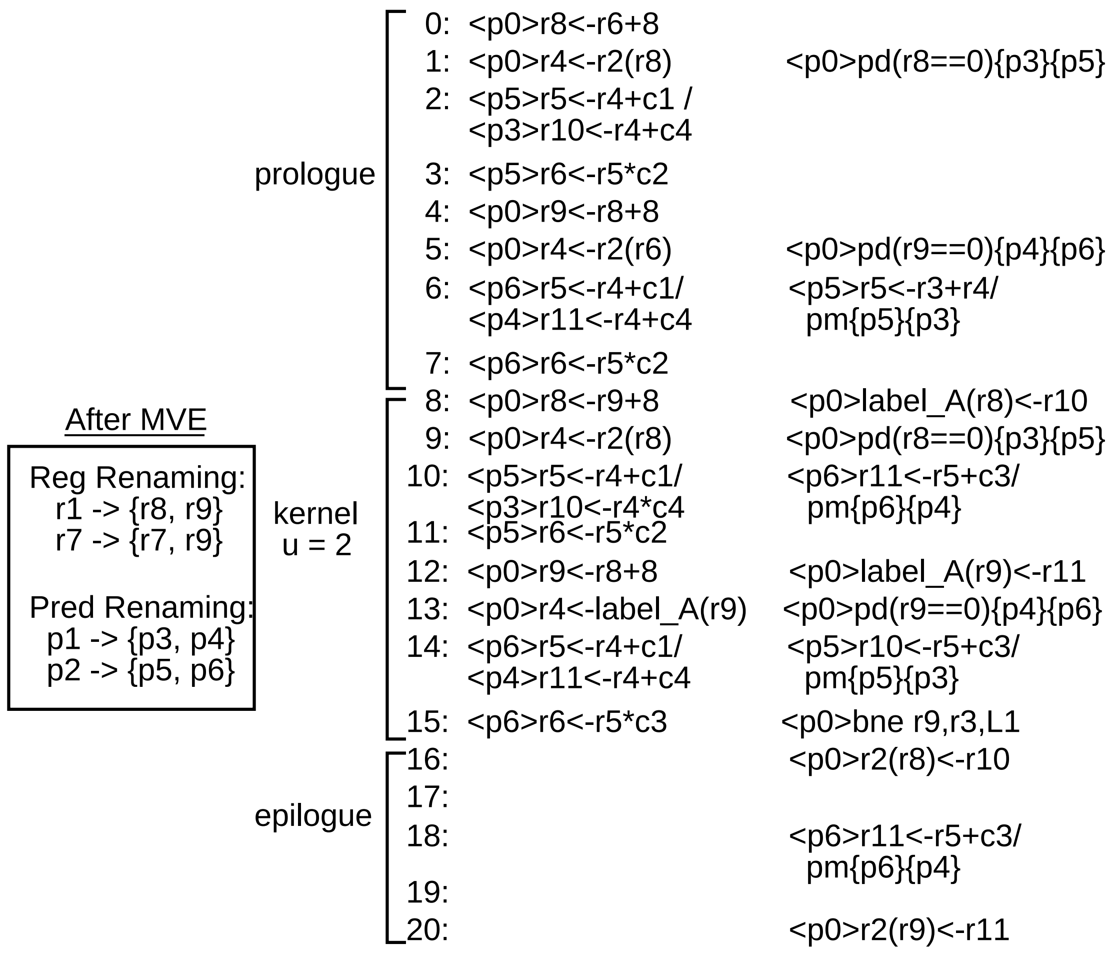
is required to execute the remaining number of iterations. This loop is referred to as the preconditioning loop. If the loop trip count is greater than $S + U$, the remaining number of iterations is (trip count $- S) \bmod U$. If the trip count is less than $S + U$, then only the non software pipelined loop is executed. If the trip count is known to be less than $S+U$ at compile time, then the software pipeline is not generated.

The disadvantage of using a preconditioning loop is that it executes relatively slow compared to the pipelined code. Thus, the performance of the loop does not monotonically increase as the loop bound increases. On the other hand, with multiple epilogues, the larger the value of the loop bound, the more amortized the prologue code becomes. With preconditioning code, the speedup is saw-toothed as the loop bound is increased. Rau, Schlansker, and Tirumalai provide a thorough overview of the code generation schema for modulo scheduling [37].

Figure 4.14 shows the resulting software pipeline after unrolling the kernel, register renaming, and generating the prologue and epilogue. Note that two register variables, **r1** and **r7**, and two predicate variables, **p1** and **p2**, are renamed. Thus, the kernel is unrolled twice (i.e., there are two stages in the kernel). Notice that **r1** was originally live at the beginning of the loop body. Thus, the value of **r1** must be renamed to **r8** in the preheader. Note that if **r1** is live at the beginning of the preheader, then a copy must be inserted instead.

4.10 Step 10. Apply Reverse If-Conversion

After the software pipeline is generated, RIC can be applied to generate the CFG. If necessary, it is also applied to regenerate the CFG of the preconditioning loop. Since the predicates have been renamed using MVE, the RIC algorithm presented in Chapter 2 can be



```
 0:  <p0>r8<-r6+8
 1:  <p0>r4<-r2(r8)            <p0>pd(r8==0){p3}{p5}
 2:  <p5>r5<-r4+c1 /
     <p3>r10<-r4+c4
 3:  <p5>r6<-r5*c2
 4:  <p0>r9<-r8+8
 5:  <p0>r4<-r2(r6)            <p0>pd(r9==0){p4}{p6}
 6:  <p6>r5<-r4+c1/            <p5>r5<-r3+r4/
     <p4>r11<-r4+c4              pm{p5}{p3}
 7:  <p6>r6<-r5*c2
 8:  <p0>r8<-r9+8              <p0>label_A(r8)<-r10
 9:  <p0>r4<-r2(r8)            <p0>pd(r8==0){p3}{p5}
10:  <p5>r5<-r4+c1/            <p6>r11<-r5+c3/
     <p3>r10<-r4*c4              pm{p6}{p4}
11:  <p5>r6<-r5*c2
12:  <p0>r9<-r8+8              <p0>label_A(r9)<-r11
13:  <p0>r4<-label_A(r9)      <p0>pd(r9==0){p4}{p6}
14:  <p6>r5<-r4+c1/            <p5>r10<-r5+c3/
     <p4>r11<-r4+c4              pm{p5}{p3}
15:  <p6>r6<-r5*c3            <p0>bne r9,r3,L1
16:                            <p0>r2(r8)<-r10
17:
18:                            <p6>r11<-r5+c3/
                                 pm{p6}{p4}
19:
20:                            <p0>r2(r9)<-r11
```

Figure 4.14 Software pipeline of example loop before RIC.

applied with only minor modifications to ensure that the prologue, kernel, and epilogue are properly connected.

Figure 4.15 shows the regenerated CFG of the software pipeline shown in Figure 4.14. The allowable predicate sets are shown for each basic block. The control flow graph is regenerated starting with the prologue. When the kernel is reached, the appropriate control flow is added between the prologue and the kernel nodes in the leaf node set. Note that it is possible to determine the kernel boundary based on the instruction number. Since there are 2 stages in the prologue and each stage has 4 instructions, instruction 8 is the first instruction of the kernel. At the kernel boundary, a new node is generated for every node in the leaf node set. This is required since the loop back branch must jump back to the beginning of the kernel. Note that these nodes will have the same allowable predicate set as their predecessors.

The same action is taken when the epilogue boundary is encountered. Since there are 2 stages in the kernel and 4 instructions per stage, the epilogue starts in instruction 16. A new set of nodes is generated for every node in the leaf node set. Again, the nodes have the same allowable predicate set as their predecessor. The epilogue boundary also corresponds to placing the loop back branch. Thus, in addition to the epilogue, control flow is inserted to jump back to the beginning of the kernel to the node with the same allowable predicate set.

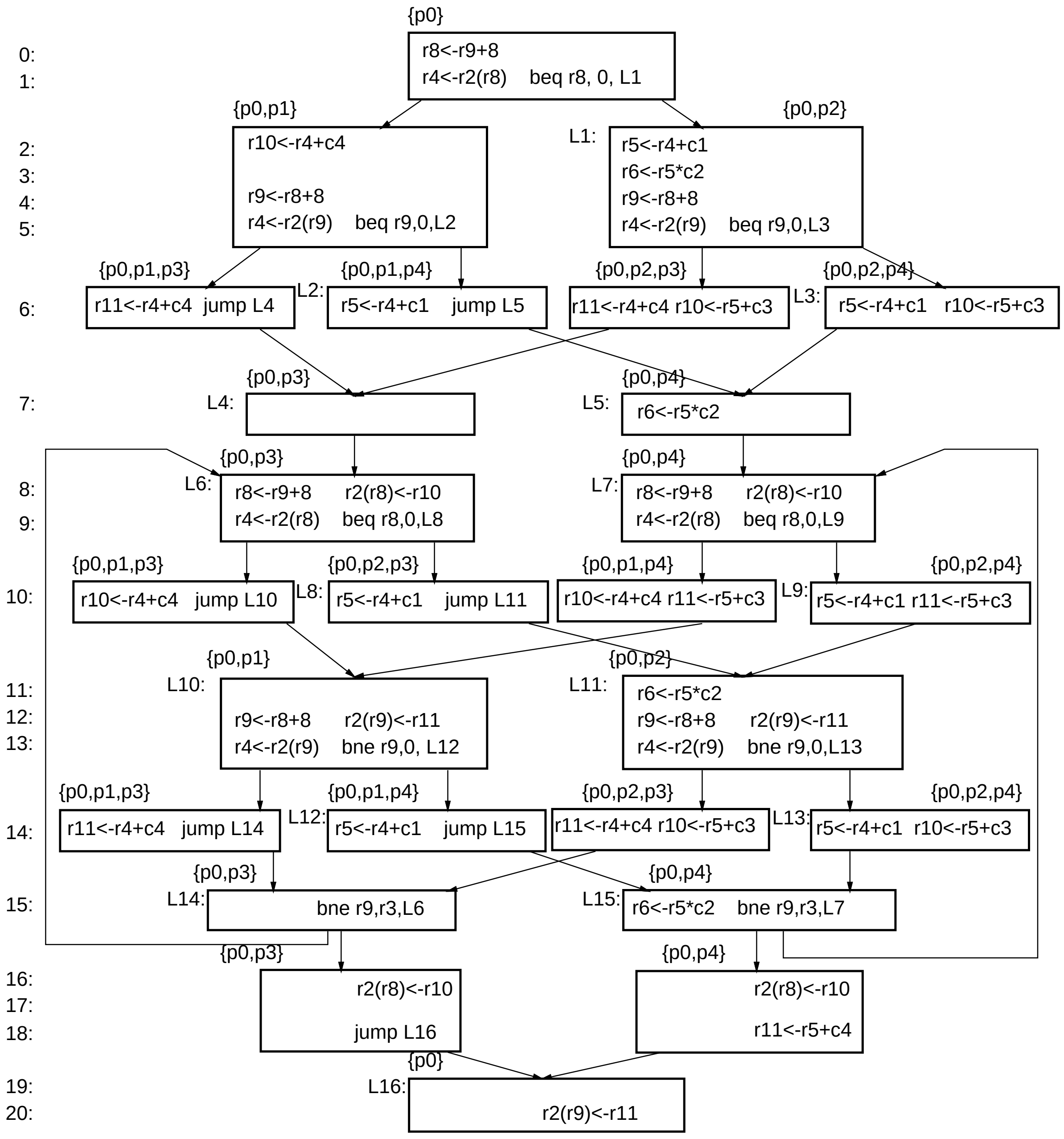


Figure 4.15 Software pipeline of example loop after RIC.

CHAPTER 5

EXPERIMENTAL RESULTS

In this chapter we chronicle the development of software pipelining in the IMPACT C compiler. This chapter focuses on techniques that do not require any special hardware support. Chapter 6 presents the benefits of Predicated Execution hardware support for Modulo Scheduling [11].

We first compare an existing Modulo Scheduling technique using Hierarchical Reduction [16], [14], [15] with a global code compaction based technique, GURPR* [18]. After illustrating the benefits of Modulo Scheduling, we compare the performance of Modulo Scheduling using Hierarchical Reduction against Modulo Scheduling with Isomorphic Control Transformations. The implementation of the ICTs was motivated by a comparison of Modulo Scheduling with Hierarchical Reduction and Modulo Scheduling with If-conversion assuming hardware support for Predicated Execution [64]. From this comparison we decided to investigate whether it would be possible to remove the restrictions that Hierarchical Reduction places on scheduling without requiring special hardware support. Based on our results we have chosen Modulo Scheduling with Isomorphic Control Transformations as the software pipelining method in IMPACT for processors without hardware support. As mentioned earlier, this technique is referred to as Enhanced Modulo Scheduling (EMS). The second part of this chapter presents a deeper analysis of EMS.

5.1 Compiler Support

The software pipelining techniques analyzed in this chapter were implemented in the IMPACT C compiler. A flow diagram of the IMPACT compiler is shown in Figure 5.1. The IMPACT compiler has four levels of intermediate representation (IR). The **PCODE** IR is a parallel C code representation with loop constructs intact [65]. Thus, memory dependence analysis and loop level transformations can be effectively applied at this level [66]. The next level is the **HCODE** IR. **HCODE** is a flattened C representation with simple if-then-else and goto control flow constructs. The Hcode environment has support for statement level profiling, which can be used to guide such optimizations as function in-line expansion [32]. The **LCODE** IR is a machine independent register transfer representation. At the Lcode level both classical and superscalar optimizations are performed [27], [32], [62]. Also, scope enlarging transformations such as superblock and hyperblock formation are performed at this level [32], [62], [20], [23], [22]. Currently, processor simulation is performed at this level. Finally, **MCODE** is a machine dependent register transfer representation. The Mcode module provides a standard interface for code generators [67]. Note that **MCODE** is structurally identical to **LCODE** and thus techniques developed in the Lcode framework can be utilized in the Mcode framework with possible machine specific extensions. Thus, at the Mcode level, machine specific optimizations, scheduling, and register allocation are performed [68]. Currently, the IMPACT compiler can generate code for the following processors: MIPS R2000, SPARC, HP-PA RISC, Intel i860, AMD 29K, and Intel X86.

Originally, the software pipeline scheduler was implemented in the Intel i860 code generator,[1] for two reasons. First, we can apply the software pipelining techniques to a more realistic

[1]The Intel i860 code generator and the MIPS R2000 code generator predate the Mcode implementation.

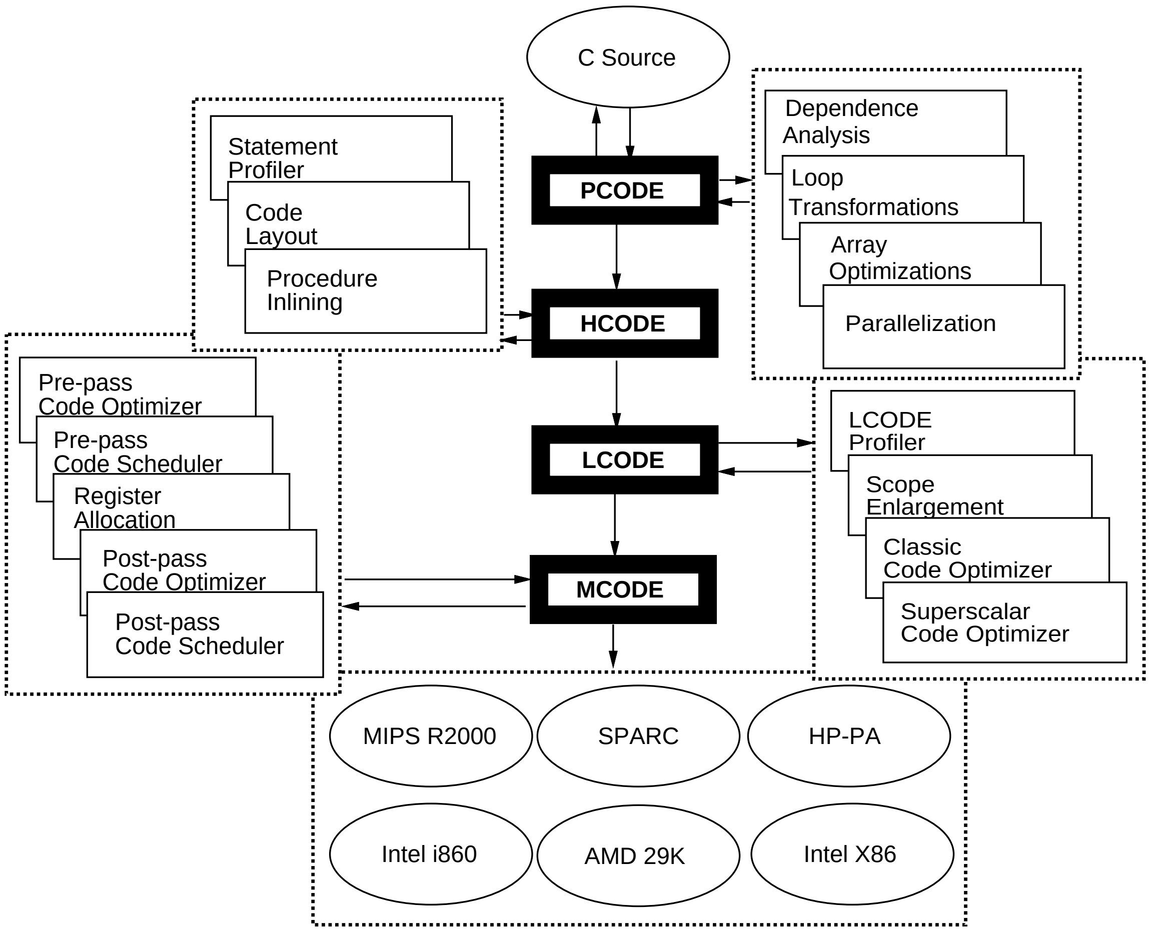


Figure 5.1 The IMPACT C Compiler.

instruction set architecture than the general Lcode architecture. **LCODE** is general not to place unnecessary constraints during optimization. This powerful instruction set unrealistically reduces the number of operations to schedule. The second motivation was the fact that I had implemented the Intel i860 code generator for the IMPACT compiler.[2] Thus, I was familiar with the environment and intrigued by the possibilities of scheduling such as complex architecture as discussed in Appendix A. Note, that while the Intel i860 instruction set and latencies were used for this implementation, to keep the results general, the underlying architecture was not. The machine model used is described in the next section.

In the second phase of the software pipeline scheduler development, the Modulo Scheduling with ICTs implementation was migrated up to the Lcode level. By this point, the Mcode level was developed. Since techniques developed for Lcode can be used in Mcode, the first motivation listed above was no longer an issue. Furthermore, the software pipeline scheduler can now be utilized to schedule a broader range of architectures.

Note that scheduling is performed after classical code optimizations are applied. For this dissertation, the scope enlargement techniques are considered part of the scheduling algorithm. Thus, the different transformations are applied depending on the software pipeline technique being analyzed. Software pipelining is performed before register allocation. There is ongoing research exploring the possibility of merging software pipelining with register allocation.

