INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6” x 9” black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
RICE UNIVERSITY

Improved Software Pipelining for Superscalar Architectures

by

Edmar Wienskoski Jr.

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy

APPROVED, THESIS COMMITTEE:

Keith D. Cooper, Associate Professor.
Chair
Department of Computer Science
Rice University

Ken Kennedy, Noah Harding Professor
Department of Computer Science
Rice University

John K. Bennett, Associate Professor
Department of Electrical and Computer Engineering
Rice University

Houston, Texas
April. 1998
Improved Software Pipelining for Superscalar Architectures

Edmar Wienskoski Jr.

Abstract

Although instruction scheduling is an NP-complete problem [27], many techniques have been developed to improve pipelining efficiency. Among them, several were proposed for VLIW machines, and were shown to be efficient and extendible to superscalar architectures. However the available resources on a superscalar can vary significantly. Our goal for this thesis is to improve the effectiveness of software pipelining scheduling for modern superscalar architectures.

I. We explore ways to improve compile time performance by producing more accurate lower bounds for $H_{min}$. Our new scheme accounts for register use and lets the scheduler provide guidance to the allocator. The scheduling process of the loops that benefited from this technique was 18% to 30% faster.

II. We explore new techniques for improving the schedules of loops that contain complex control flow. Our proposed techniques do not require the addition of specialized hardware. The initiation intervals for those loops were reduced by 25% to 36%.
Acknowledgments

I would like to thank all people that somehow got involved in my graduate studies. The contributions I received ranged from material to emotional support, from technical to moral guidance, from professional to entertaining assistance, and through many other forms that this space would not be enough to list them all.

In special, I would like to thank my wife, Fernanda, and my parents, Mercedes and Edmar, for the unconditional dedication, love and care throughout these years.

The members of the committee, Keith Cooper, Ken Kennedy, and John Bennett for the inspiration, encouragement and confidence during the development of this thesis.

The Brazilian science agency, CAPES, for providing my support in the first four years of my graduate studies, and DARPA through contract DABT63-95-C-0115 for the other three years of funding.

All the members of the Massively Scalar Compiler Project at Rice University, for the great team work environment they provide. In particular, Tim Harvey, for his patience during our many discussion sessions.
Contents

Abstract  ii
Acknowledgments  iii
List of Illustrations  vi

1 Introduction 1
1.1 Instruction Scheduling  2
  1.1.1 Instruction Scheduling Profitability  3
  1.1.2 Instruction Scheduling and Register Allocation  3
  1.1.3 Instruction Scheduling Techniques  5
1.2 Software Pipelining Overview  7
  1.2.1 Initiation Interval  8
  1.2.2 Scheduling Recurrences  9
  1.2.3 Scheduling If-then-else Constructs  9
1.3 Achievement in this Thesis  9
1.4 Organization of the Thesis  10

2 Related Work  11
2.1 Minimal Initiation Interval Estimation  11
2.2 Modulo Scheduling  12
2.3 Scheduler and Register Allocator  13
2.4 Handling Control Flow  15

3 Register Constraints  18
3.1 Register Cycle Definition  19
3.2 Computing Register-cycles  22
3.3 Recurrences in the Dependence Graph  25
3.4 Fan-out Constraints  26
3.5 Fan-in Constraints  27
3.6 Accounting for Registers as a Resource Constraint  28
4 Unwinding the Control Flow
   4.1 Overview .................................................. 33
   4.2 Unwinding the Control Flow Graph ....................... 34
   4.3 Software Pipelining Loops with Early Exits .............. 38
      4.3.1 The Hierarchical Reduction versus Unwinding CFG Trade-off. 43
   4.4 Limiting the Application of the Unwinding Transformation by Code Size ............................................. 45

5 Experimental Results ............................................. 46
   5.1 Obtaining Results .......................................... 46
   5.2 Compilation Process Improvements ......................... 47
      5.2.1 Optimization Opportunities for Registers Constraint ($H_{Lam}$ vs $H_{RC}$) ........................................ 47
      5.2.2 Compilation Time for Registers Constraints ($\sum H_{Lam}$ vs $\sum H_{RC}$) ............................................. 49
      5.2.3 Compilation Time for Unwinding CFG ($H_{Lam}$ vs $H_{ucfg}$) ................................................ 50
   5.3 Running Time Improvements ................................ 53
      5.3.1 Optimization Opportunities and Profitability for UCFG ................................................... 53
      5.3.2 Overall Execution Speed Up ............................ 55
      5.3.3 Speed Up for Loops with Control Flow ................. 58

6 Conclusions ..................................................... 63
   6.1 Suggestions for Future Work ................................ 65
      6.1.1 Estimating the $H_{min}$ under Scheduler and Allocator Interaction ............................................. 65
      6.1.2 Hypernode Reduction ................................... 66

Bibliography ...................................................... 67
Illustrations

1.1 Compiler internal structure ........................................ 1

3.1 An example ............................................................. 22
3.2 Computing lifetimes ..................................................... 23
3.3 Adjusting for fan-outs .................................................. 26
3.4 Fan-in constraints ......................................................... 27
3.5 Adjusting for fan-ins ...................................................... 29

4.1 Flattening the control flow graph ..................................... 35
4.2 Algorithm to flatten the CFG. ........................................ 36
4.3 Graphical illustration of operation split_joint on edge $x \rightarrow y$ 36
4.4 Separating the loops ...................................................... 37
4.5 Algorithm to separate the loops. ...................................... 37
4.6 Partially dead and partially useless instruction optimization ... 39
4.7 General structure for early exit loops ............................... 40
4.8 Software pipelining loops with early exits ......................... 41
4.9 Example of early exit loop ............................................. 41
4.10 Algorithm to compute the $p$ subsets ............................... 42
4.11 Example loop after software pipelining ............................. 44

5.1 Executing benchmarks in our research compiler ..................... 47
5.2 $H_{Lam}$ compared to $H_{RC}$ for inner loops of test programs 48
5.3 $H_{Lam}$ compared to $H_{RC}$ after unwinding the control flow ... 49
5.4 Number of MRT tables scheduled (Overall compilation) ........ 51
5.5 Number of operations scheduled (Overall compilation) .......... 51
5.6 Number of MRT tables scheduled (Only participants loops) .... 52
5.7 Number of operations scheduled (Only participants loops) ....... 52
5.8 Number of MRT tables scheduled (Overall compilation) ........ 52
5.9 Number of operations scheduled (Overall compilation) ........................................ 53
5.10 Operations executed for loops with control flow unwind optimization .......... 54
5.11 Path locality for loops with control flow .............................................................. 56
5.12 Results for the CPU model MPC601 ................................................................. 56
5.13 Results for the CPU model MPC604 ................................................................. 57
5.14 Results for the CPU model MPC604 with 2 float point units ......................... 57
5.15 Results for the CPU model MPC604 ................................................................. 59
5.16 II improvement for NAS, Fmm, Linpack and Livermore benchmarks .......... 60
5.17 II improvement for Spec benchmark ................................................................. 61
5.18 Summary of figures 5.16 and 5.17 ................................................................. 62
Chapter 1

Introduction

Computers are easier to use when we program them with high level programming languages. Such languages are designed to simplify coding an algorithm. Unfortunately, the translation from high level languages to machine code often introduces inefficiencies that a programmer who knew about all the machine's specific features could avoid. More typically, the compiler is given the responsibility to generate efficient machine code.

The motivation for doing this is the cost of such specialized programmers, and the extra time required to code the algorithm. On the other hand, once the compiler "learns" how to generate efficient code, it will do so for all applications. Also, the combined performance of the compiler and architecture is often a factor considered when buying a computer.

The compiler writer attacks this issue of generating efficient machine code by designing the compiler in several phases. Figure 1.1 depicts a typical choice. The first phase is the translation itself, the others analyze or transform the resulting code in order to remove inefficiencies.

Many analyses and transformations have been designed. Some of them are machine independent. By machine independence, we meant that the analysis or transformation is computed and performed in the same way whatever target architecture we have. Whether or not the transformation is profitable is arguably machine dependent (see [17] §2.1).

Most of the machine independent analysis and transformations can be precisely described in terms of equations and or theorems, and once an efficient algorithm for

![Diagram of compiler internal structure](image)

Figure 1.1 Compiler internal structure
such a problem is found, the problem becomes "solved" or "understood", especially in the case when the given algorithm is proven to be optimal.

On the other hand, machine dependent optimizations rarely have a simple mathematical structure. Many of them, like instruction scheduling, can be proven to be NP-complete. Such problems are approached in a different way: either the compiler will try to guess a solution, or will successively approximate one. If we modify the architecture, the factors that make a given guess good may change, or new factors must be considered, or even the factor in question may no longer be a concern. As architectures evolve, many compilation problems must be reviewed to catch up with the advances in architecture.

With recent introduction of multifunctional asynchronous RISC architectures, the instruction scheduling problem that seemed to be "understood" for VLIW architectures, takes on new dimensions.

1.1 Instruction Scheduling

Scheduling is an optimization that reorders the instructions of a program in an attempt to improve some aspect of its performance. Scheduling relies on detailed knowledge of the target machine's architecture and its implementation. The scheduler uses this knowledge to improve performance by taking advantage of specific machine features.

The notion of "improving performance" is vague. The specific goals of scheduling depend heavily on the target machine.

- On a simple scalar machine, like the VAX machines, the scheduler might reorder instructions to shorten the distance between definitions and uses. This can decrease the lifetimes of values and, by reducing demand for registers, decrease the amount of register spill code that must be introduced.

- On a pipelined architecture, the opposite goal may be desirable. To achieve high performance on such a machine, the compiler must keep the pipelines full. If definitions are too close to uses, the pipelines will stall while waiting for operands.\(^1\) Thus, the scheduler should try to separate definitions from uses.

\(^1\)The "stall" may be a hardware interlock, or it may take the form of compiler-inserted nops.
1.1.1 Instruction Scheduling Profitability

The importance of scheduling seems to increase with each new generation of processors.

Pipelined architectures are designed to exploit instruction level parallelism (ILP). ILP can only be achieved if, within a sequence of operations, there is no data dependence between these operations. Thus, the hardware can only deliver its peak performance if the scheduler finds such ILP and properly reorders the instructions to enable them to be executed in parallel.

Recent architectural developments increase the importance of instruction scheduling. The trend is to increase the number of functional units in the architecture. The CIDRA-5 has a total of seven functional units. The following table summarizes the evolution of the PowerPC and Alpha architectures.

<table>
<thead>
<tr>
<th>CPU</th>
<th>year</th>
<th>float point</th>
<th>integer</th>
<th>address arithmetic</th>
<th>memory</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPC601</td>
<td>93</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>MPC604</td>
<td>94</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>ALPHA21064</td>
<td>95</td>
<td>1</td>
<td>2</td>
<td>-</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>ALPHA21264</td>
<td>97</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>-</td>
<td>6</td>
</tr>
</tbody>
</table>

RISC design allows instructions to be executed at a very high rate on deeper pipelines, increasing both relative memory latency and functional unit latency. The MPC604 has memory latency of 3 cycles and branch missprediction of 3 cycles; the CIDRA-5 has memory latency of 17 cycles and branch missprediction of 3 cycles. The DEC ALPHA has branch latency of 7 cycles.

Each of these facts demands that the scheduler find more operations that can be executed in parallel.

1.1.2 Instruction Scheduling and Register Allocation

The first attempt to satisfy all those requirements is to adopt an aggressive approach to scheduling — find all possible parallelism in a program and use it all. The problem with this lies in the interaction of the scheduler with the register allocator.

As more operations are executed in parallel, the register requirement for that program will increase. There will be a point where the allocator will have to spill some registers in order to satisfy the demand. Spill code leads to a longer schedule, increasing the initiation interval, which may defeat the purpose of aggressively scheduling operations in parallel. Thus, the ultimate scheduler goal is to parallelize up
to the point of register availability, and not to the point determined by the program structure.

This observation raises a question: Where in the compiler should we locate the scheduler?

If the allocator comes first, the allocator may reuse the same register names too soon, preventing the scheduler from moving instructions without violating the data dependences. In that case the allocator must have some scheduling insight and choose register names wisely to avoid future conflicts. This approach is feasible within local scheduling. When dealing with a global scheduler, such as software pipelining, it would be more difficult to avoid constraining the scheduler.

On the other way around, the scheduler comes before the allocator. Here the problem is that the best scheduler algorithms are greedy algorithms that will try to find the best schedule assuming an unlimited number of registers. Later the allocator will have to spill registers, adding new instructions to the schedule. In this case the scheduler must have allocation insights and place operations wisely to avoid unnecessarily increasing demand for registers.

Work on combining scheduling and register allocation has largely focused on under-utilizing registers to avoid scheduler-introduced spills. Pinter [40] proposes building a dependence graph and then complementing the graph. This new graph indicates all possible operations that can be executed in parallel, and it is added to the interference graph in a such way that the allocator will try to use different register names for operations that can be executed in parallel. This approach trades off possible spills for execution slots, and relies on construction of quite large graphs.

In more recent papers [21, 25], the register allocator tries to get a better allocation for an already fixed set of scheduled instructions. This approach does not allow the scheduler to give up execution slots; the whole burden is on the register allocator.

Bradlee et al [8] attempted to integrate the scheduler and register allocator in one single phase. They implemented three opposing approaches in an attempt to evaluate the need for scheduler and register allocator integration. They used a basic block list scheduler in all three experiments, and they found that a full integration would not get better results than a partial integration approach based on passing parameters from one to the other.

Another factor to consider is the way that intermediate code treats values and registers.

Regarding register assignment, there are two major ways to do it; variables residing in memory, and variables residing in registers. In the first one the intermediate code
is generated with pairs of loads and stores for all variables uses. In this scheme, later analysis and transformations will get rid of many memory operations, effectively leaving the data residing in registers. Chow and Hennessy [16] describe a register allocator that works this way. The advantage of this scheme is that the intermediate code can be easily converted to assembler code, leaving code optimization and register allocation as an option. The second is a more aggressive approach. It assumes an unlimited set of registers, and the intermediate code is generated by associating every variable name with a register name. This representation requires some careful implementation in order to avoid using too many register names and slowing down the analysis and transformations. In this scheme a register allocator is mandatory, Chaitin [14] and Briggs [9] describe a scheme for register allocation by graph coloring that outperforms Chow’s one.

For instruction scheduling, the virtual register scheme of intermediate representation makes easier to do scheduling before register allocation. Since each computation has its own name, the scheduler has all freedom possible to move operations that can be computed in parallel. In the memory-based scheme, the scheduler must analyze memory addresses to understand when a dependence exists between two instructions.

For these reasons, and because our research compiler assumes that the variables reside in registers, we decided to work with the scheme shown in figure 1.1 — that is scheduling before allocation.

