Improving Software Pipelining with Unroll-and-Jam and Memory Reuse Analysis

By
Chen Ding

A THESIS
Submitted in partial fulfillment of the requirements
for the degree of
MASTER OF SCIENCE IN COMPUTER SCIENCE

MICHIGAN TECHNOLOGICAL UNIVERSITY
1996

This thesis, “Improving Software Pipelining with Unroll-and-Jam and Memory Reuse Analysis”, is hereby approved in partial fulfillment of the requirements for the Degree of MASTER OF SCIENCE IN COMPUTER SCIENCE.

DEPARTMENT of Computer Science

Thesis Advisor Dr. Philip Sweany

Head of Department Dr. Austin Melton

Date

Abstract

The high performance of today's microprocessors is achieved mainly by fast, multiple-issuing hardware and optimizing compilers that together exploit the instruction-level parallelism (ILP) in programs. Software pipelining is a popular loop optimization technique in today's ILP compilers. However, four difficulties may prevent the optimal performance of software pipelining: insufficient parallelism in innermost loops, the memory bottleneck, hardware under-utilization due to uncertain memory latencies, and unnecessary recurrences due to the reuse of registers (false recurrences).

This research uses an outer-loop unrolling technique, unroll-and-jam, to solve the first and second problems. It shows, both in theory and experiment, that unroll-and-jam can solve the first problem by exploiting cross-loop parallelism in nested loops. Unroll-and-jam can also automatically remove memory bottlenecks for loops. This research discovered that for 22 benchmark and kernel loops tested, a speed improvement of over 50% is obtained by unroll-and-jam.

For solving the uncertain memory latencies, this research uses a compiler technique, memory reuse analysis. Using memory reuse analysis can significantly improve hardware utilization. For the experimental suite of 140 benchmark loops tested, using memory reuse analysis reduced register cycle time usage by 29% to 54% compared to compiling the same loops assuming all memory accesses were cache misses.

False recurrences restrict the use of all available parallelism in loops. To date, the only method proposed to remove the effect of false recurrences requires additional hardware support for rotating register files. Compiler techniques such as modulo variable expansion [25] are neither complete nor efficient for this purpose. This thesis proposes a new method that can eliminate the effect of false recurrence completely at a minimum register cost for conventional machines that do not contain rotating register files.

Acknowledgments

I would like to thank my primary advisor, Dr. Phil Sweany, for giving me this interesting work and supporting me, both technically and emotionally, throughout my two-year study. Without him, this thesis would not be possible. I would like to thank Dr. Steve Carr for his tremendous help, encouragement and support. The close working relationship with them has taught me true sense of research and character of a computer scientist. Sincere thanks also go to my other committee members, Dr. David Poplawski and Dr. Cynthia Selfe, for their feedback and encouragement, to all other faculties in the Computer Science department, whose knowledge and support have greatly strengthened my preparation for a career in computer science, and to all my fellow graduate students, whose constant caring and encouragement have made my life more colorful and pleasant.

Being a foreign student, English would have been a formidable obstacle without the help from numerous friendly people. Dr. Phil Sweany and his wife, Peggie, have given me key help in the writing of my proposal and this thesis. My writing skill has been significantly improved by a seminar taught by Dr. Phil Sweany, Dr. Steve Carr and Dr. Jeff Coleman, and a course taught by Dr. Marc Deneire. Dr. Marc Deneire also provided me direct professional guidance on the writing of this thesis. I also want to acknowledge the valuable help, both on my written and spoken English skills, from various student coaches in the Writing Center of the Humanity department.

Special thanks go to my wife, Linlin, whose love has always been keeping me warm and happy. Last, I want to thank my parents. Without their foresight and never-ending support, I could not have reached this point.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computer scientists have been trying to improve the execution speed of programs since the advent of the first computer. Parallel computing has long been one major approach to achieving higher execution speed. One popular way to characterize program parallelism is to consider the program context for which parallelism is sought. Coarse-grain parallelism attempts to extract parallelism at the function level of abstraction. Medium-grain parallelism describes searching for parallelism at the loop level. Fine-grain parallelism refers to attempts to overlap low-level operations such as adds, multiplies, loads and stores. The MIMD (multiple instruction multiple data) model of computation attempts to find coarse-grain parallelism [18]. In an MIMD machine, each processor runs a separate process and carries out a sub-task. Each processor has its own memory, registers and program counter. Processors communicate with each other by message passing. In contrast to MIMD parallelism, ILP (instruction-level parallelism) exploits the smallest granularity in parallel execution. ILP machines have only one instruction stream, i.e. one program counter, and typically they have pipelined functional units which allow the issuing of one or more operations every cycle. Logically, all functional units share the same memory and register file.

MIMD machines are suited to exploiting large amounts of parallelism because they can be scaled to thousands of processors or more. However, programming for MIMD machines is much more difficult than programming for sequential machines and porting between MIMD machines typically requires redesign of algorithms and programs in order to achieve good performance. In addition, not all programs today have adequate parallelism to make efficient use of MIMD machines. These two factors make MIMD machines less cost-effective for general computing problems than alternative architectural paradigms such as ILP.

Instruction-level parallelism, although much more limited than MIMD parallelism, is more suited for general computing problems. Studies show that all programs exhibit a certain degree of instruction-level parallelism [43] [29]. More important, ILP is transparent to the programmer; existing sequential programs can be executed on ILP machines without programmer modifications.

Rau and Fisher define ILP as a combined hardware and software approach to exploiting instruction-level parallelism in sequential programs[37]. ILP hardware issues multiple operations every cycle and ILP compilers expose parallelism in a program so that the parallel hardware can be fully utilized. This thesis investigates ILP compiler techniques for loop optimization.

1.1 Instruction Scheduling

ILP compilers need to reorder the instructions to expose parallelism to ILP hardware because instruction-level parallelism is not evenly distributed in programs. Such reordering is called *instruction scheduling*. Instruction scheduling relies heavily on two data structures, namely the *control flow graph* and the *data dependence graph*. Therefore we delay further discussion of instruction scheduling until we define the *control flow graph* and the basic concepts of data dependence.

The control structure of a program can be represented graphically as a *control-flow graph*. Each node of the control-flow graph is a *basic block*, which is a sequence of code having only one entry at the beginning and one exit at the end. Each function is divided into basic blocks so that each instruction is in one and only one basic block. Each directed edge represents a possible direct execution path from one basic block to another basic block. Thus, a directed edge from a basic block, B_i, to another B_j means there is a possible execution path that goes from B_i to B_j without passing through some other basic block.

There are three types of data dependences among program operations: *true* dependence, *anti* dependence, and, *output* dependence. Two operations have a true dependence when a value generated in the first operation is used in the second operation. In that case, the second operation can not be started until the first operation finishes. An anti-dependence exists between two operations when the second operation changes a value that the first operation needs to use. Finally, an output dependence exists between two operations that both change the same value. Obviously, executing two operations with any of the three dependences at the same time may lead to an incorrect result. The direct purpose of instruction scheduling is to reorder program operations and group data independent operations together so that they can be executed in parallel. Anti and output

dependences are considered as false dependences because they are caused by the reuse of registers and they can be removed by register renaming.

Based upon the optimizing scope used in the scheduling, instruction scheduling can be divided into two categories, *local* and *global* scheduling. Local scheduling considers only one basic block at a time when scheduling. Because of its limited optimizing scope, local scheduling ignores the parallelism among basic blocks, that is, it cannot parallelize the execution of different basic blocks. Global scheduling expands the optimizing scope to include more than one basic block so that the execution of different basic blocks can overlap one another. In two similar experiments, Nicolau and Fisher found parallelism of 90 in average with global scheduling whereas Tjaden reported average parallelism of 1.8 within basic blocks [43] [29].

Although global scheduling can exploit parallelism among basic blocks, it does not fully utilize parallelism in loops. Because loops consist of many iterations, we want to execute in parallel not only multiple operations of a single iteration, but also operations from multiple iterations. Local and global scheduling can only exploit parallelism within one iteration (*intra-iteration parallelism*) but they cannot exploit parallelism among iterations (*inter-iteration parallelism*). In order to exploit inter-iteration parallelism, we need to parallelize not only different basic blocks of one iteration, but also different execution of the same basic blocks in different iterations. Neither local nor global scheduling can do this.

A natural extension of global scheduling in the exploitation of inter-iteration parallelism would be to unroll a loop and then globally schedule the unrolled loop body. For example, if an innermost loop was unrolled three times, there would be four iterations in the new loop body. Then global scheduling could parallelize the execution of every four iterations by scheduling the code of four iterations together. However, in such a scheme, parallelism between unrolled iterations would still not be utilized; that is, every four iterations would not be executed in parallel with the next four iterations.

To fully utilize inter-iteration parallelism, a special class of compiler techniques called *software pipelining* have been developed. In software pipelining, each iteration starts as early as possible without waiting for any previous iteration to finish. The execution of iterations overlaps with no interruption. The purpose of this research is to improve the performance of software pipelining for better loop optimization.

1.2 Software Pipelining

Software pipelining is a powerful ILP compiler technique that exploits inter-iteration parallelism by overlapping the execution of different iterations.

The performance of a software pipeline can be measured by the *initiation interval (II)*, which is the number of cycles between the initiation of two consecutive iterations. When at its full speed, a software pipeline completes one iteration every II cycles. The lower-bound of II is the upper-bound on the speed of the software pipeline. So the purpose of software pipelining is to achieve the lowest II for any given loop.

Three constraints determine the lower-bound of II for each loop on a given machine. The first constraint to II is the recurrence constraint. Data recurrences are cyclic data dependences in a loop. Operations in a data recurrence cannot be executed in parallel; therefore the speed of a software pipeline cannot be faster than sequential execution time of all operations in any data recurrence. The length of the "longest" data recurrence determines the lower-bound on II, the *recurrence constraint II (RecII)*, due to the recurrences in the loop.

A second constraint to software pipelining is the resource constraints of the target architecture. Because one iteration is completed each II cycles, the resources available in II cycles must greater or equal to the resource requirement of one iteration. The lower-bound on II determined by the limited machine resources in the machine is called the *resource constraint II (ResII)*.

The third constraint to software pipelining is the number of registers needed since a software pipeline can not use more registers than the target machine has. Software pipelines typically need many more registers than other schedules because of the overlapping executions. This overlapping effect and this register requirement increases as II gets smaller. The register constraint is a lower-bound on II due to the limited number of registers in the target machine.

These three software pipelining constraints present real limits on the performance of software pipelining. However, ineffective compiler techniques can inhibit software pipelining's effectiveness by either overestimating real constraints or by ignoring methods of mitigating software pipelining's "natural" constraints. Software pipelining constraints can be over-restrictive due to the ineffective compiler techniques. The goal of this research is to identify causes of unnecessary software pipelining constraints and eliminate the effect of over-restrictive constraints for software pipelining. Chapter 2 explains the concept of software pipelining, describes and surveys major software pipelining methods, defines and discusses recurrence, resource and register constraints, and explains a popular software pipelining technique, modulo scheduling.

1.3 Research Goals

This thesis identifies four major problems that cause unnecessary high constraints on software pipelining and investigates solutions to these four problems. This section presents these four problems, their effect on software pipelining constraints, and the approach of this research to solve the problems. The solving of these four problems can significantly reduce the unnecessary constraints imposed on software pipelining and achieve a higher software pipelining performance than has been evidenced in current software pipelining techniques.

The first problem is that the parallelism in innermost loops may be insufficient to fully utilize machine resources. In terms of software pipelining constraints, insufficient inter-iteration parallelism in innermost loops would lead to recurrence constraints higher than resource constraints. When this happens, machine resources cannot be fully utilized due to the high recurrence.

Although for single-nested loops, inter-iteration parallelism is the maximum level of loop parallelism, for multiple-nested loops, this is not the case. A multiple-nested loop contains not only multiple iterations of one loop but also multiple loops. The parallelism exists among not only different iterations of one loop but also different iterations of different loops. The parallelism among iterations of different loops is called *cross-loop parallelism*. The effect of insufficient inter-iteration parallelism in the innermost loop can be avoided if the optimizing scope of software pipelining can be expanded to outer loops to make use of cross-loop parallelism.

One approach to exploit cross-loop parallelism is to software pipeline the outer loops as well. However, the cyclic control structure, the possibly different loop count of different loops, and the recurrences in outer loops make it very difficult to successfully overlap the execution of different loops when each loop is software pipelined.

This research uses a different approach to exploit cross-loop parallelism for nested loops. It uses an outer-loop unrolling technique, *unroll-and-jam* [7][9], before software pipelining. Unroll-and-jam unrolls an outer loop and then jams the resulting inner loops back together [7] [9]. By jamming iterations of outer loops into the innermost loop, unroll-and-jam brings cross-loop parallelism into the innermost loop. After unroll-and-jam, software pipelining can be performed on iterations from multiple loops and both inter-iteration and cross-loop parallelism can be utilized by software pipelining. This use of cross-loop parallelism can eliminate the effect of insufficient inter-iteration parallelism in the innermost loop. As a result, after unroll-and-jam, recurrence constraints can be reduced to be lower than resource constraints and machine resources can be more fully utilized.