In the experiments, the base scheduler is a basic block scheduler.

[2]The Intel i860 code generator was based upon the MIPS R2000 code generator for IMPACT [69].

5.2 Machine Model

The basic processor used in this study is a RISC processor with in-order execution with register interlocking and deterministic operation latencies. In the first phase of experiments, the processor has an instruction set similar to the Intel i860 [21]. The only difference is that we allow more types of branch operations and assume automatically advanced load and floating-point pipelines. Table 5.1 shows the operation latencies for the i860. Note that the integer multiply and divide and the floating-point conversion and divide are implemented using approximation algorithms [21]. In the second phase of the experiments, the IMPACT processor has an instruction set similar to the MIPS R2000 [70]. Table 5.2 shows the operation latencies.

For the branch operation we assume that the compare and branch are performed in 1 cycle. Thus, there are no branch delay slots. There are four basic kinds of compare and branch operations: equal, not equal, greater than, and greater than or equal. For each kind there are three types: integer, single-precision floating-point, and double-precision floating-point. There are also signed and unsigned versions of the integer greater than and greater than or equal operations. In total, there are 14 types of branch and compare operations.

The base processor for the both phases of the experiment has an infinite register file. This allows us to measure the register requirements. We also assume an ideal cache.

The multiple instruction issue machine model for these experiments is a VLIW processor with no interlocking. There are uniform resource constraints, with the exception that only one branch can be issued per cycle. For Phase I, the experiments were performed using machines with instruction widths or issue rates of 2, 4, and 8. For Phase II, issue rate 1 was also included to show the benefit of software pipelining for single issue processors.

Table 5.1 Intel i860 operation latencies (* indicates that the operation is implemented using an approximation algorithm).

Instruction Class	Latency
integer ALU	1
integer multiply	*
integer divide	*
branch	1
memory load	2
FP load	3
memory store	1
FP/Double ALU	3
FP multiply	3
Double multiply	4
FP conversion	*
FP/Double divide	*

5.3 Benchmarks

The focus of this study is to analyze the relative performance of conditional branch handling techniques. To run our experiments, we collected a set of 26 loops with conditional branches from the Perfect benchmarks [71]. Only DOALL loops (loops without cross-iteration memory dependencies) were included in the test suite. Table 5.3 provides the loop characteristics. The FORTRAN line numbers of the loops are provided so that the results can be independently verified. The column labeled BBs gives the number of basic blocks in the loop. Note that all conditional constructs within the loops are structured and non nested. The column labeled OPs gives the number of machine operations assuming the Impact machine model.[3] The column labeled Branches gives the number of branch operations in the loop, including the loop back branch.

[3]Note that the number of operations will differ for the Phase I experiments which use the Intel i860 machine model.

Table 5.2 Impact operation latencies.

Instruction Class	Latency
integer ALU	1
integer multiply	3
integer divide	10
branch	2
memory load	2
memory store	1
FP/Double ALU	3
FP conversion	3
FP multiply	3
FP divide	10

5.4 Phase I: Preliminary Studies

This section presents the experimental results that corroborate our design decisions for EMS. For the results presented in this chapter, the speedup is the harmonic mean of the speedups for the 26 loops. We assume that each loop executes an infinite number of times. Thus, the effect of the prologue and epilogue are not accounted for in the speedup results.

The code expansion is determined by the number of instructions in the software pipeline divided by the number of instructions in the basic block schedule. Based on this calculation, the code expansion is reported as a multiple of the number of instructions in the basic block schedule. The code expansion is averaged across the 26 loops using the arithmetic mean. The code expansion reflects the prologue and epilogue code, where every technique supports one epilogue. The code expansion due to the preconditioning loop (defined in Section 4.9) is not reported. This overhead is the same for all techniques.

One final note, the code expansion reported is an upper bound on the expected code expansion for two reasons. First, we assume that the size of the instruction for an issue n machine is

Table 5.3 Loop characteristics.

Loop	Perfect Benchmark	FORTRAN Line No.	BBs	OPs	Branches
1	ADM	2191	4	38	3
2	ADM	2217	4	47	3
3	ADM	1508	4	61	3
4	ADM	3981	4	10	3
5	ADM	2253	4	79	3
6	ADM	2280	4	142	3
7	ADM	3080	7	26	5
8	ADM	2631	7	43	5
9	ADM	2662	7	48	5
10	ADM	2541	4	48	3
11	ADM	2515	4	37	3
12	ADM	1689	4	15	3
13	ARC2D	2630	4	32	3
14	ARC2D	2655	4	44	3
15	ARC2D	2558	4	31	3
16	ARC2D	2493	4	37	3
17	ARC2D	2516	10	39	7
18	ARC2D	2578	7	25	5
19	FLO52Q	1134	4	21	3
20	FLO52Q	1144	10	22	7
21	FLO52Q	1095	4	24	3
22	FLO52Q	1089	10	22	7
23	SPEC77	259	4	12	3
24	SPEC77	316	7	19	5
25	SPEC77	1201	4	14	3
26	SPEC77	1292	4	18	3

n times greater than the size of an instruction for an issue 1 machine. Second, we assume that there is no hardware support for interlocking.

5.4.1 Limited Hierarchical Reduction versus Hierarchical Reduction

The Hierarchical Reduction technique described in Section 2.7.1 was implemented in the IMPACT compiler. We assume that Hierarchical Reduction is applied once to transform the loop into straight-line code. Thus, no II sensitive heuristics are applied when forming the reduct_ops. We implemented two versions of Modulo Scheduling with Hierarchical Reduction. The limited version does not allow a reduct_op to overlap itself. The normal version does not place any such limitation.

5.4.2 Induction Variable Reversal

We have applied Induction Variable Reversal to both the Limited and regular Hierarchical Reduction Modulo Scheduling techniques. Figure 5.2 shows the performance of Modulo Scheduling with Hierarchical Reduction before and after Induction Variable Reversal (IVR). We see that the optimization improves the performance of Hierarchical Reduction by as much as 260% for an issue-8 machine. It also shows the difference between Limited and regular Hierarchical Reduction. Before IVR, there is very little difference in the performance of Limited and regular Hierarchical Reduction. The reason is that the multi-node recurrence formed by the induction variable limits the achievable II and thus there is usually no opportunity for a reduct_op to overlap itself during scheduling. After IVR, the multi-node recurrence is converted into a self-recurrence which no longer bounds II. Thus, preventing reduct_op's from overlapping themselves becomes the limiting factor as the issue rate increases. For an issue 8 machine, regular Hierarchical Reduction performs on average 145% faster.

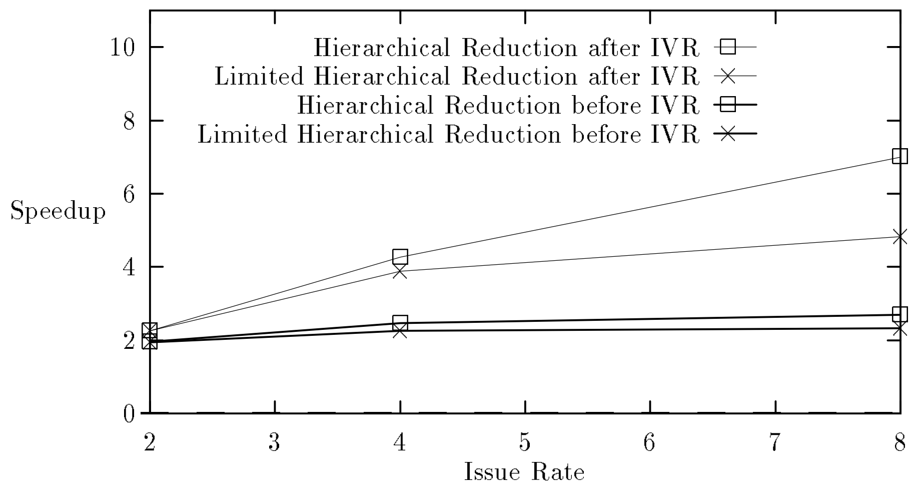


Figure 5.2 Speedup before and after Induction Variable Reversal (IVR).

The benefit of preventing a reduct_op from overlapping itself is that there is potentially lower code expansion. As Figure 5.3 shows, for an 8 issue machine, the code expansion is approximately 37% smaller for Limited Hierarchical Reduction.

5.4.3 Modulo Scheduling with Hierarchical Reduction versus GURPR*

To review, in the GURPR* algorithm, the loop body is compacted and pipelined assuming a lower bound on *II* determined by the inter-body dependence distance. From this intermediate pipeline representation, *II* is determined as the shortest interval that contains all operations in the loop. Once this interval is determined, it may contain multiple copies of an operation. Any redundant operations are deleted so that exactly one iteration is completed within one *II*. In our implementation of GURPR* no global code compaction is performed. We found that techniques such as trace scheduling and code percolation tend to increase both the longest path

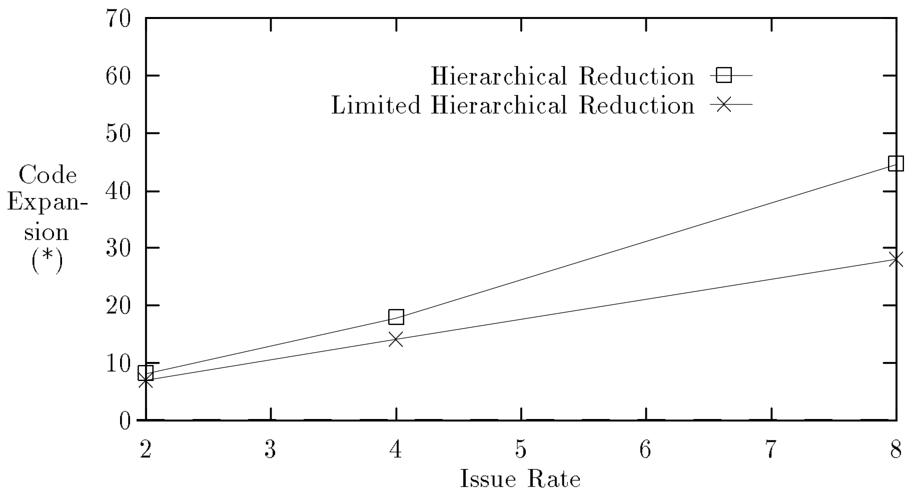


Figure 5.3 Code expansion after Induction Variable Reversal.

and the resource conflicts, thereby increasing *II*. It is possible that some heuristics could be applied to these code compaction techniques to improve the performance of GURPR* [58].

Figure 5.4 shows the performance of GURPR* versus Modulo Scheduling with Hierarchical Reduction. Modulo Scheduling performs from 1.38 to 1.64 times better than GURPR*. The relative performance of GURPR* degrades as the issue rate increases. The reason that GURPR* does not perform as well as Modulo Scheduling is that it under schedules the resources since it may schedule multiple copies of an operation into the final *II*. All but one copy will be deleted.

Figure 5.5 shows the code expansion of GURPR* and Modulo Scheduling with Hierarchical Reduction. The code expansion of GURPR* is 107%, 73%, and 130% larger than for Modulo Scheduling with Hierarchical Reduction. Even though GURPR* does not perform as well as Modulo Scheduling, it has a larger code expansion. This is because the original *II* is equal to one for loops with only self-recurrences. Thus, many iterations are overlapped during the pipelining

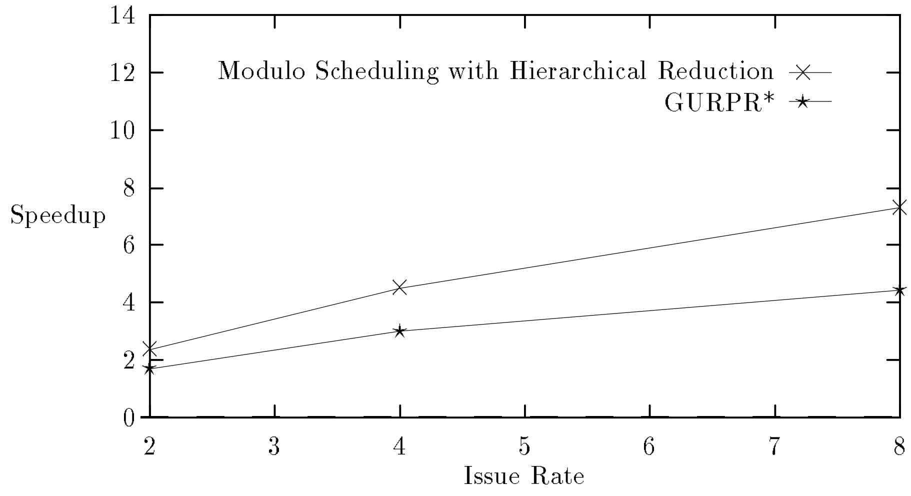


Figure 5.4 Speedup of Modulo Scheduling with Hierarchical Reduction versus GURPR*.

phase before the final *II* is determined [58]. This increases the overlap of the conditional constructs resulting in larger code expansion.

5.4.4 EMS versus Modulo Scheduling with Hierarchical Reduction

Given equivalent resource constraints, EMS should perform better than Hierarchical Reduction since Hierarchical Reduction has pseudo-operations with complicated resource usage patterns. Figure 5.6 shows the speedup of EMS and Modulo Scheduling with Hierarchical Reduction. The EMS technique performs 18%, 17%, and 19% better than Modulo Scheduling with Hierarchical Reduction for issue rates 2, 4, and 8, respectively.

The disadvantage of these techniques that require explicit conditional branches is that there may be multiple copies of an operation, each on a different control path. Furthermore, after software pipelining, a conditional construct can overlap itself. If it overlaps itself n times, the

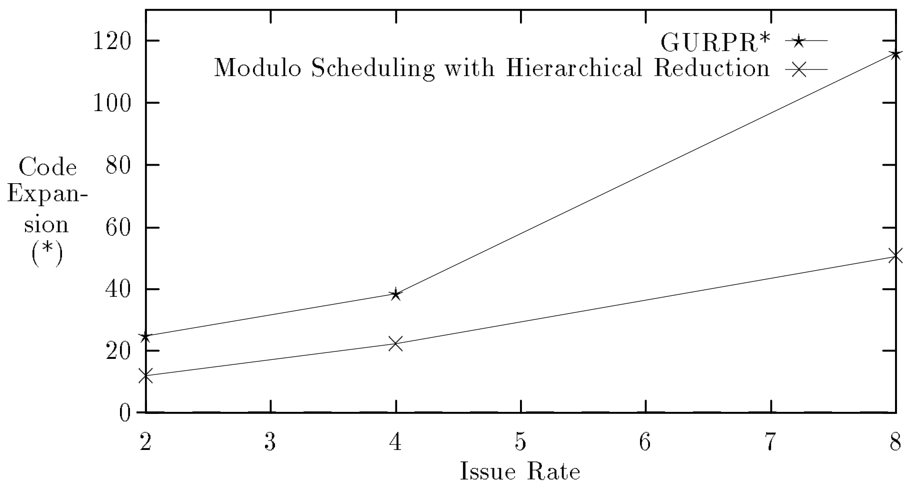


Figure 5.5 Code Expansion of Modulo Scheduling with Hierarchical Reduction versus GURPR*.

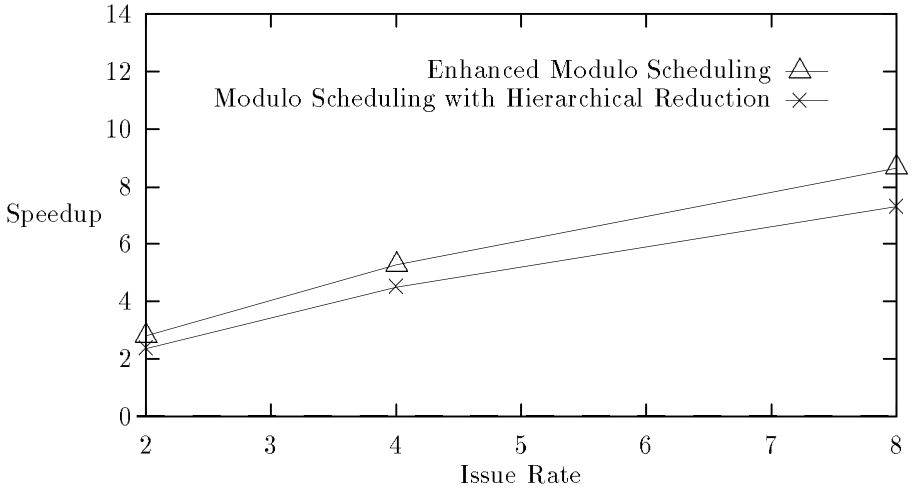


Figure 5.6 Speedup of Enhanced Modulo Scheduling and Modulo Scheduling with Hierarchical Reduction.

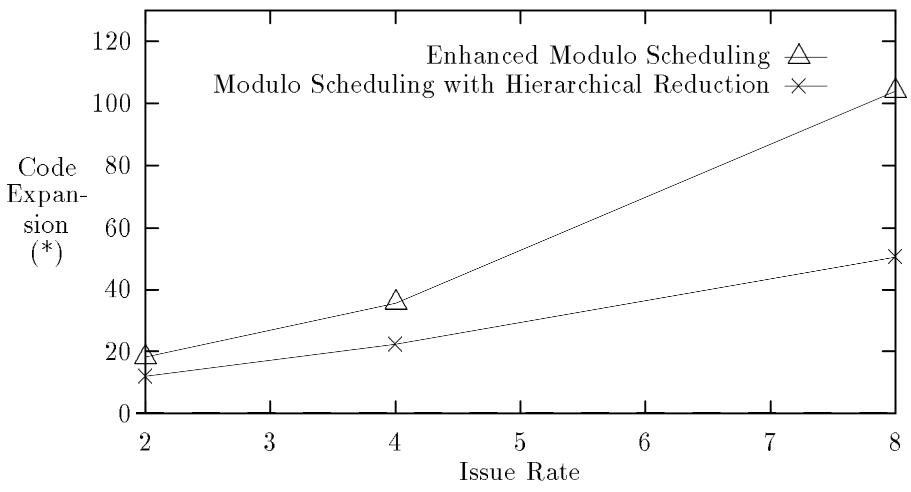


Figure 5.7 Code Expansion of Enhanced Modulo Scheduling and Modulo Scheduling with Hierarchical Reduction.

code expansion can be roughly a factor of 2^n. Figure 5.7 shows the arithmetic mean of the code expansion of the two techniques.

As expected, the code expansion for EMS is larger than for Modulo Scheduling with Hierarchical Reduction since a tighter schedule is more likely to have a larger number of overlapping register lifetimes and conditional constructs. The code expansion for EMS is 52%, 60%, and 105% larger than the code expansion for Modulo Scheduling with Hierarchical Reduction for issue 2, 4, and 8, respectively. If the underlying architecture supported multiple branches per cycle, then the code expansion for EMS would be even larger.