1.1.3 Instruction Scheduling Techniques

Although instruction scheduling is an NP-complete problem [27], many techniques have been developed to approximate its solution. Several of them were proposed for VLIW machines, and were shown to be efficient and extendible to superscalar architectures. However the available resources on the superscalar architecture can vary significantly. For example, the Cidra 5 [6] had one register set for each functional unit, while superscalar processors like the RS6000 [39] share a single register set among multiple functional units of same type. Our goal for this thesis is to improve the effectiveness of instruction scheduling for modern superscalar architectures.

As a starting point, we must select a basic approach on which we can build. Several different approaches have been proposed. These include list scheduling, trace scheduling, direct placement and software pipelining.

List scheduling [23, 27] is a greedy algorithm driven by heuristics. It is applied over basic blocks and can achieve good performance relative to optimal basic block scheduling. However, relative to the potential of the hardware pipelines, it is disap-
pointing most of the time. The longer the pipeline, the more parallelism is needed to fill in the pipeline slots, and the average basic block doesn't contain enough independent computation to supply the pipeline. Several variations have been proposed that extend the basic blocks to improve performance [30, 43, 7], and that minimize the penalty across basic block boundaries [24, 1].

Trace scheduling [22] tries to automate ad-hoc techniques developed for VLIW machines. The process begins with an execution trace. The trace is used to identify the most frequently executed path, which is scheduled first. Next, other paths are added in decreasing frequency order, moving instructions and duplicating code as necessary. Trace scheduling is suitable for programs executed many times with a small data set variation. It was originally intended to optimize the whole program. Global scheduling techniques, such as this, attempt to minimize the total execution time of all the basic blocks. Thus, effective trace scheduling might actually increase the size of the program by greatly lengthening an infrequently executed basic block in order to slightly reduce the length of a high frequently executed basic block.

Direct placement [26] uses the data dependence graph to compute directly the earliest possible execution time for a particular instruction. This scheme was designed to be coupled directly to the code generator. As instructions are generated, their final positions are computed from the final positions of previously issued instructions. The problem with this technique is that when a computed slot is already occupied, the instruction is placed in the next available one, regardless of which dependence chain is longer (i.e., instructions cannot be assigned priorities, since instructions are scheduled as they come out of the code generator). Direct placement is also based on basic blocks. Its main advantage over list scheduling is its simplicity and almost linear speed.

Software pipelining [32, 42] was specifically designed for loops. Its main goal is to achieve high pipeline performance across the iterations of a single loop nest. It treats the sequence of iterations to be executed as one long basic block. It takes advantage of this view to create a schedule where several iterations are in progress at one time. This lets it cover pipeline latencies in one iteration with independent operations from another. It focuses on the inter-iteration aspects of scheduling, relying on another discipline, like list scheduling, to handle local placement of instructions.

Software pipelining is the approach that appears to make the biggest impact in CPU performance. First, most processor time is spent executing loops. Second, it effectively extends the basic blocks (not statically but dynamically), improving the
effectiveness of the list scheduler. Also, code growth is limited to a prologue and an epilogue for each loop.

1.2 Software Pipelining Overview

The general formulation of the software pipelining process for single basic block loops was given by Rau [42]. He also presented a condition that has to be met by any legal software pipelined schedule, namely the \textit{modulo constraint}. and derived lower bounds on the rate that successive iterations of the loop can be started, namely the \textit{initiation interval}.

Rau [41] gives us a taxonomy for software pipelining algorithms today:

- \textit{move-then-schedule} - This approach moves instructions, one by one, across the back edge of the loop. The instruction can be rotated either in forward or in backward direction. Ebcioğlu's pipeline-percolation scheduling [20], Jain's circular scheduling [31], and Allan's enhanced region scheduling [4] are examples of such schedulers.

- \textit{schedule-then-move} - This approach focuses on directly creating a schedule that maximizes performance, and later doing the necessary code motion determined by the schedule. There are two ways to achieve this:

  - \textit{unroll-while-scheduling} simultaneously unrolls and schedules the loop until a repetition pattern is found. At this point we have the kernel of the software pipelined loop. Aiken's perfect pipelining [3] is one such technique.

  - \textit{modulo scheduling} builds a schedule for one iteration of the loop. It must be done in such a way that repeating the schedule at regular intervals (the \textit{initiation interval}) neither violates any dependences nor introduces any resource conflicts. Research in modulo schedule has being done by Lam [32], Tirumalai [45], Huff [29] among others.

Compared to unroll-while-scheduling, the code expansion of modulo scheduling is limited to prologue and epilogue generation. On the other hand, there is no simple way to compute the initiation interval for modulo scheduling — if the scheduler fails to build a modulo reservation table for a given $H$, it has to start over again with a new value for $H$.

Monica Lam [32] introduced an aggressive approach for software pipelining arbitrary loops. Put simply, her algorithm computes both upper and lower bounds on
the schedule length and tries to construct a schedule of shortest length. If the construction fails, it increases schedule length by one and tries again. The construction must succeed by the time schedule length reaches the upper bound.

1.2.1 Initiation Interval

One key to an efficient implementation of her technique is the accuracy of the lower bound. Finding an upper bound is relatively easy: the instructions can be scheduled (naively) in the sum of the number of instructions plus their delays. Tightening the upper bound will not speed up the scheduler: the upper bound simply proves that the scheduler will terminate. Finding a lower bound is harder. Lam suggests using two constraints. For each recurrence in the dependence graph, compute the total delay around the recurrence divided by the number of iterations that it spans. The schedule for a loop must contain at least that many cycles to execute that recurrence. For each type of functional unit, compute the number of instructions of that type in the loop divided by the number of issue slots of that type per cycle. The schedule for a loop must contain at least that many cycles to issue each instruction. Lam's lower bound is the larger of these two lower bounds.

The scheduler proceeds by setting the loop length, or initiation interval, to the lower bound. Next, it tries to construct a schedule of that length. It may fail for any one of several reasons. These include

1. Data dependences may prevent the scheduler from making full use of the functional units: in this case, the functional units will be idle in some cycles. Since the lower bound assumes ideal utilization, this can force a longer initiation interval. This case can arise from an interaction between the components of the estimated lower bound.

2. The functional units may be well utilized, but in a way that generates excessive demand for registers. If insufficient registers are available, spill code must be introduced. Each spill instruction requires another slot on a functional unit: this can increase the lower bound for issue slots. This case arises from a kind of resource constraint not represented in the lower bound estimate – the demand for registers.
1.2.2 Scheduling Recurrences

The software pipelining algorithm uses list scheduling to place operations in the modulo reservation table. When the dependence graph of the operations in the loop has recurrences, the list scheduler will not find a root in the graph to start with. Lam's proposed solution is to build a strongly connected component graph from the dependence graph, where each node in the SCC graph summarizes a set of operations that are scheduled independently by removing the edges that complete the recurrence. The resulting SCC graph is acyclic and can be used by the list scheduler.

1.2.3 Scheduling If-then-else Constructs

Loops with arbitrary control flow in the loop body are dealt with by performing IF-conversion \([5]\) to replace all branching by predicated operations. This transformation yields a loop with a single basic block that is then submitted to the modulo scheduling algorithm. The disadvantages of predicated modulo scheduling are that it requires special hardware, and that the \(II_{\text{min}}\) must be computed as if all the operations in the body of the loop are executed in every iteration, although in reality, only those along one of the control flow are actually executed.

Lam introduced hierarchical reduction. In her scheme, each if-then-else construct is reduced to a single node representing all the constraints of its components. After reducing the construct, its single node becomes part of an outer basic block. This process is repeated until the loop body is reduced to a single basic block, then, software pipelining is applied. This technique can only be applied to loop bodies with simple, well structured control flow graphs.

1.3 Achievement in this Thesis

The primary weaknesses of software pipelining are its poor compile time efficiency and its inability to handle well loops that contain control flow.

The problem with compile time arises from the iterative nature of the technique. The speed of the software pipelining process is directly proportional to the distance between \(II_{\text{min}}\) and the \(II\) that actually succeeds. Our new register constraint will lead to a tighter \(II_{\text{min}}\) estimation and, so, to a faster compilation process. The idea of registers as a resource constraint is motivated by the fact that register pressure rises with the degree of pipelining. As \(II\) decreases, parallelism is increased, and thus register pressure. Moreover, as we satisfy data dependences by interleaving
one iteration between definition and use, we stretch the register lifetimes, increasing register pressure. Given a register pressure estimate, we come up with a limit on the II below which it can be proven that no register allocation can succeed.

Control flow within the loop body has been a weakness of software pipelining. We will explore a series of ideas for exploiting path locality within the loop to improve performance inside loops that contain complex control flow. The basic idea is simple: duplicate code to improve the schedules of all paths through the loop, to the detriment of path changes on successive iterations. After the transformation each execution path in the original loop becomes an individual loop with early exits, but no internal control flow. These resulting loops are then scheduled independently.

1.4 Organization of the Thesis

In chapter 2 we provide more details of Lam’s software pipelining method and compare it to more recent work. Chapter 3 presents our new method for estimating the initiation intervals. Chapter 4 deals with improvements to loops with control flow. Finally, chapter 5 shows the experimental results we obtained from compilation and simulation. Chapter 6 gives a summary of the results and suggestions for future work.
Chapter 2

Related Work

Monica Lam pioneered software pipelining for VLIW machines [32]. She introduced an aggressive approach. At a high-level, the algorithm is easy to understand.

1. compute a lower bound on the loop initiation interval, $I_{\text{min}}$. The initiation interval is the scheduling length of the loop.

2. let $II \leftarrow I_{\text{min}}$

3. try to schedule the loop into $II$ cycles, using a modulo reservation table of length $II$.
   
   If either the scheduler or the allocator fails, increment $II$ and try again.

This technique is very effective on loops that contain a single block [32]. Its compile time efficiency depends largely on the distance between $I_{\text{min}}$ and the value of $II$ for which scheduling succeeds.

2.1 Minimal Initiation Interval Estimation

The first step of the algorithm is to estimate a lower bound, $I_{\text{min}}$. Lam proposes to take the larger of two estimates that are based on resource constraints as $I_{\text{min}}$.

- The first constraint is based on the availability of functional units to execute the instructions. For each functional unit type, the instructions that it must execute are counted. This tally is divided by the number of functional units of that type. This shows the minimal number of cycles required to issue all instructions of that type, ignoring any control flow or data dependences. Let $II_{fu}$ be the largest such number, taken over all functional unit types. $II_{fu}$ must be a lower bound on the smallest initiation interval for the loop, since all the instructions must execute.
• The second constraint is based on the latency of recurrences in the loop. When the data dependence graph contains a cycle (representing a recurrence), its length in CPU cycles can be computed by adding the latency of each instruction in the recurrence. Let $\ell_s$ be the largest such number, taken over all the recurrences in the loop. $\ell_s$ must be a lower bound on the initiation interval since the loop must be long enough to execute all its recurrences.

Thus, Lam's minimal initiation interval is:

$$H_{\text{Lam}} = \max(H_{\text{fr}}, H_s)$$

(2.1)

Ebcioğlu proposed pipeline-percolation scheduling [20]. He assumes the hardware is a VLIW machine able to execute a tree in one single instruction. In this kind of hardware, instruction execution has two phases. First the conditionals are evaluated and the final path is determined. Then the instructions in the execution path are executed, and the others are discarded. Ebcioğlu does not propose a minimum $H$ instead each VLIW instruction is scheduled in its own iteration. The effective number of stages that the schedule takes is the final number of VLIW instructions. That means every register defined by a VLIW instruction will remain alive at least until the next iteration, putting enormous pressure on register allocation.

Allan et al. have introduced some refinements [4] to the above technique. She computed a minimum $H$ to minimize register pressure and executed the Ebcioğlu technique over a set of instructions instead of each one at a time. The estimate differs from Lam's work in that $H_{\text{min}}$ is not iterated, it is used only once to alleviate the register pressure from percolation. She also combined other techniques to improve parallelism, like the brake — a heuristic that divides a long dependence chain in several pieces in such a way that each piece spans only one iteration of the loop.

### 2.2 Modulo Scheduling

The modulo scheduling algorithm uses a basic block list scheduler and a modulo reservation table to determine if there is a resource conflict.

Initially, the modulo reservation table is set to have $H_{\text{min}}$ entries. If an instruction cannot be scheduled in a given time slot nor in any of the $H-1$ next time slots, then a schedule for the target $H$ fails. Lam's approach to this problem is to iterate over $H$. She increments the value of $H$ and tries to schedule again. One of the arguments for choosing this strategy relies on the fact that schedulability is not monotonic.
A binary search on $II$ might fail to find the smallest possible $II$, because, given that the scheduler fails at $II_i$, we can not prove it would fail for all initiation intervals smaller then $II_i$. That is precisely the point that one of our improvements exploits. Our measure of register pressure (as defined in the next chapter) is trivially monotonic. Thus, if we know that the scheduler would fail at $II_i$ because of register shortage then we can prove all intervals shorter than $II_i$ will fail for the same reason.

When the dependence graph has recurrences, the scheduler first computes the strongly connected components of the graph. Next, each SCC is represented by a single node, reducing the graphic to an acyclic one, and, only then, it proceeds as above. Each node representing a SCC summarizes all constraints of all instructions in this SCC.

The problem with this approach is that when the constraint information is gathered to one single node, all the operations are assumed to happen at the same moment. or more precisely, during the whole time frame predicted by the partial solution. If recurrences are small, it is likely some resource will be allocated for more time than really needed.

### 2.3 Scheduler and Register Allocator

Lam does not mention any particular technique for register allocation. In fact, when discussing modulo variable expansion and possible exponential register requirements, she assumes that the target machine has a large number of registers, as did Warp. This machine has 64 general purpose registers that can be accessed by any operation, plus a set of 64 registers for each functional unit, namely: adder, multiplier, two memory access functional units, two address arithmetic functional units, giving a total of 7x64 registers.

She also shows empirical results on the Warp machine to conclude that register shortage is a problem for a small number of programs and these programs are invariably large loops where other scheduling techniques can perform well.

In superscalar architectures, the register shortage can be significant. The transformations that cover latency typically increase demand for registers.

- Scheduling to cover functional unit latency moves uses farther away from definitions; this can increase the demand for registers.

- Register blocking and cache blocking try to hide memory latency; this almost always introduces new address expressions and induction variables that, in turn,
increase the demand for registers. Blocking also moves uses closer together, making it more likely that the value itself will stay in a register. Carr's [13] register blocking and Lu's [34] register promotion both move values directly into registers, increasing demand.

- Modern optimizers take advantage of as many registers as possible. The scheduler tries to reorder the resulting code, but reordering itself can increase demand of registers, too.

A simple solution to this problem is to build machines, like the Warp used in Lam's research, that have sufficiently large register sets. This is unlikely to occur in the commodity microprocessor market for several reasons. First, increasing the register set size would require additional bits in the instruction format; this alone makes the task difficult. Second, the marketplace pressures architects to maintain backward compatibility; a major change like doubling the number of registers is less likely in this environment. In fact, the opposite trend seems likely; hardware designers will add more functional units, increasing the demand for registers without increasing the number of bits allowed to address them.