Although in theory cross-loop parallelism can be brought into the innermost loop by using unroll-and-jam, software pipelining may not be able to utilize cross-loop parallelism due to possible new recurrences created by unroll-and-jam. When jamming iterations of outer loops into the innermost loop, unroll-and-jam also brings in new data dependences. These new dependences can form new recurrences. Moreover, the length of the new recurrences can be in proportion to the degree of unrolling, which would make outer-loop unrolling ineffective. However, as we shall see in Chapter 3, steps can be taken to eliminate the effect of these additional recurrences.

The first work of this research is to investigate using unroll-and-jam with software pipelining to exploit cross-loop parallelism. Chapter 3 studies cross-loop parallelism and the effect of unroll-and-jam on software pipelining. It analyzes the possible new recurrences caused by unroll-and-jam and investigates methods to eliminate the effect of these new recurrences so that software pipelining can utilize cross-loop parallelism without restrictions.

In addition to the benefit of exploiting cross-loop parallelism, unroll-and-jam has another benefit to software pipelining, which is to automatically balance the machine resource usage in the innermost loop. This leads to the discussion of the second problem on software pipelining, the memory bottleneck. A memory bottleneck occurs when the number of memory resources, i.e. the number of load and stores that can be issued every cycle, is not adequate for the high demand for memory resources in the loop. In such a situation, the execution is delayed due to the competition for memory resources being the limiting factor on software pipelining; all other machine resources are then under-utilized.

To remove memory bottlenecks, compilers must eliminate memory loads and stores so that the memory resource demand of the loop can be in proportion to the available memory resources in the machine. *Scalar replacement,* or *load-store-elimination* is a technique that can be used to replace memory loads and stores with scalar operations [7] [5] [15] [10]. Scalar replacement catches the reuse of memory locations across iterations of the innermost loop so that instead of storing a value to the memory location in one iteration and loading the value in the next iteration, scalar replacement would use a scalar to pass the value from one iteration to another. This, in turn reduces the ratio of memory operations to other computation and allows for more “balanced" parallelization. Thus, higher performance is obtained for software pipelining.

Scalar replacement, although it catches the reuse of memory locations among iterations of the innermost loop, does not catch the reuse among iterations of different loops in a nested loop. However, unroll-and-jam can catch such reuse among iterations of different loops. Moreover, an unroll-and-jam algorithm developed by Carr and Kennedy can automatically compute the optimal degree of unrolling to remove memory bottlenecks for a given machine [9].

The effect of removing memory bottlenecks with unroll-and-jam has not been examined in the context of software pipelining. The second work of this research is to use unroll-and-jam with software pipelining, and measure the improvement in the performance of software pipelining after the removal of memory bottlenecks. One difficulty of using unroll-and-jam with software pipelining is that the prediction of register pressure of the transformed loop is more difficult. An accurate estimation of register pressure is needed for unroll-and-jam to determine an appropriate degree of unrolling. Too much register pressure would significantly degrade the performance of the transformed loop due to the high cost of register spilling. Carr and Kennedy developed a method that predicts the register pressure after unroll-and-jam and thus controls the degree of unrolling [9]. However, their formula does not consider the effect of software pipelining. This research examines the amount of additional register pressure in software pipelining and investigates a new prediction method. Chapter 3 explains how unroll-and-jam removes memory bottlenecks and investigates the register problem. Chapter 6 provides experimental evidence of the improvement obtained by unroll-and-jam and the increase in register pressure after unroll-and-jam.

The third problem of software pipelining is the hardware misuse due to uncertain latency of memory operations. The uncertainty of the memory latencies is rooted in the dramatic difference between the CPU speed and the speed of the main memory. To shorten the latency of the communication between CPU and memory (memory load/stores), one or more levels of caches are used in almost all modern computer architectures. As a result, the data to be accessed can reside in some level of cache or in the main memory. Thus the time needed for accessing the data varies depending on the location of the data. In modern machines, the latency difference between an access to the first-level cache (a cache hit) and a access to the main memory (a cache miss) is usually a factor of 10 or greater. This causes significant uncertainty.

The uncertain memory latencies can lead to severe hardware misuse by software pipelining: on the one hand, if a load that is assumed to be a cache hit turns out to be a cache miss, the software pipeline is stalled and all hardware resources wait until the cache miss is resolved; on the other hand, if a load that is assumed to be cache miss turns out to be cache hit, it occupies many more registers than it needs which causes unnecessary high register usage.

The problem of uncertain memory latencies can be solved by using an advanced compiler analysis technique, *memory reuse analysis*. Memory reuse analysis is a compiler analysis technique used to predict the cache behavior by analyzing the reuse of the data in the program. It can predict whether a memory load or store is a cache hit or a cache miss. By using memory reuse analysis, software pipelining can accurately predict the latency of memory operations. Precise knowledge of the latency of all program operations can eliminate both stalls and excessive register use in software pipelines. The generated software pipeline can have a faster execution and still make better use of available registers. Of course, this better utilization of registers reduces the register constraint of software pipelining.

The third goal of this research is to measure the benefit of using memory reuse analysis in software pipelining. In particular, since assuming all memory operations as cache misses is used by other researchers in studying register assignment for software pipelining [34] [22] [17] [16], we want to measure the decrease of register overuse attributed to assuming all memory operations are cache misses. The result of this measurement will show the significance of using memory reuse analysis with software pipelining. Chapter 4 explains the hardware reuse caused by uncertain memory latencies and Chapter 6 presents an experimental study of the register use for different memory latency models.

Finally, the fourth problem of software pipelining is that current software pipelining algorithms cannot always utilize maximum parallelism available in the innermost loop due to the existence of false recurrences. False recurrences are caused not by the true flow of values, but by the reuse of some locations, i.e. variables. False recurrences can be eliminated by variable renaming, that is, using separate variables to avoid the reuse of the same variable. However, the overlapping execution of software pipelining makes it very difficult to perform adequate and efficient renaming.

In software pipelining, traditional hardware renaming [41] is not sufficient for eliminating false recurrences because the definition of a new value may happen before the use of the old value in the overlapping execution. Traditional compiler renaming is insufficient as well in that when a value is defined in the loop body, it does not assign a different variable to hold the value produced in each iteration [14]. Although special hardware support, namely a rotating register file, can eliminate all false recurrences [13] [36] [35], most of today's machines do not have the benefit of such rotating register files.

A special software technique, modulo variable expansion (MVE)[25], was developed by Lam in order to eliminate false recurrences. But MVE has two shortcomings. First, it only handles a special class of false recurrences and cannot deal with the general case of false recurrences. For

example, modulo variable expansion [25] can not handle the case when the reused variable is not defined at the beginning of the loop. Second, MVE may unnecessarily increase the register pressure since not all false recurrences need renaming. MVE blindly eliminates the effect of all applicable false recurrences. However, if a false recurrence is not more restrictive than other constraints, it does not need to be renamed. Excessive renaming causes unnecessary register pressure since renaming consumes additional registers.

This research uses a leading software pipelining algorithm, iterative modulo scheduling [25] [33]. The final piece of this work is to develop a modified iterative modulo scheduling algorithm to (1) eliminate the effect of all false recurrence on software pipelining and (2) minimize the additional registers used by the renaming in modulo scheduling. Chapter 5 presents this new algorithm.

In summary, the major goals of this research are to improve existing software pipelining techniques to increase the amount of parallelism in the innermost loop, to eliminate or reduce the effect of the memory bottleneck and hardware under-utilization due to uncertain memory latencies, and to eliminate the effect of all false recurrences for modulo scheduling. As a result, all major software pipelining constraints — recurrence, resource and register — can be significantly reduced. Software pipelining can achieve a higher performance than currently popular methods provide.

Chapter 2

Software Pipelining

This chapter focuses on a popular loop optimization technique in today's ILP compilers, namely software pipelining. Section 1 uses an example loop to explain the basic ideas and concepts of software pipelining. Section 2 surveys and compares various software pipelining methods. Section 3 discusses software pipelining constraints due to the data recurrences in the loop and limited resources of the target machine. These constraints set a limit on the performance of software pipelining. Section 3 presents the heart of this thesis, namely our desire to eliminate unnecessary software pipelining constraints, and thus improve the performance of software pipelining. A summary of four sources of over-restrictive software pipelining constraints is provided at the end of Section 3. The last section explains in detail the algorithm of a leading software pipelining technique, iterative modulo scheduling, which is the software pipelining method used and studied in this research.

2.1 Software Pipelining Concepts

To illustrate and motivate the concepts of software pipelining, we will first investigate the problem of computing the sum of integers from 0 to 9 on a machine which has separate addition and branch functional units that operate concurrently. We will assume the latency of addition and branch is one machine cycle and that all variables are zero before the execution of this loop.

To determine whether two operations can be executed in parallel, the compiler will check if there is any data dependence between the two operations. The three types of data dependences (true, anti, output) were defined in Chapter 1.

```
x       A:      S = S + I
y               I = I + 1
z               if I<=9 goto A
```

Figure 2.1: Example Loop 1, computing sum from 0 to 9

The example loop presented in Figure 2.1 computes the sum from 0 to 9. There is an anti dependence from x to y and a true dependence from y to z. Because of these two dependences, instruction scheduling will give the schedule in Figure 2.1, where no two operations in one iteration can be executed in parallel. Thus, the execution of the loop would be:

$$x_0 y_0 z_0 x_1 y_1 z_1 ... x_9 y_9 z_9.$$

where the subscript indicates the iteration of the operation. This schedule will take 30 machine cycles. However, if we issue an iteration every two cycles and use the following schedule for each iteration:

```
a       A:      no-op
b               S = S + I
c               I = I + 1
d               if I<=9 goto A
```

then the execution of the loop will be:

$$a_0 b_0 (\{c_0 a_1\}\{d_0 b_1\})(\{c_1 a_2\}\{d_1 b_2\})...(\{c_8 a_9\}\{d_8 b_9\}) c_9 d_9,$$

where $\{\}$ means that the operations within a bracket are issued at the same cycle. The execution can also be shown as:

$$a_0 b_0 (\{c_i a_{i+1}\}\{d_i b_{i+1}\})^{i=0..8} c_9 d_9.$$

Notice the pattern of $(\{c_i a_{i+1}\}\{d_i b_{i+1}\})^{i=0..8}$. This pattern leads to a complete, software pipelined loop in Figure 2.1, which only takes 20 cycles.

The first loop schedule issues one operation per cycle because no operation can be executed in parallel within one iteration. In such a scheme, the parallelism provided by the architecture is not used due to insufficient parallelism within one iteration. The second loop schedule executes two iterations at a time. In each cycle, it issues one operation from iteration i and another operation

```
        S = S + 0
A:      I = I + 1       #       no-op
        S = S + I       #       if I<=8 goto A
        I = I + 1
```

Figure 2.2: Software Pipeline Schedule of Example Loop 1

from iteration $i + 1$. By exploiting inter-iteration parallelism, the second schedule makes a better use of the parallel hardware. It only takes 20 cycles, a speedup of 33% over the first loop schedule.

The second schedule is a software pipeline. The fixed time difference between the issuing of two successive iterations is called *initiation interval (II)*. II for this example is 2 cycles. The execution time of a loop depends on its II. In the example pipeline, execution time for the entire loop is:

$$II * N + 2,$$

where N is equal to 9. The additional two cycles in the equation is the time used to initialize and complete the software pipeline.

Because in software pipelining, the execution of multiple iterations overlaps, it uses special code constructs to initialize and drain the “software pipe". As shown in the example, a software pipeline consists of three parts.

- The loop part is called the *kernel*. It contains all the overlapped operations. In the example, it begins with the third operation in the first iteration and terminates at the second operation in the last iteration. The total is 9 iterations.
- The code leading to a kernel is the *prelude*. In the example, it is $S = S + 0$, which contributes one cycle to the execution time.
- The code after a kernel is the *postlude*. In the example, the postlude is $I = I + 1$, which contributes one cycle to the execution time.

As the execution speed of a software pipeline mainly depends on its II, the goal of all software pipelining techniques is to find the smallest feasible II for any loop on any machine. For each loop, there is a lower bound on its II. Section 2.3 will show how the lower bound II is determined by software and hardware constraints and how this research reduces these constraints.

2.2 Software Pipelining Methods

This section reviews various software pipelining methods described in the literature. Software pipelining methods can be divided into two categories, scheduling while unrolling and scheduling without unrolling which is also known as modulo scheduling. The strengths and weaknesses of these two approaches are discussed. Some major methods in each category are briefly introduced.

2.2.1 Unrolling while Scheduling

Because software pipelining overlaps the execution of different iterations, it is natural to unroll a loop and schedule multiple loop iterations until a repeating pattern appears. Then a software pipeline schedule can be obtained by simply repeating this pattern[39]. For example, if a pattern contains 4 iterations and has a length of 6 cycles, the software pipeline will achieve a peak rate of finishing 4 iterations every 6 cycles, that is, a II of $\frac{3}{2}$.

The major advantage of unrolling-while-scheduling methods is that iterations are scheduled in a natural, unrestricted way. Iterations can overlap freely and adequately. Branches are handled easily. Global scheduling techniques can also be used to help the scheduling. However, in practice, two obstacles exist to all unrolling-while-scheduling methods.

First, since pattern recognition is necessary, each method needs to remember previous states and search them for a repeating pattern. Searching for a pattern is time consuming. Moreover, a repeating pattern may never appear when different data recurrences have different lengths. Some restrictions have to be used to force a pattern. However, such forcing may create imperfect patterns.