5.4.5 Final comparison

Figure 5.8 shows the arithmetic mean of *RII* and the achieved *II* for EMS, Modulo Scheduling with Hierarchical Reduction, and GURPR*. Since loops without cross-iteration memory

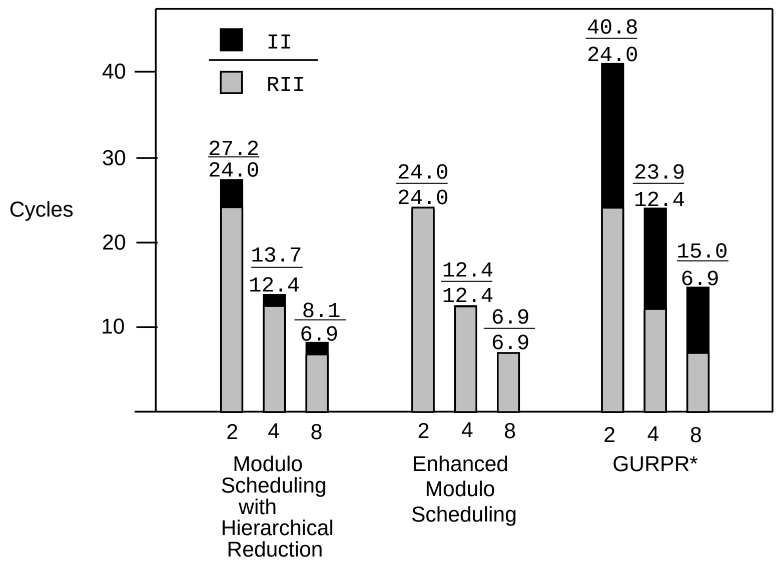


Figure 5.8 Average lower bound on *II* (*RII*) versus achieved *II* for EMS, Modulo Scheduling with Hierarchical Reduction, and GURPR*.

dependences were used, the lower bound on *II* is always *RII*. Furthermore, each technique assumes the same hardware support, and thus *RII* is the same for each approach. The graph shows the number of cycles per *II*, and thus, the size of the bar is inversely proportional to the performance of the technique. Figure 5.8 illustrates why EMS performs better than both Hierarchical Reduction and GURPR*.

For EMS, *RII* is determined by the most heavily utilized resource along any execution path. For our simple machine model, *RII* can always be achieved for loops without conditional constructs. For loops with conditional constructs, however, *RII* may not be achieved. The *RII* calculation assume that two operations from different control paths can be scheduled to the same resource. This is only possible when the two operations can be scheduled in the same cycle. Since dependences may force an operation to be scheduled later, it may not be possible to achieve *RII* for loops with conditional constructs. In the worst case, the scheduled *II* will be the sum of the resource usages along both paths. Our results indicate that EMS almost always achieves its lower bound on II. Although the precision in Figure 5.8 does not show any difference between the average RII and average II, there are some cases where II is larger than RII. On average II is 0.1% and 0.3% percent larger than the the average RII for issue rates 2 and 4, respectively.

Hierarchical Reduction often cannot achieve RII due to the complex resource usage patterns of the pseudo-operations. There are several optimizations that could be applied to Modulo Scheduling with Hierarchical Reduction that may improve its performance. For instance, if a reduct_op spans more than one II, then it may conflict with itself and thus no schedule for that II can be found. It is possible to schedule the reduct_ops with a knowledge of II. This requires that Hierarchical Reduction be applied for every value of II attempted. Also, a backtracking algorithm such as those proposed for scheduling recurrences could be employed. One of the benefits of the ICT approach is that it has a greater degree of scheduling freedom since operations are scheduled only once. Thus, for more realistic machine configurations, we would expect EMS to perform even better than Modulo Scheduling with Hierarchical Reduction.

While GURPR* does not use RII to determine the lower bound on II during the pipelining stage, it does represent a lower bound for the achieved II. Due to the insertion of cycles when a resource conflict arises, GURPR* rarely achieves this lower bound on II.

One final note on the comparison of the different techniques. To fully understand the benefits of EMS over other methods, a more detailed simulation is required that accounts for more constrained functional units, specific register file configurations, and specific cache configurations.

5.5 Phase II: Analysis of Enhanced Modulo Scheduling

In Phase I we discovered that EMS is an effective technique for software pipelining. By using the ICTs, the modulo scheduler has more freedom to schedule operations of a conditional construct with operations outside the construct. Thus, it is possible to achieve a tighter schedule. Since EMS has more scheduling freedom, it is interesting to analyze the effects of increasing *II* versus 1) performance, 2) pipeline latency, 3) register pressure, and 4) code expansion. In addition, we would like to analyze the experimental complexity of RIC for both the original loop and the modulo scheduled loop.

Before getting into the detailed analysis, it is useful to understand the basic interactions between the machine model, *II*, speedup, code expansion, and register pressure. As the issue rate increases, we assume that the number of function units increases correspondingly. Thus, as the issue rate increases, *II* will decrease, which results in fewer cycles in the steady state and thus higher performance. Intuitively, if we have a fixed schedule for a loop (which is not the case with Modulo Scheduling, but it illustrates the basic principle) and decrease *II*, then the loop will span more stages and the pipeline latency increases. This has several ramifications. First, it will take more iterations to fill the pipeline. Second, the likelihood of a register overlapping itself increases and thus, the register pressure increases. Third, if the loop has conditional constructs, then the likelihood that they overlap increases and thus, the code expansion increases.

5.5.1 Software pipeline characteristics for EMS

Tables 5.4 through 5.7 report the software pipeline characteristics for each loop for issue rates 1 through 8, respectively. The LP column provides the loop number. *II* and *RII* are shown for each loop. Note that *II* is not always equal to *RII*. *II* is greater than *RII* when we predict that we can schedule operations with mutually exclusive predicates to the same resource but are not able to because the operations are scheduled more than one *II* apart. This will occur more often in loops with 1) fewer operations and hence a smaller *II*, 2) a higher issue rate and hence a smaller *II*, and 3) when the loop has a larger number of operations that use a constrained resource such as the branch unit in our study.

The LAT column gives the latency of the software pipeline in cycles. That is, one iteration completes in LAT cycles. The S column gives the number of stages in the prologue/epilogue. S + 1 is the latency of the pipeline in terms of *II*. LAT is actually the latest issue time of an operation in the first iteration of then loop. Thus, it is a more accurate latency than II(S + 1).

The U column gives the number of times the loop is unrolled for Modulo Variable Expansion (MVE). Some numbers in the U column have a "+" appended that denotes that the loop was unrolled one extra time to reduce the register usage. Given a loop with three variables where two variables need to be renamed once (2 values) and the other twice (3 values), the loop needs to be unrolled a minimum of 3 times. If the loop is unrolled 3 times, then to be able to rename the variables correctly, 9 registers are needed since 2 does not divide 3 evenly. On the other hand, if the loop is unrolled 4 times, then 8 registers are needed (2 + 2 + 4). It is debatable whether register pressure or code expansion is more critical in terms of achieving a software pipelined loop that can execute on the given machine resources (i.e., without using spill code or incurring cache misses, respectively).

The R column gives the number of registers required after MVE, where LB is the lower bound and UB is the upper bound. Rau, Lee, Tirumalai, and Schlansker report that a simple and surprisingly tight lower bound can be obtained by calculating the total number of registers live during each cycle of one *II* in the kernel and taking the maximum of these totals [36].

Table 5.4 The software pipeline characteristics of EMS for issue rate 1.

LP	II	RII	LAT	S	U	R LB	R UB	P	OPS (1)	OPS (2)	OPS (3)
1	37	37	72	1	2	18	27	1	214	139	141
2	46	46	127	2	4+	23	34	1	446	297	302
3	60	60	170	2	3	22	29	1	485	312	316
4	10	10	17	1	2	5	7	1	43	22	22
5	78	77	133	1	2	17	24	1	351	241	243
6	140	140	329	2	3	24	37	1	1080	734	738
7	26	26	47	1	2	11	15	2	116	69	71
8	42	42	83	1	2	17	24	2	243	154	158
9	47	47	126	2	4+	24	34	2	466	307	312
10	47	47	126	2	4+	23	34	1	412	293	298
11	36	36	97	2	3	19	28	1	337	226	230
12	14	14	26	1	2	8	11	1	62	41	43
13	31	31	83	2	4+	22	34	1	278	197	202
14	43	43	84	1	2	16	21	1	241	167	169
15	30	30	87	2	4+	19	27	1	312	224	224
16	36	36	106	2	4+	20	29	1	342	246	246
17	36	36	71	1	2	13	18	3	200	133	135
18	23	23	40	1	2	11	15	2	116	79	79
19	18	18	44	2	3	9	13	1	216	140	140
20	19	19	37	1	2	9	10	3	142	84	84
21	23	23	61	2	4+	14	20	1	304	194	200
22	19	19	37	1	2	9	10	3	142	84	84
23	12	12	18	1	2	5	6	1	69	44	44
24	19	19	45	2	3	8	11	2	171	75	75
25	14	14	37	2	3	8	11	1	158	94	94
26	18	18	48	2	3	11	16	3	360	205	205
AVG	35.54	35.50	82.73	1.54	2.81	14.81	20.96	1.50	281.00	184.65	186.73

Table 5.5 The software pipeline characteristics of EMS for issue rate 2.

LP	II	RII	Lat	S	U	R LB	R UB	P	Ops (1)	Ops (2)	Ops (3)
1	19	19	75	3	4	30	51	1	646	500	415
2	23	23	113	4	6+	38	65	1	1044	790	689
3	30	30	134	4	6+	32	50	1	1340	1062	958
4	5	5	14	2	3	7	11	1	80	52	36
5	39	39	128	3	4	30	45	1	1018	780	684
6	70	70	240	3	4	41	59	1	1652	1278	1118
7	13	13	37	2	4+	16	24	2	316	258	209
8	21	21	79	3	4	31	51	2	644	472	399
9	24	24	115	4	6+	37	63	2	1092	856	694
10	24	24	95	3	4	33	55	1	536	392	347
11	18	18	81	4	6+	33	55	1	564	422	369
12	7	7	13	1	2	8	13	1	62	38	34
13	16	16	74	4	6+	34	58	1	528	426	353
14	22	22	65	2	4+	24	37	1	508	404	347
15	15	15	71	4	6+	29	49	1	480	344	304
16	18	18	88	4	6+	29	49	1	864	646	571
17	18	18	63	3	4+	21	34	3	550	414	335
18	12	12	41	3	4	18	30	2	376	256	194
19	9	9	51	5	6	19	33	3	942	588	541
20	10	10	34	3	4+	11	15	4	662	564	400
21	12	12	45	3	4	20	32	1	408	296	236
22	10	10	34	3	4+	11	15	4	662	564	400
23	6	6	16	2	3	7	11	1	144	114	82
24	10	10	34	3	4	8	12	2	298	238	164
25	7	7	26	3	4	13	21	1	224	172	132
26	9	9	32	3	4	14	24	4	586	334	282
AVG	17.96	17.96	69.15	3.12	4.46	22.85	37.00	1.69	624.08	471.54	395.89

Table 5.6 The software pipeline characteristics of EMS for issue rate 4.

LP	II	RII	Lat	S	U	R LB	R UB	P	Ops (1)	Ops (2)	Ops (3)
1	10	10	57	5	6	47	85	2	1584	1348	955
2	12	12	91	7	8	62	110	2	2124	1900	1342
3	15	15	131	8	10+	59	101	2	4040	3508	2969
4	4	3	11	2	3	6	11	1	132	88	36
5	20	20	105	5	6	46	79	1	1896	1620	1254
6	35	35	261	7	8	61	103	1	4280	3600	2974
7	7	7	34	4	6+	26	45	2	1052	984	665
8	11	11	72	6	8+	51	91	2	1764	1608	1030
9	12	12	92	7	8	60	108	2	2528	2212	1586
10	12	12	82	6	8+	56	100	1	1128	992	718
11	9	9	53	5	6	42	76	1	840	732	541
12	4	4	14	3	4	15	25	1	188	96	78
13	8	8	61	7	8	53	95	1	884	740	511
14	11	11	54	4	6+	38	67	1	736	676	469
15	8	8	69	8	10+	48	87	2	1548	1348	936
16	10	9	68	6	8+	45	80	1	1360	1092	781
17	10	9	45	4	4	30	51	3	960	728	403
18	7	6	33	4	6+	24	42	2	1032	800	505
19	5	5	38	7	8	26	46	4	2200	1892	1119
20	7	6	31	4	6+	15	25	4	2172	1744	693
21	6	6	35	5	6	29	50	2	984	792	557
22	7	6	31	4	6+	15	25	4	2172	1744	693
23	4	3	14	3	4	10	16	1	272	228	114
24	6	5	28	4	5	11	17	2	620	408	208
25	4	4	18	4	6+	15	26	1	384	332	177
26	5	5	34	6	8+	25	45	8	4840	3164	2032
AVG	9.58	9.27	60.08	5.19	6.62	35.19	61.77	2.08	1604.62	1322.15	897.92

Table 5.7 The software pipeline characteristics of EMS for issue rate 8.

LP	II	RII	LAT	S	U	R LB	R UB	P	OPS (1)	OPS (2)	OPS (3)
1	5	5	54	10	12+	82	156	2	3880	3472	2273
2	6	6	77	12	14+	98	189	2	5112	4760	3159
3	8	8	78	9	10	74	132	2	5376	5048	3533
4	3	2	11	3	4	9	15	1	296	336	81
5	10	10	95	9	10	77	142	2	4648	4504	3144
6	18	18	172	9	10	97	178	1	6768	6336	4717
7	5	4	25	4	6+	26	47	2	880	1032	323
8	7	6	55	7	8	66	117	3	4128	4344	2056
9	7	6	80	11	12	94	179	2	5320	5408	2814
10	6	6	77	12	14+	96	185	1	2656	2392	1602
11	5	5	49	9	10	69	129	1	1576	1336	813
12	3	2	14	4	6+	19	33	1	496	520	159
13	5	4	40	7	8	57	109	1	1480	1344	704
14	6	6	47	7	8	61	113	1	1776	1704	1040
15	5	4	48	9	10	58	111	1	1880	1592	824
16	5	5	59	11	12+	71	131	2	2640	2616	1400
17	8	6	36	4	4	30	52	3	1664	1232	367
18	5	4	28	5	6	27	52	2	1552	1296	436
19	4	3	35	8	10+	29	54	5	4672	4536	1540
20	7	6	31	4	4	12	22	4	3416	3224	736
21	4	3	31	7	8	37	68	2	1648	1544	654
22	7	6	31	4	4	12	22	4	3416	3224	736
23	3	2	11	3	4	10	16	1	408	400	115
24	5	4	32	6	8+	14	23	2	1760	1976	430
25	3	2	16	5	6+	19	33	1	648	472	165
26	3	3	26	8	8	29	57	8	13816	9968	4509
AVG	5.89	5.23	48.39	7.19	8.31	48.96	90.96	2.19	3150.46	2869.85	1474.23

The upper bound on the number of registers is sum of the maximum and minimum number of registers that are live simultaneously [72].

The P column gives the number of predicates in the loop after applying MVE. Note that the maximum number of predicates needed is 8. This justifies the use of bit operations for the allowable predicate set during RIC.

The OPs (1) column gives the number of operations in the resultant software pipelined loop assuming a VLIW without interlocking. The OPs (2) column gives the number of operations in the resultant software pipelined loop assuming a VLIW with interlocking. Thus, instructions with all no-op operations are deleted from the resultant code. Finally, the OPs (3) column gives the number of operations in the software pipelined loop assuming a VLIW with interlocking and a mechanism to insert no-op's. Whether or not such an architecture is realistic, the results provide insight into the code expansion for superscalar processors. Note that this is not the code expansion for superscalar since a VLIW machine model was assumed during scheduling.

The AVG reported is the arithmetic mean over the 26 loops. It is provided as a quick reference. The individual benchmark data is provided for a more detailed understanding.

5.5.2 II versus performance

It is interesting to study the effect of increasing *II* on the parameters of the software pipeline. Figure 5.9 shows the speedup as *II* is increased. *AII* is the minimum achieved *II*. While a larger *II* corresponds to decreased performance, it has some positive side effects which are explored in the following sections.

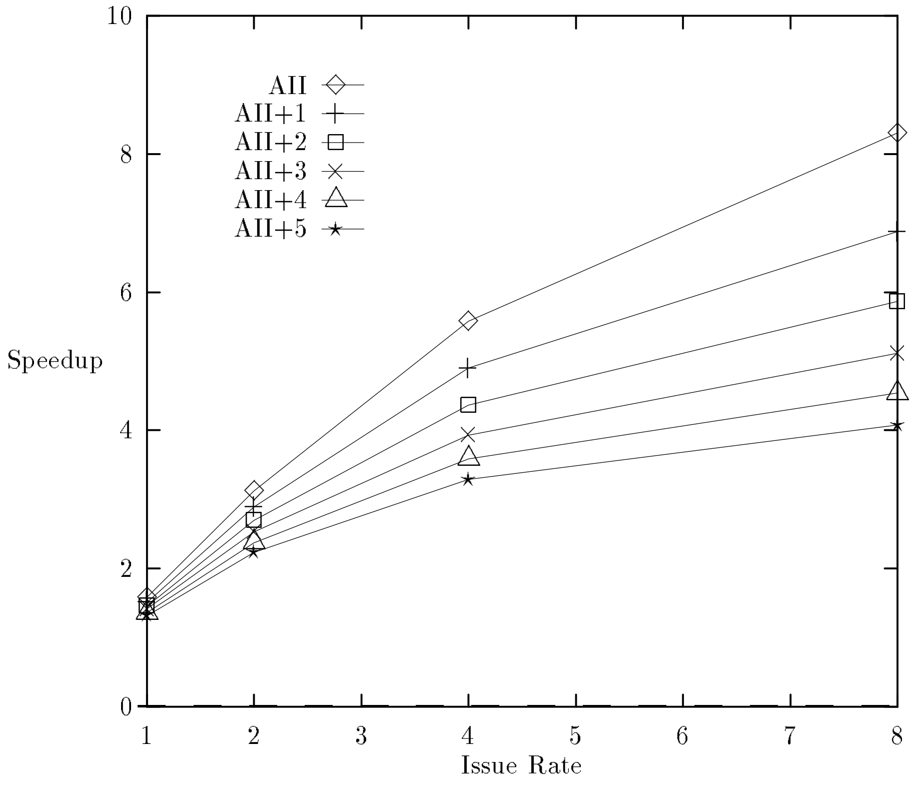


Figure 5.9 Speedup of EMS for increasing values of II. AII is the minimum achievable II.