Bradlee, Eggers and Henry [8] attempted to integrate the scheduler and register allocator. They implemented three opposing approaches in an attempt to evaluate the need for scheduler and register allocator integration. They used a basic block list scheduler in all three experiments, and they found that a full integration would not get better results than a partial integration approach based on passing parameters from one to the other.

Huff [29] introduced the idea of using registers as constraints. But he only computed such a constraint after scheduling the code, and uses this figure only to guide the register allocator. In his work, Huff suggests that a very small portion of the loops would benefit of exploring this idea further. We believe that was the case due the large number of small loops in the test programs. Moreover, there is no indication of how much optimization was applied to the test programs; and many global optimizations increase register pressure.

A computation similar to our register-cycle figure is also made in Huff's work. The biggest difference is that this number is computed for each individual instruction instead of a global figure, and it is used to guide a heuristic to place a given instruction within its slack frame, instead of considering register pressure as factor to help scheduling by guiding spilling and register allocation.
2.4 Handling Control Flow

Lam introduced Hierarchical reduction in [32]. In her scheme each construct is reduced to a single node representing all the constraints of its components. For example, the constraints for an if construct would be the union of the constraints from the then and else parts. Similarly, the length of the new node would be the maximum of that of the two branches. After reducing the if construct, its single node becomes part of an outer basic block. Repeating this process one can schedule all loops in the program.

Hierarchical reduction is a pessimistic approach. It doesn’t allow the scheduler to mix instructions from other parts of the program. This is especially bad if the code has moderate sized basic blocks where a long chain of dependences exists and few of them can be computed in parallel. Hierarchical reduction must reduce such sets of instructions to a single complex instruction before any parallelism can be explored by software pipelining. That way, if we have many “if”s in the loop body, hierarchical reduction will leave many holes in the schedule. This situation hurts compile time because the initial II doesn’t take this in account. Since it simply counts all the instructions, taking the maximum for each side of the branch, in the end, a loop body with some control flow may require many scheduler iterations to succeed, and the pipeline still may be under-utilized. Moreover, when hierarchical reduction takes place, many dependences edges are concentrated to one point. This change in the dependence graph tends to create new recurrences. The more conditional code that must be reduced, the more likely and larger these false recurrences will occur. The final schedule must satisfy these recurrences too, resulting in even more compilation time and larger II achieved.

Although requiring special hardware, Ebcioğlu [20] handles control flow within the loop body in a more efficient way than Lam’s. The hardware is able to execute a tree of instructions, i.e., it executes several branches and operations in one single cycle provided the conditionals are evaluated independently. The first part of an execution cycle determines which path will be executed and, so, which operation’s results will be committed; the others will be discarded. The scheduler builds trees of operations moving them in a greedy fashion, no attempt is made to gather instructions equally likely to execute. That way, the effective use of the hardware can be compromised by a large number of discarded instructions.

Moon and Ebcioğlu propose a new framework [35] to extract parallelism and represent VLIW transformations. The framework allows representation of code duplication, which can be used to delay actual code duplication until scheduling is
done. Eventually, after issuing long instructions for VLIW machines, the VLIW instructions reflect the replicated code. After extracting all possible parallelism, they use Ebcigolu's percolation scheduler. The improvement resides in the fact that fewer operations are likely to be discarded in the second part of instruction execution.

Another proposal to increase parallelism comes from Chen. Chen and King [15]. They propose an algorithm to transform a sequence of nested conditional branches to a multiway branch, in such way that they are effectively executed in parallel.

A different approach to improve hardware utilization is proposed by Tang et al. [44]. They propose special hardware and a new algorithm to minimize the number of functional units not utilized at each cycle. In this case, the algorithm takes advantage of the fact that the two sides of a conditional branch are never executed at the same time and builds a VLIW word longer than what the architecture can execute. For example, if the architecture has $n$ functional units, the instruction can be built with $2n$ operations: $n$ from each side of the branch. The special hardware is required to decide which path will be taken and assign the $n$ operations in this path to the $n$ functional units on the fly.

Since Lam takes the union of the resource usages in a conditional construct while predicated modulo scheduling takes the sum of the usages, the former should yield the smaller $II_{min}$. However, since Lam separately list schedules each side of the conditional branch, the $II$ that she actually achieves should deviate from the $II_{min}$ greatly. Warter et al [46] have implemented both techniques and came to the same conclusion. They combined the best of both approaches in their enhanced modulo scheduling algorithm. They derive the modulo schedule as if predicated execution were available, except that two operations of the same iteration are allowed to be scheduled on the same functional unit, at the same time, if their predicates are mutually exclusive. This is equivalent to taking the union of the resource usages. Enhanced modulo scheduling results in $II_{min}$ that are small as for hierarchical reduction, but, as with predicated modulo scheduling, the achieved $II$ is better than Lam's approach.

Our proposal to improve software pipeline performance on loops with control flow consists of unwinding the control flow through a series of edge splitting and block replication transformations. The idea is to explore path locality within a loop. For each possible path through the loop body we generate a copy of the basic blocks in the path. That way each path forms a new independent loop that has only two possible exit points: one that terminates the loop, and the second that branches to a different path within the same loop.
The resulting loops are then scheduled independently. If we choose to software pipeline these new loops, we must create some prologue and epilogue basic blocks between different paths. As we show in chapter 4 this technique also changes which chain of data dependences get pipelined, bringing more opportunities to find ILP.

Whalley [48] uses code replication in loops with conditional expressions to transform the loop to one with a single conditional variable. In most cases the transformation applied to one single loop still results in one single loop, where some conditional branches in the conditional expression are moved to other program segments, effectively reducing the number of conditional branches executed.

Warter [47] proposes a scheme to achieve multiple initiation intervals with a modulo reservation schedule. She observes that a loop with conditional code and fixed initiation interval has some drawbacks. A short path in the execution may take longer to execute because the operations of this shorter path may be scattered in the reservation table. She proposes to partition the MRT into the same number of execution paths in the loop. Operations are then re-grouped in these partitions. To get a real gain with this technique, one must profile the loop and find which path is executed more often and order the partitions within the MRT in decreasing order of frequency.

The technique we present in this thesis does not require this. Since each path is transformed in a independent loop, each path will have its own initiation interval and less executed paths are not penalized by more frequent executed paths. The possible drawback in our scheme happens when code growth is excessive or when the loop does not present enough path locality to sustain the pipeline.

Ebcioğlu is also aware of the effective initiation interval problem [20]. Although enhanced pipeline scheduling does not build a MRT, he points out that it allows gaps between instructions in the same iteration to maintain performance if path locality is not achieved. In other words, iteration \( n \) and \( n + 1 \) will both take paths such that iteration \( n \) will provide data in time for subsequent operations of iteration \( n + 1 \), and iteration \( n + 1 \) will continue without pausing.
Chapter 3

Register Constraints

This chapter details our technique to account for the availability of registers in the lower bound computation of Lam's software pipelining algorithm.

Suppose we want to determine the register usage of a given program. The problem in getting this information from the register allocator is that the register allocator builds the interference graph based on a final instruction order, so a success/fail result will be known only after scheduling. Moreover, the interference graph is expensive to build and gives too much local information. What we need is a simple way to understand register usage in a program region, i.e., for a particular loop. Graph coloring allocators cannot do this.

Given a loop body, we want to determine the register usage for this particular piece of code. We can divide the register demand in three types:

$R_G$ are those registers used for holding loop invariant values that are not constants.\(^1\)

These values are defined outside the loop and referenced one or more times within the loop. Each of these requires a register for the entire loop.

$R_I$ are those registers allocated to recurrences. Each recurrence and each iteration spanned requires allocation of one register for the entire loop.

$R_s$ are those registers required to satisfy short lived values. They can be live across iteration boundaries or not, depending on software pipelining, but are always live for less than the duration of one iteration.

The first two types of register demand are easy to compute. It is $R_s$ that requires our attention.

\(^1\)Constant values should be "rematerialized" [10].
3.1 Register Cycle Definition

Individually examined, each instruction requires a register to hold the result of the operation and registers to hold each operand. Moreover, the duration of this allocation must be at least equal to the instruction latency. A register-cycle is the product of the number of registers required by the allocation and the number of cycles the allocation must be taken.

To count register-cycles, the compiler should consider, for each value, the schedule that minimizes the value’s lifetime. That lifetime provides a lower bound on the number of cycles that the value must occupy a register. By summing the lower bound for each value, we can obtain an underestimate of the demand for registers. It is an underestimate precisely because the schedules that minimize each value’s lifetime may not all be achievable in a single schedule for the loop.

A simple example may help. Assume a target machine where the results of a load instruction become available on the second cycle after its issue and the result of an add is available on the next cycle. Then, the sequence

<table>
<thead>
<tr>
<th>cycle</th>
<th>slot 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>r1 ← MEM[a]</td>
</tr>
<tr>
<td>2</td>
<td>r2 ← MEM[b]</td>
</tr>
<tr>
<td>3</td>
<td>nop</td>
</tr>
<tr>
<td>4</td>
<td>r3 ← r1 + r2</td>
</tr>
</tbody>
</table>

has three distinct values. Register r1 is defined in cycle 1 and used in cycle 4. It uses 3 register-cycles. Register r2 is defined in cycle 2 and used in cycle 4. It uses 2 register-cycles. Register r3 is defined in cycle 4 and (presumably) discarded. It uses 1 register cycle. Thus, the code fragment effectively use 6 register-cycles. Of course this value will vary according to the given instruction order. In the case of an unlimited number of issue slots, an absolute minimum value can be derived from the data dependence graph, where each edge is labeled with latency information. The previous example has the following dependence graph

```
       r1 ← ⋯
      /   \
2 → 2
       \   /
        r3 ← r1 + r2
```

which gives rise to 4 register-cycles in the edges and 1 register-cycle in the leaf node. for a total of 5 register-cycles.

On a two register machine, register constraints would require at least three cycles in the schedule. If we assume an unlimited number of issue slots, this schedule can be realized.

<table>
<thead>
<tr>
<th>cycle</th>
<th>slot 1</th>
<th>slot 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>r1 ← MEM[a]</td>
<td>r2 ← MEM[b]</td>
</tr>
<tr>
<td>2</td>
<td>nop</td>
<td>nop</td>
</tr>
<tr>
<td>3</td>
<td>r1 ← r1 + r2</td>
<td>nop</td>
</tr>
</tbody>
</table>

During cycle 2, no instructions are issued. If there are other computations in the loop body, these cannot overlap due the lack of registers. From a different angle, we can say that we need at least 5 registers to realize the schedule with initiation interval equal 1.

<table>
<thead>
<tr>
<th>cycle</th>
<th>slot 1</th>
<th>slot 2</th>
<th>slot 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>r1</td>
<td>r2</td>
<td>nop</td>
</tr>
<tr>
<td>2</td>
<td>r3</td>
<td>r4</td>
<td>nop</td>
</tr>
<tr>
<td>3</td>
<td>r2</td>
<td>r5</td>
<td>r1 ← r1 + r2</td>
</tr>
<tr>
<td>4</td>
<td>r1 ...</td>
<td>r2 ...</td>
<td>r3 ← r3 + r4</td>
</tr>
<tr>
<td>5</td>
<td>r3 ...</td>
<td>r4 ...</td>
<td>r2 ← r2 + r6</td>
</tr>
</tbody>
</table>

Notice that loop unrolling and modulo variable expansion were applied. Also, in the loop steady state (cycle number 3), all 5 registers are busy, which indicates that no schedule is possible with fewer registers.

That brings to light the importance of this new lower bound. The other two constraints will report a possible initiation interval of 1, leaving to the register allocator the task of finding the necessary registers to realize the schedule. On a two register machine, either three registers will be spilled, or the scheduler will fail and the initiation interval will be increased.

Thus, the register-cycle count not only gives the minimum initiation interval that can be obtained for a given register set, but also gives an insight into the cost (number of spills to be introduced) to obtain a shorter schedule. Spilling increases the demand for instruction issue slots; this can raise the initiation interval. As memory latencies grow, a more pernicious effect surfaces – spill code adds a lot of latency
to the dependence graph for each issue slot that it requires. This may increase the importance of a spill-free schedule.

As another example, if the original computation was

\[
\begin{array}{c|c}
\text{cycle} & \text{slot 1} \\
1 & r1 \leftarrow \text{MEM}[a] \\
2 & r2 \leftarrow \text{MEM}[b] \\
3 & \text{nop} \\
4 & r3 \leftarrow r1 + r2 \\
5 & r4 \leftarrow \text{MEM}[c] \\
6 & r5 \leftarrow \text{MEM}[d] \\
7 & \text{nop} \\
8 & r6 \leftarrow r4 + r5 \\
\end{array}
\]

the heuristic would suggest a total of 10 register-cycles. On a machine with two registers, five cycles would be the minimum number required to meet the register constraint. The schedule

\[
\begin{array}{c|c|c}
\text{cycle} & \text{slot 1} & \text{slot 2} \\
1 & r1 \leftarrow \text{MEM}[a] & r2 \leftarrow \text{MEM}[b] \\
2 & \text{nop} & \text{nop} \\
3 & r1 \leftarrow r1 + r2 & \text{nop} \\
4 & r1 \leftarrow \text{MEM}[c] & r2 \leftarrow \text{MEM}[d] \\
5 & \text{nop} & \text{nop} \\
6 & r1 \leftarrow r1 + r2 & \text{nop} \\
\end{array}
\]

uses six cycles and no shorter schedule can be realized. The observation here is that the number of registers required to generate the add instruction is equal to the number of available registers, so no pipelining is possible and the longest path in the data dependence graph determines the length of the schedule. Of course, with four registers, the bound predicts a three cycle body.

\[
\begin{array}{c|c|c|c|c}
\text{cycle} & \text{slot 1} & \text{slot 2} & \text{slot 3} & \text{slot 4} \\
1 & r1 \leftarrow \text{MEM}[a] & r2 \leftarrow \text{MEM}[b] & r3 \leftarrow \text{MEM}[c] & r4 \leftarrow \text{MEM}[d] \\
2 & \text{nop} & \text{nop} & \text{nop} & \text{nop} \\
3 & r1 \leftarrow r1 + r2 & r3 \leftarrow r3 + r4 & \text{nop} & \text{nop} \\
\end{array}
\]
The lower bound highlights the fact that we need more than four registers before we can issue any instructions in cycle two.

Note that this bound, like Lam’s two bounds, is a lower bound. The prescribed schedule may not exist because register-cycles may not be the critical constraint on the schedule. Alternately, the assumptions underlying the estimate may be unrealistic. For example, the individual schedules assumed to compute register-cycles per value may not be simultaneously realizable. In our example, with two issue slots, the function unit constraint predicts an initiation interval of three cycles although an optimal schedule requires four cycles.