Second, resource constraints are hard to deal with. If consideration of resource constraints is postponed until after pattern recognition, resource conflicts may degrade the performance of the software pipeline by inserting new instructions into the schedule. Alternatively, if resource constraints are considered earlier, it will significantly complicate the task of scheduler and pattern matcher. Moreover, if a pattern is not perfect, extra operations exist in the pattern and they do not represent resource requirements. This problem leads to sub-optimal resource usage in software pipelining.

Perfect Pipelining

Perfect pipelining is motivated by a global compaction method, percolation scheduling[2] and assumes hardware support of multiple branches per cycle[1]. The algorithm first performs code motion. In scheduling, infinite resources are assumed and operations are scheduled as early as possible. To force a pattern to occur, the algorithm limits the scheduling scope to a fixed number of iterations. The fixed number is determined by experiment. Once a pattern is formed, resource conflicts are solved by delaying operations as little as possible. Perfect pipelining can effectively deal with general loop bodies. Global scheduling techniques are combined smoothly. However, its method of forcing a pattern is still ad hoc and finding a pattern is expensive.

Petri-Net Pacemaker

Allan et al. use the Petri-Net model for software pipelining in part to solve the following two major problems of scheduling-while-unrolling methods, pattern generation and pattern recognition [19][31].

A Petri Net is a net of nodes where tokens travel from node to node obeying certain rules. Software pipelining can be naturally modeled as a Petri Net due to the cyclic nature of Petri Nets. In this model, each operation is a node and the issuing of a node happens when a token travels to this node. Dependences are modeled as rules that restrict the transition of tokens from one node to another. In the model of Rajagopalan and Allan [31], the pace of different recurrences are synchronized to be the same in a Petri Net, so a pattern can always be formed. By checking the state information of the Petri Net, a repeating pattern can be found relatively easily. Rajagopalan and Allan's model can also handle resource constraints and predicates in a smooth way. In comparing with other methods, Rajagopalan and Allan claimed that their model achieved the best performance in unrolling-while-scheduling methods and performance similar to the best modulo scheduling method available [32] [3].

2.2.2 Modulo Scheduling

In contrast to scheduling-while-unrolling methods which unroll a loop and schedule multiple iterations, another class of scheduling methods, *modulo scheduling*, generates a pipelined schedule for a single iteration of a loop. Modulo scheduling imposes two restrictions on software pipelines.

- 1. Each iteration uses an identical schedule.
- 2. New iterations are initiated in a fixed integral II.

As modulo scheduling reduces the freedom of scheduling, it may result in a worse performance than that of scheduling-and-unrolling methods. However, extensive experimentation has shown that modulo scheduling methods can achieve near-optimal II. No experimental evidence has been shown that a scheduling-and-unrolling method can achieve a clearly better performance than the current leading modulo scheduling methods. The following two reasons are a possible explanation for this experimental result. First, for loops that are not recurrence bounded and have no complicated recurrences, the difference in scheduling freedom is not very significant. More than half of the loops tested by Rau[33] are resource-bounded. The second possible reason is that current unrolling-while-scheduling methods cannot deal with resource constraints as effectively as modulo scheduling can. Treating resource constraints separately may result in sub-optimal performance.

A significant restriction of modulo scheduling is that II must be an integer. For example, if MinII of a loop is 3/2, then the lowest II can be achieved by modulo scheduling is 2. Unrolling cannot solve this problem when the exact achievable lower bound of II is unknown. However, scheduling-while-unrolling can naturally form a software pipeline with a non-integer II.

The handling of branches is undoubtedly the greatest weakness of modulo scheduling. Though Hierarchical Reduction[25] and Enhanced Modulo Scheduling[44] can deal with branches without special hardware support, the amount of code expansion involved can be exponential.

The major strength of modulo scheduling over scheduling-while-unrolling techniques is that modulo scheduling does not need to determine the degree of unrolling or search for a repeating pattern, which are very time consuming in scheduling-while-unrolling methods. If the restricted scheduling freedom does not prevent modulo scheduling from achieving software pipeline's performance limit, modulo scheduling is a very good choice for implementation. Also, a complete code generation algorithm of modulo scheduling methods has been published [35].

A leading heuristic-based method, iterative modulo scheduling, will be described in detail in Section 2.4 since it is the method used in this research.

Integer Linear Programming

In parallel with heuristic-based research on modulo scheduling, another work, Integer Linear Programming (LP) [21] [4], was developed under a strict mathematical formulation, namely integer programming. Though the method has a possible exponential compilation time, it can

find a software pipeline schedule that uses a given number of resources at the smallest initiation interval while minimizing the number of registers it uses. LP can also be used to generate pipelined schedules for imperfect hardware architectures. Altman et al. found that for many loops they tested, the compilation time is in seconds[4].

2.3 Software Pipelining Constraints

The II (initiation interval) of a loop is constrained by both the parallelism in the loop and the limited hardware resources. The amount of parallelism in the loop is determined by data recurrences. Hardware constraints are determined by the characteristics of a target machine.

Software pipelining constraints are important because they establish a performance limit for any loop. No matter how effective a software pipelining algorithm or implementation is, higher performance cannot be achieved without reducing the software pipelining constraints. This section introduces all major software pipelining constraints. Section 2.3.5 outlines how these constraints can be unnecessarily high due to various reasons.

2.3.1 Recurrence Constraint

The recurrence constraint determines the amount of parallelism in a loop. The computation of the recurrence constraint is relatively expensive and needs information about a special class of data dependences, called loop-carried dependences, that are data dependences between two operations of two different iterations.

The following discussion of the recurrence constraint considers data dependences only for a straight-line loop body. For a description of how control constructs can be handled in software pipelining and how control dependences are converted to data dependences, the reader can refer to [44] [45].

Recurrence Initiation Interval

The *recurrence initiation interval (RecII)* of a loop is defined as the lowest II that satisfies all recurrences of the loop. In order to describe the recurrences of a loop, we need to introduce an important program representation form, namely the data-dependence graph (DDG). It is the DDG for a loop that will enable us to compute RecII.

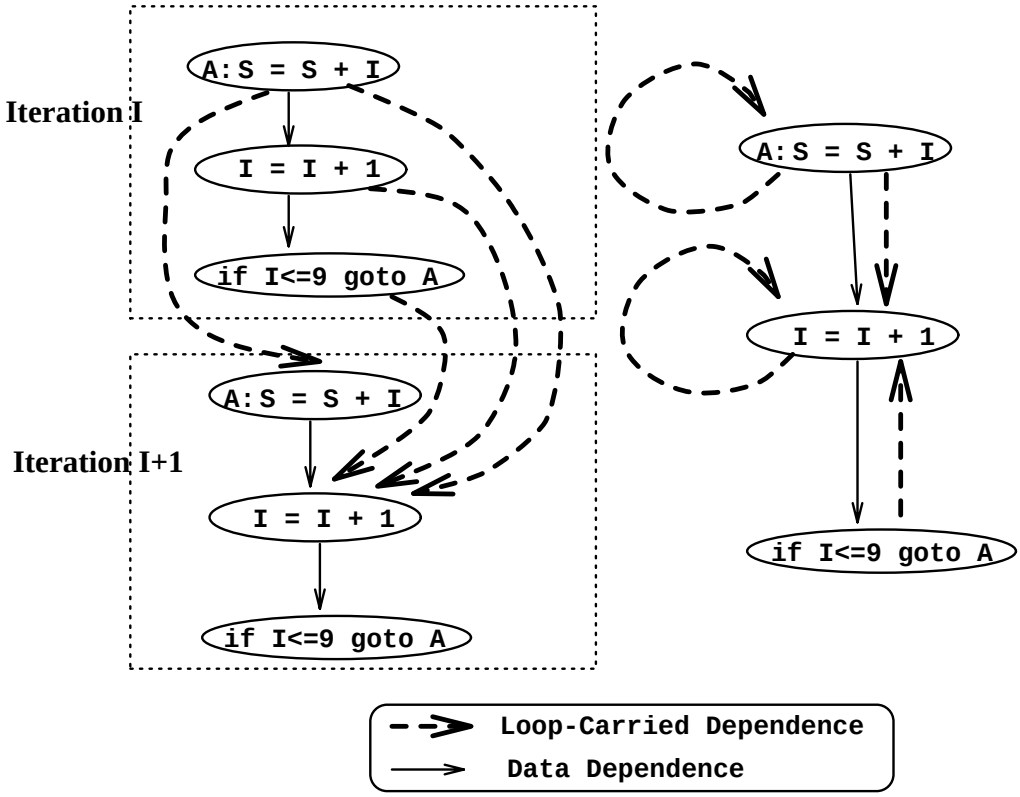


Figure 2.3: DDG of Example Loop 1

A DDG is a directed graph $G =< N, E >$, where N is a set of nodes and each node represents a machine operation. E is a set of edges, each of which represents a data dependence between the two nodes it connects. Because a loop consists of many iterations, data dependences can exist between two operations within one iteration and also between two operations from different iterations. A dependence between two operations within one iteration is a *loop-independent dependence*; a dependence that connects operations from different iterations is called a *loop-carried dependence*. The number of iterations between two operations of a dependence is the *iteration distance*. Loop-invariant dependences have an iteration difference of 0. In this thesis, each edge of a DDG is represented as a tuple <node x, node y, Dependence type d, Timing t, Iteration difference i>, where dependence type d is either true, anti or output, and timing t is the minimum number of machine cycles that the issuing of node x of this iteration must precede node y of i iterations later in order to satisfy the data dependence d.

Before software pipelining, the DDG of a loop is generated. For example, the DDG of the example loop in Figure 2.1 is shown in Figure 2.3.

If we consider only loop-independent dependences, the DDG for a loop would be acyclic; i.e. it would contain no cycles. As can be seen in Figure 2.3, however, loop-carried dependences may create cycles in a DDG. Such cycles are *loop-carried dependence cycles* since there must be at least one loop-carried dependence in each cycle. Each loop-carried dependence cycle represents a recurrence and imposes a constraint on the initiation interval. For example, consider the loop-carried dependence cycle from node x to node y and back to node x. The cycle determines that node y of this iteration must follow node x by one cycle and node x of the next iteration must follow node y of this iteration by one cycle. The recurrence of this cycle determines that the issuing of node x of the next iteration must be at least two cycles later than the issuing of node x of this iteration. That is, at most one node x can be issued every two machine cycles. For a loop execution of N iterations, the node x of the Nth iteration cannot be issued until 2 cycles after the x of the $(N-1)$th iteration, which means that II must be at least 2 cycles long.

In a loop, II cannot be less than the length of any loop-carried dependence cycle. Recurrence constraint, RecII, is the lowest II that satisfies all dependence cycles in the DDG. Intuitively, the RecII is determined by the longest loop-carried dependence cycle. When considering non-unit latencies and iteration differences, RecII is strictly defined as the largest quotient, for any recurrence cycle, of the sum of the edge timing for each edge in the cycle, divided by the sum of the iteration difference for each edge in the cycle, or more formally:

$$RecII = \max_{\forall cycle\ c \in C} \{ \frac{\Sigma_{\forall edge\ e \in c} Timing(e)}{\Sigma_{\forall edge\ e \in c} Iteration\_difference(e)} \} \tag{2.1}$$

where C is the set of all loop-carried dependence cycles in the DDG. The loop-carried dependence cycle that determines RecII is called the longest cycle in the DDG.

To calculate RecII, dependence analysis is needed to provide the information of both loop-independent dependences and loop-carried dependences. Loop dependences can also be divided into scalar dependences and subscript-variable dependences. The computing of dependences can be done by well-known dependence analysis techniques [23] [20]. However, for software pipelining, not all scalar loop-carried dependences are needed.

Pruning Loop-Carried Scalar Dependences

This section shows that, for the purpose of computing RecII and thus, for performing modulo scheduling, many of the loop-carried scalar dependences can be ignored. Minimizing the number of dependences considered can improve the compilation time while guaranteeing cor-

rect code generation. The following lemma and four theorems describe several forms of scalar dependences that can be ignored.

The first lemma shows that if there is a loop-carried scalar dependence with an iteration distance of k, where $k > 1$, there must be the same kind of loop-carried dependence of the same type with an iteration distance of one. This lemma will be used to show that all loop-carried scalar dependence with an iteration distance of more than one can be ignored. In the following text, a loop-carried scalar dependence with an iteration distance of one is called a *unit loop-carried dependence*; any other loop-carried dependence is called a *non-unit loop-carried dependence*.

Lemma 1 *If there is a scalar dependence, <node x, node y, Dependence Type d, Timing t, Iteration Difference i>, where i is greater than one, then there must be a dependence of the same type from node x to node y with timing t and iteration difference of exactly one. That is, there must be a dependence $< x, y, d, t, 1 >$ in the DDG.*

Proof: The scalar values that are used and defined are the same for each operation of all loop iterations. If there is a dependence d between node x of iteration k and node y of iterations $k + i$, then there is a dependence d between node x of iteration k and node y of iteration $k + 1$. That is, a dependence $< x, y, d, t, 1 >$. □

The first theorem shows that ignoring non-unit loop-carried scalar dependences in computing RecII does not affect the accuracy of RecII. The length of a recurrence cycle that contains any non-unit loop-carried dependence must be less than RecII; therefore, recurrence cycles containing non-unit loop-carried dependences can be ignored.