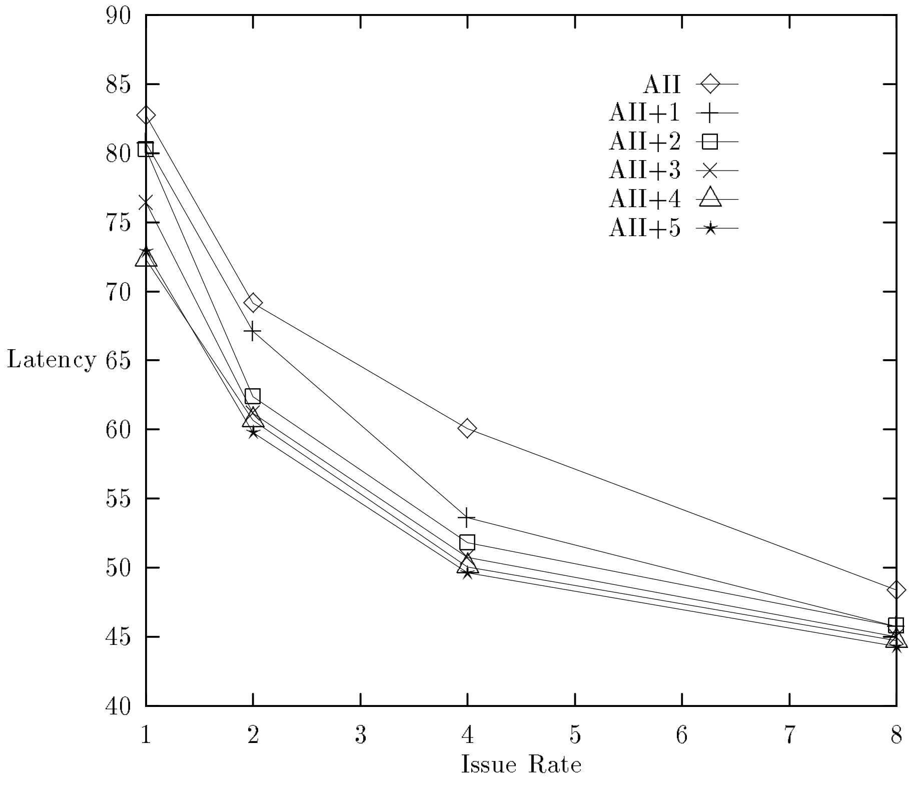


Figure 5.10 The pipeline latency (cycles) for EMS for increasing values of II. AII is the minimum achievable II.

5.5.3 II versus pipeline latency

As II is increased, the pipeline latency decreases and thus the minimum number of iterations for software pipelining decreases. Figure 5.10 and Figure 5.11 show the effect of increasing II on the pipeline latency in terms of cycles and stages, respectively. Figure 5.10 shows the average latest issue time across all loops and Figure 5.11 is the average number of stages in the prologue/epilogue across all loops.

As II increases, the number of stages, S, in the prologue decreases. Since S + 1 corresponds to the minimum number of iterations the loop must have to be software pipelined, for loops

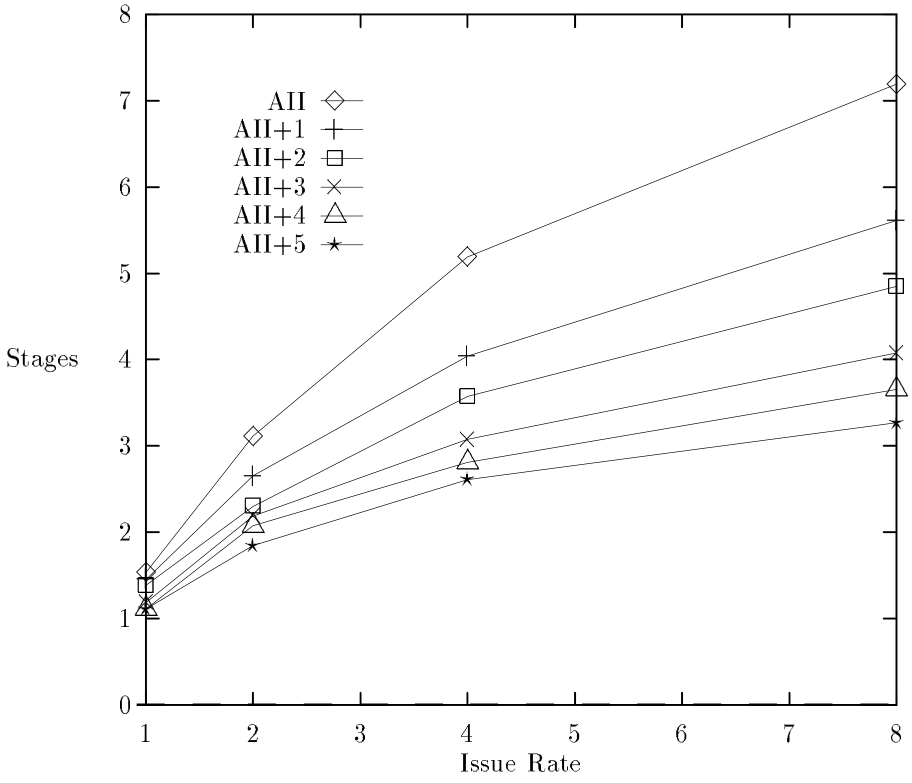


Figure 5.11 The stages in the prologue for EMS for increasing values of *II*. *AII* is the minimum achievable *II*.

with few iterations, it may be desirable to increase II until the number of iterations equals S + 1. On the other hand, simply unrolling the loop and scheduling may achieve a tighter schedule.

5.5.4 II versus register pressure

As the pipeline latency decreases, one iteration spans fewer stages and thus, the register pressure will decrease. As discussed in Section 5.5.1 lower and upper bounds on the number of registers can be calculated. In this section we divide the registers into three classes: integer, single precision floating-point, and double precision floating-point. Figures 5.12 and 5.13 show the average and maximum for the lower bound on integer registers. Figures 5.14 and 5.15 show the average and maximum for the upper bound on integer registers. Figures 5.16 through 5.23 show the corresponding data for single precision and double precision floating-point register classes.

Since Rau et al. have found that the lower bound is tight, we will focus our discussion on these results. It is interesting to note that the register pressure is not excessive. A 32 entry integer register file is sufficient for machines with issue rates 1 and 2 and a 64 entry integer register file is sufficient for machines with issue rates 4 and 8. For single precision floating-point registers, a 32 entry register file is again sufficient for issue 1 and 2 machines and a 64 entry register file is sufficient for issue 4 and 8 machines. For double precision floating-point registers, a 32 entry register file is sufficient for machines with issue rates as large as 8.

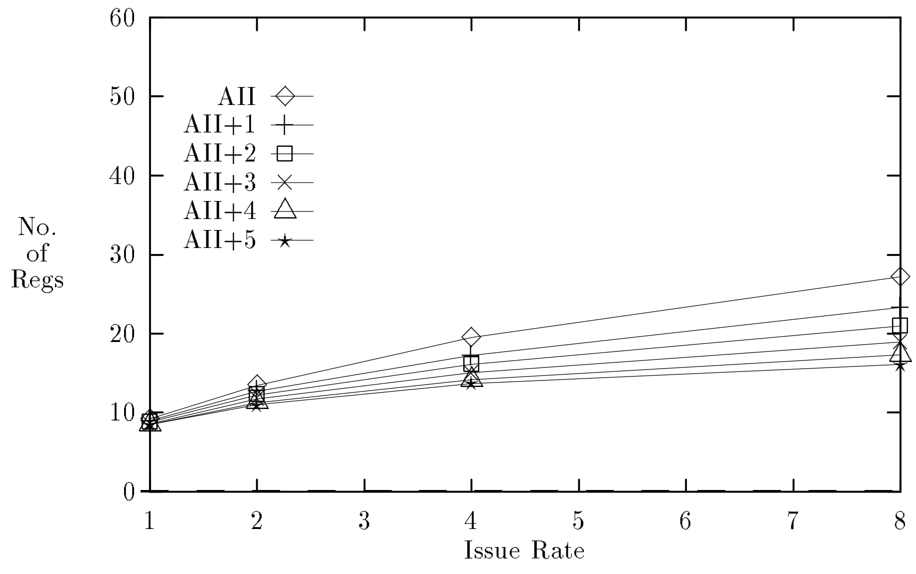


Figure 5.12 Average lower bound on number of integer registers for increasing values of II. AII is the minimum achievable II.

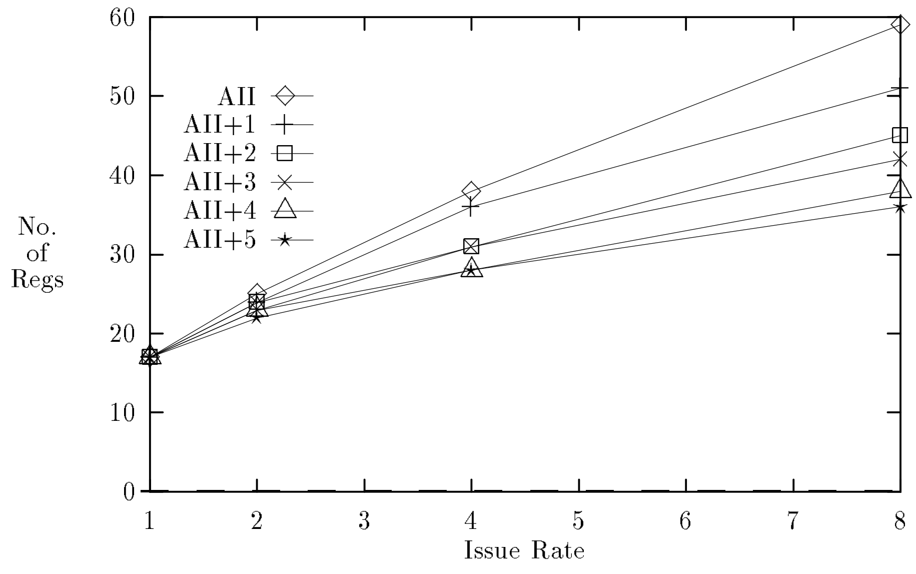


Figure 5.13 Maximum lower bound on number of integer registers for increasing values of II. AII is the minimum achievable II.

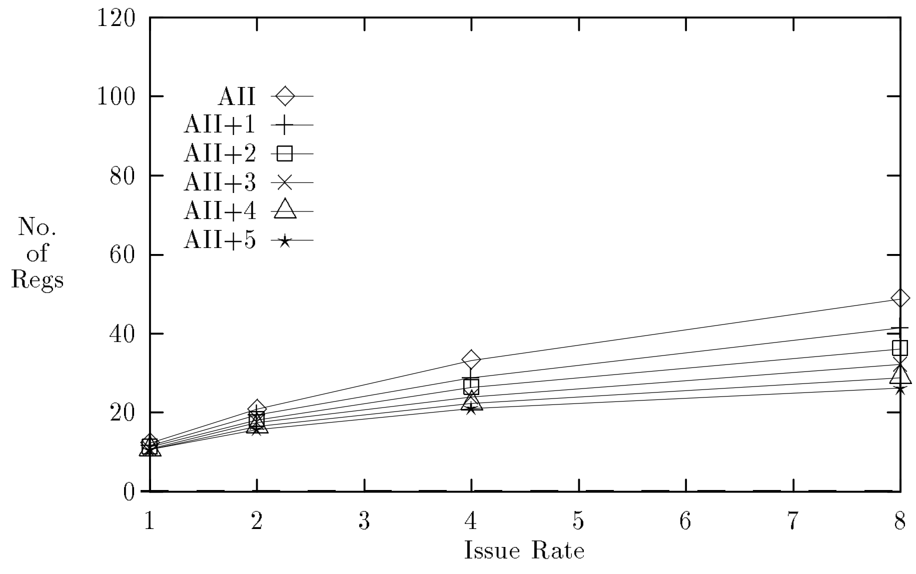


Figure 5.14 Average upper bound on number of integer registers for increasing values of II. AII is the minimum achievable II.

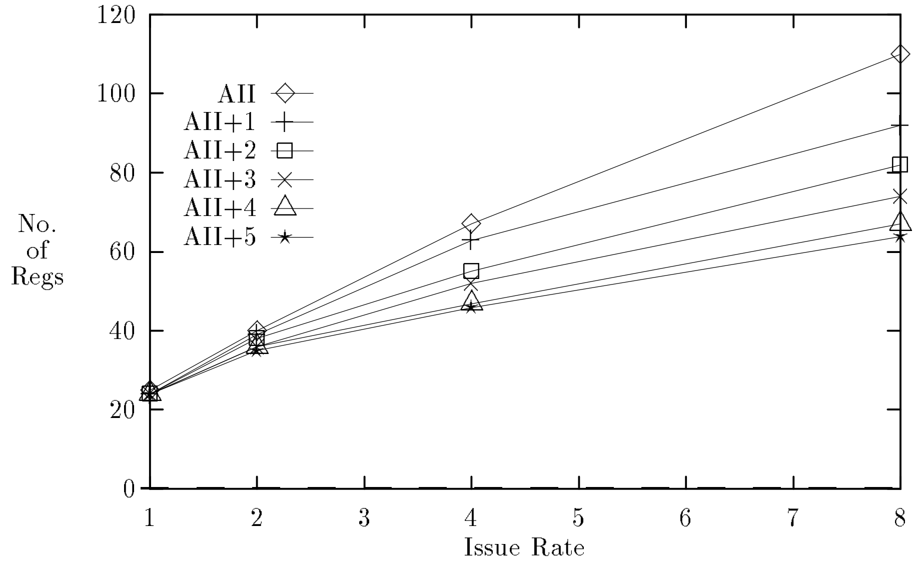


Figure 5.15 Maximum upper bound on number of integer registers for increasing values of II. AII is the minimum achievable II.

There are cases when the register usage will exceed the machine resources. In a software pipelined loop it is important to avoid this situation. Otherwise, spill code is required to load/store the value into a temporary register. Unless register allocation is integrated with scheduling, adding spill code after the schedule is found will decrease the performance of the software pipelined loop since it introduces extra cycles into the steady state. In addition, these cycles do little useful work other than load/store the register value. Finally, the extra cycles may violate the schedule for machines without interlocking.

One solution is to use the lower bound estimates during scheduling in order to increase II if the requirements exceed the machine resources. Note that while in general the register requirements decrease as II increases, this is not always true. Figure 5.19 shows that the maximum number of registers required increases from $II + 1$ to $II + 2$ for an issue 8 machine.

5.5.5 II versus unrolling

As II increases, the register pressure decreases because the lifetime of a variable is likely to span fewer IIs. Thus, the number of times the loop is unrolled due to register renaming decreases. Figure 5.24 shows the number of times the kernel is unrolled as II is increased.

5.5.6 II versus code expansion

Since both the number of times the loop is unrolled and the number of stages in the prologue/epilogue decrease as II increases, we expect that the code expansion will also decrease. Figure 5.25 shows the code expansion assuming no interlocking as II is increased. Figure 5.26 shows the code expansion assuming interlocking as II is increased. Figure 5.27 shows the code expansion assuming interlocking with no no-op operations as II is increased. As mentioned

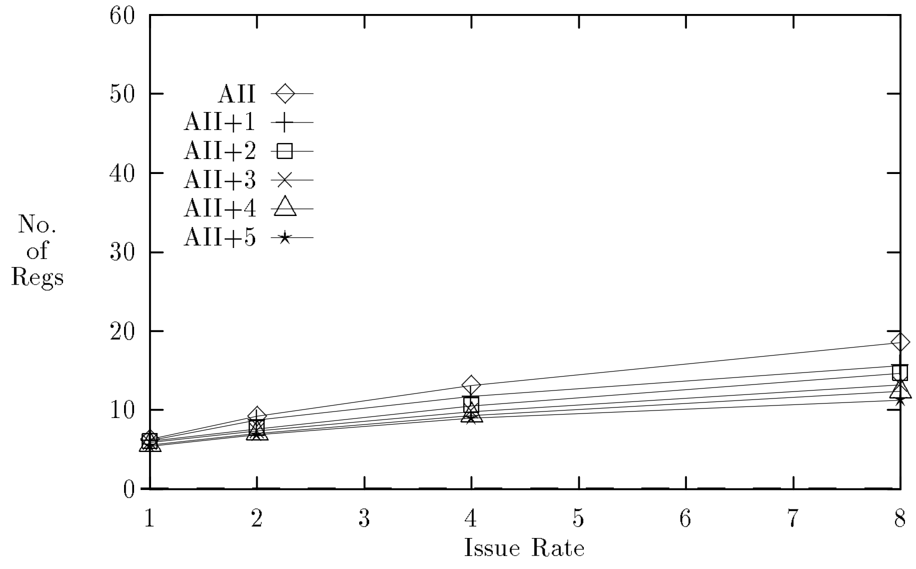


Figure 5.16 Average lower bound on number of single precision floating-point registers for increasing values of *II*. *AII* is the minimum achievable *II*.

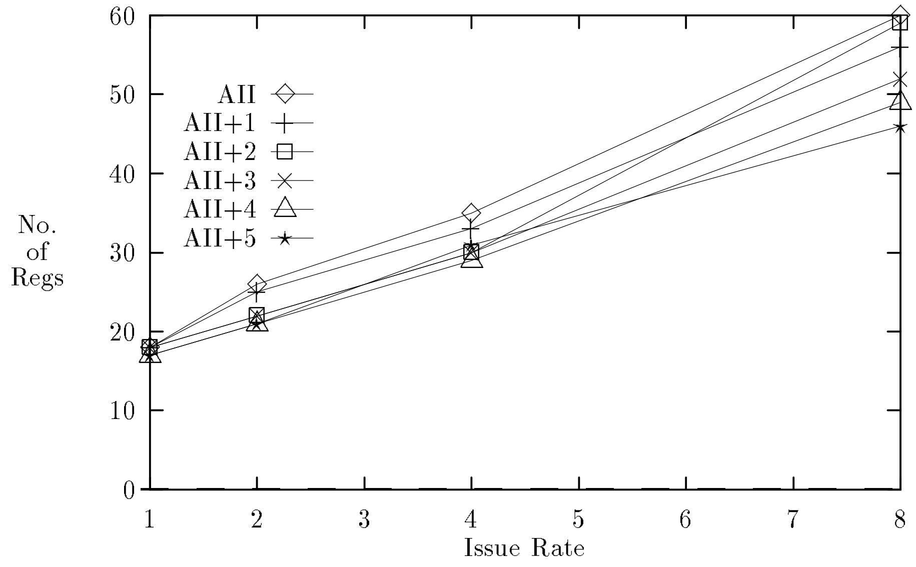


Figure 5.17 Maximum lower bound on number of single precision floating-point registers for increasing values of *II*. *AII* is the minimum achievable *II*.

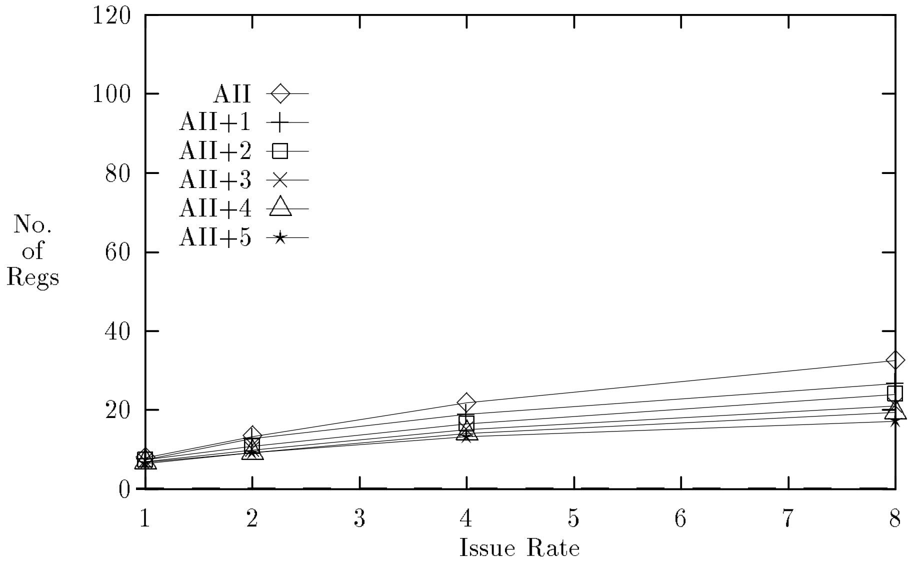


Figure 5.18 Average upper bound on number of single precision floating-point registers for increasing values of *II*. *AII* is the minimum achievable *II*.