3.2 Computing Register-cycles

While the computation is obvious in the simple example, real programs are more complex. In particular, we are concerned about a loop body with the structure shown in Figure 3.1. Assume that the delays from the assignment to r2 are two cycles and that all other delays are one cycle. Clearly, the minimum lifetime of r1 is three cycles (assuming enough issue slots exist). The value of r1 computed in the initial assignment is used as input to the definition of r2 and again in the definitions of r17 and r18. The definition of r2 cannot occur until the cycle after r1 is defined. Similarly, the definitions of r17 and r18 cannot execute until two cycles after r2 is defined, effectively stretching the live range of r1. The lifetime of r2 is two cycles, while r17 and r18 each have a lifetime of one cycle. The lifetime of r37, r53, and r38 are also one cycle. This gives a total register-cycle requirement of ten for the code fragment.

Assume for the moment that the dependence graph for the loop body is a tree, and that our target machine has infinite resources. Under those conditions, the register-
ALGORITHM RC: Counts register-cycles for each definition

INPUT: Assume a dependence dag \(d\) where each node is an instruction. Instructions (definitions) are numbered uniquely and compactly. Each instruction \(i\) has an associated delay, \(\text{delay}(i)\)

OUTPUT: The minimum number of register-cycles required by each definition

METHOD: Transform the graph by combining tree edges, until forward/cross edges are found in parallel with tree edge. Save the necessary stretch in the node and delete forward/cross edge. Repeat until last forward/cross edge is deleted.

NOTATION: 
- \(\text{parent}[i]\) Denotes the number of parents of node \(i\)
- \(\text{parent}_j[i]\) Denotes the \(j^{th}\) parent of node \(i\)

DFS the dependence graph.
Return tree/forward/cross edge information.

for each node \(i\)
  if \(|\text{parent}[i]| \neq 0 \text{ and } |\text{child}[i]| \neq 0\)
    worklist \(\leftarrow i\)

for each edge \(e\)
  latency\([e]\) \(\leftarrow\) latency of instruction on head of \(e\)

while (exist forward/cross edges in graph)
  remove \(i\) from the worklist
  Remove\((i)\)

Remove\((i)\):

for each incoming edge of \(i\), \(In_j[i]\)
  Make a copy of outgoing edges of \(i\), \(Out_k[i]\)
  Add \(\text{latency}[In_j[i]]\) to copied edges
  If \(Out_k[i] \in \text{forward/cross edges set}\)
    \(\text{stretch}[i] = \max(\text{stretch}[i], |\text{latency[edge]} - \text{latency}[Out_k[i]]|)\)
    delete forward/cross edge

Figure 3.2 Computing lifetimes
cycle requirement is trivially computed by adding the latency of each instruction in the loop. In a more general case though, the dependence graph will be an acyclic graph (a dag). The difference from the previous scenario is the possible existence of cross edges and forward edges, which introduces alternate paths for some pairs of nodes in the graph. That means, if one of the paths is longer then the other one\(^2\), the shorter one will be stretched to match the longer one, and that stretching causes more register-cycles to be used. Our algorithm consists in computing how much stretch is necessary to satisfy all paths in the dependence dag. Our lower bound is then

\[
\frac{\sum_{d \in \text{def}_s} \text{latency}(d) + \text{stretch}(d)}{|\text{registers}|}
\]  

(3.1)

A simple algorithm for computing lifetimes is shown in Figure 3.2. The general idea of the algorithm is that whenever a forward/cross edge is found in a dependence graph, it is “short-circuiting” a longer path in the graph, and thus, the register allocation that the forward edge represents must be “stretched” to match the longest path. Our algorithm consists of computing all necessary “stretching” in a given DAG.

The algorithm starts by building the SSA representation of the given program, and giving each edge a weight representing the latency of the originating instruction.

The main algorithm works by removing nodes with \(n\) incoming edges and \(m\) outgoing edges from the graph, and replacing them by \(n \cdot m\) edges connecting all \(n\) predecessors to all \(m\) successors. When these edges are put back in the graph, an edge duplication may occur. If this is the case, one of the edges is a forward/cross edge, and the other is representing the accumulated latency of some “short-circuited” path in the original graph. The difference of accumulated latencies between these edges is the number of register cycles required to satisfy the stretching in question. The algorithm finishes when all forward edges are processed.

The algorithm is linear in the size of the SSA graph. Since nodes are collapsed only once, the algorithm performs \(O(\text{nodes})\) collapse operations. Each collapse requires \(O(\text{incoming-edges} \cdot \text{outgoing-edges})\) operations, but \textit{incoming-edges} is a small ILOC dependent constant\(^4\). Therefore, the total number of operations for collapsing is \(O(\text{edges})\), and for the whole algorithm is \(O(\text{nodes} + \text{edges})\).

At this point we have computed the register-cycle requirement for each individual instruction. To find a global figure for the whole loop, control flow must be considered.

\(^2\)Length measured in instruction latencies on the path

\(^4\)For most operations, \texttt{JMPx} operations has arbitrary number of registers, but no outgoing edge, therefore collapsing of \texttt{JMPx} operations never generates edges.
In order to do that, we compute the \( RC \) for each basic block adding the \( RC \) of each of its instructions, then we use the single-source longest path algorithm with the basic block \( RCs \) as edges weights.

The single-source longest path algorithm is modified to take advantage of having a graph with no cycles as input, \( i.e.\), we substitute the \( O(V \log V) \) sorting algorithm by a topological sort, reducing the running time to \( O(V + E) \).

Applying the algorithm to the dependence graph shown in Figure 3.1 yields the following values for latency, stretch and register-cycle.

<table>
<thead>
<tr>
<th></th>
<th>r1</th>
<th>r2</th>
<th>r17</th>
<th>r18</th>
<th>r37</th>
<th>r38</th>
<th>r63</th>
</tr>
</thead>
<tbody>
<tr>
<td>latency</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>stretch</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>register-cycles</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Note that \( r17, r18, \) and \( r38 \) each require a single register-cycle, even though they are never used. Thus, the entire computation requires at least ten register-cycles.

### 3.3 Recurrences in the Dependence Graph

A recurrence in dependence graph indicates that one or more registers are required to hold the value over the entire loop body, independent of its length. The recurrence will require a register for each iteration that the cycle spans. (A register is necessary for each active iteration. See Carr’s thesis [12]). More precisely, each source of an anti-dependence will require a register for each iteration that the cycle spans. If some definition is source of more then one anti-dependence, then it suffices to provide registers for the longest one. To account for this, subtract the set of registers required for recurrences from the number of machine registers available for allocation.

In our experimentation we used a low level intermediate language and variables allocated to registers. We also use the SSA representation of the program [19]. In that environment, all recurrences will consist of cycles in the SSA graph involving register names and \( \phi \)-nodes. The number of iterations spanned by a cycle can be found by examining the \( \phi \)-nodes in the cycle. If we count the number of \( \phi \)-nodes that refer to a definition point outside the loop, that is the number of iterations spanned by the cycle.\(^4\)

\(^4\)In our representation, we carry along enough information about memory references to let us ignore memory-based values that occur in cycles.
ALGORITHM FO: Adjusts register-cycles for each definition for fan-outs constraints.

INPUT: Assume inputs and outputs from algorithm RC in figure 3.2.

OUTPUT: fan-out cycles required by each definition

METHOD: Visit all nodes in the dependence graph. For each node, count all children that use the same functional unit type. Apply formula for $O$

NOTATION: $uses$ is an array with an entry for each $type$ of functional unit existent in the architecture; $type(j)$ returns the functional unit type of instruction $j$, and $k_f$ is the number of functional units of type $f$.

\[
\begin{align*}
\text{fan-out-cycles} & \leftarrow 0 \\
\text{for } 1 \leq i \leq |\text{definitions}| \\
& \quad \text{for } 1 \leq j \leq |\text{child}[i]| \\
& \quad \quad \text{uses[type(j)]} \leftarrow \text{uses[type(j)]} + 1 \\
& \quad \quad \text{for each functional unit type, } f \\
& \quad \quad \quad \text{fan-out-cycles} \leftarrow \text{fan-out-cycles} + \left\lceil \frac{\text{uses}[i]}{k_f} \right\rceil - 1
\end{align*}
\]

**Figure 3.3** Adjusting for fan-outs

### 3.4 Fan-out Constraints

So far, we have assumed an infinite number of registers and functional units to satisfy the dependence graph. Real processors have finite resources. In practice, this limits the number of uses that can occur in a single cycle. We can use this observation to tighten our register-based lower bound.

If the machine has $n$ registers and $k_{f_i}$ functional units of type $f_i$, and $m_i > k_{f_i}$ instructions of type $f_i$ use the value in register $r_i$, then we must add some register-cycles to the estimate computed in Figure 3.2. Specifically, we can compute the number $O$ of extra register-cycles as

\[
O = \max(\left\lceil \frac{m_i}{k_{f_i}} \right\rceil, \ldots, \left\lceil \frac{m_j}{k_{f_j}} \right\rceil) - 1
\]

For example, consider scheduling the sequence
\[
\begin{array}{c|c}
\text{cycle} & \text{slot 1} \\
\hline
1 & r1 \leftarrow \text{MEM}[a] \\
2 & r2 \leftarrow r1 \ldots \\
3 & r3 \leftarrow r1 \ldots \\
4 & r4 \leftarrow r1 \ldots \\
\end{array}
\]

on a machine with four registers, but with 2 units of the type used in the operations. The heuristic in Section 3.2 computes a lifetime of four register-cycles, suggesting a initiation interval of one. The formula for Fan-out constraints adds one extra cycle. Scheduling yields the following code.

\[
\begin{array}{c|c|c}
\text{cycle} & \text{slot 1} & \text{slot 2} \\
\hline
1 & r2 \leftarrow r1 \ldots & r3 \leftarrow r1 \ldots \\
2 & r4 \leftarrow r1 \ldots & r1 \leftarrow \text{MEM}[a] \\
\end{array}
\]

The algorithm FO given in figure 3.3 is executed immediately after the algorithm in figure 3.2. Each definition (node) and each child (edge) is visited only once. so the algorithm is linear in the size of the dependence graph. The number of functional unit types is typically a small constant.

### 3.5 Fan-in Constraints

Another factor is the limit that the number of functional units and available instruction issue slots imposes on a schedule. The algorithm given in the previous sections computes the minimum number of register-cycles required by the dependence graph. If the predecessors of a given node require more resources than are available, then not all predecessors can be issued immediately before their uses. This will increase the lifetimes of some operands. An example will help. Consider, for example, the dependence graph labeled "simple case" in Figure 3.4. Both load instructions must

![Simple case](image)

\[\text{Simple case}\]

![Cascade effect](image)

\[\text{Cascade effect}\]

**Figure 3.4** Fan-in constraints
be issued at least two cycles before instruction \( x \). Assuming we can issue only one load per cycle, one of these loads must be issued one slot before the other, effectively increasing the number of register-cycles required for one of the edges.

If the machine can issue \( k_f \) instructions of type \( f \) per cycle, and some instruction \( n \) has \( m_f \) predecessors of type \( f \), then we must add \( k_f \) register-cycles, where \( k_f \)

\[
I = \lceil \frac{m_f}{k_f} \rceil - 1
\]
to the register-cycle count. One question that remains is where to put these extra cycles, because the fan-in phenomena has a cascade effect. As the dependence graph labeled “cascade effect” in Figure 3.4 shows, resolving one fan-in conflict may introduce others. To complicate matters further, placing an instruction to satisfy a fan-out may also satisfy a fan-in. Efficiency dictates that the compiler cannot explore all these issues just to compute a lower bound, so we adopt an oversimplification. We compute fan-ins considering only immediate predecessors that have not been adjusted for fan-out.

The algorithm FI given in figure 3.5 is executed immediately after algorithm in figure 3.3. Each definition (node) and each child (edge) is visited once, so the algorithm is linear in the size of the dependence graph. The second loop also executes in linear time. At most there may exist as many different latencies as the number of incoming edges for that node. The number of functional unit types is a small constant.

As result of these adjustments, equation 3.1 has to be modified:

\[
RC = \frac{\sum_{d \in \text{defS}}(\text{latency}(d) + \text{stretch}(d)) + \text{fan-out-cycles} + \text{fan-in-cycles}}{| \text{registers} |}
\]  

(3.2)

Following digital design, we call these fan-out / fan-in constraints.

### 3.6 Accounting for Registers as a Resource Constraint

\( R_s \), the number of registers needed for short-lived computations, can now be obtained by dividing the total register-cycle number by some initiation interval \( II \).

In summary, the equation:

\[
R_s = \frac{RC}{II}
\]  

(3.3)

estimates the register requirement for short lived values of a given loop, for a given initiation interval. The immediate consequence of this is that we can check all three
ALGORITHM F1: Adjusts the total number of register-cycles for fan-in constraints.

INPUT: Assume inputs and outputs from algorithm RC in figure 3.2 and algorithm FO in figure 3.3.

OUTPUT: fan-in-cycles, the number of register-cycles to be added to the total number of register-cycles used by the program.

METHOD: Visit all nodes in the dependence graph. For each node, count all parents that use the same functional unit type and same latency. Apply formula for I and total it.

NOTATION: defs is a sparse array with an entry for each type of functional unit existent in the architecture and for each size of latency. type(j) returns the functional unit type of instruction j, latency(j) returns the latency of instruction j, and k_f is the number of functional units of type f.

fan-in-cycles ← 0
for 1 ≤ i ≤ |definitions |
  for 1 ≤ j ≤ |parent[i] |
    if node j has not being adjusted by FO algorithm
      defs[type(j)][latency(j)] ← defs[type(j)][latency(j)] + 1
  for each functional unit type, f
    t ← 0
    for each latency in defs, d
      t ← max(t, ⌈defs[f][d]/k_f⌉) - 1
    fan-in-cycles ← fan-in-cycles + t

Figure 3.5 Adjusting for fan-ins
register requirement components \((R_G + R_5 + R_s)\) against the number of registers furnished in the architecture, \(R_T\), and decide if allocation is possible, or estimate a number of register to be spilled \(((R_G + R_5 + R_s) - R_T)\).

On other hand, this notion of register-cycles can also be used to derive a register-based lower-bound on the initiation interval. Using equation 3.3, and the total number of registers in the architecture, we come up with:

\[
II_{RC} = \frac{RC}{R_T - (R_G + R_s)} \tag{3.4}
\]

The formula is still an estimate. In particular, it produces numbers on average register demand rather than the demand at any specific point.\(^5\)

In general, there are three possible outcomes for equation 3.4:

1. When \(R_G + R_5 > R_T\) it is not possible to compute \(II_{RC}\). In this case, register spilling must take place.

2. When \(II_{RC} < \max(II_{fu}, II_5)\) registers do not impose constraints to scheduling. We proceed with software pipelining as usual.

3. When \(II_{RC} \geq \max(II_{fu}, II_5)\) a choice of action must be taken. Either we choose to increase the initiation interval taking

\[
II_{min} = \max(II_{fu}, II_5, II_{RC}) \tag{3.5}
\]

as the starting point to software pipeline, or, we spill some registers.

Inserting spill code has several effects.

1. It adds instructions. This can increase \(II_{fu}\); whether or not it does depends on the busyness of the individual functional units. If the unit that issues loads and stores did not give rise to \(II_{fu}\), some spills may be, in effect, free.