Theorem 1 *In computing RecII, all non-unit loop-carried scalar dependences can be ignored.*

Proof: Define function f to be a mapping from a loop-carried dependence cycle to an integer.

$$f(cycle\ c) = \frac{\Sigma_{\forall edge\ e \in c} Timing(e)}{\Sigma_{\forall edge\ e \in c} Iteration\_difference(e)}$$

From Formula 2.1, we have

$$RecII = \max_{\forall cycle\ c \in C} \{f(c)\}$$

To show that any dependence cycle having any non-unit loop-carried scalar dependence can be ignored, we will show that for any such cycle c_1, there exists another loop-carried dependence cycle c_2 that has only unit loop-carried scalar dependence, and $f(c_2) > f(c_1)$. Let's assume that a loop-carried dependence cycle c_1 has non-unit loop-carried scalar dependences. According to Lemma 1,

for each such loop-carried scalar dependence, there is a corresponding unit scalar dependence. By replacing each non-unit loop-carried scalar dependence with the corresponding unit dependence, we form a loop-carried dependence cycle, c_2, that must also exist in the DDG of the loop. The numerator of $f(c_1)$ is same as the numerator of $f(c_2)$, however, the denominator of $f(c_2)$ is smaller than that of $f(c_1)$. So $f(c_2) > f(c_1)$. Therefore, non-unit loop-carried scalar dependences can be ignored in computing RecII. □

The second theorem shows that non-unit loop-carried scalar dependences can also be ignored for a certain class of scheduling methods, including modulo scheduling, because in these scheduling methods, non-unit loop-carried scalar dependences are shadowed by unit loop-carried dependences.

Theorem 2 *Any scheduling method that always schedules a node of iteration k before the same node of any later iteration can ignore non-unit loop-carried scalar dependences.*

Proof: By Lemma 1, for each non-unit loop-carried scalar dependence, there is a corresponding unit dependence. We will show that any loop-carried scalar dependences with a non-unit iteration difference is satisfied if the corresponding unit loop-carried dependence is satisfied.
Assume a dependence <node x, node y, d, t, i> where $i > 1$. For any iteration k, according to this dependence, the issuing of node y of iteration $k + i$ must be t cycles later than the issuing of node x of iteration k. Because the scheduling method will observe the dependence <node i, node j, d, t, 1>, the issuing of node y of iteration $k + 1$ will be t cycles later than the issuing of node y of iteration k. Because the issuing of node y in any later iteration of $k + 1$ will be no sooner than the issuing of node y in iteration $k + 1$, dependence <node x, node y, d, t, i> will be satisfied. □

Current compiler renaming techniques can eliminate loop-independent scalar output dependences at least for loops of a single basic block. When the loop body does not have loop-independent output dependences, we may also ignore loop-carried scalar output dependences in modulo scheduling if the II is more than the timing of the output dependence.

Theorem 3 *Assume t_{output} is the timing constraint of the scalar output dependence. If the scheduling method always issues a node of a iteration k t_{output} cycles before the same node of its next iteration, and if the loop does not have loop-independent output scalar dependences, then the scheduler can ignore all loop-carried scalar output dependences.*

Proof: A loop-carried scalar output dependence is either between the same node of two different iterations or two different nodes of two different iterations. That is, it is either

<node x, node y, $output$, t_{output}, i> or <node x, node x, $output$, t_{output}, i>, where $x \neq y$ and $i = 1$ (from Theorem 2). For the first case, there must be a loop-independent scalar dependence $< x, y, output, t_{output}, 0 >$. From the condition, this case is impossible. For the second case, from the condition, the x of next iteration won't be issued until $i * t_{output}$ cycles after the issuing of the x of current iteration. So the dependence will hold. □

Similar to the last theorem, it is easy to prove that when there is no loop-independent output dependence and RecII is greater than the timing of an output dependence, the loop-carried output dependence can be ignored in computing RecII. The reason is that loop-carried scalar dependences do not form recurrence cycles other than themselves.

Theorem 4 *In computing RecII, all loop-carried scalar output dependences can be ignored if* $RecII > t_{output}$ *and if there is no loop-independent scalar output dependence.*

Proof: Because there is no loop-independent output dependence, every loop-carried output dependence must start and finish at the same operation of different iterations. Then, all loop-carried dependence cycles which contain a loop-carried output dependence must be a cycle with only one edge, the output dependence. The length of all such cycles is less than or equal to t_{output}. Therefore, if the $RecII > t_{output}$, we can ignore all loop-carried output dependences.

From the theorems introduced above, the non-unit scalar loop-carried dependences and all scalar output dependences can be ignored in modulo scheduling. With either special hardware support of rotating registers or the compiler technique described in Chapter 5, the effect of both loop-independent and loop-carried anti dependences can be eliminated as well. Then, for modulo scheduling, the scalar dependences that cause recurrences include only loop-independent and unit loop-carried scalar true dependences.

2.3.2 Resource Constraint

Recurrence constraints reflect the maximum parallelism in a loop. However, limited resources in a machine may prevent a loop from using all the available loop parallelism. *Resource constraints (ResII)* define the limitation on initiation interval imposed by limited machine resources. For a loop with a ResII higher than RecII, the loop parallelism cannot be fully exploited due to the insufficient machine resources because the ResII will limit the minimum II. This section defines ResII and explains how the lack of memory functional units and the intense memory resource requirements in loops can cause memory bottlenecks and waste machine resources.

Resource Initiation Interval (ResII)

For modern machines with pipelined functional units, the resource limit can be seen as the number of machine operations that can start execution in a machine cycle. For example, if a machine can issue two memory load/store operations per cycle, we say that the machine has two memory resources. Other typical resources include integer arithmetic, floating point arithmetic, and branch functional units.

The resource requirement of the loop needs to be calculated before computing the resource constraint. From the DDG of a loop, it is not difficult to find the resource requirement of each iteration. Each node in a DDG represents an operation and its resource requirement. The total resource requirement of one iteration can be obtained by simply summing the resource requirements of all DDG nodes.

Given the available resources on a machine and resource requirement of one loop iteration, the resource constraint on initiation interval, ResII, is computed as shown in Formula 2.2.

$$ResII = \max_{\forall resource\ r \in machine} \{ \frac{\Sigma_{\forall node\ n \in DDG} resource\_requirement_n(r)}{available(r)} \} \qquad (2.2)$$

For example, consider a loop that requires 4 floating point multiplies, 8 floating point adds, 10 integer operations and 6 memory operations to complete the execution of a loop iteration. If we are pipelining such a loop for a machine that can start 1 floating point add, 1 floating point multiply, and 2 integer operations each cycle and that can start a memory operation every other cycle, then ResII would be 12, because the limiting resource would be the ability to schedule the memory operations. The individual resource limits for this example would be:

- 4 cycles needed for floating multiplies since 4 floating multiplies in the loop divided by 1 floating multiply started per cycle equals 4.
- 8 cycles needed for floating adds since 8 floating adds in the loop divided by 1 floating add started per cycle equals 8.
- 5 cycles needed for integer operations since 10 integer operations divided by 2 integer operations started per cycle equals 5.
- 12 cycles needed for memory operations since 6 memory operations divided by .5 memory operations started per cycle equals 12.

$ResII = \max\{4, 8, 5, 12\} = 12\ cycles$; the maximum constraint for any resource is the 12 cycles required for the issuing of all memory operations.

Memory Bottleneck

Most modern architectures allow initiation of more *computational* operations (floating-point, integer, or branches operations) than memory operations (loads, stores) in each machine cycle. This is due in part to the hardware difficulty in providing greater memory bandwidth and in part to the expectation that less than 20% of a program's executed operations will be memory operations. In loops, however, loads and stores often constitute considerably more than 20% of the operations because array computations typically dominate loops and code generation for arrays requires large numbers of loads and stores. Thus, we often see a mismatch between the resources required for a loop and those provided by the hardware. When such a mismatch occurs, memory resources become a bottleneck and other resources become under utilized. Consider the example loop and the machine given in last section, where the resource constraint is determined by memory resource constraint. Assume the loop can be executed at an II equal to ResII, which is 12 machine cycles. We can see that in each 12 machine cycles, 8 cycles of floating-multiply resources, 4 cycles of floating-add resources and 7 cycles of integer-operation resources are wasted, because there are insufficient computation operations to balance the requirement for memory operations. This is not an unusual situation.

Resource Requirement of Loops with Conditionals

In loops with conditionals, there are multiple possible execution paths. The resource requirement is the one having the heaviest resource usage, which is defined by Formula 2.3

$$ResII = \max_{\forall execution\ path\ p \in loop} \{ \max_{\forall resource\ r \in machine} \{ \frac{\Sigma_{\forall DDG\ node\ n \in path\ p} resource\_requirement_n(r)}{available(r)} \} \} \tag{2.3}$$

2.3.3 Interaction between Resource and Recurrence Constraints

Given ResII and RecII, a lower bound on initiation interval, MinII, can be computed as in Formula 2.4.

$$MinII = \max\{ResII, RecII\} \tag{2.4}$$

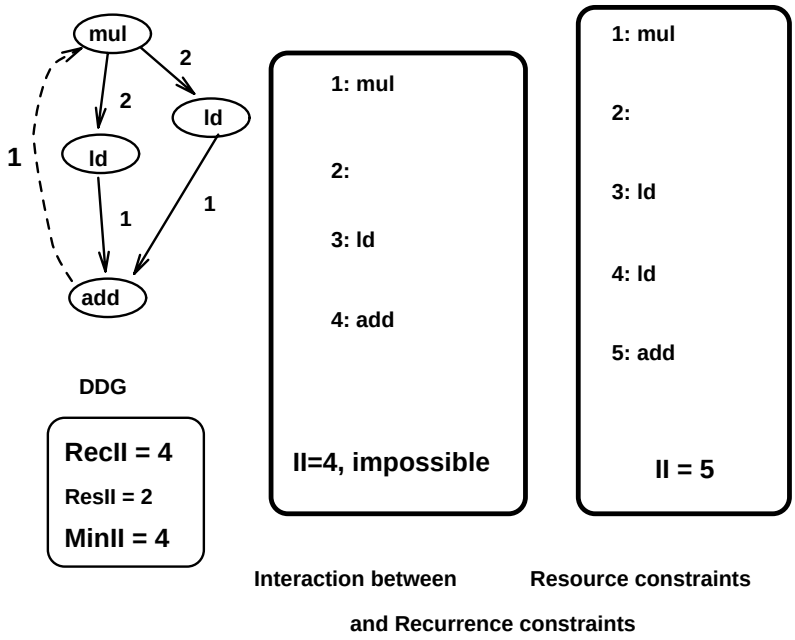


Figure 2.4: Interaction between Resource and Recurrence Constraints

Ideally, the software pipeline of a loop can executed at an initiation interval of MinII. However, even using an exhaustive search, we may still not find a schedule with an II equal to MinII. A major reason is the interaction between resource and recurrence constraints.

Let's consider the example DDG in Figure 2.4 and assume the machine can issue one memory load and two arithmetic operations per cycle. According to Formulas 2.1 and 2.2, RecII is 4 cycles and ResII is 2 cycles. So MinII should be 4 cycles. However, it can be proved by contradiction that it is impossible to find a schedule with an II of 4. Assume there is a schedule with an II of 4 and the multiply operation is scheduled at cycle i. Because the multiply operation of the next iteration will be started at cycle i+4, the recurrence determines that the addition operation must be scheduled at i+3 and two load operations must be scheduled at the same cycle i+2. However, scheduling two loads in one cycle violates the memory resource constraint. So II of 4 cycles is impossible. Instead, the lower bound of II for this loop is 5 machine cycles.

As can be seen from Figure 2.4, the interaction of recurrence and resource constraints may result in a higher constraint than either resource or recurrence constraints. The interaction is triggered by high recurrences. In the example, the recurrence dominates the constraint on the initiation interval. When recurrence is higher than other constraints, the operations on the longest

loop-carried dependence cycles have one fixed position relative to others in the schedule. When a set of operations, which are determined to be scheduled in one machine cycle by a recurrence, violate the resource constraint, an II of RecII can not be achieved.

If a loop has a higher ResII, then the interaction will be reduced and even eliminated. Consider the operations in the longest loop-carried dependence cycle. Each operation has more freedom when ResII is greater than RecII. When operations have more possible positions instead of only one position, resource conflicts are more likely to be solved.

The key to reduce the interaction between recurrence and resource constraints is to reduce the restriction of recurrences on scheduling so that all machine resources can be more fully used. Ideally, if no recurrence exists, the ResII can always be achieved and no interaction could exist. In practice, we cannot completely eliminate all recurrences. However, we can often reduce the interaction by making the recurrence constraint much lower than the resource constraint. The lower the recurrence, the more freedom the scheduler has in scheduling resources. Therefore, a relatively low recurrence constraint compared to resource constraint is desirable in order to reduce the effect of interaction between recurrence and resource constraints.