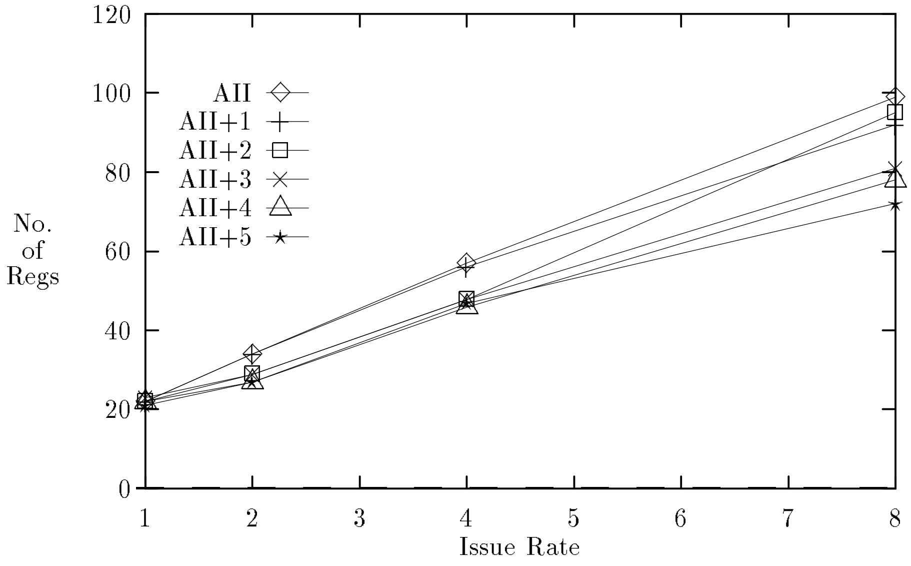


Figure 5.19 Maximum upper bound on number of single precision floating-point registers for increasing values of *II*. *AII* is the minimum achievable *II*.

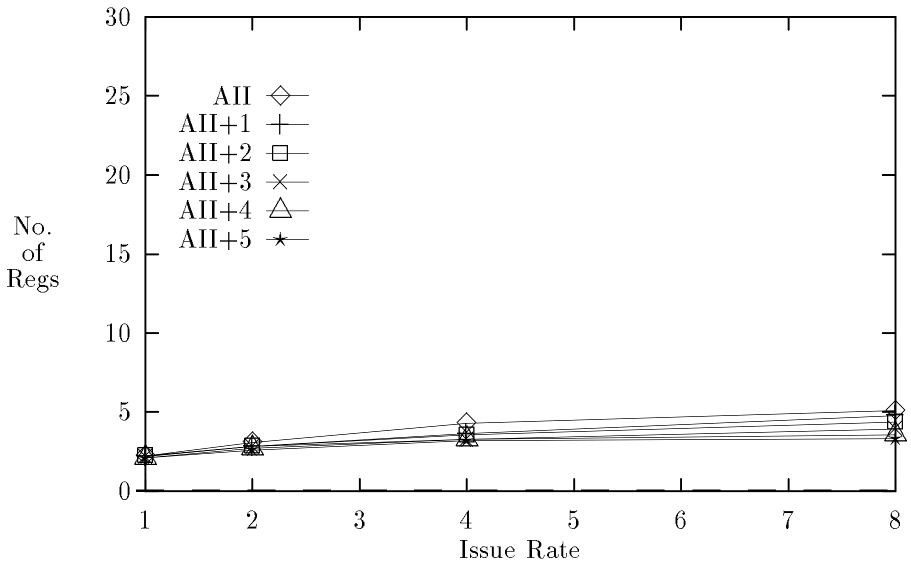


Figure 5.20 Average lower bound on number of double precision floating-point registers for increasing values of *II*. *AII* is the minimum achievable *II*.

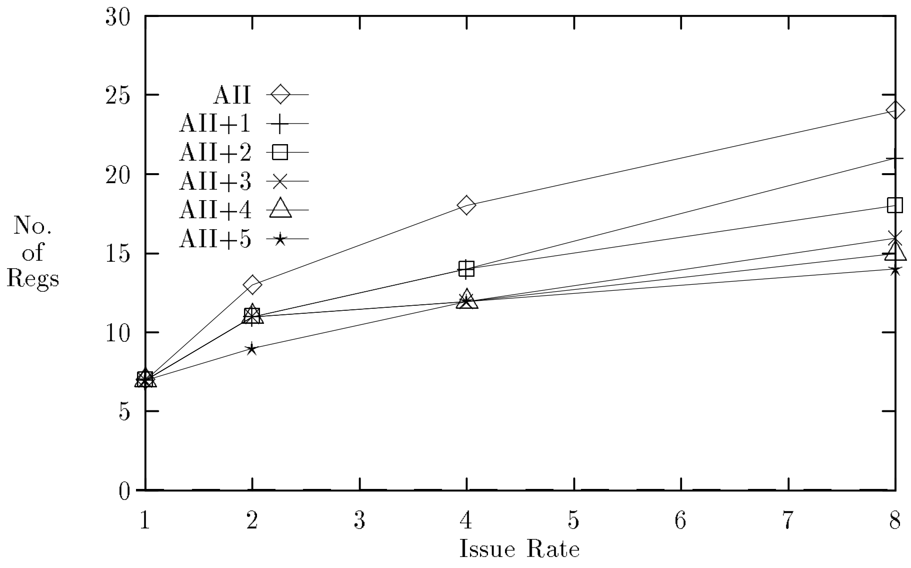


Figure 5.21 Maximum lower bound on number of double precision floating-point registers for increasing values of *II*. *AII* is the minimum achievable *II*.

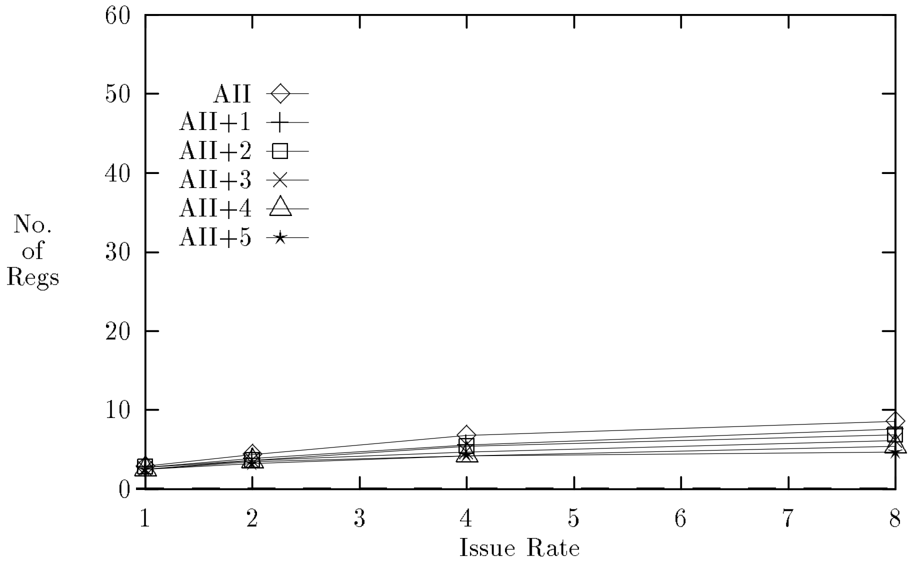


Figure 5.22 Average upper bound on number of double precision floating-point registers for increasing values of II. AII is the minimum achievable II.

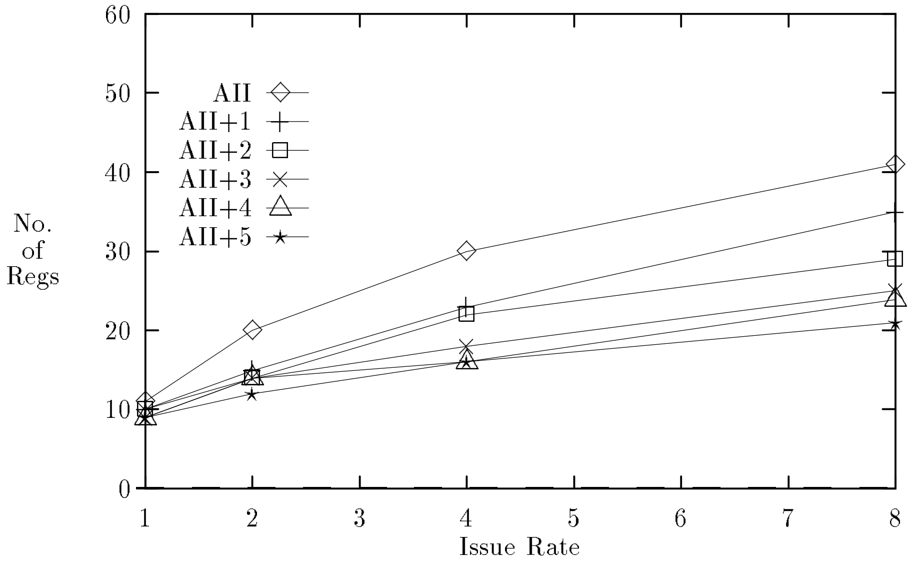


Figure 5.23 Maximum upper bound on number of double precision floating-point registers for increasing values of II. AII is the minimum achievable II.

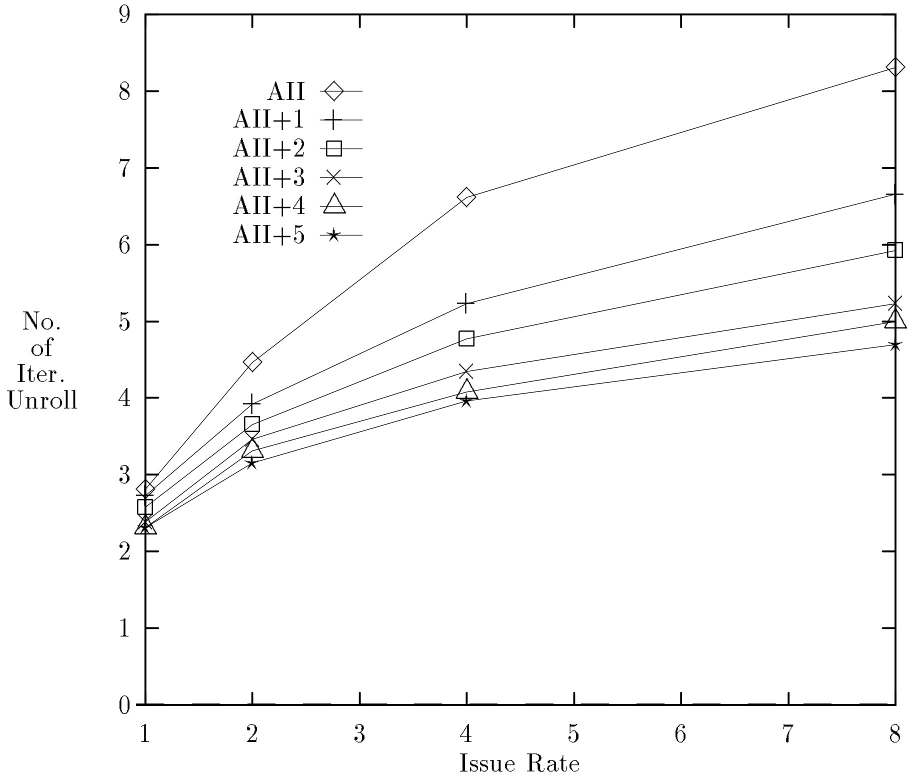


Figure 5.24 Number of times kernel is unrolled for increasing values of II. AII is the minimum achievable II.

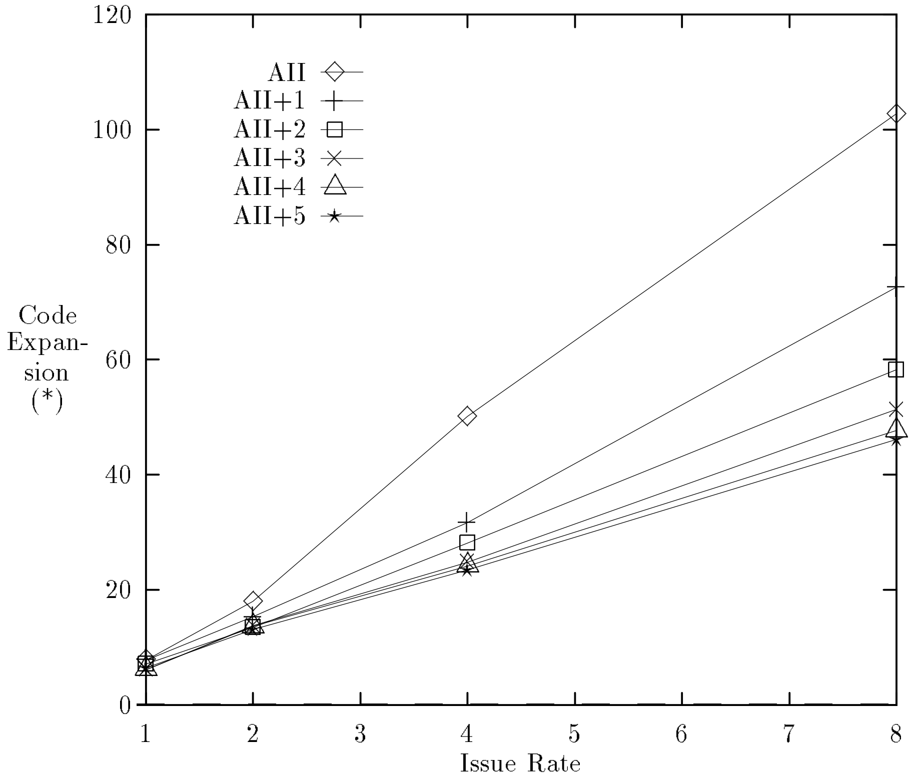


Figure 5.25 Code expansion assuming no interlocking. *AII* is the minimum achievable *II*.

earlier, the results in Figure 5.27 can be used to estimate the code expansion for an equivalent superscalar processor.

5.5.7 Experimental complexity

As stated in Section 2.6 the time complexity of the RIC algorithm is

$$O(H_p L_{max} + H_d L_{max}^2 + 2H_m L_{max}^2),$$

where H_p is the number of operations in the hyperblock (or equivalently software pipeline for EMS) excluding pd operations and pm operations, H_d is the number of pd operations in the hyperblock, and H_m is the number of pm operations. $H = H_p + H_d + H_m$. L_{max} is the maximum

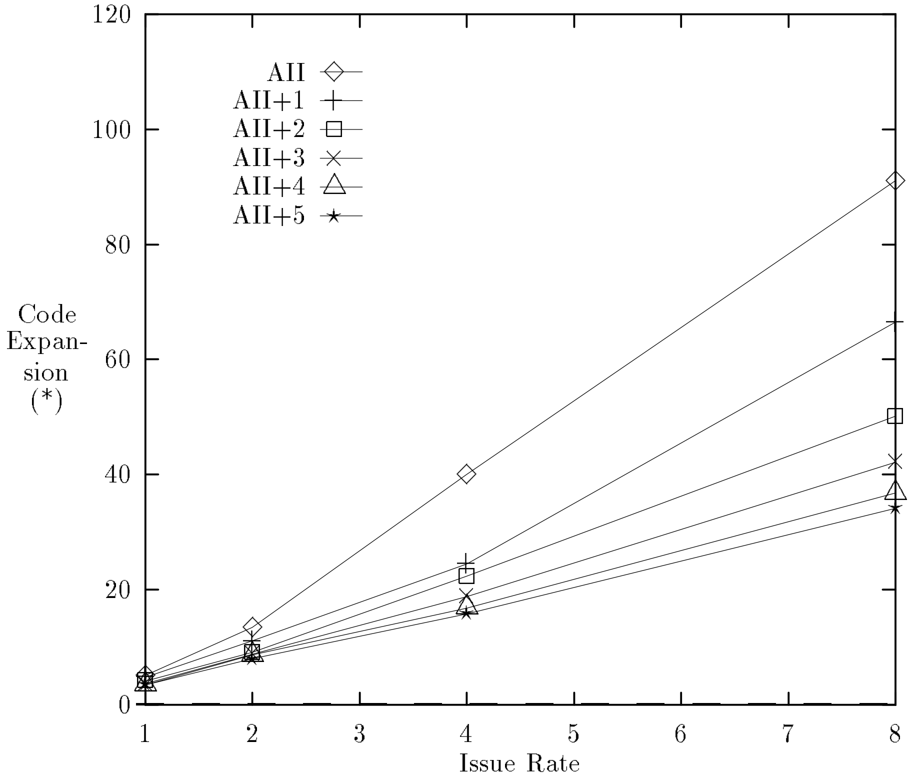


Figure 5.26 Code expansion assuming interlocking. *AII* is the minimum achievable *II*.

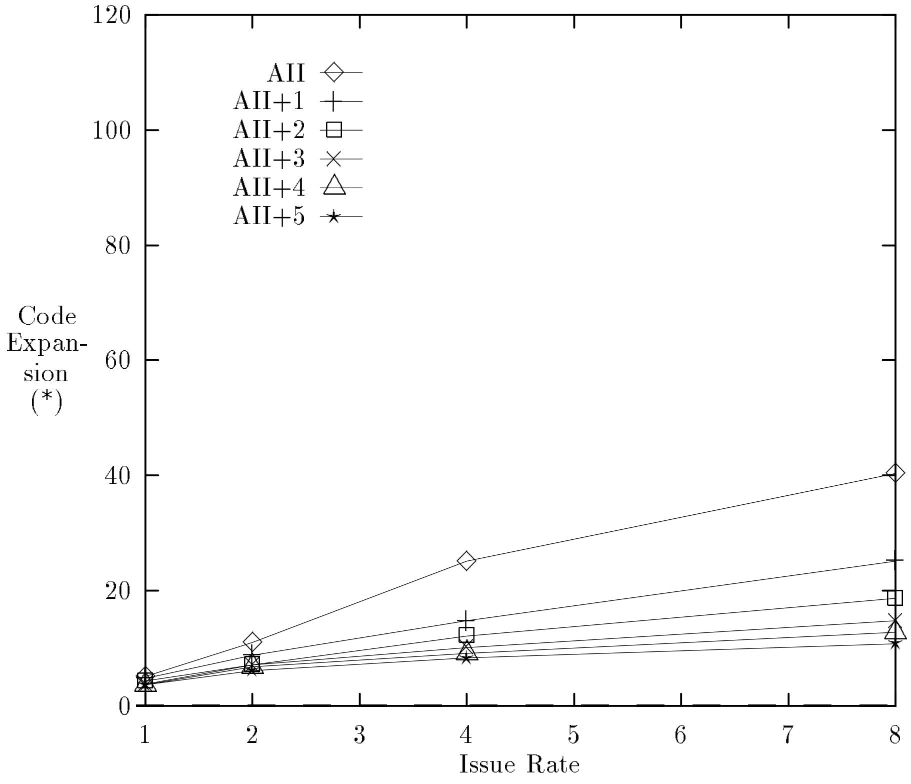


Figure 5.27 Code expansion assuming interlocking without no-op's. *AII* is the minimum achievable *II*.

number of nodes in the leaf node set. The complexity of RIC can also be expressed in terms of the number of nodes in the resultant control flow graph $|N|$: The complexity is thus $O(H|N|^2)$.