2. It decreases demand for registers. This will lower \(RC\) and the numbers derived from it, including \(II_{RC}\).

\(^5\) Average demand can be small in a loop body where peak demand is still excessive. Unfortunately, we know of no method for determining demand for registers precisely, except for performing a complete register allocation
In this case, we can view the process of choosing an initial interval as a negotiation between scheduler and register allocator. As spill code is introduced, both \( \max(I_{fu}, I_{bs}) \) and \( I_{RC} \) will converge to a point. This converging point is the one we propose as start point to software pipeline the loop.

This scheme requires two different versions of the register allocator. The first would accept the number of registers to spill as a parameter, would focus on a region (the loop being scheduled) instead of the whole program, would do only spilling and not allocation, and would be called iteratively from the scheduler. The second would color the graph and do the actual allocation of registers but would never spill registers within a inner loop, instead it should fail by returning to the scheduler. The scheduler then, increments the initiation interval and start the scheduling process over again.

In the second allocator, if the allocation process is done globally, then it would be necessary to recompute register allocation for all loops, pipelined or not, whenever the scheduler changes the schedule of one single loop. To avoid this situation, one would have to content with less powerful register allocation scheme designed for use within a region, or would have to develop new techniques to do global register allocation incrementally.

We realized that the actual implementation of these register allocators would be a research project by itself and would deal with many issues beyond the scope of this thesis, so we left it for future research. Moreover, the contribution of the full implementation of these register allocators would be a precise measurement of how much compilation time would be saved from using registers as a third constraint in the minimal initiation interval computation. All other relevant performance data was acquired and reported on chapter 5.

Although not including all loops, the following measurement of compilation time improvement due to the register constraint can be precisely done with our current software pipelining implementation.

If we start the scheduling process with \( I_{\min} = \max(I_{fu}, I_{bs}) \) and software pipeline the loop assuming an infinite number of registers, \( i.e. \), using only data dependence constraints, we end up with a range of initiation intervals, namely \( I_{\min} \cdots I_{dep} \). Because \( I_{dep} \) was achieved only from data dependences constraints, we know that in the implementation with scheduler negotiating spill code with the register allocator, the final initiation interval achieved for that loop has to be \( I \geq I_{dep} \). Therefore, whenever the \( I_{RC} \) computed in equation 3.4 falls in between \( I_{\min} \cdots I_{dep} \), we can be sure that
\begin{align*}
\frac{H_{RC} - H_{min}}{H_{dep} - H_{min}}
\end{align*}

of compilation time can be saved from the register constraint. Although only a fraction of loops will fall in this case, the figures in section 5.2.2 shows speed-up of 18% to 30% for the scheduling process of the loops in that interval. We believe these figures are significant enough to justify the usage of registers constraint technique in a modulo scheduler, and any further experiments would contribute little to this particular issue.
Chapter 4

Unwinding the Control Flow

4.1 Overview

Our approach to improve software pipelining performance on loops with control flow consists of unwinding the control flow through a series of edge splitting and block cloning transformations. The idea is to exploit path locality within a loop. For each possible path through the loop body we generate a copy of the basic blocks in the path. That way each path forms a new independent loop that has only two possible exit points: one that terminates the loop, and the second that branches to a different path within the same loop.

The resulting loops are then scheduled independently. If we choose to software pipeline these new loops, we must create some prologue and epilogue basic blocks between different paths.

Unwinding paths through a loop creates an opportunity for performance improvement in several ways. First, it increases the length of basic blocks. This will lead to a decrease in the dynamic branch frequency. Second, the technique produces code with no merge points. This creates longer extended basic blocks [2, 30, 43] which are amenable to list scheduling. Every path in the original loop becomes a single extended basic block. Finally, if the code displays any path locality within the loop, the execution will avoid some of the cycles that are inevitably lost when matching up the schedules of multiple paths at a merge point.

A critical question is, do real programs exhibit path locality? By this, we mean, do consecutive iterations take the same path through the loop? We expect that they do. Simulation results obtained from FMM, SPEC, and NAS benchmarks, shown in section 5.2.1, suggest that most of the loops present locality profitable to the scheduler (see section 5.2.1 for discussion). In particular, we expect this technique to work well in loops that contain tests for boundary conditions, error conditions, and out of range subscripts. If, however, the loop shows little path locality, the loss from the epilogue and prologue code inserted along path-to-path transitions may well be
similar to the losses encountered in either hierarchical reduction or the instruction
discarding in VLIW predicated execution schemes.

As a side benefit, as far as the number of executed instructions is concerned, the
execution of a program transformed this way should be shorter or equal to any other
scheme. That is because partially dead and partially useless instructions are moved
from within the loops into the epilogue or prologue blocks.

This technique relies heavily on code replication. Thus, a reasonable concern is
the extent to which it causes code growth. An examination of innermost loops found
in the SPEC, FMM, LINPACK, and Livermore benchmarks, showed that 96% of the
inner loops have at most five different paths, and 98% have ten or fewer paths. Also,
no inner loop had conditional statements nested more than two deep, and 99% of the
inner loops contained four or fewer conditional statements. These statistics suggest
that the code space problems are likely to be manageable. Moreover, the application
is restricted to the body of inner loops, which represents a small part of the total
code. The results supporting these statements are shown in section 5.3.1.

4.2 Unwinding the Control Flow Graph

Control flow unwinding is performed in two steps. First, we transform the loop body
CFG into a tree (ignoring back edges). To achieve this, starting at the loop head, we
duplicate all path subgraphs for each merge point. As we duplicate basic blocks, back
edges leading to the loop head are also duplicated.

Figure 4.1 illustrates that process. Note that in the last phase we still have a
single loop, but now with four back edges leading to the head of the loop. Also, due
to the construction, each back edge corresponds to a different execution path within
the loop.

The algorithm is presented in figure 4.2. A graphical representation of operation
split.joint used in the algorithm is shown on figure 4.3. The algorithm starts with
a depth first search in the CFG to annotate the edges of the graph with one of four
possible values: tree, back, or forward edges as defined in [28], or exit edges for those
tree edges crossing loop boundaries. Next it lists the merge points in the CFG, and
for each one, it clones the basic block at this point (let's call it $H$) and recursively
clones all blocks in the sub-tree headed by the $H$ clone.

The second step of control flow unwinding consists of making each back edge an
independent loop. That is achieved by duplicating the header of the loop at the end
of each path in the CFG. As we do this, other blocks becomes loop headers and more
edges leaving the loop are added, but no new back edges are created. The process is repeated until each path has its unique head and back edge. Figure 4.4 shows the unwinding process on the last graph in figure 4.1. The dotted edges denote path-to-path transitions. In the second half of this figure the path-to-path edges leaving the blocks labeled A were omitted for clarity.

Note that after unwinding, each new loop retains a single entry point. This is an important condition that ensures pipelining feasibility. The fall-through exit is actually a path change in program execution.

Scheduling the resulting loops is easier than the original loop, since each one becomes a single extended basic block. Several methods for global scheduling can be used on extended basic blocks [30, 43], but in this thesis we will deal with the resulting loop as it was a regular single basic block. The difference arises when instructions from blocks A, B or C are hoisted above the exit point at end of block E, in which case speculative execution takes place. If this is not desired, one can software pipeline the loop and avoid speculative execution by choosing a minimal number of iterations to be spanned at least equal the number of exit branches in the extended basic block. The next section shows in detail how this is done.

The unwinding algorithm is presented in figure 4.5. Initially the algorithm picks the loop head and each back edge entering the node is submitted to the split-joint operation. This duplicates the head block at the end of the loop (let's call this block A') and creates new edges leaving A' (let's call these edges e1 \cdots e_n). Walking up the tree from block A', we will reach one of the edges e_1 \cdots e_n. That edge becomes the
ALGORITHM  \texttt{flat.cfg}: Transform a \texttt{CFG} DAG into a tree.

INPUT: The \texttt{CFG} DAG of a loop body

OUTPUT: The \texttt{CFG} in a tree form.

METHOD: Find joints in the \texttt{CFG}. Split each joint and recur on the sub-dag headed by the joint.

DFS the \texttt{CFG} and annotate edges with one of:
- tree, back, forward, or exit
- Create linked list $\texttt{fw.list}$ of forward edges during DFS

For each edge $x \to y$ in $\texttt{fw.list}$ do
  \texttt{split.joint}($x \to y$)
  \texttt{flat.visit}(y')

\texttt{flat.visit}(y):
  for each \texttt{tree} edge $y \to z$ leaving $y$
    \texttt{split.joint}($y \to z$)
    \texttt{flat.visit}(z')

\texttt{split.joint}($x \to y$):
  Clone $y$ as $y'$
  Modify edge $x \to y$ to be $x \to y'$
  For each edge $y \to z$ leaving $y$
    Create edge $y' \to z$
    If $y \to z$ is a forward edge
      Add $y' \to z$ to $\texttt{fw.list}$

\textbf{Figure 4.2}  Algorithm to flatten the \texttt{CFG}.

\textbf{Figure 4.3}  Graphical illustration of operation \texttt{split.joint} on edge $x \to y$
**Figure 4.4** Separating the loops.

**ALGORITHM unwind\_cfg:** Separate a loop with multiple back edges into several loops with one back edge each.

**INPUT:**

CFG in a tree form: routine split\_joint from figure 4.2: array of counters with number of incident back edges of each node.

**OUTPUT:**

Set of loops each with one back edge and one entry point.

**METHOD:**

Replicate the head of the loop, spreading the back edges to descendant nodes in the tree. Recur until no head has more than one back edge.

```plaintext
w ← loop head
while (w ≠ ∅)
    Pick n ∈ w
    If back\_edges[n] > 1
        For each back edge x → y of n
            split\_joint (x → y)
            back\_edges[n]--
            back\_edges[y/successor]++
            insert y/successor in w
```

**Figure 4.5** Algorithm to separate the loops.
new back edge, while the others represent path-to-path transitions and are labeled as tree edges. The original back edge also becomes a tree edge. Next, the block to which the new back edge points is inserted in the worklist and its back edge counter is incremented. The algorithm is repeated until the worklist is empty.

Each blocks is inserted in the worklist only once, and new back edges always point to blocks further down in the tree. Eventually they will reach the leaves of the original CFG tree. By construction in the first step, the leaves in the original CFG tree are in the same number of back edges. At this point no loop head has more than one back edge and the algorithm terminates.

The figure 4.6 illustrates a smaller but complete example annotated with some instructions. The smaller nodes represent landing pads for partially dead and partially useless instructions.

Note that after unwinding the control flow each individual loop has one instruction less than the original loop. The $d$ definition on block $A'$ is partially dead because in path $AB$, $d$ gets redefined before any usage. The $x$ definition on block $A''$ is partially useless because $x$ is not used in path $AC$. When the paths get separated in different loops, those instructions get moved to the landing pads; partial redundancy elimination [36] cannot do that.

### 4.3 Software Pipelining Loops with Early Exits

Some research papers address only loops where the exact number of iterations can be determined at compilation time; others claims that the algorithm given can be extended to loops with multiple exits and cite Tirumalai [45] as a way to do it. This seems to be an engineering problem. In this thesis, I present a new solution for it.

Tirumalai [45] address the problem of software pipelining loops with early exits and proposes a solution that requires both special hardware and a source code transformation. The hardware described in his paper is capable of predicated execution and eager execution. Eager execution means that an operation can be executed prior to the resolution of the predicate under which the operation would normally execute. If the predicate evaluates to false, the side effects of the operation are reverted.

The software transformation changes a loop with multiple exits to a loop with a single exit. This is accomplished by substituting an early exit branch operation with operations that set a flag and jump to the end of the loop. At the end of the loop, all exit conditions are tested. If any of them is true, the program executes a jump table operation to the proper destination.
Figure 4.6 Partially dead and partially useless instruction optimization

Figure 4.7 shows a general model for loops with early exits. In this model, a loop containing $n$ basic blocks, enumerated as $A \cdots N$, has $n$ conditional branches, each of them with some destination out of the loop ($L_1 \cdots L_n$). Within the dependence DAG of each block, we will find a sub-graph that evaluates the branch condition and thus determines the final execution path. We call the subset of operations in this sub-graph $p$, such that for each basic block $A \cdots N$ we have the corresponding $A_p \cdots N_p$ subsets. Since we don’t have the special hardware that Tirumalai proposes, nor any other kind of hardware to support speculative execution, we must schedule all operations in $A_p \cdots N_p$ in the first iteration. The other operations are free to be software pipelined. After the modulo reservation table is built, these operations may remain in the first iteration or may be delayed to the next one, etc. This is denoted in figure 4.7 by the subsets with numeric indexes. Thus $A_1B_1 \cdots N_1$ denotes the subsets of operations from blocks $AB \cdots N$ that are not part of path determination and are scheduled in the first iteration. $A_2B_2 \cdots N_2$ denotes the operations scheduled in the next iteration, and so on. Block $A$ spans $a_n - 1$ iterations, $B$ spans $b_n - 1$ iterations, and so on. The maximum of $a_n \cdots n_n$ determines the number of prologue and epilogue loop copies necessary to software pipeline this loop.

---

$^1$Which stands for path determination.
Since the operations contained in the $A_p \cdots N_p$ sets impose a strong restriction on scheduling, they are the first ones to be inserted in the MRT. Each subset is listed and summarized as a complex operation. The compound latency of this operation is the result of the longest path in the dependence sub-graph\(^2\), and resource conflicts. If the sum of all compound latencies is larger than $H_{\min}$, we adjust the $H$ to be at least that much. This may not lead to the best possible $H$, but under the constraints imposed by the lack of speculative execution, this is the best $H$ that can be achieved. See section 4.3.1 for more details on this.

The software transformation proposed by Tirumalai [45] would reverse the effect of unwinding the CFG; we don’t want this to happen. We solve this by adding one more iteration to the prologue. This has the effect of pipelining the control dependences. If we delay the scheduling of $A_1 \cdots N_1$ to the next iteration, they will be free to be scheduled in any basic block without incurring in speculative execution. Of course, delaying $A_1 \cdots N_1$ will push everything else one iteration ahead. Figure 4.8 illustrates this process.

A short example will make this more clear. Figure 4.9 shows a very simple loop with one conditional branch. The right side of the figure has the unwinding partially shown. The portion drawn corresponds to the left path of the original loop, and forms an independent loop with two basic blocks. We named them $A$ and $B$. The $p$ part of each basic block is identified and grouped in $A_p$ and $B_p$, the algorithm to compute the $p$ sets is given in figure 4.10.