2.3.4 Register Constraint

The *register constraint* of software pipelining is due to the fact that the number of registers needed in a software pipeline cannot exceed the number of registers available on the machine. Register constraints can be represented as the smallest II possible given a fixed number of registers. If we know the minimum need of registers of software pipeline schedules of any II, we can find the register constraint. However, determining the minimum register need for a given II or a given loop is difficult because:

- We need to find the software pipeline schedule that requires the lowest number of registers. However, there could be exponential number of possible schedules for a given II.
- Given a software pipeline schedule, finding an optimal register assignment is NP complete.

Various methods have been proposed to circumvent the above difficulties and find a good estimation for the register constraint. Two estimation methods, MaxLive, which is independent of register assignment methods[35], and, MinAvg, which is independent of scheduling [22], are introduced in this section.

MaxLive is a lower bound on the register demand of a given schedule. At any point of the given schedule, there are certain number of values that are live at that point. Because no two of them can share a register, the register need at that point must be at least the number of values that are live. MaxLive is is defined as the maximum number of values that are live at any cycle of a given software pipeline schedule, as in Formula 2.5.

$$MaxLive = \max_{\forall cycle\ t \in schedule} \{number\ of\ values\ living\ at\ cycle\ t\} \tag{2.5}$$

MaxLive may not be an accurate estimate because the minimum number of registers needed for a schedule can be higher than MaxLive. However, MaxLive can serve as a fast estimation, since it is typically computed before doing any register assignment. Rau, et al., claimed that MaxLive is fairly accurate because for loops they tested, their best register allocation algorithm achieved an average of 1.3 registers more than MaxLive [35].

Due to the reason that computing of MaxLive must be based on a given schedule, there are two disadvantages,

- The scheduler cannot compute MaxLive before scheduling. The scheduling effort may be wasted if it turns out that the schedule uses too many registers. However, the waste of scheduling time can be avoided if the register demand can be estimated before scheduling.
- MaxLive is only a lower bound for a given schedule. However, other schedules of the same II may require fewer number of registers. So when MaxLive for some schedules exceeds the number of available registers, we cannot conclude that no schedule is possible under the register constraint.

To avoid the two disadvantages of MaxLive, Huff proposed a schedule-independent lower bound on register requirement for a given II, MinAvg [22]. MinAvg is computed based on the lower bound on the length of all lifetimes in the loop. Because each register in II cycles can only support a lifetime of length II, the lower bound, MinAvg, is defined as the lower bound on the total length of lifetimes divided by II, as in Formula 2.6.

$$MinAvg = \frac{\Sigma_{\forall value\ v\ \in loop} MinLT(v)}{II} \tag{2.6}$$

where $MinLT(v)$ is the minimum lifetime of variable v. The minimum schedule-independent lifetime for each value is computed using MinDist (see Section 5.2) [22]. The sum of the minimum lifetime of all variables is MinLT.

2.3.5 Sources of Over-Restrictive Constraints

There are four ways in which the major software pipelining constraints, recurrence, resource, and register constraints, can be over-restrictive:

- limited optimizing scope of software pipelining,
- the memory bottleneck,
- hardware under-utilization due to the uncertain memory latencies,
- and, false recurrences in the loop.

Recurrence constraints can be unnecessarily high for either of two reasons. First, the optimizing scope of software pipelining is limited to the innermost loop. It ignores the outer loops in a nested loop, which may have additional parallelism. Second, current software pipelining techniques cannot eliminate all false recurrences without the special hardware support of rotating registers. These false recurrences impose unnecessary recurrence constraints on software pipelining.

High recurrence constraint also triggers more performance degradation due to interaction between recurrence and resource constraints, which causes a larger II than necessary. If the recurrence constraint is unnecessarily high, the interactions will also be unnecessarily high.

The resource constraint can be very restrictive in the situation of the memory bottleneck. Although memory resources are fully utilized, other resources remain idle. This obviously wastes available hardware parallelism.

Register constraints can be over-restrictive if software pipelining uses a higher latency for memory operations than is actually necessary. Many researchers assume that all memory operations are cache misses in order to avoid execution time penalty when assuming all memory operations are cache hits. But, when assuming all memory operations are misses, values may occupy registers much longer than necessary. Because of the large latency difference between a cache miss and a cache hit, the over-estimation of memory latency can lead to a considerably over-restrictive register constraint.

2.4 Iterative Modulo Scheduling Steps

Iterative modulo scheduling is a popular software pipelining technique that has been extensively studied in both academic and in industrial settings. As do other modulo scheduling

approaches, iterative modulo scheduling combines the consideration of recurrence and resource constraints in scheduling. It is a heuristic-based method that is practical for use in a general purpose compiler. Its scheduling, code generation and register assignment algorithms have been well-studied [25] [33] [22] [35] [44] [34] [16]. Several experimental evaluations have shown that the iterative modulo scheduling method can achieve near-optimal performance for a large number of benchmark loops [33] [22] [16]. No other heuristic-based software pipelining algorithm has been shown to have a better performance than iterative modulo scheduling. Therefore, we have chosen iterative modulo scheduling as the basis for our software pipelining. Chapter 6 presents evaluation results obtained by using iterative modulo scheduling. Chapter 5 describes two improvements to the current iterative modulo scheduling technique.

This section describes, in more detail, the basic iterative modulo scheduling steps. The techniques introduced are an assembly of previous work on iterative modulo scheduling. The first step of iterative modulo scheduling is if-conversion, which converts conditional constructs into straight-line code. Modulo scheduling can only schedule a straight-line loop body; therefore loops containing branches must be if-converted into straight-line code. The second step is iterative scheduling where a software pipeline schedule is found. This step considers both recurrence and resource constraints. The third step, after a software pipeline schedule is found, unrolls the kernel to avoid the conflicts due to the overlapping lifetime of some variables. This step makes modulo variable expansion feasible. The next step is code generation. The prelude and postlude are generated at this step. For if-conversioned loops, this step needs to recover the conditional constructs. The last step is register assignment. Register assignment in modulo scheduling has been an area of active research and many questions remain unsolved. The last section gives a relatively detailed survey of several register assignment approaches used with iterative modulo scheduling.

Special hardware supports can eliminate the need for one or more of the above steps. These hardware features are mentioned where needed; but complete descriptions of them are beyond the scope of this thesis.

2.4.1 If-conversion

If-conversion is a technique used to generate branch-free loop bodies. It adds a predicate to hold the condition for the execution for each operation.

The following example shows the idea of if-conversion.

```
if (i>0)
        A = TRUE;
else
        A = FALSE;
```

To deal with the branch statement in the program, if-conversion uses a predicate P to indicate the result of the condition in execution. The value of P is p if the condition is true and the value is $\overline{p}$ if the condition is false. The two assignment operations each holds a value of P. Each operation is executed if the value of P in execution is the same as the value it holds. The code after if-conversion is:

if (i>0) P = p; else P = $\overline{p}$

(p) A = TRUE;

($\overline{p}$) A = FALSE;

Some architectures support if-conversion by providing dedicated predicate registers (PRs). For conventional architectures that do not have PRs, a special code generation scheme, reverse if-conversion was developed for Enhanced Modulo Scheduling [44]. Reverse if-conversion generates all possible execution paths for a software pipeline after scheduling.

To understand reverse if-conversion, assume a single branch construct is denoted as $\{A or \overline{A}\}$, where A is the branch target basic block and $\overline{A}$ is the fall through basic block. For example, one possible loop is $\{A or \overline{A}\}B$, where B is scheduled to be overlapped with $\{A or \overline{A}\}$ in the software pipeline. Reverse if-conversion will generate software pipeline code as $\{AB or \overline{A}B\}$. If the loop body is $\{A or \overline{A}\}\{B or \overline{B}\}$, the generated software pipeline will be $\{AB, A\overline{B}, \overline{A}B, or \overline{AB}\}$. A limitation of reverse if-conversion is the possible exponential code explosion that can result. When there are n conditional constructs overlapped, 2^n paths need to be generated.

2.4.2 Iterative Scheduling

Iterative modulo scheduling assumes that before this step, a loop body has been converted into straight-line code, and all data dependences and resource requirements have been specified in the DDG. First, RecII, ResII and MinII are computed and rounded to an integer. Then the scheduler will try to schedule the loop body for MinII. If it fails, II is increased and scheduling is repeated

until an II is found. This iterative process will eventually finish since II cannot be higher then the length of a locally scheduled loop.

For each trial II, timings of loop-carried dependences are computed using the trial II. Following the restrictions of modulo scheduling, when a node x is scheduled at time t, the node x of iteration i will be issued at time $(i-1)II + t$. Because the scheduler knows the issuing time of all DDG nodes of all iterations, it can ensure that all loop-carried dependences are satisfied.

To ensure that the resource constraints are met, the scheduler uses a *resource reservation table (RRT)* of length II. Each slot contains all machine resources available in one cycle. The resource requirement of every scheduled node is placed into the resource reservation table to ensure no resource conflicts. A conflict-free RRT will guarantee a conflict-free software pipeline. For an if-converted loop, the resource reservation table will consider the effect of branches by allowing operations of the same iteration but disjoint execution to share a resource[44].

2.4.3 Kernel Unrolling and Code Generation

After a schedule and an II have been found, some degree of kernel unrolling is needed because loop variants that have a lifetime longer than II will cause cross-loop value conflicts in the overlapped execution of the software pipeline. (The lifetime of a variable is the time between it first definition and its last use in the schedule of one iteration.) Consider an example loop $\{A_{i+2}B_{i+1}C_i\}^{i=1..n}$. If a variable is defined in A, used in C, and has a lifetime longer than II, for any i, the value defined in A_i can not be used by C_i because it has already been changed by A_{i+1}.

To solve this problem of overlapping variable lifetimes, some have proposed a hardware solution, namely rotation registers (RRs) in which a machine can use multiple physical registers for one value. For conventional machines that do not have RRs, a compiler technique called modulo variable expansion [25], or kernel unrolling, can be used to solve this problem. After the schedule of one iteration is found, a certain degree of kernel-unrolling is performed. The least amount of unrolling is determined by dividing the longest variable life time by II.

In the above example, modulo variable expansion will unroll the loop body 3 times. The resulting software pipeline will be:

$A_0\{B_0A_1\}$ (prelude)

$(\{A_{i+2}B_{i+1}C_{i+0}\}\{A_{i+3}B_{i+2}C_{i+1}\}\{A_{i+4}B_{i+3}C_{i+2}\})^{i=0..k}$

$\{A_{3k+2}B_{3k+1}C_{3k}\}\{B_{3k+2}C_{3k+1}\}C_{3k}$ (postlude),

where k is one less than the integer part of n divided by 3. The new II will be 3 times the original II.

An obvious concern here is what to do with the rest of iterations when loop count n is not a multiple of 3. One solution is loop conditioning, which adds an appropriate number of non-pipelined iterations before the prelude (preconditioning), or after the postlude (postconditioning) to ensure that the new kernel *will* be executed k times where k is a multiple of the unroll factor. If n in the above example is 20, a non-pipelined schedule of loop that has a loop count 2 needs to be added either before the prelude or after the postlude of the software pipeline.

When there are multiple lifetimes that are greater than II and each lifetime requires a different degree of unrolling, the degree of kernel-unrolling can be determined by two different methods[25]. One needs a larger number of unrolling (code expansion) but requires fewer registers; the other requires more registers but less unrolling.

After kernel unrolling, the code generation step generates prelude, kernel and postlude. Pre-conditioning or post-conditioning is needed for DO-loops if the kernel is unrolled. For WHILE-loops, special techniques have been developed by Rau et al[35].

If a loop has undergone if-conversion and the machine does not have hardware support for predicate execution, the branch constructs must be recovered in the code generation step. Warter et al. describes this technique in [44].

2.4.4 Register Allocation

Current popular register assignment methods in ILP compilers are based on a graph coloring formulation in which each scalar is a graph node, and each edge indicates that the lifetimes of two values connected by the edge overlap with each other. So the register assignment problem becomes a node-coloring problem where we want to use the fewest number of colors (registers) to color all nodes such that no adjacent nodes share a color.

To minimize the register usage of software pipelines, a scheduling method should choose a schedule that attempts to conserve registers. However, it is difficult for scheduling methods to estimate register usage or to predict the result of register assignment because until after scheduling, complete information about register lifetimes is unavailable. This section introduces three heuristic-based register-assignment approaches for software pipelining. For optimal methods, readers are referred to [17] [21].

Register Assignment after Register-Need-Insensitive Scheduling

Rau et al. investigated the register assignment problem of software pipelining[34]. They studied various post-schedule register assignment methods. But they did not try to reduce the register need in the scheduling step.

They tried different heuristics on both ordering lifetimes and assigning lifetimes to available registers. The three lifetime-ordering heuristics they tested are: ordering by starting time, ordering by the smallest time difference with the previously scheduled one, and ordering by the number of conflicts with other lifetimes. To map a lifetime to a register, there are also three heuristics, best fit, first fit, and last fit.

For traditional machines that do not have rotating registers, the best heuristic, which generates the best code and uses the least compile time, is ordering lifetimes by the number of conflicts and mapping lifetimes to registers by first fit. In Rau et al.'s experiment, the best heuristic used the same number of registers as MaxLive for 40% loops; and on average, it required 1.3 more registers than MaxLive.

This work presented two notable results. First, the best heuristic is similar to graph-coloring. So a general graph-coloring register assignment algorithm would probably do as well as their best heuristic. Second, the measurement used for the lower bound on register demand, MaxLive, is not necessarily the lower bound for the loop. MaxLive is dependent on a particular schedule. It is possible that another schedule could have a lower MaxLive. Because the scheduler used in Rau et al.'s research was not sensitive to register demand, it did not try to find that schedule with the lowest MaxLive.