Table 5.8 shows the parameters of RIC for the original loop for issue rate 1. Tables 5.9 through 5.12 report the parameters of RIC for the software pipelined loop for the different issue rates. Also, the experimental complexity of RIC is calculated using L_{max}. The table also presents $H|N|^2$ to illustrate that it is not a good bound on the complexity of RIC. The actual experimental complexity is much more reasonable and is closer to H than $H|N|^2$. Note that the experimental complexity reported is actually high since it assumes L_{max} which is somewhat larger than the average L as illustrated by L_{avg}.

Table 5.8 Complexity of RIC for original loop with issue rate 1.

LP	H_P	H_D	H_M	$\|N\|$	L_{AVG}	L_{MAX}	$H\|N\|^2$	$H_P L_{MAX}$ $+ H_D L^2_{MAX}$ $+ 2H_M L^2_{MAX}$
1	36	1	1	4	1	2	608	84
2	45	1	1	4	1	2	752	102
3	59	1	1	4	1	2	976	130
4	8	1	1	4	1	2	160	28
5	77	1	1	4	1	2	1264	166
6	140	1	1	4	1	2	2272	292
7	22	2	2	7	1	2	1274	68
8	39	2	2	7	1	2	2107	102
9	44	2	2	7	1	2	2352	112
10	46	1	1	4	1	2	768	104
11	35	1	1	4	1	2	592	82
12	13	1	1	4	1	2	240	38
13	30	1	1	4	1	2	512	72
14	42	1	1	4	1	2	704	96
15	29	1	1	4	1	2	496	70
16	35	1	1	4	1	2	592	82
17	33	3	3	10	1	2	3900	102
18	21	2	2	7	1	2	1225	66
19	19	1	1	4	1	2	336	50
20	16	3	3	10	1	2	2200	68
21	22	1	1	4	1	2	384	56
22	16	3	3	10	1	2	2200	68
23	10	1	1	4	1	2	192	32
24	15	2	2	7	1	2	931	54
25	12	1	1	4	1	2	224	36
26	16	1	1	4	1	2	288	44
AVG	33.85	1.42	1.42	5.27	1.00	2.00	1059.58	84.77

Table 5.9 Complexity of RIC for software pipeline loop with issue rate 1.

LP	H_P	H_D	H_M	$\|N\|$	L_{AVG}	L_{MAX}	$H\|N\|^2$	$H_P L_{MAX}$ $+ H_D L_{MAX}^2$ $+ 2H_M L_{MAX}^2$
1	106	3	3	13	1	2	18928	248
2	265	6	6	21	1	2	122157	602
3	291	5	5	18	1	2	97524	642
4	22	3	3	9	1	2	2268	80
5	229	3	3	12	1	2	33840	494
6	696	5	5	18	1	2	228744	1452
7	64	6	6	18	1	2	24624	200
8	115	6	6	21	1	2	56007	302
9	259	12	12	39	1	2	430443	662
10	271	6	6	21	1	2	124803	614
11	171	5	5	19	1	2	65341	402
12	37	3	3	12	1	2	6192	110
13	175	6	6	21	1	2	82467	422
14	124	3	3	12	1	2	18720	284
15	169	6	6	21	1	2	79821	410
16	205	6	6	21	1	2	95697	482
17	97	9	9	36	1	4	149040	820
18	61	6	6	21	1	2	32193	194
19	91	5	5	20	1	2	40400	242
20	46	9	9	40	1	4	102400	616
21	127	6	6	23	1	2	73531	326
22	46	9	9	40	1	4	102400	616
23	28	3	3	11	1	2	4114	92
24	71	10	10	31	1	4	87451	764
25	56	5	5	19	1	2	23826	172
26	76	5	5	33	2	4	93654	544
AVG	149.92	5.81	5.81	21.92	1.04	2.38	84484.04	453.54

Table 5.10 Complexity of RIC for software pipeline loop with issue rate 2.

LP	H_P	H_D	H_M	$\|N\|$	L_{AVG}	L_{MAX}	$H\|N\|^2$	$H_P L_{MAX} + H_D L_{MAX}^2 + 2H_M L_{MAX}^2$
1	246	7	7	26	1	2	175760	576
2	441	10	10	35	1	2	564725	1002
3	581	10	10	35	1	2	736225	1282
4	36	5	5	13	1	2	7774	132
5	533	7	7	26	1	2	369772	1150
6	974	7	7	24	1	2	569088	2032
7	127	12	12	39	1	2	229671	398
8	267	14	14	46	1	2	624220	702
9	431	20	20	65	1	2	1989975	1102
10	316	7	7	24	1	2	190080	716
11	341	10	10	33	1	2	393129	802
12	37	3	3	9	1	2	3483	110
13	291	10	10	33	1	2	338679	702
14	247	6	6	22	1	2	125356	566
15	281	10	10	32	1	2	308224	682
16	341	10	10	35	1	2	442225	802
17	225	21	21	77	1	4	1583043	1908
18	141	14	14	60	1	4	608400	1236
19	199	11	11	90	3	8	1790100	3704
20	106	21	21	179	3	8	4742068	4880
21	148	7	7	26	1	2	109512	380
22	106	21	21	179	3	8	4742068	4880
23	46	5	5	19	1	2	20216	152
24	99	14	14	45	1	2	257175	366
25	78	7	7	25	1	2	57500	240
26	106	7	7	58	3	8	403680	2192
AVG	259.38	10.62	10.62	48.27	1.31	3.08	822390.31	1257.46

Table 5.11 Complexity of RIC for software pipeline loop with issue rate 4.

LP	H_P	H_D	H_M	$\|N\|$	L_{AVG}	L_{MAX}	$H\|N\|^2$	$H_P L_{MAX} + H_D L^2_{MAX} + 2H_M L^2_{MAX}$
1	386	11	11	72	2	4	2115072	2072
2	661	15	15	92	2	4	5848624	3364
3	1045	18	18	114	2	4	14048676	5044
4	36	5	5	13	1	2	7774	132
5	837	11	11	38	1	2	1240396	1806
6	2086	15	15	50	1	2	5290000	4352
7	211	20	20	95	2	4	2265275	1804
8	533	28	28	128	2	4	9650176	3476
9	646	30	30	141	2	4	14035986	4024
10	631	14	14	45	1	2	1334475	1430
11	375	11	11	38	1	2	573268	882
12	85	7	7	17	1	2	28611	254
13	436	15	15	48	1	2	1073664	1052
14	411	10	10	33	1	2	469359	942
15	505	18	18	108	1	4	6310224	2884
16	477	14	14	47	1	2	1115545	1122
17	257	24	24	125	1	8	4765625	6664
18	201	20	20	90	2	4	1952100	1764
19	271	15	15	182	4	8	9970324	5048
20	151	30	30	309	4	16	20146492	25456
21	232	11	11	67	2	4	1140206	1456
22	151	30	30	309	4	16	20146492	25456
23	64	7	7	24	1	2	44928	212
24	127	18	18	64	1	4	667648	1372
25	111	10	10	33	1	2	142659	342
26	211	14	14	394	12	32	37101404	49760
AVG	428.35	16.19	16.19	102.92	2.08	5.46	6210962.00	5852.69

Table 5.12 Complexity of RIC for software pipeline loop with issue rate 8.

LP	H_P	H_D	H_M	$\|N\|$	L_{AVG}	L_{MAX}	$H\|N\|^2$	$H_P L_{MAX} + H_D L_{MAX}^2 + 2H_M L_{MAX}^2$
1	771	22	22	136	3	4	15074240	4140
2	1145	26	26	162	2	4	31414068	5828
3	1103	19	19	120	3	4	16430400	5324
4	50	7	7	23	1	2	33856	184
5	1445	19	19	116	2	4	19955248	6692
6	2642	19	19	62	1	2	10301920	5512
7	211	20	20	63	1	2	996219	662
8	571	30	30	254	3	8	40709596	10328
9	990	46	46	276	2	4	82422432	6168
10	1171	26	26	83	1	2	8425247	2654
11	647	19	19	61	1	2	2548885	1522
12	121	10	10	35	1	2	172725	362
13	436	15	15	49	1	2	1118866	1052
14	616	15	15	50	1	2	1615000	1412
15	533	19	19	61	1	2	2124691	1294
16	783	23	23	138	1	4	15787476	4236
17	257	24	24	123	1	8	4614345	6664
18	221	22	22	106	2	4	2977540	1940
19	325	18	18	218	5	8	17156164	6056
20	121	24	24	289	6	16	14115049	20368
21	316	15	15	91	2	4	2865226	1984
22	121	24	24	289	6	16	14115049	20368
23	64	7	7	25	1	2	48750	212
24	197	28	28	120	1	4	3643200	2132
25	122	11	11	35	1	2	176400	376
26	241	16	16	858	23	64	200972768	212032
AVG	585.38	20.15	20.15	147.81	2.81	6.85	19608282.00	12673.15

CHAPTER 6

BENEFIT OF PREDICATED EXECUTION FOR MODULO SCHEDULING

In the previous chapter we saw that software pipelining yields high code expansion. There are three sources of code expansion: 1) code expansion due to unrolling for Modulo Variable Expansion, 2) code expansion due to the prologue/epilogue, and 3) code expansion due to overlapping conditional constructs. Hardware support can be used to eliminate all code expansion. The Cydra 5 architecture provides two forms of hardware support to eliminate code expansion [11], [12]. The rotating register file can dynamically rename registers and thus eliminate the need to unroll the kernel [36]. Predicated Execution support can be used to conditionally execute operations. If predicates are assigned to loop conditions, then prologue and epilogue code can be eliminated [37]. Eliminating the prologue and epilogue prevents the compiler from scheduling operations outside the loop with the prologue and epilogue code. Finally, predicated execution can be used to avoid code duplication due to overlapping conditional constructs. This last form of hardware support is explored in this chapter [64].

6.1 Architecture Support for Predicated Execution

The architecture support required for predicated execution consists of: 1) predicate defining (pd) operations, 2) predicate invalidating (pi) operations, 3) a predicate register file, and 4) predicated operations. An example of an architecture similar to the Cydra 5 is shown in Figure 6.1 [11], [12], [52]. We assume 14 types of pd operations: integer, single precision

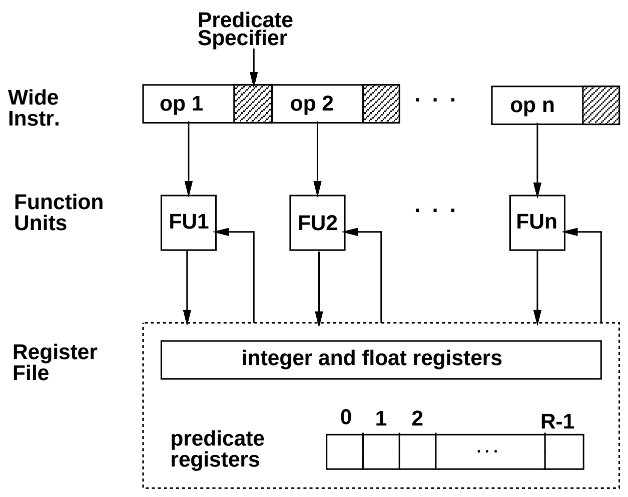


Figure 6.1 Architecture support for predicated execution.

floating-point; double precision floating-point versions of equal, not equal, greater than, and greater than equal; and unsigned integer versions for greater than and greater than equal. These operations compare two source operands and set the value of the destination predicate register accordingly. A pi operation is used to invalidate the specified predicate register.

The predicate register file has **R** 1-bit registers. The register is cleared if the predicate is not defined and set if it is defined. A pi operation clears the register. A pd operation sets the bit if the condition is true, otherwise, it clears it. Each operation within the wide instruction word has a predicate register specifier of width $\log_2 \mathbf{R}$. The width can be reduced if the predicate register file is implemented as a rotating register file as in the Cydra 5 [52].

All operations within the wide instruction are executed. After the predicate register file access delay, an operation in the execution pipeline that refers to a cleared predicate register will be squashed.

6.2 Experimental Results

The pd operations have a one cycle latency. We assume that the machine has the Intel i860 instruction set and latencies used in Phase-I experiments of Chapter 5. To measure the predicate register file size requirements, we use an unlimited predicate register file. The model has an infinite register file and an ideal cache. The experiments were performed using machines with instruction widths or, equivalently, issue rates of 2, 4, and 8.

With Predicated Execution, all operations in the loop are fetched, and those with their predicates set complete execution. Thus, whereas EMS can schedule two operations from different control paths in the same slot, Predicated Execution allows only one operation per slot. For this reason, the lower bound on II is the sum of the resources constraints along all paths. Thus, compared with EMS, we would expect Modulo Scheduling with Predicated Execution to have a larger RII.

Figure 6.2 shows the speedup for Modulo Scheduling with Predicated Execution and for EMS. The speedup results presented are the harmonic mean of the speedup for each loop. We assume that the loop executes an infinite number of times. Thus, the effect of the prologue and epilogue are not accounted for. Modulo Scheduling with Predicated Execution performs 2%, 6% and 27% better than EMS for issue rates 2, 4, and 8, respectively. The reason that Modulo Scheduling performs better than EMS as the issue rate increases is that the machine model for EMS assumes one branch per cycle and thus the branch unit becomes a limiting resource.

Figure 6.3 shows the arithmetic mean of RII and the achieved II for EMS and Modulo Scheduling with Predicated Execution. Note that RII for EMS is larger than RII for Predicated Execution. This supports the hypothesis that the one branch per cycle constraint often limits the lower bound on II for EMS (especially for issue-8 machines). If the underlying architecture

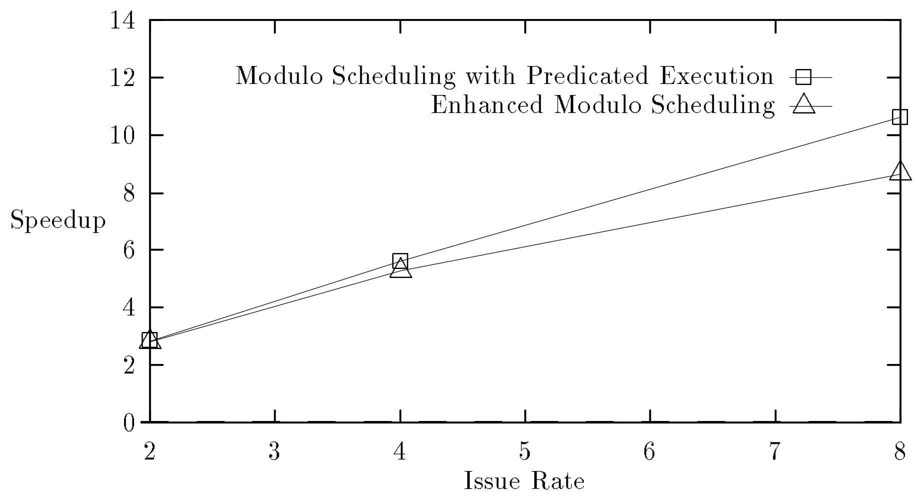


Figure 6.2 Speedup of Modulo Scheduling with and without support for Predicated Execution.

supports multiple branches per cycle, then EMS should perform as well as or better than Predicated Execution since EMS almost always achieves its lower bound on II.

Multiple branches per cycle would, however, increase the code expansion of EMS. Figure 6.4 shows the code expansion for Modulo Scheduling with Predicated Execution and for EMS. Note that the code expansion for Modulo Expansion with Predicated Execution is the minimum code expansion given that the prologue and epilogue are generated and that there is no support for dynamic register renaming. The code expansion for EMS is 75%, 103%, and 257% times larger than for Predicated Execution for issue 2, 4, and 8, respectively. Note that these results are slightly misleading since the code expansion due to the predicate register specifiers has not been included since it is implementation dependent.

Figure 6.5 gives the percentage of the loops that can fit into the specified predicate register file size. As expected, the predicate register requirements increase as the issue rate increases

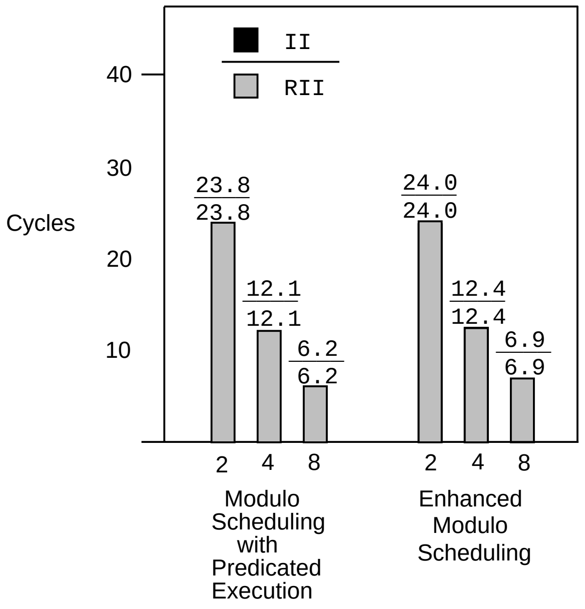


Figure 6.3 Average lower bound on II (RII) versus achieved II for EMS and Modulo Scheduling with Predicated Execution.

and more iterations are overlapped. A predicate register file with R registers requires $\log_2 R$ bits in the predicate register specifier. To schedule all of the loops for an issue-8 machine would require a 6-bit predicate register specifier. The majority of the loops scheduled, however, have only one conditional construct per loop. Thus, if a rotating register file is used [12], then a 1-bit predicate register specifier would be adequate.