---

\(^2\)Where the edges are weighted with the latency of the corresponding definition.
Figure 4.8  Software pipelining loops with early exits

Figure 4.9  Example of early exit loop
**ALGORITHM** \( \text{make}_p \): Builds all \( A_p \cdots N_p \) sets at once

**INPUT:**
Basic blocks \( A \cdots N \)

**OUTPUT:**
Sets \( A_p \cdots N_p \)

**METHOD:**
For each basic block start from the branch operation and walk the use-def chains.

For each basic block \( B \)
\[
B_p \leftarrow \emptyset
\]

For each basic block \( B \)
Identify the branch operation \( b \)
\[
w \leftarrow b
\]
while \( (w \neq \emptyset) \)
Pick \( x \) from \( w \)
\[
X_p \leftarrow X_p \cup \{x\} \quad \text{where } X \text{ is the basic block that contains } x
\]
for all uses \( u \) of \( x \)
if \( u \) is defined within the loop
\[
w \leftarrow w \cup \{u\}
\]

**Figure 4.10** Algorithm to compute the \( p \) subsets
The next step is to list schedule each of $A_p \cdots N_p$. To make this as efficient as possible, we must observe that not all operations in set $X_p$ are essential to compute the conditional branch in block $X$. There may be operations originally located in block $X$ that are used in the computation of a conditional branch in other block than $X$.

We modified the list scheduler algorithm to take operations from two different sources. For each set $X_p$, the list scheduler will first take operations that compute basic block $X$ conditional branch. If none is ready to be scheduled, then some other operation in the set $X_p$ is scheduled. Once all essentials operations are scheduled the scheduler terminates, and if there are operations remaining in the $X_p$ set, they are pushed ahead to the next block to be list scheduled. Eventually, each operation will reach a basic block where they are essential or will be scheduled in an empty slot of a previous block.

In the example of figure 4.9, we made very simple assumptions. The operations that compute the conditional branches have a latency of one cycle. The real opportunity for software pipelining is in the dependence chain $i \rightarrow j \rightarrow k \rightarrow z$. Assume that each of $i, j, k$ has latency of 2. The figure 4.11 shows how such a loop would be scheduled. The operations in boxes before the loop kernel form the prologue, the operations without boxes forms the epilogue. In the first iteration in the prologue, only operations from $A_p$ and $B_p$ are present, if the execution leaves the loop, the remaining operations of this first iteration must be executed in the epilogue.

4.3.1 The Hierarchical Reduction versus Unwinding CFG Trade-off.

The reason Tirumalai proposes hardware for eager execution is in his observation that the compound latency of sets $A_p \cdots N_p$ is usually long and often has many empty slots that can not be properly filled by the modulo scheduler.

In fact, the operations that are free to be software pipelined under early exit loops are the same ones that were list scheduled under hierarchical reduction, and the ones that are list scheduled in the sets $A_p \cdots N_p$ are the ones that are free to be software pipelined in the original loop with control flow. One tends to complement the other.

Fortunately the origin of the early exit loops is from our unwinding transformation. Thus we can try both unwinding and hierarchical reduction techniques, one at a time and also combined, and choose the schedule that leads to smaller initiation interval. The tables in section 5.3.3 show these results.
Figure 4.11  Example loop after software pipelining
4.4 Limiting the Application of the Unwinding Transformation by Code Size

The fundamental question is *How often does the compiler need to stop unwinding because of code growth?* The potential for exponential code growth always limits this kind of transformation, but a close examination in our benchmarks revealed that the expansion factor (the number of different paths in a loop body) is often a small number. Moreover, the largest expansion factors were found in the smallest programs. Taking the examined benchmarks as reference, this suggests that the code size will not be a crucial limiting factor.

Techniques to measure and limit code growth may seem somewhat arbitrary. Let's review some factors that should be considered:

**Absolute code size.** Given the size of the program, the size of each inner loop, and the number of different paths within each loop body, simple arithmetic determines the final size of the program. The compiler could compare the final size of the program to some arbitrary number, related to system capacity, and decide on the feasibility of the transformation.

**Relative code size.** Given the same inputs as above, the compiler compares the final program size before and after the transformation. There still is an arbitrary factor involved, but the user decision is somehow easier. The user could specify, for example, 50% maximum increase.

**Absolute path expansion factor.** A decision could be taken solely by the number of different paths within each loop body. The user could determine that a loop with more than, say, 100 different paths, would rarely exhibit some path locality worth exploring, and that factor would be passed to the compiler. Taking our test suite as a reference, from a total of 385 loops, we found 83 with some control flow, among those, only 6 with more than 100 different paths.
Chapter 5

Experimental Results

To validate our ideas, we designed and implemented a scheduler for ILOC capable of both regular and improved software pipelining. Extensions to the current ILOC instruction simulation must be provided to account for pipeline execution cycles, pipelining interlocks, functional units reservations, and the like. Figure 5.1 shows an overview of our experimental setup. The boxes represent programs, and floating text denotes file formats.

5.1 Obtaining Results

The existing ILOC simulator was augmented to account for more detailed instruction execution. For each simulated instruction, ic2c can generate a call to an external function, whose name is UserFunct, and pass the opcode and register names of the current operation as parameters.

The external function we provide simulates the pipeline behavior and any other necessary low level architecture detail for a given target machine. It works as a state machine where each given instruction changes the current state.

During the initialization of the state machine we read a configuration file containing several architecture parameters: a latency table, which maps each ILOC mnemonic to a integer number; a functional unit descriptor table, which details how many different types of functional units exist in the architecture and the number of units for each type; a functional unit usage table, that maps each ILOC operation to a functional unit; a register set descriptor, which defines how many different types of register sets exist and the number of registers in each set; and a register set usage table, which maps each ILOC operation to a register set type.

We assume that each ILOC operation will correspond to one or to a fixed set of target machine instructions, for which we can determine a fixed set of resource usage. Also, each register set type can be accessed by all functional units with capability of accessing that type. That way a detailed hardware usage for each ILOC operation can be idealized and encoded in the configuration file.
5.2 Compilation Process Improvements

5.2.1 Optimization Opportunities for Registers Constraint ($II_{Lam}$ vs $II_{RC}$)

Figures 5.2 and 5.3 show the results of investigating the register cycle requirements for some benchmarks. The tables compare results obtained for three CPU models: MPC601, MPC604, and the MPC604-2fp.

For each architecture, the tables show three columns. Column I is the percentage of inner loops for which the initial interval based solely on Lam’s constraints, $II_{Lam}$, is already greater than the initial interval based on register usage, $II_{RC}$. Column II
depicts the opposite situation. Note that the $II_{RC}$ computed here is the one from equation 3.4.

The last column shows the percentage of loops that cannot be allocated without spilling. This situation arises when the number of registers necessary to satisfy recurrences and globals is larger than the number of registers available in the architecture, which leaves no free registers to be used on short lived computations.

In summary, columns II and III together express the register constraint’s usefulness for improving the initiation interval estimation.

Figure 5.2 shows results for all inner loops found in the benchmarks, and results when loops with no control flow in the body are discarded. If we examine column II for each CPU model, we will notice a sudden increase from the MPC601 to the MPC604. That happens because there are a large number of initialization loops in the benchmarks. Those loops, when executed on a MPC604 or better, have an $II$ of one which is the minimum initiation interval for both the functional unit constraint and the register constraint.

Single basic block loops are easier to software pipeline, and any modulo scheduling technique can manage them well. When we exclude those loops from the statistics, we obtain the results on the second part of figure 5.2. The percentages on columns II and III suggests the relevance of considering registers constraints: 26% to 34% of loops with control flow would have their compilation time improved.

Figure 5.3 shows results for all inner loops found in the benchmarks after the unwinding transformation. Notice the larger number of inner loops. A comparison with the previous table shows a decrease on register pressure.

<table>
<thead>
<tr>
<th># loops</th>
<th>MPC601</th>
<th>MPC604</th>
<th>MPC604-2fp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>I</td>
<td>II</td>
<td>III</td>
</tr>
<tr>
<td>All loops</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FMM</td>
<td>57</td>
<td>93.0</td>
<td>3.5</td>
</tr>
<tr>
<td>SPEC</td>
<td>283</td>
<td>96.1</td>
<td>1.1</td>
</tr>
<tr>
<td>Only loops with control flow</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FMM</td>
<td>5</td>
<td>60.0</td>
<td>40.0</td>
</tr>
<tr>
<td>SPEC</td>
<td>41</td>
<td>73.2</td>
<td>7.3</td>
</tr>
</tbody>
</table>

**Figure 5.2** $II_{Lam}$ compared to $II_{RC}$ for inner loops of test programs
<table>
<thead>
<tr>
<th></th>
<th># loops</th>
<th>MPC601</th>
<th></th>
<th>MPC604</th>
<th></th>
<th>MPC604-2fp</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>I</td>
<td>II</td>
<td>III</td>
<td>I</td>
<td>II</td>
<td>III</td>
</tr>
<tr>
<td>All loops</td>
<td></td>
<td>I</td>
<td>II</td>
<td>III</td>
<td>I</td>
<td>II</td>
<td>III</td>
</tr>
<tr>
<td>FMM</td>
<td>114</td>
<td>97.4</td>
<td>2.6</td>
<td>0.0</td>
<td>89.5</td>
<td>10.5</td>
<td>0.0</td>
</tr>
<tr>
<td>SPEC</td>
<td>498</td>
<td>96.4</td>
<td>0.8</td>
<td>2.8</td>
<td>63.0</td>
<td>34.1</td>
<td>2.8</td>
</tr>
<tr>
<td>NAS</td>
<td>76</td>
<td>97.4</td>
<td>0.0</td>
<td>2.6</td>
<td>86.8</td>
<td>13.1</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Figure 5.3 \( II_{Lam} \) compared to \( II_{RC} \) after unwinding the control flow

### 5.2.2 Compilation Time for Registers Constraints (\( \Sigma II_{Lam} \) vs \( \Sigma II_{RC} \))

Figures 5.4, 5.5, 5.6, and 5.7 summarize all compilation time results. We choose to present the measurements in the form of number of MRTs scheduled and number of operations scheduled instead of time in seconds. The number of MRTs is more representative of the improvement opportunities, while the number of operations scheduled represents the actual improvement. For example, given a set of loops, suppose we find a small number of improvement opportunities (\( i.e. \) the number of MRT scheduled decreased only a little bit) but, if those opportunities represent the largest loops in the set (\( i.e. \) the MRT is a large one) the total compilation time improvement can be significant.

The tables have results for conventional hierarchical reduction modulo scheduling (sections labeled Monica Lam's Software Pipelining), the same algorithm but also including the pipelining of loops with early exits (sections labeled Software Pipelining with early exit loops), and the result of software pipelining after unwinding the control flow (sections labeled Unwinding Control Flow and Software Pipelining). For each algorithm we scheduled the same tests for three CPU models: the MPC601, MPC604, and the MPC604-2fp.

As mentioned in section 3.6, only a small number of loops were used to compute the compilation time improvement reported in this section. Starting the scheduling process with \( II_{min} = \max(II_{fu}, II_{so}) \) and software pipelining the loop assuming an infinite number of registers, \( i.e. \) using only data dependence constraints, we end up with a range of initiation intervals, namely \( II_{min} \cdots II_{dep} \). Whenever the \( II_{RC} \) computed in equation 3.4 falls in between \( II_{min} \cdots II_{dep} \), we can be sure that

\[
\frac{II_{RC} - II_{min}}{II_{dep} - II_{min}}
\]
of compilation time can be saved from the register constraint. Only a fraction of
loops will fall in this case: the columns labeled *Loops* in the figures 5.4 and 5.5 shows
the number of loops involved.

Figures 5.4 and 5.5 includes all inner loops from FMM, LINPACK, LIVERMORE,
NAS, and SPEC benchmarks. We can observe that our technique that accounts for
register in the initiation interval estimation gives a consistent compilation improve-
ment through out the tests.

Figures 5.6. and 5.7 reports only the loops involved in the computation of the
compilation time improvement. We can see that for this class of loops the impact
of the register constraint on compilation time is quite significant. Observing the
hierarchical algorithm results, the more aggressive the CPU, the larger is the number
of operations scheduled. This is particular clear in figure 5.6 between the lines for the
MPC604 and the MPC604-2fp. This is the result of data dependences being the limiting
factor during the scheduling of the MRT. Since the MPC604 and MPC604-2fp have more
functional units, the initiation interval estimations (which are based on resources)
will be smaller, but eventually the final $II$ (that also depends on data dependences)
will be the same. This data dependence factor is more accentuated in the hierarchical
reduction algorithm due the fact that false recurrences are introduced when a complex
operation representing a if-then-else construct is summarized. Dependences that were
pointing to different operations from within the if-then-else, are now pointing to the
same complex operation, possibly creating cycles.

**5.2.3 Compilation Time for Unwinding CFG ($II_{lam}$ vs $II_{ucfg}$)**

Figures 5.8 and 5.9 have essentially the same tables from figures 5.4 and 5.5 but with
lines and columns rearranged. Because the unwinding transformation duplicates code
and increases the number of loops, we expect the overall compilation to be slower.
In fact the tables show up to a 200% increase in compilation-time. Notice that the
number of MRT tables scheduled is smaller when unwinding is used. This is possible
because the UCFG algorithm does not suffer from the false recurrences problem as
mentioned in the previous section. Thus, the number of MRT tables processed are
smaller. On the other hand, the loops with control flow are the larger ones, so
after unwinding there will be more long MRTs to be scheduled which increases the
compilation time.
<table>
<thead>
<tr>
<th>CPU models</th>
<th>$II_{Lam}$</th>
<th>$II_{RC}$</th>
<th>Loops (%)</th>
<th>Reduction (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Monica Lam's Software Pipelining</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPC601</td>
<td>5283</td>
<td>5247</td>
<td>0.7</td>
<td>0.7</td>
</tr>
<tr>
<td>MPC604</td>
<td>5922</td>
<td>5883</td>
<td>0.9</td>
<td>0.7</td>
</tr>
<tr>
<td>MPC604-2fp</td>
<td>6108</td>
<td>6059</td>
<td>2.3</td>
<td>0.8</td>
</tr>
<tr>
<td><strong>Software Pipelining with early exit loops</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPC601</td>
<td>5766</td>
<td>5730</td>
<td>0.6</td>
<td>0.6</td>
</tr>
<tr>
<td>MPC604</td>
<td>6472</td>
<td>6433</td>
<td>0.8</td>
<td>0.6</td>
</tr>
<tr>
<td>MPC604-2fp</td>
<td>6647</td>
<td>6598</td>
<td>2.0</td>
<td>0.7</td>
</tr>
<tr>
<td><strong>Unwind Control Flow and Software Pipelining</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPC601</td>
<td>4557</td>
<td>4646</td>
<td>0.1</td>
<td>0.2</td>
</tr>
<tr>
<td>MPC604</td>
<td>6106</td>
<td>6086</td>
<td>0.4</td>
<td>0.3</td>
</tr>
<tr>
<td>MPC604-2fp</td>
<td>5543</td>
<td>5519</td>
<td>1.0</td>
<td>0.4</td>
</tr>
</tbody>
</table>