Register Assignment after Lifetime-Sensitive Scheduling

Huff [22] proposed a schedule-independent lower bound for register requirement. He also modified iterative modulo scheduling so that the scheduling step attempts to minimize the lifetime of values. The schedule-independent lower bound, MinAvg, has been described in Section 2.3.4. When an operation is scheduled in Huff's scheduling method, a set of possible time slots are identified. The heuristic favors a placement as early as possible in order to avoid lengthening lifetimes involved in this operation.

Huff's experimental results showed that 46% of loops achieved a MaxLive equal to MinAvg and 93% of loops had a MaxLive within 10 registers of MinAvg. In his experiment, hardware support of rotating registers is assumed [22]. For conventional hardware, the register need is probably higher due to the effect of kernel unrolling.

Stage Scheduling to Minimize Register Requirement

Recently, a set of heuristics were developed by Eichenberger and Davidson [16] to find a near optimal register assignment. Their method, called *stage scheduling*, is a post-pass scheduler invoked after a software pipeline schedule has been found. They define a *stage* to be II cycles. In stage scheduling, operations can only be moved in an integral number of stages (thus a multiple of II cycles). For any given software pipeline schedule, there is an optimal stage schedule that requires the fewest registers among all possible schedules. Eichenberger and Davidson showed that

their heuristics achieves a performance only 1% worse than the optimal stage schedules. However, optimal stage scheduling is not necessarily the optimal schedule with the lowest register needs since stage schedules of a software pipeline do not include all possible software pipelines for a given II.

Chapter 3

Improving Software Pipelining with Unroll-and-Jam

Unroll-and-jam is a loop transformation technique that unrolls outer loops and jams the iterations of outer loops into the innermost loop. This chapter shows that unroll-and-jam has two benefits to software pipelining. First, unroll-and-jam can exploit cross-loop parallelism for software pipelining. Second, unroll-and-jam can automatically remove memory bottlenecks in loops. Section 1 describes the transformation of unroll-and-jam. Section 2 examines how unroll-and-jam can exploit cross-loop parallelism for software pipelining. It shows that unroll-and-jam may create new recurrences but the effect of these new recurrences can be eliminated by appropriate renaming techniques. Thus, unroll-and-jam can exploit cross-loop parallelism for software pipelining without restrictions. Section 3 examines the other benefit of unroll-and-jam to software pipelining. Unroll-and-jam has been shown to be very effective in automatically removing memory bottlenecks [7] [9]. Section 3 shows that the removing of memory bottlenecks can significantly decrease the resource constraint of software pipelining and thus improve software pipelining's performance. One difficulty of using unroll-and-jam with software pipelining is that the prediction of register pressure of the transformed loop is difficult. Previous register prediction schemes do not consider the effect of the overlapping execution in software pipelining [9]. Section 4 discusses several possible approaches that can help predict register pressure in software pipelining after unroll-and-jam. The final section compares the effect of unroll-and-jam on software pipelining with some other loop transformation techniques.

3.1 Unroll-and-Jam

Unroll-and-jam is an outer-loop unrolling technique [7][9]. The transformation unrolls an outer loop and then jams the resulting inner loops back together. In this thesis, the loop body before unrolling is called the *original loop body*; the loop body after unroll-and-jam is called the *unrolled loop body*. The unrolled loop body contains multiple copies of the original loop body, each of which is called an *embedded loop body*.

Figure 3.1 gives a graphical representation of the effect of unroll-and-jam on a two-nested loop. The original loop before unroll-and-jam is in Figure 3.1 (a). The execution of the original loop is shown in Figure 3.1 (d). Each iteration is labeled using the values of its index variables, i and j.

Unroll-and-jam consists of two steps; unrolling and jamming. Unrolling the outer-loop, J loop, once will result in a new loop with two innermost loops (Figure 3.1 (b)). The jamming step then merges the two innermost loops together, as shown in Figure 3.1 (c). After unroll-and-jam, the unrolled loop body contains two embedded loop bodies, each from a different I loop. The execution of the transformed loop is shown in Figure 3.1 (e), where the iterations of every two different I loops are executed together.

In summary, unroll-and-jam brings the computation of multiple iterations together. Moreover, the embedded loop bodies in an unrolled loop are from different loops. Therefore, unroll-and-jam makes it possible for loop optimization methods to exploit cross-loop parallelism.

In this thesis, the discussion of multiple-nested loops is limited to two-nested loops, an outer loop and an innermost loop, for the purpose of simplicity. However, for loops with nests of more than two, the loop transformation techniques and improvements to software pipelining will be mostly the same. The rest of this chapter will describe how unroll-and-jam can enable software pipelining to exploit cross-loop parallelism and how to eliminate the effect of recurrences in software pipelining.

3.2 Exploiting Cross-Loop Parallelism with Unroll-and-Jam

This section shows that using unroll-and-jam transforms loops so that sufficient cross-loop parallelism can be obtained in the innermost loop. However, unroll-and-jam may create new recurrences, specifically cross-loop dependence cycles. Cross-loop dependence cycles can nullify the benefit of unroll-and-jam since their length can be proportional to the degree of unrolling.

```
for ( j = 0; j < M; j ++)
     for ( i = 0; i < N; i ++)
     {
          loop body ( i, j )
     }
```

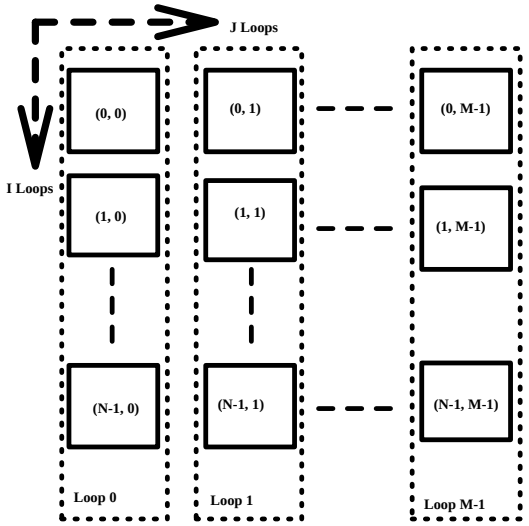
(a) Original Loop

```
for ( j = 0; j < M; j += 2 )
{    for ( i = 0; i < N; i ++)
     {
          loop body ( i, j )
     }
     for ( i = 0; i < N; i ++)
     {
          loop body ( i, j+1 )
     }
}
```

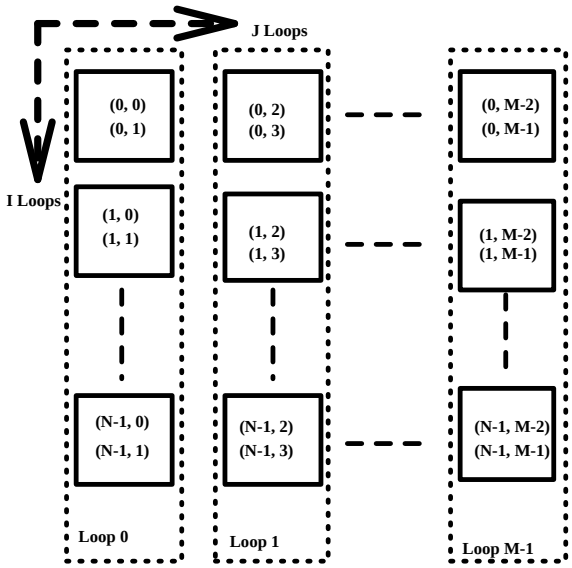
(b) after unrolling

```
for ( j = 0; j < M; j += 2 )
{    for ( i = 0; i < N; i ++)
     {
          loop body ( i, j )
          loop body ( i, j+1 )
     }
}
```

(c) after jamming



(d) before Unroll-and-Jam



(e) after Unroll-and-Jam

Figure 3.1: Illustration of Unroll-and-Jam

This section categorizes cross-loop dependence cycles and describes how to eliminate the effect of such cross-loop dependence cycles. As a result, unroll-and-jam can exploit cross-loop parallelism without difficulties and eliminate the possible negative effect on the recurrence constraint of software pipelining.

3.2.1 Cross-Loop Parallelism

Cross-loop parallelism is the parallelism existing among iterations of different loops in a multiple-nested loop. Unroll-and-jam increases the amount of parallelism in the innermost loop by exploiting cross-loop parallelism. As the amount of parallelism is measured by loop recurrences, the effect of unroll-and-jam can be examined by comparing the data dependence graph (DDG)[1] of the original loop body with the DDG of the unrolled loop body. Figure 3.2 shows how the DDG of the unrolled loop body is changed after unroll-and-jam.

The DDG nodes in the unrolled loop body are copies of the DDG nodes of the original loop body. The DDG nodes in the original loop body can be divided into two parts, *control nodes* and *computing nodes*. Control nodes consist of the increment of the index variable, and the test on the index variable deciding if the loop is finished. These nodes are part of the control structure of the loop. Computing nodes are the remaining DDG nodes; those perform the computation during each iteration. After unroll-and-jam, the computing nodes of the original loop body are duplicated in each embedded loop body; however, only a single copy of the control nodes is required for the entire unrolled loop body. For example, if a loop is unrolled N times, there will be $N + 1$ copies of the computing nodes and 1 copy of control nodes in the DDG of the unrolled loop body. We call each copy of the computing nodes in the unrolled loop body an *embedded DDG* and call the control nodes the *control DDG*.

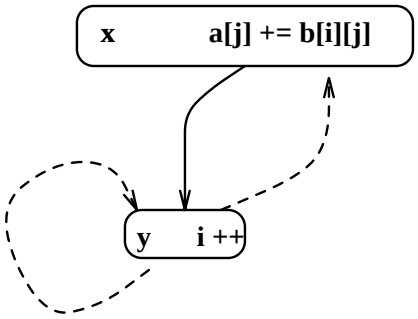
Figure 3.2 (c) and (d) show the DDGs of the original loop body and the unrolled loop body of the example loop. Because the loop is unrolled once, there are two embedded loop bodies in the unrolled loop body. The computing node of the original loop body is the assignment of $a[i][j]$ and the control node is the increment of i. The DDG of the unrolled loop body contains two copies of the computing node (two embedded DDGs) and one copy of the control node (one control DDG).

The data dependences in the original loop body are also copied to the DDG of the unrolled loop body. In the original loop body, a dependence is either (1) between two computing nodes, (2) between a computing node and a control node, or (3) between two control nodes. Loop-invariant

[1] See Section 2.3.1 for a definition of DDG

```
for ( j = 0; j < M; j ++)
    for ( i = 0; i < N; i ++)
    {
        a[j] += b[i][j]
    }
```

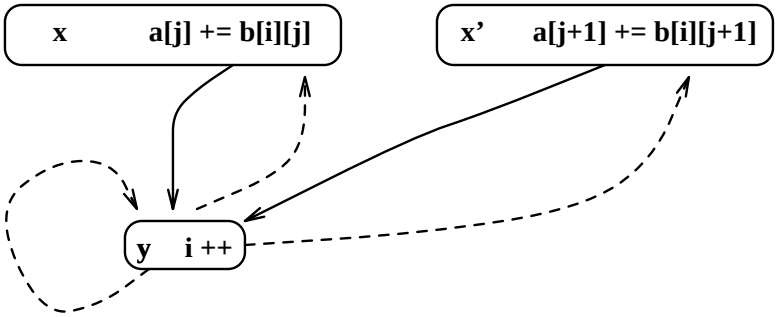
(a) Loop before transforming



(c) Recurrences in the innermost loop, before transforming

```
for ( j = 0;  j < M;  j += 2 )
    for ( i = 0; i < N; i ++)
    {
        a[j] += b[i][j]
        a[j+1] += b[i][j+1]
    }
```

(b) Loop after transforming



(d) Recurrences in the innermost loop, after transforming

Figure 3.2: Example Loop 2

dependences in the original loop body are copied exactly in the unrolled loop body: each dependence between two computing nodes in the original loop body is duplicated in each embedded DDG; each dependence between a computing node and a control node is duplicated between the computing node in each embedded DDG and the control node in the control DDG; and each dependence between two control nodes is copied in the control DDG. However, unroll-and-jam may add new dependences to the unrolled loop body. These additional dependences are *cross-loop dependences*. The next section shows how cross-loop dependences affect the DDG. However, loop-carried dependences that are not cross-loop dependences are copied to the unrolled loop body in the same way as loop-independent dependences. The example loop in Figure 3.2 does not have any cross-loop dependences; so all data dependences, loop-independent and loop-carried, are simply duplicated in the DDG of the unrolled loop body.

The important characteristic of the example DDG after unroll-and-jam is that no direct data dependence exists between any two computing nodes of different embedded DDGs. This determines that any loop-carried dependence cycle in the transformed loop is a copy of some loop-carried dependence cycle in the original loop. So the length of the longest loop-carried dependence cycle in the unrolled loop is the same as that in the original loop. This means that the recurrence remains the same after unroll-and-jam. As the amount of computing is increased, the amount of parallelism is increased. When unrolling N times, the amount of parallelism in the innermost loop will be increased $N + 1$ times.