6.3 Using Control Flow Profiling

One drawback to some software pipelining techniques, including Modulo Scheduling, is that the length of the steady-state schedule is fixed for all paths through the loop [60]. This can particularly limit the performance if the most frequently executed path is much shorter than other paths through the loop. Control-flow profiling information can be used to determine which paths to include in the software pipelined loop body. Figure 6.6(a) shows the weighted

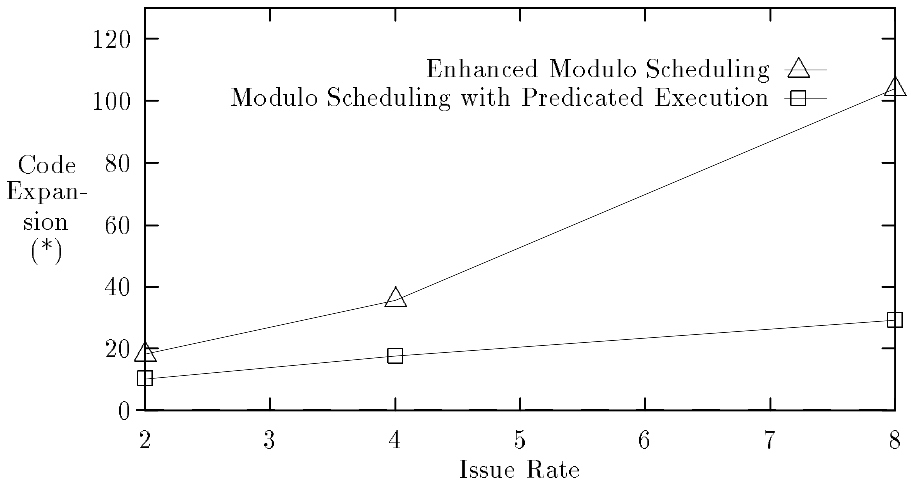


Figure 6.4 Code Expansion of Enhanced Modulo Scheduling and Modulo Scheduling with Predicated Execution.

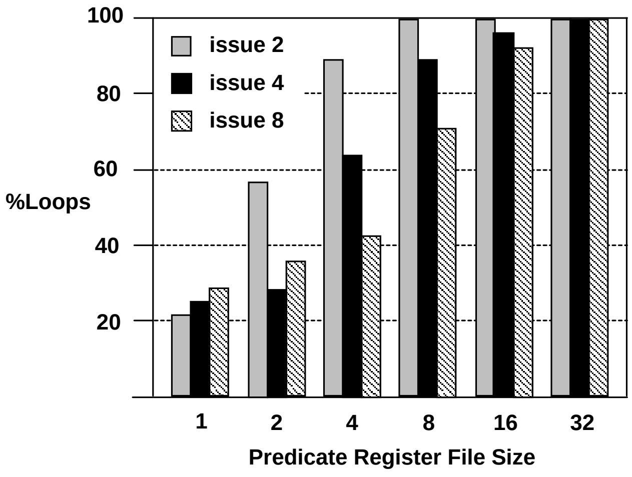


Figure 6.5 Predicated register requirements.

control flow graph of a simple loop. The loop body consists of four operations, where operation **A** is a conditional branch.[1] Since operation **C** is executed only 10% of the time, only the path {**A, B, D**} is software pipelined.

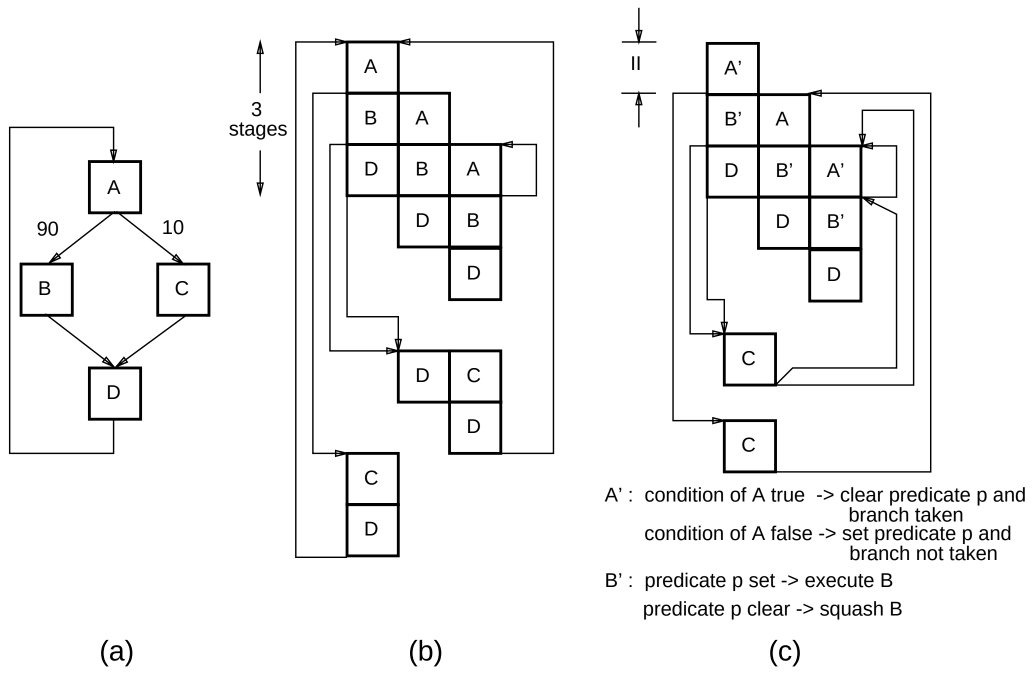
When control paths are excluded from the software pipeline, a mispredicted branch can break the software pipeline. The simplest approach to handling this hazard is to empty the software pipeline, execute the code along the taken path, and refill the pipeline. Figure 6.6(b) shows the software pipelined loop using this hazard resolution technique. This is a software hazard resolution technique since the compiler generates the necessary code to empty the pipeline and to execute the code along the taken path. Note that this example is overly simplified to illustrate the order in which operations are executed when a branch misprediction hazard occurs. Thus, explicit branch operations other than **A** are not shown, but, their corresponding control flow arcs are shown.[2]

In this simple example, there are only two stages in the software pipeline until the steady-state execution is reached. Typically, the number of stages is higher. For example, Section 5.5.3 reported that the average number of stages is approximately seven for an issue 8 machine. Thus, it can be costly to recover from a mispredicted branch using a software resolution technique.

It would be ideal if the execution could jump out of the pipeline, execute the taken path code, and jump back into the pipeline. In order to do so, the operations in the pipeline that are along the not taken path of the branch need to be squashed. Squashing means that the operations are fetched but not executed. Predicated hardware support can be used to squash operations in the software pipeline [11], [12]. Figure 6.6c shows the software pipeline schedule

[1]To keep the example simple, the loop back branch is ignored.

[2]When the loop back branch is considered, code is also required to handle early exits from the software pipeline [37].



A
90
10
B
C
D
3 stages
A
B
A
D
B
A
D
B
D
D
C
D
C
D
II
A'
B'
A
D
B'
A'
D
B'
D
C
C
A' : condition of A true -> clear predicate p and branch taken
condition of A false -> set predicate p and branch not taken
B' : predicate p set -> execute B
predicate p clear -> squash B
(a)
(b)
(c)

Figure 6.6 Using control-flow profile for software pipelining. a) Weighted control-flow graph. b) Software hazard resolution. c) Hardware hazard resolution.

assuming predicated hardware support. In this example, $\mathbf{A}'$ is assumed to be a conditional branch operation that also sets a predicate **p**. Operation $\mathbf{B}'$ executes **B** if predicate **p** is set, otherwise **B** is squashed. When the condition of **A** is false, the predicate **p** is set and operation **B** executes. When the condition of **A** is true, the predicate **p** is cleared and control branches to execute operation **C**. After **C** executes, control branches back to the operation following the mispredicted branch. Since the predicate **p** is cleared, operation **B** will not execute but other operations scheduled with $\mathbf{B}'$ will.

6.3.1 Modulo Scheduling with profiling and predication

Control-flow profiling information is used to modulo schedule the most frequently executed paths in the loop. After the most frequently executed paths have been selected, If-conversion is applied to these paths. For conditional branches where both successor basic blocks are included

Algorithm Modulo Schedule with Profile and Predication: Given a hyperblock loop with infrequent or undesirable paths eliminated, apply Modulo Scheduling to software pipeline the loop.

```
construct dependence graph
/* $RII$ is the sum of the resource constraints */
determine lower bound on $II$
while modulo schedule of hyperblock for given $II$ is not found
    increment $II$
/* use MVE to rename registers that span more than one $II$ (U = number of times to unroll loop) */
U = MVE(hyperblock)
create kernel given hyperblock and U
rename registers in kernel according to MVE
/* determine number of stages in prologue and epilogue */
$S = \lceil \text{latest\_issue\_time}/II \rceil - 1$
create softpipe given kernel and S
$\forall$ I $\in$ softpipe {
    if I has a branch and predicate define (bpd) operation and not loop back branch {
        schedule taken path code
        branch back to instruction following I
    }
}
generate remainder loop to execute the remainder of (loop_bound - S)/U iterations
append softpipe to remainder loop
```

Figure 6.7 Algorithm Modulo Scheduling with Profile and Predication software pipelines loops with some execution paths removed.

in the modulo schedule, the branch is converted to a pd operation and operations along both paths are predicated. For conditional branches where only one successor is in the modulo schedule, the branch is converted into a branch and predicate define (bpd) operation and only the not taken path (the path in the modulo schedule) is predicated.

Figures 6.7 and 6.8 present the algorithms used to modulo schedule profiled loops without recurrences. These algorithms are discussed at a fairly high level of abstraction. The detailed information of Modulo Scheduling is provided in Chapter 4.

Algorithm Copy Operation: Given an operation, copy and insert appropriate bookkeeping code.

```
copy op
if (op is a bpd operation) {
    copy taken path of op
    change branch destination of op to point to copy of taken path
}
```

Figure 6.8 Algorithm Copy Operation for handling off path code during scheduling.

The basic Modulo Scheduling algorithm with profiling is presented in Figure 6.7. The first step in this algorithm is to find an II that can be scheduled. Only the operations along the selected paths need to be considered. The list scheduling algorithm presented in Figure 4.12 is used to modulo schedule the hyperblock.

The only difference between the Modulo Scheduling algorithms with and without profiling is that the taken path of the branch must be copied every time the bpd operation is copied. Figure 6.8 shows the copy operation algorithm. After the software pipeline has been constructed, the taken paths of each bpd operation are scheduled taking into account the dependence and resource constraints of the pipelined schedule. Once the taken path is scheduled, a branch operation is inserted to branch to the instruction following the instruction containing the bpd operation.

During scheduling, the constraints of the excluded paths are ignored. This is possible since 1) speculative execution is not allowed, and 2) control branches back to the instruction following the bpd operation. Since speculative execution is not allowed, control dependent operations will not be moved above the bpd operation. Disallowing speculative execution will not affect the pipeline throughput, but may lengthen the pipeline latency. Since control branches back to the instruction following the bpd operation, operations moved from above to below the branch during scheduling will always be executed. Furthermore, since the paths merge at the instruction after the branch, the branch operation prevents operations from being moved above the merge point. Operations below the merge point that are control dependent on the branch will be predicated.

6.3.2 Results

To evaluate the effectiveness of using control-flow profiling, we applied Modulo Scheduling with Profile and Predication to VLIW processors with predicated hardware support. The benchmarks used in this study are 25 loops selected from the Perfect Suite [71]. These loops have no cross-iteration dependences and have at least one conditional construct. The profiling optimization is applied for branches where one path is taken at least 80% of the time. The loops are assumed to execute a large number of times and thus only the steady-state (kernel) execution is calculated for the modulo scheduled loops.

The base processor used in this study is a RISC processor which has an instruction set similar to the Intel i860 [21]. Again, Intel i860 instruction latencies are used. The processor has an unlimited number of registers and an ideal cache.

The benefit of control-flow profiling for Modulo Scheduling with predicated hardware support is illustrated by comparing the speedup with and without profiling information for a VLIW processor with issue rates 2, 4, and 8.[3] No limitation is placed on the combination of operations that can be issued in the same cycle. For each machine configuration, the loop execution time is reported as a speedup relative to the loop execution time for the base machine configuration.

[3]Induction Variable Reversal was not applied to these loops.

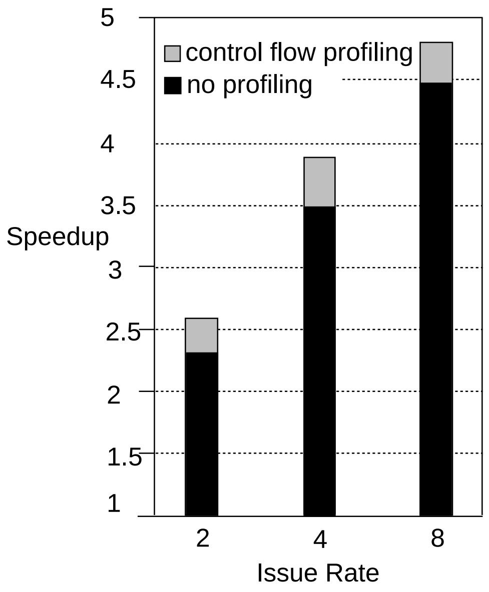


Figure 6.9 Performance improvement with profiling for Modulo Scheduling with predicated hardware support.

The benefit of using control-flow profile information to improve the performance of modulo scheduling with predicated hardware support is shown in Figure 6.9. The speedups are calculated using the harmonic mean. Overall, profiling improves the performance by approximately 12% for the issue-2 machine, 11% for the issue-4 machine, and 7% for the issue-8 machine.

It is interesting to note the effect of control-flow profiling. By eliminating paths before modulo scheduling, the scheduling constraints along excluded paths can be ignored and a tighter *II* can be found. There are two types of scheduling constraints, recurrences and resources. With predicated hardware support, reducing the resource constraints can be particularly important since all operations along all control paths are fetched, and thus, the resource constraint is determined by the most heavily used resource along *all* paths. By eliminating some of the less frequently executed control flow paths, the lower bound on *II* may be reduced. As shown in Figure 6.9, this particularly benefits the lower issue rates which incur more resource conflicts.

One final point to note is that while this technique has been presented as a way to exclude infrequently executed paths, it can also be used to exclude paths which contain software pipeline preventing code. For instance, loops with subroutine calls are often not software pipelined. If the subroutine is called only along some execution paths, then they can be excluded using predicated hardware support and the remaining paths can be efficiently scheduled.

CHAPTER 7

CONCLUSIONS

7.1 Summary

In this dissertation we have presented a new approach to scheduling in the presence of conditional branches. The Isomorphic Control Transformations (ICTs) combine the well-known If-conversion technique with a new Reverse If-conversion technique to simplify the task of global scheduling to one that looks like local scheduling. Since the control flow graph is regenerated after scheduling, this approach can be used for processors that do not have special hardware support for conditional execution.

In this dissertation we have shown that Modulo Scheduling is a practical and robust software pipelining technique based on local scheduling. To apply Modulo Scheduling to loops with conditional branches, the global scheduling problem must be reduced to a local scheduling problem. Enhanced Modulo Scheduling (EMS) is Modulo Scheduling with ICTs. The EMS technique has been implemented in IMPACT compiler. In this dissertation, we have presented the algorithms and implementation details of ICTs in general and EMS. Since ICTs do not assume special hardware support, EMS can be applied to existing processors and their future superscalar implementations. We have shown that EMS is a more flexible scheduling algorithm than Modulo Scheduling with Hierarchical Reduction, which is a technique that can also be applied to existing processors. Since EMS is more flexible, it can achieve a tighter schedule, and thus EMS performs better than Modulo Scheduling with Hierarchical Reduction.

We have also analyzed the benefit of Predicated Execution for Modulo Scheduling. In processors that support only one branch per cycle, the branch unit becomes a bottleneck as the issue rate increases. Predicated Execution eliminates this bottleneck. Furthermore, it reduces the code expansion considerably. Finally, we have shown how predicated hardware support can be used to allow control flow to branch out of and back into the software pipeline to support frequency-based scheduling or to handle loops with software pipeline inhibitors such as subroutine calls.

7.2 Future Work

7.2.1 Isomorphic Control Transformations

In this dissertation we have presented an application of ICTs to Modulo Scheduling and have shown the benefits of this approach for processors without hardware support for conditional execution. When paths are combined before scheduling, it is important that no artificial constraints are imposed. Thus, we have insured that the resource constraints do not have to be summed by using the control path information from the Predicate Hierarchy Graph (PHG). Similarly, we have to remove the sum of paths dependence constraints. When an operation is moved above a merge point, it should be scheduled independently on each path. Currently, it is scheduled as one operation and thus, it is scheduled based on the worst-case dependence along both paths.

Another extension to the ICT approach is to improve the concept of a predicate merging operation such that it retains the property that no jumps be inserted during RIC but eliminates the dependence on the original control flow graph. Currently, paths are merged only when they were merged in the original control flow graph. This may yield larger code expansion then necessary since paths may be able to be merged earlier or in a different order than from the original graph.

The use of ICTs for acyclic global scheduling has to be more fully explored, specifically, the use of ICTs with general hyperblock scheduling. To do so, there must be a way to incorporate data flow information into the predicate intermediate representation. This allows for speculative execution and other aggressive scheduling techniques. For example, it is currently not possible to move an operation from above to below a branch and schedule it on only one path of the branch. This is allowed when the result of the operation is in the live-in set of only one successor of the branch.

Finally, the ICTs can be extended to support partial predication support which is supported by superscalar implementations.

7.2.2 Modulo Scheduling

The current implementation of EMS does not support loops with recurrences. This is not a limitation of the technique but rather is due to the fact that memory dependence information is not available at the IMPACT back-end. When such information is available, loops with recurrences can be supported.

There is a limitation of the fixed-*II* techniques that variable-*II* techniques [19] do not encounter [60]. Fixed-*II* techniques create gaps in the schedule for execution paths not constrained by the minimum *II*. One solution is to use profiling information to remove the infrequently executed paths. This approach is particularly suited for processors with Predicated Execution. For processors without hardware support for conditional execution, it may be possible to use ICTs to eliminate the fixed-*II* constraint.

There are other optimizations that may improve the performance of EMS. First, the loop can be unrolled before scheduling to achieve a closer fit to the processor resource constraints (e.g., the remainder of resources requirements divided by machine constraints should be close to zero). Furthermore, heuristics to control code expansion, register pressure, register file port accesses, and cache port accesses can be employed. Modulo Scheduling is particularly suited for applying such heuristics since it is a local scheduling based technique.

Another important extension to EMS is to integrate register allocation and scheduling. This is particularly important since register spilling may violate the resultant software pipeline schedule. Techniques such as those developed by Rau et al. [36] and Huff [50] can be employed.

Finally, EMS can be extended to processors with partial predication support and to processors which support multiple branches per cycle.

7.2.3 Task granularity adjustment

Software pipelining is an aggressive loop scheduling technique for general-purpose high-performance processors. While extensive research has been performed developing techniques to vectorize loops for vector processors, similar research has to be done for software pipelining loops for general purpose processors. First, the scheduling characteristics have to be identified, and then loop transformations can be applied to adjust the loop structure accordingly. The Pcode level of the IMPACT compiler is designed to support such transformations for back-end optimization and scheduling techniques.