**Figure 5.4** Number of MRT tables scheduled (Overall compilation)

<table>
<thead>
<tr>
<th>CPU models</th>
<th>$II_{Lam}$</th>
<th>$II_{RC}$</th>
<th>Loops (%)</th>
<th>Speed-up (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Monica Lam's Software Pipelining</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPC601</td>
<td>470241</td>
<td>468359</td>
<td>0.7</td>
<td>0.4</td>
</tr>
<tr>
<td>MPC604</td>
<td>467739</td>
<td>465383</td>
<td>0.9</td>
<td>0.5</td>
</tr>
<tr>
<td>MPC604-2fp</td>
<td>533091</td>
<td>530418</td>
<td>2.3</td>
<td>0.5</td>
</tr>
<tr>
<td><strong>Software Pipelining with early exit loops</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPC601</td>
<td>486516</td>
<td>484634</td>
<td>0.6</td>
<td>0.4</td>
</tr>
<tr>
<td>MPC604</td>
<td>485399</td>
<td>483043</td>
<td>0.8</td>
<td>0.5</td>
</tr>
<tr>
<td>MPC604-2fp</td>
<td>549250</td>
<td>546577</td>
<td>2.0</td>
<td>0.5</td>
</tr>
<tr>
<td><strong>Unwind Control Flow and Software Pipelining</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPC601</td>
<td>887956</td>
<td>887560</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>MPC604</td>
<td>1612852</td>
<td>1612168</td>
<td>0.4</td>
<td>0.1</td>
</tr>
<tr>
<td>MPC604-2fp</td>
<td>1074976</td>
<td>1074253</td>
<td>1.0</td>
<td>0.1</td>
</tr>
</tbody>
</table>

**Figure 5.5** Number of operations scheduled (Overall compilation)
<table>
<thead>
<tr>
<th>CPU models</th>
<th>$H_{Lam}$</th>
<th>$H_{RC}$</th>
<th>Reduction(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monica Lam's Software Pipelining</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPC601</td>
<td>140</td>
<td>104</td>
<td>25.7</td>
</tr>
<tr>
<td>MPC604</td>
<td>201</td>
<td>162</td>
<td>19.4</td>
</tr>
<tr>
<td>MPC604-2fp</td>
<td>554</td>
<td>505</td>
<td>8.8</td>
</tr>
<tr>
<td>Software Pipelining with early exit loops</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPC601</td>
<td>140</td>
<td>104</td>
<td>25.7</td>
</tr>
<tr>
<td>MPC604</td>
<td>201</td>
<td>162</td>
<td>19.4</td>
</tr>
<tr>
<td>MPC604-2fp</td>
<td>554</td>
<td>505</td>
<td>8.8</td>
</tr>
<tr>
<td>Unwind Control Flow and Software Pipelining</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPC601</td>
<td>63</td>
<td>52</td>
<td>17.5</td>
</tr>
<tr>
<td>MPC604</td>
<td>135</td>
<td>115</td>
<td>14.8</td>
</tr>
<tr>
<td>MPC604-2fp</td>
<td>149</td>
<td>125</td>
<td>16.1</td>
</tr>
</tbody>
</table>

Figure 5.6 Number of MRT tables scheduled (Only participants loops)

<table>
<thead>
<tr>
<th>CPU models</th>
<th>$H_{Lam}$</th>
<th>$H_{RC}$</th>
<th>Speed-up(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monica Lam's Software Pipelining</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPC601</td>
<td>8246</td>
<td>6364</td>
<td>29.6</td>
</tr>
<tr>
<td>MPC604</td>
<td>10308</td>
<td>7952</td>
<td>29.6</td>
</tr>
<tr>
<td>MPC604-2fp</td>
<td>27656</td>
<td>24983</td>
<td>10.7</td>
</tr>
<tr>
<td>Software Pipelining with early exit loops</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPC601</td>
<td>8246</td>
<td>6364</td>
<td>29.6</td>
</tr>
<tr>
<td>MPC604</td>
<td>10308</td>
<td>7952</td>
<td>29.6</td>
</tr>
<tr>
<td>MPC604-2fp</td>
<td>27656</td>
<td>24983</td>
<td>10.7</td>
</tr>
<tr>
<td>Unwind Control Flow and Software Pipelining</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPC601</td>
<td>2268</td>
<td>1872</td>
<td>21.1</td>
</tr>
<tr>
<td>MPC604</td>
<td>4500</td>
<td>3816</td>
<td>17.9</td>
</tr>
<tr>
<td>MPC604-2fp</td>
<td>4636</td>
<td>3913</td>
<td>18.5</td>
</tr>
</tbody>
</table>

Figure 5.7 Number of operations scheduled (Only participants loops)

<table>
<thead>
<tr>
<th>CPU models</th>
<th>$H_{Lam}$</th>
<th>$H_{ucfg}$</th>
<th>Reduction(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPC601</td>
<td>5766</td>
<td>4657</td>
<td>19.2</td>
</tr>
<tr>
<td>MPC604</td>
<td>6472</td>
<td>6106</td>
<td>5.7</td>
</tr>
<tr>
<td>MPC604-2fp</td>
<td>6647</td>
<td>5543</td>
<td>16.6</td>
</tr>
</tbody>
</table>

Figure 5.8 Number of MRT tables scheduled (Overall compilation)
<table>
<thead>
<tr>
<th></th>
<th>$L_{Lam}$</th>
<th>$L_{ucfg}$</th>
<th>Speed-up(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPC601</td>
<td>486516</td>
<td>887956</td>
<td>-82</td>
</tr>
<tr>
<td>MPC604</td>
<td>485399</td>
<td>1612852</td>
<td>-232</td>
</tr>
<tr>
<td>MPC604-2fp</td>
<td>549250</td>
<td>1074976</td>
<td>-96</td>
</tr>
</tbody>
</table>

**Figure 5.9** Number of operations scheduled (Overall compilation)

### 5.3 Running Time Improvements

For the reasons we mentioned in the previous chapter, the unwinding transformation allows software pipelining to achieve better performance for loops with control flow. In this section we present the overall execution speed improvements and the speed-up for each inner loop with control flow.

#### 5.3.1 Optimization Opportunities and Profitability for UCFG

Figure 5.10 shows the improvements on number of operations executed on several benchmarks after the control flow unwinding optimization. The speed-ups obtained were small because we used the best optimization sequence we could find before unwinding, leaving very little room for partial dead code elimination. The column labeled *Operations moved* gives the actual number of operations moved out of the loops with this optimization. The real benefit of unwinding will be to the scheduler.

The loop profile column is divided in three parts. The column labeled I has the number of inner loops that had already only one path within the loop body, and so did not need to be processed by UCFG (354). The column II has the number of inner loops that were processed by UCFG (52 total), and column III is the number of loops that could be processed but were not due code size expansion (2 total). The limit settings used were: 50K operations or 500 paths.

These numbers implies that 12.8% of all loops are candidates for unwinding (*i.e.* they have some control flow) and that the transformation is feasible on 96.3% of them.

Figure 5.11 depicts the path locality distribution found during an instrumented execution of the benchmarks. The path locality is expressed as a percentage of hits over the total number of iterations. A branch is defined to be a hit if it jumps to the head of its corresponding loop. The first column on the table shows the number of loops with less than 50% hit rate; unwinding those loops can degrade execution time. The loops on the second column are likely to not present any improvement after software pipelining, since the number of iterations spent in the kernel is small
<table>
<thead>
<tr>
<th>FMM benchmark</th>
<th>Operations executed</th>
<th>Operations moved</th>
<th>Loop profile</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>no ucfg</td>
<td>w/ ucfg</td>
<td>Speed-up</td>
</tr>
<tr>
<td>fmin</td>
<td>876</td>
<td>876</td>
<td>0</td>
</tr>
<tr>
<td>rkf45</td>
<td>237897</td>
<td>237894</td>
<td>0</td>
</tr>
<tr>
<td>seval</td>
<td>460242</td>
<td>460208</td>
<td>0</td>
</tr>
<tr>
<td>solve</td>
<td>9430</td>
<td>9413</td>
<td>0.2</td>
</tr>
<tr>
<td>svd</td>
<td>4101</td>
<td>4101</td>
<td>0</td>
</tr>
<tr>
<td>urand</td>
<td>711</td>
<td>711</td>
<td>0</td>
</tr>
<tr>
<td>zeroin</td>
<td>3495</td>
<td>3241</td>
<td>0.7</td>
</tr>
</tbody>
</table>

| SPEC benchmark (k) | | | | 363 | 204 | 34 | 1 |
| doduc | 7299 | 7267 | 0.4 | 0 | 15 | 5 | 0 |
| fpppp | 88735 | 88648 | 0.1 | 0 | 12 | 0 | 0 |
| matrix | 9703 | 9703 | 0 | 0 | 0 | 0 | 0 |
| tomcatv | 240937 | 240920 | 0 | 11 | 1 | 0 | 0 |

| NAS benchmark (k) | | | | 1 | 17 | 4 | 0 |
| cg | 417248 | 416964 | 1 | 0 | 17 | 4 | 0 |
| ep | 28905 | 28082 | 0.3 | 0 | 0 | 1 | 0 |
| ft | 104868 | 104868 | 0 | 23 | 2 | 0 | 0 |
| is | 12697 | 12697 | 0 | 4 | 0 | 0 | 0 |
| mg | 54068 | 54066 | 0 | 20 | 1 | 0 | 0 |

**Figure 5.10** Operations executed for loops with control flow unwind optimization
compared to the prologue and epilogue overhead. Loops from column three may get
some improvement depending on the real distribution of the hits (Example: 70% of
locality means that for each 10 iterations, 7 is a hit and 3 is a miss, i.e., if the real
occurrence is HHHMMHHHHM, we will see little or no gain. On the other hand
MHMMHMHHMM can have some.) For loops from last column we expect to see some
real gain.

If we consider the last two columns, then we have 74% of all loops where unwinding
was applied displaying path locality. Note that this is a static count, closer inspection
in the profile data indicates that the loops with lower locality are the ones that execute
less iterations.

5.3.2 Overall Execution Speed Up

Since our experiments are based on a simulator, we choose to present our results in
the form of number of operations executed per CPU cycle. This format not only allows
us to do immediate comparisons on a given benchmark, but also gives insight of how
much of the hardware the benchmark uses. For example, in the figures that follows,
the EP test from the NAS benchmark, is the one with lower rate of pipelining usage.
in the other extreme is the FT test from the same benchmark.

Figures 5.12, 5.13 and 5.14 show the results for CPU models: MPC601, MPC604
and MPC604-2fp respectively. Each figure consists of simulation results for the FMM,
LINPACK, LIVERMORE, NAS, and SPEC benchmarks. Each benchmark is executed
under three different conditions: first with no instruction scheduling, the operations
are kept in the order it comes from the optimizer; second, with basic block scheduling
implemented as in Gibbons [23]; and third, with software pipeline implemented as in
Lam [32]. The relation “operations executed per CPU cycle” plotted in the graphs
were taken directly from the total number of operations simulated. Examining the
figures, one could expect more performance from software pipelining over basic block
scheduling. The main reason our results doesn’t reflect that improvement is the
lack of dependence analysis in our research compiler. That implies too conservative
SSA [18, 11] graphs and consequently higher initiation intervals.

As mentioned in section 4.3.1, there is a trade-off between hierarchical reduction
and unwinding control flow. In the next section we will show those results in de-
tail. Here we show the overall execution speeds for the CPU model MPC604 only.
Figure 5.15 consists of simulation results for the FMM, LINPACK, LIVERMORE, NAS,
and SPEC benchmarks. Each benchmark is executed under three different conditions:
first with software pipelining implemented as in Lam [32]; second, with loops un-
<table>
<thead>
<tr>
<th>Path locality</th>
<th>(&lt; 50)</th>
<th>(&lt; 70)</th>
<th>(&lt; 90)</th>
<th>(&lt; 100)</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmin</td>
<td>N.A.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rkf45</td>
<td>Only 3 iterations executed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>seval</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>solve</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>svd</td>
<td>N.A.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>urand</td>
<td>N.A.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>zeroin</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>doduc</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>17</td>
</tr>
<tr>
<td>fpppp</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>matrix</td>
<td>N.A.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tomatv</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>cg</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>ep</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ft</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>is</td>
<td>N.A.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mg</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>(\bar{r})</td>
<td>3</td>
<td>6</td>
<td>25</td>
</tr>
</tbody>
</table>

**Figure 5.11** Path locality for loops with control flow.

![Graph showing path locality for loops with control flow.](image)

**Figure 5.12** Results for the CPU model MPC601
Figure 5.13  Results for the CPU model MPC604

Figure 5.14  Results for the CPU model MPC604 with 2 float point units
wound and software pipelined with our scheme for early exit loops: and third, with Lam's software pipeline and unwind and software pipeline after that. From the figure we can observe that sometimes unwind loses to Lam's method: sometimes it wins: but the combination of both is often the best one.

5.3.3 Speed Up for Loops with Control Flow

Figures 5.16 and 5.17 details the results for loops with control flow. Figure 5.16 lists all loops with control flow from NAS, FMM, LINPACK, and LIVERMORE benchmarks that were both software pipelined by hierarchical reduction and by unwinding, so a direct comparison is possible. The next table contains the inner loops with control flow from the SPEC benchmark. For each CPU model we give three columns, the first reports the $\alpha$ achieved with hierarchical reduction, the second reports the maximum $\alpha$ achieved in each group of loops resulting of unwinding the same loop, and the third the maximum $\alpha$ achieved in each group of loops resulting of unwinding the same loop after pipelined with hierarchical reduction.