However, as previously mentioned unroll-and-jam may create new recurrences due to the existence of cross-loop dependences. The new recurrences may restrict the available cross-loop parallelism. The next two sections show that although unroll-and-jam may cause new recurrences, the negative effect of the new recurrences on software pipelining can be eliminated.

3.2.2 Cross-Loop Dependences

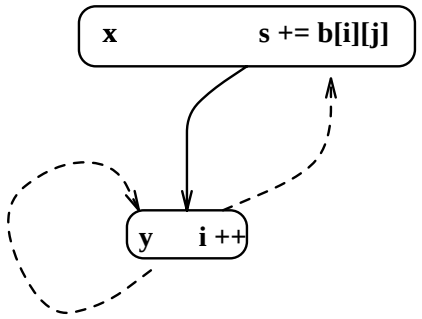
A loop-carried dependence can exist between computing nodes of different iterations of the same loop. Loop-carried dependence can also exist between computing nodes of different iterations of different loops. The latter type of loop-carried dependence is called *cross-loop dependence*. For example, in Figure 3.1 (e), if a dependence exists between an iteration in $Loop$ 0 and an iteration in $Loop$ 1, it is a cross-loop dependence. Any dependence carried by the J loop must be a cross-loop dependence. In fact, cross-loop dependences are dependences carried by outer loops.

```
for ( j = 0; j < M; j ++)
    for ( i = 0; i < N; i ++)
    {
        s += b[i][j]
    }
```

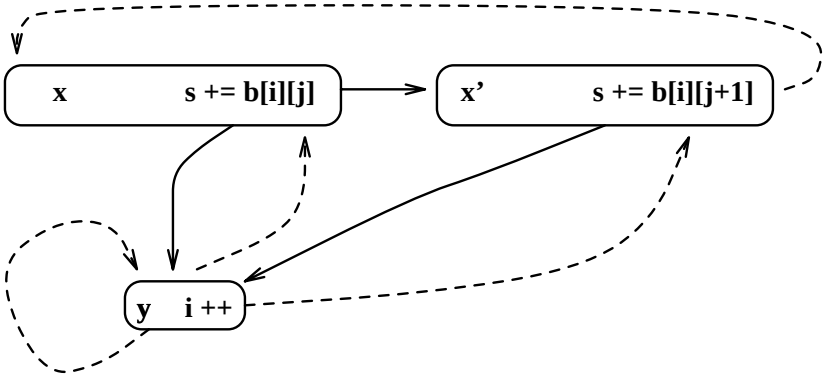
(a) Loop before transforming

```
for ( j = 0;  j < M;  j += 2 )
    for ( i = 0; i < N; i ++)
    {
        s += b[i][j]
        s += b[i][j+1]
    }
```

(b) Loop after transforming



(c) Recurrences in the innermost loop, before transforming



(d) Recurrences in the innermost loop, after transforming

Figure 3.3: Example Loop 3

Because unroll-and-jam brings iterations of different loops together, cross-loop dependences may be brought into the innermost loop and form new loop-carried dependence cycles in the unrolled DDG. A loop-carried dependence cycle that has a cross-loop dependence edge is a *cross-loop dependence cycle*.

For the example loop in Figure 3.2, there is no cross-loop dependence in the original loop, so no dependence links the two embedded DDGs in the unrolled loop. However, for the other example loop of Figure 3.3, there is a cross-loop dependence between iterations of unrolled I loops. The cross-loop dependence is brought into the innermost loop by unroll-and-jam of the original loop. It becomes two dependences: the loop-independent and the loop-carried dependences between node x and x'. (Actually, each of the two dependences is three dependences, true, anti and output. For simplicity, we assume they are one dependence.)

The cross-loop dependence causes two cross-loop dependence cycles in Figure 3.3 (d). The first cross-loop dependence cycle includes node x and node x'. Both edges in this cycle are

cross-loop dependences. The second cross-loop dependence cycle links node x, x' and y. The cross-loop dependence from node x to node x' links with the loop-independent dependence from node x' to y and the loop-carried dependence from node y to x. Though the latter two dependences are just copies of existing dependences in the original loop body, they form a new recurrence when linked with the cross-loop dependence.

Cross-loop dependence cycles are the new recurrences added by unroll-and-jam. In this thesis, cross-loop dependence cycles are divided into two types, true cycles and false cycles. In a true dependence cycle, all data dependences are true dependence edges; in a false cycle, at least one dependence is a “false" dependence — that is at least one is either an anti-dependence or an output dependence. The first new recurrence cycle in Figure 3.3(d) is a true cycle because all edges in the cycle are true dependences. The next section will show that true cycles are caused by scalar variables shared by both the innermost and the outer loop. In the example in Figure 3.3, the true cycle is caused by variable S, which is shared by the I and J loops. The second new recurrence cycle in Figure 3.3(d) is a false cycle since the loop-carried dependence, from node x' to y, is an anti dependence. False cycles are caused by the reuse of some scalar variable in the innermost loop. In the example in Figure 3.3, the false cycle is caused by the reuse of the index variable i.

False cycles are of little interest since they are not inherent recurrences. False cycles are a subset of false recurrences discussed in Chapter 5, and as such, can be eliminated in modulo scheduling either by hardware support of rotating registers or by the renaming technique described in Chapter 5. From now on, cross-loop dependence cycles only refers to true cycles in the loop after unroll-and-jam.

True cross-loop cycles can nullify the benefit of cross-loop parallelism obtained by unroll-and-jam. Unlike other dependence cycles that only include nodes of a single embedded DDG, such true cross-loop cycles include nodes from all embedded DDGs. The length of the cross-loop dependence is proportional to the degree of loop unrolling. New recurrences introduced by true cycles limit the amount of parallelism in the innermost loop and makes more unrolling useless.

The following discussion of cross-loop dependence cycles focuses on the ‘global’ true cycles that include the computing nodes of all embedded DDGs for any degree of unrolling. These true cycles are the recurrences that can nullify the benefit of cross-loop parallelism obtained by unroll-and-jam. A ‘global’ cross-loop dependence cycle (1) links every embedded DDG, and, (2) links the same group of computing nodes within each embedded DDG.

3.2.3 Cross-Loop Dependence Cycles

True cycles must be caused by variables shared by both the innermost and the outer loop. Consider Figure 3.1 (d). The iterations of the outer loop are called J loops and the iterations of the inner loop are called I loops. A true cycle starts from each iteration in an I loop, goes through the unrolled iterations in the corresponding J loop, and ends at the next iteration of the same I loop. The data dependences of the true cycle mean that there is a scalar variable used and defined in both the I and J loop. In other words, this scalar variable is shared in all loop iterations.

True cycles, which are caused by a shared variable, can be eliminated by shared-variable renaming. To remove these cross-loop dependences, the shared variable should be renamed in each embedded loop body. After using a different variable in each embedded loop body, the cross-loop dependences are removed, and so is the cross-loop dependence cycle. Shared-variable renaming is a special case of a well known technique, called *tree height reduction*[23]. Tree height reduction divides the computation of the shared variable in the loop and completes the computation in a minimum number of steps. The shared-variable renaming used in unroll-and-jam is equivalent to dividing the overall computation into a number of parts equal to the unroll degree, which reduces the recurrence length to $\frac{1}{n}$, where n is the degree of unrolling.

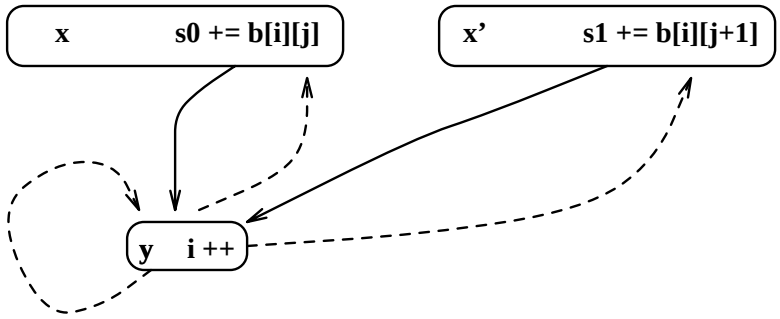
Figure 3.4 shows how shared variable renaming can eliminate the true cycle in Figure 3.3 (d). The true cycle in Figure 3.3 (d) is caused by variable S, which is shared in both I and J loops. Assume iteration (i, j) and $(i, j + 1)$ are executed together in the unrolled loop body. The loop-carried data dependence from node x' to x means that the computing of S in iteration $(i, j+1)$ must precede the computing of S in iteration $(i+1, j)$. This is not a recurrence in the original loop, since iteration $(i + 1, j)$ (in loop j) precedes iteration $(i, j + 1)$ (in loop j+1) in the original loop. The dependence is caused by the reuse of variable S. The order of computing S in two iterations does not affect the semantics of the loop. That is, the computation of S in each iteration can proceed in parallel if they do not reuse the variable S. So the true cycle can be removed if variable S is renamed into two variables, S_0 and S_1 as in Figure 3.4 (b).

After renaming the shared variable S in the example loop in Figure 3.3, the renamed loop and its new DDG are shown in Figure 3.4 (a) and (b). Now the DDG of the renamed loop body contains no cross-loop dependence. By renaming the variable S, all cross-loop dependences of the true cycle are removed.

If an outer loop is unrolled N times and a variable x is defined in each iteration, then variable x should be renamed into x_0, x_2, ..., and x_{N-1} to eliminate the cross-loop dependence. In

```
for ( j = 0;  j < M;  j += 2 )
{
    for ( i = 0; i < N; i ++)
    {
        s0 += b[i][j]
        s1 += b[i][j+1]
    }
}
s += s0 + s1
```

(a) Loop after renaming



(b) Recurrences in the innermost loop, after renaming

Figure 3.4: Example of Shared-Variable Renaming

the unrolled loop body, each embedded loop body uses a different renamed variable. By eliminating the cross-loop dependences among the embedded loop bodies in the unrolled loop, true cycles can be removed.

3.2.4 Reducing Recurrence Constraints

Unroll-and-jam can increase the amount of parallelism in the innermost loop by exploiting the cross-loop parallelism in the multiple-nested loop. By unrolling the outer loop N times, the parallelism in the innermost loop can be increased to N times higher than before. To measure the decrease on software pipelining constraints, we define the *unit initiation interval (unit II)* as II divided by the unroll factor. *Unit RecII* is RecII divided by the unroll factor and *Unit ResII* is the ResII divided by the unroll factor.

After removing cross-loop dependence cycles, the RecII is the same regardless of the degree of the unrolling. But the unit RecII is decreased with respect to the degree of unrolling. When the innermost loop is unrolled N times, the unit RecII of the unrolled loop body is $\frac{1}{N+1}$ of the RecII in the original loop. Therefore, unroll-and-jam can always obviate the recurrence constraint for nested loops by reducing the unit RecII until it is less than the unit ResII. This then guarantees that the minimum II will be determined by the unit ResII rather than the unit RecII.

3.3 Removing the Memory Bottleneck with Unroll-and-Jam

Memory bottlenecks in software pipelining are caused by the limited number of memory functional units and the high memory resource demand in loops. Unroll-and-jam, by unrolling

an outer loop and jamming iterations of different loops together, can effectively remove memory bottlenecks and reduce the memory resource demand of the loop. This section describes how unroll-and-jam removes memory operations and how software pipelining resource constraints can be reduced by unroll-and-jam.

3.3.1 Removing Memory Operations

Scalar replacement, or load-store-elimination, has been used to reduce the number of memory load/store operations by reusing values across innermost loop iterations[7] [5] [15] [10]. That is, instead of storing a value at this iteration and loading it at the next iteration, the value can be maintained in a register and the load and store operations of that value can be eliminated.

For multiple-nested loops, more opportunities for scalar replacement can be obtained by reusing values not only across different iterations in the innermost loop but also across different loops. Unroll-and-jam, by bringing iterations of different loops into the innermost loop, can capture many more chances for scalar replacement. For example, consider matrix multiply. Assume the target machine can issue only one memory operation per machine cycle. Assume the machine has enough other functional units so that the number of memory operations in the innermost loop determines the ResII.

```
for(i=O;  i<4; i++)
  for(j=0; j<4; j++)
    {
     c[i][j] = 0;
     for(k=0; k<4; k++)
       c[i][j] = a[i][k]*b[k][j]+c[i][j];
    }
```

The example loop has 4 memory operations in the innermost loop and a ResII of 4. Scalar replacement can remove the store and the load of C[i][j] in the innermost loop. After scalar replacement, the loop becomes:

```
for(i=O;  i<4; i++)
  for(j=0; j<4; j++)
    {
     c[i][j] = 0;
     t = C[i][j];

     for(k=0; k<4; k++)
     {
       t = a[i][k]*b[k][j]+t;
     }

     C[i][j] = t;
    }
```

After scalar replacement, the number of memory operations in the innermost loop is decreased to 2 and the ResII is 2 machine cycles. In this manner, unroll-and-jam can remove some memory operations from the innermost loop by considering multiple iterations of both inner and outer loops and, thus, increasing the scope of scalar replacement. In this example, unroll-and-jam will find that the next J loop iteration is $c[i][j+1] = a[i][k] * b[k][j+1] + t$. The loaded value of $a[i][k]$ is used in both iterations. So by unrolling the J loop once, a load of $a[i][k]$ can be eliminated. After unrolling the J loop once by unroll-and-jam, the loop becomes:

```
for(i=0;  i<4; i++)
  for(j=0; j<4; j+=2)
   {
    c[i][j] = 0;
    t0 = C[i][j];
    c[i][j+1] = 0;
    t1 = C[i][j+1];

    for(k=0; k<4; k++)
    {
      w = a[i][k];
      t0 = w * b[k][j] + t0;
      t1 = w * b[k][j+1] + t1;
    }

    C[i][j] = t0;
    C[i][j+1] = t1;
   }
```

Now there are 3 memory operations in the innermost loop. Because the transformed loop body does twice the work of the original one, the unit ResII (ResII divided by the unroll factor) becomes $\frac{3}{2}$, 25% lower than the unit ResII before using unroll-and-jam. Again this reduction is due to unroll-and-jam leading to more aggressive scalar replacement.