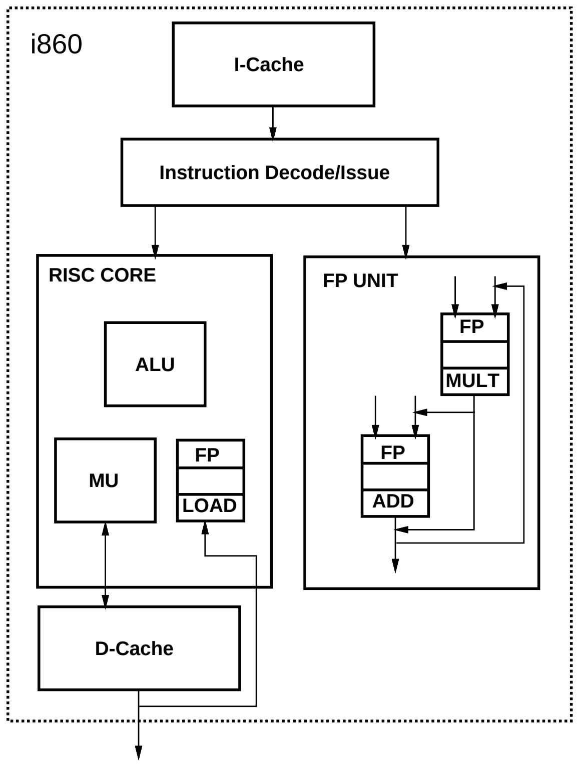


Figure .1 Intel i860 processor block diagram.

This appendix shows how modulo scheduling can be applied to a processor with a *manual pipeline*. A manual pipeline is one in which an operation advances to the next stage of the pipeline when another operation is inserted into the pipeline. The destination of the operation leaving the pipeline is specified by the operation entering the pipeline. The Intel i860 processor shown in Figure .1 has two modes for the floating-point add and floating-point multiply: non pipelined or manual pipelined. Figure .2 illustrates the difference between these two modes. In the non pipelined mode, a single precision multiply takes 3 cycles to execute and no other floating-point multiply can be executing at the same time. In the manual pipelined mode, the time to execute a pipelined instruction depends on when the other floating-point operations enter the pipeline. Since Modulo Scheduling takes into account the exact resource constraints during scheduling, it can be used to software pipeline loops utilizing the manual pipeline.

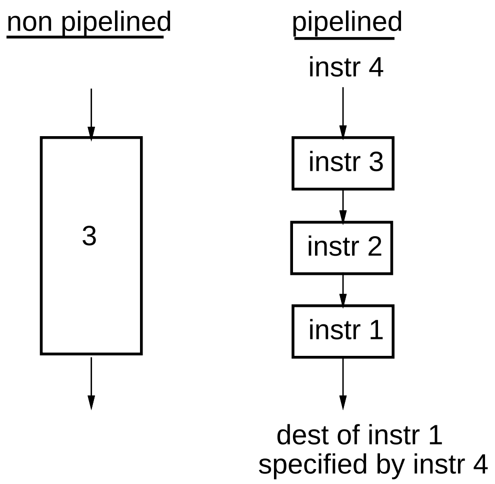


Figure .2 Manual pipeline.

Figure .3 illustrates how a loop can be software pipelined with an ordinary hardware pipeline that automatically advances. Since the pipeline automatically advances, the latency is guaranteed to be 3 and thus $ST1$ can be scheduled in cycle 6. In a manual pipeline, however, operations must be pushed through the hardware pipeline. Thus, in most schedulers, the manual pipeline causes an indeterministic latency. But, with modulo scheduling, we know that every operation in the loop body is executed every II cycles. Based on this observation, we can determine the *manual latency*. The following theorem, stated without proof, defines the manual latency.[1]

Theorem: For a loop with an iteration interval II and N_X type X instructions, where X is a manual pipeline with S_X stages, if the manual latency μ_X is used for all type X instructions within the loop, then a correct software pipeline schedule can be found, where

$$\mu = \lceil \frac{S_X}{N_X} \rceil * II$$

For the i860, X can be

- Floating-Point Multiply

[1]Note that the chaining mechanism in the i860 allows a floating-point multiply (add) to push a floating-point add (multiply) through the pipeline.

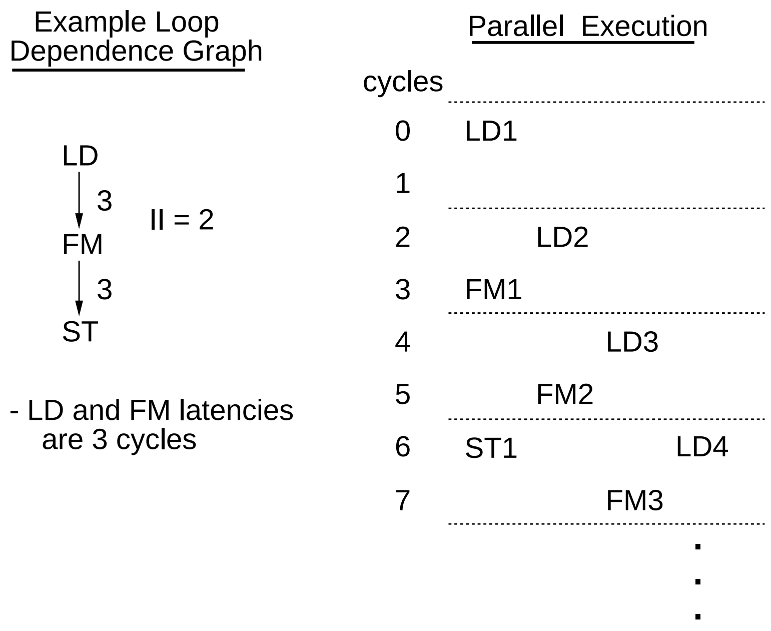


Figure .3 Software pipelining regular pipelines.

- Single Precision - $S_X = 3$
- Double Precision - $S_X = 2$

• Floating-Point Add

- Single/Double Precision = $S_X = 3$

To modulo schedule a loop, the *manual latency* is calculated and the dependence graph updated. Figure .4 shows the software pipelined loop for a manual pipeline. Note that by simply changing the operation latency, the operation $ST1$ is not scheduled until the result of $FM1$ has been pushed through the manual pipeline.

Preliminary results for the i860 show that for 10 frequently executed loops, Modulo Scheduling can achieve a speedup of 1.9 using the non pipelined mode and 2.8 using the pipelined mode.

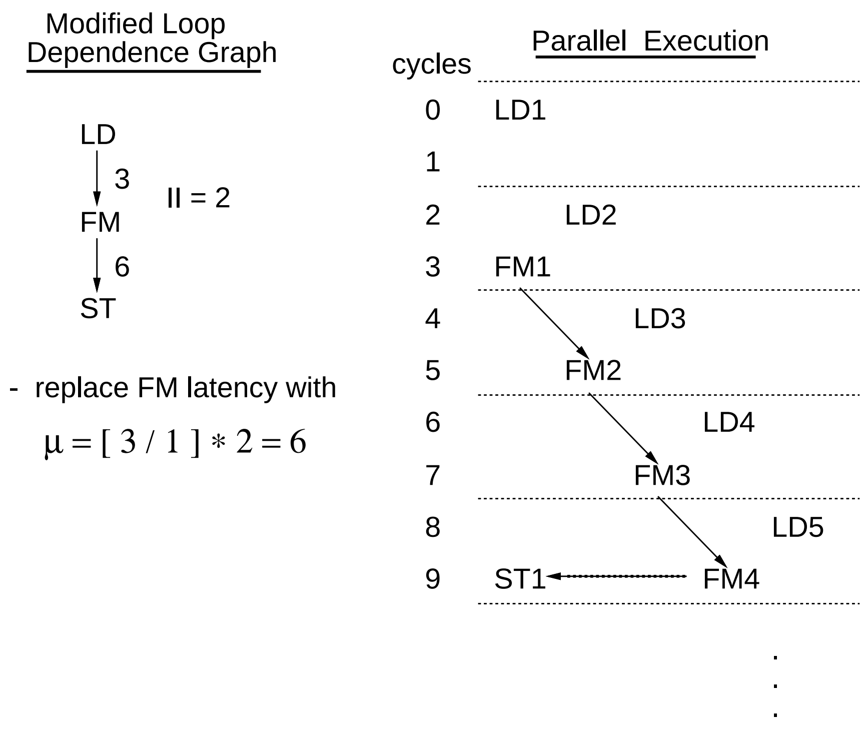


Figure .4 Software pipelining manual pipelines.

REFERENCES

[1] 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.

[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] R. Gupta and M. L. Soffa, "Region scheduling: An approach for detecting and redistributing parallelism," *IEEE Transactions on Software Engineering*, vol. 16, pp. 421–431, April 1990.

[4] D. Bernstein and M. Rodeh, "Global instruction scheduling for superscalar machines," in *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pp. 241–255, June 1991.

[5] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Transactions on Computers*, vol. c-30, pp. 478–490, July 1981.

[6] J. Ellis, *Bulldog: A Compiler for VLIW Architectures.* Cambridge, MA: The MIT Press, 1985.

[7] 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," *Journal of Supercomputing*, January 1993.

[8] R. Towle, "Control and data dependence for program transformations." Ph.D. dissertation, Department of Computer Science, University of Illinois, Urbana, IL, 1976.

[9] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, pp. 177–189, January 1983.

[10] J. C. H. Park and M. Schlansker, "On Predicated Execution," Tech. Rep. HPL-91-58, Hewlett Packard Software Systems Laboratory, May 1991.

[11] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle, "The Cydra 5 departmental supercomputer," *IEEE Computer*, pp. 12–35, January 1989.

[12] 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.

[13] G. Wood, "Global optimization of microprograms through modular control constructs," in *Proceedings of the 12th Annual Workshop on Microprogramming*, pp. 1–6, 1979.

[14] M. Lam, "A systolic array optimizing compiler." Ph.D. dissertation, Carnegie Mellon University, Pittsburg, PA, 1987.

[15] 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.

[16] 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.

[17] 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.

[18] B. Su and J. Wang, "GURPR*: A new global software pipelining algorithm," in *Proceedings of the 24th Annual Workshop on Microprogramming and Microarchitecture*, pp. 212–216, November 1991.

[19] K. Ebcioğlu and T. Nakatani, "A new compilation technique for parallelizing loops with unpredictable branches on a VLIW architecture," in *Second Workshop on Languages and Compilers for Parallel Computing*, August 1989.

[20] D. C. Lin, "Compiler support for predicated execution in superscalar processors." M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1992.

[21] Intel, *i860 64-Bit Microprocessor*. Santa Clara, CA, 1989.

[22] N. J. Warter, S. A. Mahlke, W. W. Hwu, and B. R. Rau, "Reverse if-conversion," in *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, to appear June 1993.

[23] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the hyperblock," in *Proceedings of 25th Annual International Symposium on Microarchitecture*, December 1992.

[24] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, pp. 310–349, July 1987.

[25] R. Cytron, J. Ferrante, and V. Sarkar, "Experience using control dependence in PTRAN," in *Proceedings of the Second Workshop on Languages and Compilers for Parallel Computing*, August 1989.

[26] W. Baxter and I. H. R. Bauer, "The program dependence graph and vectorization," in *Conference Record of the Sixteenth ACM Symposium on the Principles of Programming Languages*, pp. 1–10, January 1989.

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

[28] T. Lengauer and R. E. Tarjan, "A fast algorithm for finding dominators in a flowgraph," in *ACM Transactions on Programming Languages and Systems*, pp. 121–141, July 1979.

[29] A. Aiken and A. Nicolau, "A development environment for horizontal microcode," *IEEE Transactions on Software Engineering*, vol. 14, pp. 584–594, May 1988.

[30] K. Ebcioğlu and A. Nicolau, "A global resource-constrained parallelization technique," in *Proceedings of the International Conference on Supercomputing*, pp. 154–163, June 1989.

[31] K. E. S. Moon, "An efficient resource-constrained global scheduling technique for superscalar and VLIW processors," in *Proceedings of the 25th International Workshop on Microprogramming and Microarchitecture*, pp. 55–71, December 1992.

[32] P. Chang, "Compiler support for multiple instruction issue architectures." Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1991.

[33] R. L. Lee, A. Kwok, and F. Briggs, "The floating point performance of a superscalar SPARC processor," in *Proceedings of the 4th International Conference on Architecture Support for Programming Languages and Operating Systems*, pp. 28–37, April 1989.

[34] K. Ebcioğlu, "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.

[35] R. M. Lee, "Advanced software pipelining on a program dependence graph." M.S. thesis, Department of Computer Science, Utah State University, Logan, UT, 1992.

[36] B. R. Rau, M. Lee, P. P. Tirumalai, and M. S. Schlansker, "Register allocation for software pipelined loops," in *Proceedings of the ACM SIGPLAN 92 Conference on Programming Language Design and Implementation*, pp. 283–299, June 1992.

[37] 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.

[38] P. Kogge, "The microprogramming of pipelined processors," in *Proceedings of the 4th Annual Symposuim on Computer Architecture*, pp. 63–69, 1977.

[39] D. A. Padua, "Multiprocessors: Discussion of some theoretical and practical problems." Ph.D. dissertation, Department of Computer Science, University of Illinois, Urbana, IL, 1979.

[40] J. R. B. Davies, "Parallel loop constructs for multiprocessors," Tech. Rep. UIUDCS-R-81-1070, University of Illinois at Urbana-Champaign, Illinois, May 1981.

[41] R. Cytron, "Doacross: Beyond vectorization for multiprocessors," in *Proceedings of the 1985 International Conference on Parallel Processing*, pp. 836–844, August 1986.

[42] S. Jain, "Circular scheduling: A new technique to perform software pipelining," in *Proceedings of the ACM SIGPLAN 91 Conference on Programming Language Design and Implementation*, pp. 219–228, June 1991.

[43] G. R. Gao, H. H. J. Hum, and Y. Wong, "Towards efficient fine-grain software pipelininig," in *Proceedings of the 1990 International Conference on Supercomputing*, pp. 369–379, July 1990.

[44] J. Hoogerbrugge, H. Corporaal, and H. Mulder, "Software pipelining for transport-triggered architectures," in *Proceedings of the 24th International Workshop on Microprogramming and Microarchitecture*, pp. 74–81, November 1991.

[45] A. Charlesworth, "An approach to scientific array processing: The architectural design of the AP-120B/FPS-164 family," in *IEEE Computer*, September 1981.

[46] R. Touzeau, "A Fortran compiler for the FPS-164 Scientific Computer," in *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, June 1984.

[47] P. Y. T. Hsu and E. S. Davidson, "Highly concurrent scalar processing," in *Proceedings of the 13th International Symposium on Computer Architecture*, pp. 386–395, June 1986.

[48] C. Eisenbeis, "Optimization of horizontal microcode generation for loop structures," in *International Conference on Supercomputing*, pp. 453–465, July 1988.

[49] J. H. Patel and E. S. Davidson, "Improving the throughput of a pipeline by insertion of delays," in *Proceedings of the 3rd International Symposium on Computer Architecture*, pp. 159–164, 1976.

[50] R. A. Huff, "Lifetime sensitive modulo scheduling," in *Proceedings of the ACM SIGPLAN 93 Conference on Programming Language Design and Implementation*, pp. 258–267, June 1993.

[51] P. Y. T. Hsu, "Highly concurrent scalar processing." Ph.D. dissertation, Department of Computer Science, University of Illinois, Urbana, IL, 1986.

[52] P. Tirumalai, M. Lee, and M. Schlansker, "Parallelization of loops with exits on pipelined architectures," in *Supercomputing*, November 1990.

[53] J. C. Dehnert and R. A. Towle, "Compiling for the Cydra 5," *Journal of Supercomputing*, January 1993.

[54] F. Bodin and F. Charot, "Loop optimization for horizontal microcoded machines," in *Proceedings of the 1990 International Conference on Supercomputing*, pp. 164–176, July 1990.

[55] 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.

[56] A. Nicolau, "Uniform parallelism exploitation in ordinary programs," in *Proceedings of the 1985 International Conference on Parallel Processing*, pp. 614–618, August 1985.

[57] V. H. Allan, J. Janardhan, R. M. Lee, and M. Srinivas, "Enhanced region scheduling on a program dependence graph," in *Proceedings of the 25th International Workshop on Microprogramming and Microarchitecture*, pp. 72–80, December 1992.

[58] J. W. Bockhaus, "An implementation of GURPR*: A software pipelining algorithm." M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1992.

[59] D. A. Padua and M. J. Wolfe, "Advanced compiler optimizations for supercomputers," *Communications of the ACM*, vol. 29, pp. 1184–1201, December 1986.

[60] F. Gasperoni, "Compilation techniques for VLIW architectures," Tech. Rep. 66741, IBM Research Division, T.J. Watson Research Center, Yorktown Heights, NY 10598, August 1989.

[61] N. J. Warter, J. W. Bockhaus, G. E. Haab, and K. Subramanian, "Enhanced Modulo Scheduling for loops with conditional branches," in *Proceedings of the 25th International Symposium on Microarchitecture*, pp. 170–179, November 1992.

[62] 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.

[63] B. R. Rau, C. D. Glaeser, and R. L. Picard, "Efficient code generation for horizontal architectures: Compiler techniques and architectural support," in *Proceedings of the 9th Annual International Symposium on Computer Architecture*, pp. 131–139, May 1982.

[64] N. J. Warter, D. M. Lavery, and W. W. Hwu, "The benefit of Predicated Execution for software pipelining," in *Proceedings of the 26th Hawaii International Conference on System Sciences*, vol. 1, pp. 497–506, January 1993.

[65] N. J. Warter, G. E. Haab, and W. W. Hwu, "Pcode manual," Tech. Rep. Internal IMPACT Report, Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, November 1992.

[66] K. Subramanian, "Loop transformations for parallel compilers." M.S. thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1993.

[67] R. A. Bringmann, "A template for code generation development using the impact-i c compiler." M.S. thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1992.

[68] R. E. Hank, "Machine independent register allocation for the impact-i c compiler." M.S. thesis, Department of Electrical Engineering, University of Illinois, Urbana, IL, 1993.

[69] W. Y. Chen, "An optimizing compiler code generator: A platform for risc performance analysis." M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1991.

[70] G. Kane, *MIPS R2000 RISC Architecture*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1987.

[71] M. Berry et al., "The PERFECT club benchmarks: Effective performance evaluation of supercomputers," Tech. Rep. CSRD-827, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, May 1989.

[72] L. J. Hendren et al., "Register allocation using cyclic interval graphs: A new approach to an old problem," Tech. Rep. ACAPS Technical Memo 33, Advanced Computer Architecture and Program Structures Group, McGill University, Montreal, Canada, 1992.

VITA

Nancy Jeanne Warter was born on April 13, 1963, in Princeton, New Jersey. She attended high school in Newark, Delaware. In the fall of 1981, she began her undergraduate studies in the Electrical Engineering Department at Cornell University in Ithaca, New York. While at Cornell, she was an engineering cooperative student at the Thomas J. Watson IBM Research Center Yorktown Heights, New York. She received her Bachelor of Science degree in Electrical Engineering in 1985. In the fall of 1985 she began her graduate studies in the Electrical Engineering Department at the University of Illinois in Urbana, Illinois. From 1985 to 1986 she was a Teaching Assistant in the Electrical Engineering Department. In the summer of 1986, she joined the Center for Supercomputing Research and Development as a Research Assistant. In May of 1989, she received her Master of Science degree in Electrical Engineering. In January of 1989, she joined the Center for Reliable and High-Performance Computing as a member of the IMPACT project directed by Professor Wen-mei W. Hwu. After completing her Ph.D., she will join the Department of Electrical Engineering at the California State University at Los Angeles in Los Angeles, California. Ms. Warter is a member of the ACM and the IEEE Computer Society.