The summary of these tables is in figure 5.18. If we assume that each loop is of equal importance, we can average the execution time spent on them by just adding their initiation intervals. For each CPU model, the sum of all initiation intervals obtained from hierarchical reduction is compared to the sum of all initiation intervals obtained with unwinding, and also with the sum of all initiation intervals obtained from applying both hierarchical reduction and unwinding. Since these initiation intervals are obtained at compilation time, the ultimate choice would be to compute all three and choose the smaller one. The next two rows in the table provide the comparisons with the sum of all initiation interval obtained from the best of the first two, and from the best of all three.
Figure 5.15  Results for the CPU model MPC604
<table>
<thead>
<tr>
<th>Routine</th>
<th>Loop</th>
<th>(II)</th>
<th>(II_{ufcg})</th>
<th>(II_{u-s})</th>
<th>(II)</th>
<th>(II_{ufcg})</th>
<th>(II_{u-s})</th>
<th>(II)</th>
<th>(II_{ufcg})</th>
<th>(II_{u-s})</th>
</tr>
</thead>
<tbody>
<tr>
<td>sparse</td>
<td>1</td>
<td>28</td>
<td>22</td>
<td>20</td>
<td>14</td>
<td>19</td>
<td>18</td>
<td>14</td>
<td>19</td>
<td>18</td>
</tr>
<tr>
<td>vecset</td>
<td>1</td>
<td>11</td>
<td>7</td>
<td>8</td>
<td>11</td>
<td>8</td>
<td>7</td>
<td>11</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>gaussian</td>
<td>1</td>
<td>207</td>
<td>66</td>
<td>64</td>
<td>198</td>
<td>61</td>
<td>60</td>
<td>197</td>
<td>61</td>
<td>60</td>
</tr>
<tr>
<td>norm2u3</td>
<td>3</td>
<td>18</td>
<td>17</td>
<td>14</td>
<td>16</td>
<td>14</td>
<td>14</td>
<td>16</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>decomp</td>
<td>10</td>
<td>25</td>
<td>17</td>
<td>15</td>
<td>20</td>
<td>15</td>
<td>13</td>
<td>20</td>
<td>15</td>
<td>13</td>
</tr>
<tr>
<td>rks</td>
<td>1</td>
<td>148</td>
<td>147</td>
<td>147</td>
<td>144</td>
<td>143</td>
<td>143</td>
<td>144</td>
<td>143</td>
<td>143</td>
</tr>
<tr>
<td>decomp</td>
<td>10</td>
<td>26</td>
<td>18</td>
<td>15</td>
<td>21</td>
<td>16</td>
<td>13</td>
<td>21</td>
<td>16</td>
<td>13</td>
</tr>
<tr>
<td>dgbcf</td>
<td>3</td>
<td>103</td>
<td>94</td>
<td>96</td>
<td>98</td>
<td>80</td>
<td>87</td>
<td>98</td>
<td>80</td>
<td>87</td>
</tr>
<tr>
<td>dgbfa</td>
<td>1</td>
<td>41</td>
<td>44</td>
<td>41</td>
<td>39</td>
<td>35</td>
<td>38</td>
<td>39</td>
<td>35</td>
<td>38</td>
</tr>
<tr>
<td>dgbdi</td>
<td>1</td>
<td>44</td>
<td>38</td>
<td>37</td>
<td>37</td>
<td>25</td>
<td>26</td>
<td>37</td>
<td>25</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>30</td>
<td>29</td>
<td>31</td>
<td>29</td>
<td>23</td>
<td>23</td>
<td>29</td>
<td>23</td>
<td>23</td>
</tr>
<tr>
<td>dgeco</td>
<td>3</td>
<td>100</td>
<td>91</td>
<td>92</td>
<td>98</td>
<td>79</td>
<td>86</td>
<td>98</td>
<td>79</td>
<td>86</td>
</tr>
<tr>
<td>dgefa</td>
<td>1</td>
<td>32</td>
<td>37</td>
<td>32</td>
<td>34</td>
<td>31</td>
<td>34</td>
<td>34</td>
<td>31</td>
<td>34</td>
</tr>
<tr>
<td>dgesl</td>
<td>1</td>
<td>42</td>
<td>35</td>
<td>33</td>
<td>37</td>
<td>25</td>
<td>22</td>
<td>37</td>
<td>25</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>29</td>
<td>29</td>
<td>27</td>
<td>28</td>
<td>23</td>
<td>23</td>
<td>26</td>
<td>23</td>
<td>23</td>
</tr>
<tr>
<td>idamax</td>
<td>1</td>
<td>8</td>
<td>16</td>
<td>11</td>
<td>7</td>
<td>14</td>
<td>9</td>
<td>7</td>
<td>14</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>8</td>
<td>17</td>
<td>11</td>
<td>7</td>
<td>14</td>
<td>9</td>
<td>7</td>
<td>14</td>
<td>9</td>
</tr>
<tr>
<td>kernel</td>
<td>1</td>
<td>22</td>
<td>21</td>
<td>19</td>
<td>21</td>
<td>19</td>
<td>18</td>
<td>21</td>
<td>19</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>35</td>
<td>42</td>
<td>48</td>
<td>48</td>
<td>34</td>
<td>41</td>
<td>40</td>
<td>34</td>
<td>41</td>
<td>40</td>
</tr>
</tbody>
</table>

**Figure 5.16**  \(II\) improvement for *NAS Fmm*, *Linpack* and *Livermore* benchmarks.
<table>
<thead>
<tr>
<th>Routine</th>
<th>Loop</th>
<th>MPC601</th>
<th>MPC604</th>
<th>MPC604-2fp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$\bar{H}$</td>
<td>$H_{u \text{cfg}}$</td>
<td>$H_{s-u}$</td>
</tr>
<tr>
<td>bilan</td>
<td>1</td>
<td>52</td>
<td>60</td>
<td>49</td>
</tr>
<tr>
<td>ddeflu</td>
<td>1</td>
<td>111</td>
<td>120</td>
<td>119</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>188</td>
<td>164</td>
<td>166</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>124</td>
<td>70</td>
<td>70</td>
</tr>
<tr>
<td>debflu</td>
<td>1</td>
<td>128</td>
<td>108</td>
<td>105</td>
</tr>
<tr>
<td>debico</td>
<td>2</td>
<td>107</td>
<td>21</td>
<td>30</td>
</tr>
<tr>
<td>deseco</td>
<td>5</td>
<td>114</td>
<td>170</td>
<td>146</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>32</td>
<td>53</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>314</td>
<td>206</td>
<td>217</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>52</td>
<td>47</td>
<td>46</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>121</td>
<td>29</td>
<td>29</td>
</tr>
<tr>
<td>ihbtr</td>
<td>2</td>
<td>46</td>
<td>42</td>
<td>39</td>
</tr>
<tr>
<td>inideb</td>
<td>1</td>
<td>85</td>
<td>77</td>
<td>76</td>
</tr>
<tr>
<td>inisla</td>
<td>7</td>
<td>11</td>
<td>13</td>
<td>11</td>
</tr>
<tr>
<td>initx</td>
<td>7</td>
<td>9</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>paroi</td>
<td>1</td>
<td>80</td>
<td>16</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>45</td>
<td>29</td>
<td>29</td>
</tr>
<tr>
<td>pastem</td>
<td>1</td>
<td>55</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>prophy</td>
<td>1</td>
<td>226</td>
<td>178</td>
<td>195</td>
</tr>
<tr>
<td>twldrv</td>
<td>2</td>
<td>175</td>
<td>184</td>
<td>186</td>
</tr>
</tbody>
</table>

Figure 5.17  $\bar{H}$ improvement for Spec benchmark.
<table>
<thead>
<tr>
<th></th>
<th>( \Sigma )</th>
<th>Speed-up (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MPC601</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( II )</td>
<td>3023</td>
<td>-</td>
</tr>
<tr>
<td>( II_{ucfg} )</td>
<td>2402</td>
<td>25.8</td>
</tr>
<tr>
<td>( II_{s-u-s} )</td>
<td>2360</td>
<td>28.1</td>
</tr>
<tr>
<td>Best of two</td>
<td>2266</td>
<td>33.4</td>
</tr>
<tr>
<td>Best of three</td>
<td>2226</td>
<td>35.8</td>
</tr>
<tr>
<td><strong>MPC604</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( II )</td>
<td>2685</td>
<td>-</td>
</tr>
<tr>
<td>( II_{ucfg} )</td>
<td>2028</td>
<td>32.4</td>
</tr>
<tr>
<td>( II_{s-u-s} )</td>
<td>2036</td>
<td>31.9</td>
</tr>
<tr>
<td>Best of two</td>
<td>1931</td>
<td>39.0</td>
</tr>
<tr>
<td>Best of three</td>
<td>1896</td>
<td>41.6</td>
</tr>
<tr>
<td><strong>MPC6042fp</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( II )</td>
<td>2646</td>
<td>-</td>
</tr>
<tr>
<td>( II_{ucfg} )</td>
<td>1935</td>
<td>36.7</td>
</tr>
<tr>
<td>( II_{s-u-s} )</td>
<td>1939</td>
<td>36.4</td>
</tr>
<tr>
<td>Best of two</td>
<td>1811</td>
<td>46.1</td>
</tr>
<tr>
<td>Best of three</td>
<td>1777</td>
<td>48.9</td>
</tr>
</tbody>
</table>

**Figure 5.18** Summary of figures 5.16 and 5.17
Chapter 6

Conclusions

The primary weaknesses of software pipelining are its poor compile time efficiency and its inability to handle well loops that contain control flow.

The problem with compile time arises from the iterative nature of the technique. The speed of the software pipelining process is directly proportional to the distance between $I_{min}$ and the $I$ that actually succeeds. This thesis proposes a third constraint based on register availability for estimating the lower bound initiation interval in a modulo scheduling algorithm.

The idea of registers as a resource constraint is motivated by the fact that register pressure rises with the degree of pipelining. As $I$ decreases, parallelism is increased, and thus register pressure. Moreover, as we satisfy data dependences by interleaving one iteration between definition and use, we stretch the register lifetimes, increasing register pressure. It is shown that there are three principle components to intra-loop register pressure. Given a register pressure estimate, we come up with a limit on the $I$ below which it can be proven that no register allocation can succeed. Finally, we show that for 34% of the inner loops with control flow our new estimator predicts that either values must be spilled or the initiation interval suggested by Lam's original estimators must be lengthened.

Control flow within the loop body has been a weakness of software pipelining.

Loops with arbitrary control flow in the loop body can be dealt with by performing IF-conversion [5] to replace all branching by predicated operations. The disadvantages of predicated modulo scheduling are that it requires special hardware, and that the $I_{min}$ must be computed as if all the operations in the body of the loop are executed in every iteration, although in reality, only those along one of the control flow paths is actually executed.

Lam introduced hierarchical reduction. In her scheme, each if-then-else construct is reduced to a single node representing all the constraints of its components. Hierarchical reduction is a pessimistic approach. It doesn't allow the scheduler to mix instructions from other parts of the program. This is especially bad if the code has moderate sized basic blocks where a long chain of dependences exists and few of
them can be computed in parallel. Hierarchical reduction must reduce such sets of instructions to a single complex instruction before any parallelism can be explored by software pipelining. That way, if we have many "if"s in the loop body, hierarchical reduction will leave many holes in the schedule. This situation also hurts compile time, because the initial computation of the \( II \) doesn’t take this into account. Since it simply counts all the instructions, taking the maximum for each side of the branch. in the end, a loop body with some control flow may require many scheduler iterations to succeed, and the pipeline still may be under-utilized. Moreover, when hierarchical reduction takes place, many dependences edges are concentrated to one point. This change in the dependence graph tends to create new recurrences; the more conditional code to be reduced, the more likely and larger these recurrences will occur. The final schedule must satisfy these recurrences too, resulting in even more compilation time and larger \( II \) achieved.

Our proposal to improve software pipeline performance on loops with control flow consists of \textit{unwinding} the control flow through a series of edge splitting and block replication transformations. The idea is to explore \textit{path locality} within a loop. For each possible path through the loop body we generate a copy of the basic blocks in the path. That way each path forms a new independent loop that has only one execution path, but two or more possible exit points: one that terminates the loop, and others that branch to different paths within the same loop.

The resulting loops are then scheduled independently. If we choose to software pipeline these new loops, we must create some prologue and epilogue basic blocks between different paths. As we show in chapter 4, this technique also changes which chains of data dependences get pipelined, bringing more opportunities to find ILP.

The unwinding technique brings to light two concerns:

\textbf{Path Locality} Simulation results obtained from FMM, SPEC, and NAS benchmarks, suggest that most of the loops present locality over 90%. In particular, we expect this technique to work well in loops that contain tests for boundary conditions, error conditions, and out of range subscripts. As a side benefit, as far as the number of executed instructions is concerned, the execution of a program transformed this way should be shorter or equal to any other scheme. That is because partially dead and partially useless instructions are moved from within the loops into the epilogue or prologue blocks.

\textbf{Code Growth} This technique relies heavily on code replication. Thus, a reasonable concern is the extent to which it causes code growth. An examination
of innermost loops found in the SPEC, FMM, LINPACK, and LIVERMORE benchmarks, showed that 96% of the inner loops have at most five different paths, and 98% have ten or fewer paths. Also, no inner loop had conditional statements nested more than two deep, and 99% of the inner loops contained four or fewer conditional statements. If we impose an arbitrary limit of 50K operations or 500 paths to do the unwinding, only 2 out of 54 loops would not be unwound. These statistics suggest that the code space problems are likely to be manageable. Moreover, the application is restricted to the body of inner loops, which represents a small part of the total code.

6.1 Suggestions for Future Work

6.1.1 Estimating the $II_{\text{min}}$ under Scheduler and Allocator Interaction

Inserting spill code has several effects.

1. It adds instructions. This can increase $II_{fu}$; whether or not it does depends on the busyness of the individual functional units. If the unit that issues loads and stores did not give rise to $II_{fu}$, some spills may be, in effect, free.

2. It decreases demand for registers. This will lower $RC$ and the numbers derived from it, including $II_{RC}$.

In this case, we can view the process of choosing an initial interval as a negotiation between scheduler and register allocator. As spill code is introduced, both $\max(II_{fu}, II_{s})$ and $II_{RC}$ will converge to a point. This converging point is the one we propose as start point to software pipeline the loop.

The difference

$$R_{spilled} = R_T - (R_G + R_s + R_{s})$$

is the number of registers the scheduler will guide the register allocator, to spill for this loop before scheduling.

We propose a scheme that prioritizes registers for spilling, letting the scheduler propose a target for reducing $R_G$ and $R_s$ to achieve a number of available registers that will let scheduling proceed. In this scheme, the scheduler would compute its various estimates, invoke a restricted form of register allocator to generate spill code only, while providing it with specific targets for the number of globals, recurrences, and temporary values, and then perform the software pipelining. After scheduling all
loops, the global register allocator is invoked. This cycle might need to be repeated, either because of the inaccuracies involved in computing our various estimates, or simply because of the complexity of the underlying scheduling problem. In this case, we hope that our approach to scheduler-guided spill code insertion will help produce tighter schedules and better spill choices.

In summary, this scheme requires two different versions of the register allocator. The first would accept the number of registers to spill as a parameter, would focus on a region (the loop being scheduled) instead of the whole program, would do only spilling and not allocation, and would be called iteratively from the scheduler. The second would color the graph and do the actual allocation of registers but would never spill registers within a inner loop; instead it should fail, returning to the scheduler. The scheduler then, increments the initiation interval and start the scheduling process over again. In the second allocator, if the allocation process is done globally, then it would be necessary to recompute register allocation for all loops, pipelined or not, whenever the scheduler changes the schedule of one single loop. To avoid this situation, one would have to be content with less powerful register allocation scheme designed for use within a region, or would have to develop new techniques to do global register allocation incrementally.

### 6.1.2 Hypernode Reduction

Llosa et al [33] gives a heuristic to transform a register insensitive scheduling algorithm into a register sensitive one. The idea consists of establishing a scheduling order among the operations such that at the time a given operation is scheduled either only the successors of that operation have been scheduled or only the predecessors of that operation have been scheduled. That way, the operation in question can be scheduled either as late as possible or as soon as possible, respectively, minimizing register usage for short lived computations.

Note that this operation ordering does not invalidate the register constraint results obtained in this thesis because the latter does not depend on any particular operation order. On the contrary, minimizing register usage during scheduling has the effect of approximating the final II to the one estimated using registers constraints.

We expect that combining the register allocation scheme proposed in the previous section and modulo scheduling with the hypernode reduction heuristic has the potential to bound, in practice, the number of scheduling attempts to a small constant.
Bibliography