3.3.2 Removing the Memory Bottleneck of Software Pipelining

Memory bottlenecks happen when the memory resource demand in a loop and the capacity of the memory functional units of a machine are mismatched. As a result, the memory functional units in the machine are fully used; but other machine resources, like floating-point functional units, are not. Unroll-and-jam, by unrolling outer loops and replacing memory accesses with scalars, can effectively reduce the memory resource demand in the innermost loop. Therefore, it is effective in removing the memory bottleneck. However, different loops need different degrees of unrolling and replacing to achieve optimal machine resource usage. This section describes the heuristics that are used to achieve the optimal resource utilization. The basic criteria was defined by Callahan et al[7].

Carr and Kennedy gave the algorithm that automatically computes the optimal degree of unrolling for unroll-and-jam[9]. The criteria is the same when using unroll-and-jam to remove memory bottlenecks to software pipelining. However, in computing the optimal degree of unrolling, the effect of overlapping execution needs to be considered.

To illustrate the relation between the resource demand of a loop and the resources available in a machine, Callahan et al[7] defined the machine balance as

$$\beta_M = \frac{number\ of\ memory\ operations\ can be\ issued\ per\ cycle}{number\ of\ floating-point\ operations\ can\ be\ issued\ per\ cycle}$$

and the loop balance as

$$\beta_L = \frac{number\ of\ memory\ operations\ in\ the\ loop}{number\ of\ floating-point\ operations\ in\ the\ loop}$$

[7]. When $\beta_L = \beta_M$, the memory functional units and the floating-point functional units can all be fully used. However, when $\beta_L \geq \beta_M$, a loop is *memory bound* and the floating-point functional units cannot be fully utilized due to the high memory resource demand. For memory bound loops, β_L needs to be reduced to be equal to β_M in order to achieve a balanced use of both memory and floating-point resources. Because of the effect that unroll-and-jam can reduce β_L by allowing more aggressive scalar replacement, we can choose an unroll amount to set β_L as close to β_M as we wish for memory-based nested loops.

However, unroll-and-jam leads to a higher register demand for the nested loops since it jams several iterations together and replaces memory references with scalars. If the register demand were increased to be higher than the number of available registers, the benefit of unroll-and-jam would be counteracted by the negative effects of register spilling. So, typically, unroll-and-jam heuristics also consider the number of registers available in the target machine. If the “optimal" degree of unrolling requires too many registers, heuristics for unroll-and-jam pick the highest degree of unrolling that does not require more registers than the machine has.

When using unroll-and-jam to remove the memory bottleneck for software pipelining, the aim is still to decrease the β_L of a memory bound loop to β_M. However, the register demand of software pipelined loops is different from that of a locally or globally scheduled loop. In software pipelining, the execution of several iterations overlap. This overlapping leads to higher register demand. For modulo-scheduled loops, a variable may need several registers as a result of kernel unrolling (Section 2.4.3). Unroll-and-jam needs to consider the overlapping effect of software pipelining in order to precisely estimate the register pressure of an unrolled loop after software pipelining.

3.4 Estimating the Register Pressure of Unroll-and-Jam

Estimating the register pressure of an unrolled loop after software pipelining is crucial for using unroll-and-jam with software pipelining. If the estimation is higher than the actual register pressure, unroll-and-jam will restrict the degree of unrolling heuristically and unroll-and-jam's benefit will not be fully exploited. If the estimation is too low, unroll-and-jam will use too many registers and potentially cripple software pipelining due to register spilling [12].

Two register-pressure estimation methods for modulo scheduled loops, MaxLive and MinAvg, were described in Section 2.3.4. This section discusses the possible use of these two estimation methods with unroll-and-jam and proposes a new approach that may be more effective than simply using MaxLive or MinAvg. To date, the effectiveness of these methods has not been examined experimentally.

Using MaxLive

MaxLive (described in Section 2.3.4) is the closest estimation to the register demand for a software pipeline schedule. Rau et al. found that for the loops they tested, the actual register demand of a schedule was rarely 5 registers more than its MaxLive for a register allocation method similar to graph-coloring register assignment [34].

Although MaxLive is a fairly precise estimation, it is not practical for unroll-and-jam to use it. Computing MaxLive can only be done after a software pipeline schedule is found, i.e. after generating an unrolled loop body and scheduling the generated loop body. It is too time consuming for unroll-and-jam to try different unrolling based on the value of MaxLive.

Using MinAvg

As described in Section 2.3.4, MinAvg is an estimation that does not depend on any particular software pipeline schedule. The accuracy of an estimation using MinAvg is affected by several factors.

The first factor is the accuracy of the predicted II because the computing of MinAvg depends on a specific II. Due to the fact that the II of a loop cannot be determined until after software pipelining, the value of II has to be estimated. Fortunately, current leading software pipelining algorithms can achieve optimal or near-optimal II for most of the loops. So $MinII$, which is the larger of $RecII$ and $ResII$, can be used as a good estimate of II. Both $RecII$ and

$ResII$ can be computed before scheduling. Moreover, $RecII$ can be computed at the source level if the latencies of the machine are known. The false recurrences caused by the code generation of a compiler can be ignored since they are not 'inherent' recurrences and can be eliminated by appropriate techniques.

The second factor affecting the computing of MinAvg is how close the MinAvg is to the actual register demand. Unfortunately, the optimal assumption used in computing MinAvg sometimes makes it far lower than the actual register pressure in the generated software pipeline schedule. The basic assumption used in computing MinAvg is that, in a software pipeline schedule, each variable has a minimum-length lifetime and each register is fully used in every cycle of the software pipeline. However, resource constraints can lead to longer lifetimes than the minimum length. Especially for loops in which $ResII$ is much higher than $RecII$, the lifetime length of variables can be much longer than the minimum length. Therefore, for loops with relatively high resource constraints, MinAvg can be quite inaccurate. Moreover, unroll-and-jam increases the amount of parallelism in the innermost loop. This transformation results in a higher $ResII$ relative to $RecII$. So MinAvg is probably not a precise estimation for unroll-and-jam.

Using MinDist

MinDist can be computed once the II is known. As stated previously, II can be estimated with good accuracy. Given II, MinDist gives the minimum lifetime of each variable. This information can be very helpful in estimating the register pressure.

Without considering the overlapping effect of software pipelining, the register pressure of the unrolled loop can be precisely estimated using the algorithm given by Carr and Kennedy [9]. So if the additional register demand caused by the overlapping effect is known, the register pressure after software pipelining can be precisely estimated. MinDist can be used to compute the effect of overlapping on register pressure for modulo scheduled software pipeline schedules. The overlapping in software pipelining requires additional registers only when the lifetime of a register is longer than II. MinDist provides the lower bound of the lifetime of each variable and this lower bound can serve as an estimation directly. If the minimum lifetime of a variable is L, the variable adds $\lceil \frac{L}{II} \rceil - 1$ registers to the register pressure in the software pipeline schedule.

The disadvantage of using MinDist is that the resource constraints can lead to register lifetimes much longer than their minimum value in the unrolled loops. However, using MinDist is

more accurate than using MinAvg because the prediction using MinDist does not assume an optimal register usage. MinDist is only used to compute additional registers needed due to the overlapping of software pipelining.

3.5 Comparing unroll-and-jam with Other Loop Transformations

This section describes the advantages of unroll-and-jam, with regard to software pipelining, over the two other popular loop transformations, namely tree height reduction and loop interchange.

3.5.1 Comparison with Tree-Height Reduction

Tree-height reduction (THR) is a general technique that considers the computation carried by the loop as a computing tree and restructures the computing tree so that more computations can be parallelized and the height of the tree can be reduced[23]. Tree-height reduction can be applied to not only nested loops but also single loops. It generally involves innermost loop unrolling and back-substitution. Tree-height reduction has recently been used and evaluated on ILP architectures[38] and in modulo scheduling [26].

Both unroll-and-jam and THR increase the amount of parallelism in the innermost loop. However, the sources of the increased parallelism are totally different. Unroll-and-jam transforms a nested loop and puts parallelism from outer loops into the innermost loop, whereas THR transforms the computation in the innermost loop and exposes more parallelism from within the innermost loop. Each method deals with different sources of parallelism and each can be used with the other to obtain both benefits without problem.

The major advantage of THR is that it can be applied to any loops whereas unroll-and-jam can only be used for nested loops. Although THR is not specifically designed for nested loops, it may exploit cross-loop parallelism by interchanging the nested loop. However, as described in the next section, there are two shortcomings with the use of loop interchange: loop interchange is not always legal, and it can lead to poor cache performance.

Unroll-and-jam, however, can exploit cross-loop parallelism without difficulties caused by loop interchange. The amount of cross-loop parallelism is quite large. For architectures with high degree of hardware parallelism, exploiting cross-loop parallelism is essential. Compared to THR, unroll-and-jam can more effectively reduce resource constraints in the transformed loop.

The unrolling performed by both unroll-and-jam and THR brings in more computations that allow traditional optimization to reduce more operations. But unroll-and-jam can enhance chances for scalar replacement across loop boundaries. THR, since it limits its scope only to the innermost loop, cannot achieve the benefit of optimization across loops. Therefore, unroll-and-jam is more effective in reducing resource constraints for software pipelining than THR.

Unroll-and-jam has another advantage over THR because the effect of unroll-and-jam on RecII can be accurately predicted and the degree of unrolling can be determined before transformation. However, THR cannot predict the resulting RecII before transformation, and, thus, the degree of innermost loop unrolling cannot be optimally determined.

3.5.2 Comparison with Loop Interchange

Loop Interchange can switch an outer loop to be the innermost loop. To minimize the recurrence in the innermost loop, loop interchange should switch the loop that has the least recurrence to be the innermost loop. By doing so, loop interchange may decrease the recurrence in the innermost loop without increasing the size of the innermost loop. However, it may not be legal to perform loop interchange. Even when loop interchange can be performed freely, it still has three disadvantages compared with unroll-and-jam.

The first disadvantage of loop interchange is that it may not always eliminate the effect of recurrence in the innermost loop. If a multiple-nested loop does not have a loop that has no recurrence, loop interchange cannot eliminate the effect of the recurrence in the innermost loop as unroll-and-jam can. The unit RecII will be the same as the lowest RecII in all loops, but it cannot be reduced further. The second disadvantage of loop interchange is that by changing the order of the loop, cache performance may be worsened significantly. Poor cache performance will cause a much higher register pressure in software pipelining.

Chapter 4

Improving Software Pipelining with Memory Reuse Analysis

The deep memory hierarchy in today's microprocessors causes uncertain latencies for memory operations. In the past, software pipelining algorithms either assumed that each memory operation takes the shortest possible time to finish or they assumed that each memory operation takes the longest possible time to finish. This chapter shows that both assumptions are undesirable because they each degrade software pipelining's performance. The behavior of a memory operation can and should be predicted by using a compiler technique called memory reuse analysis. By taking advantage of predictions provided by such analysis, software pipelining should be able to achieve a much better performance than that possible with either optimistic (all memory operations take the shortest time possible) or pessimistic (all memory operations require the longest latency possible) assumptions.

4.1 Memory Hierarchy

The speed of today's microprocessors is much greater than the speed of today's memory systems. Over the past ten years, the processor speed has increased at a higher rate than that of the memory system. As a result, today's machines have a significant gap between CPU speed and memory speed. Although an integer addition typically takes only one machine cycle in the CPU, a load from the main memory may take 30 cycles. To reduce the average time required to load a value from main memory to CPU, cache is used in all modern machines to serve as a buffer between the processor and the main memory. Cache is much faster than the main memory. A load from a

first level (usually on-chip) cache normally takes only 2 to 3 machine cycles. However, cache is much smaller than main memory and thus, it is sometimes impossible to have all the data needed by a program fit into the cache.

Memory hierarchy is the term to describe current memory systems that consist of different levels of data storage. In such a hierarchy, registers are the fastest, but smallest 'memory'. Main memory is the slowest, but largest. Cache has a speed and a size in between. Most recent machines use a multiple-level cache, i.e. a small, fast, on-chip cache and a larger, slower, off-chip cache. With the speed difference between the processor and the main memory getting larger, the memory hierarchy necessarily becomes deeper to maintain sufficient memory speeds.

The deep memory hierarchy causes significant uncertainty in the time needed for a memory operation. During execution, some data come from cache and some from main memory. The *memory latency*, which is the time needed for a memory load or store, varies drastically depending upon whether the data comes from cache or from main memory. A load or store from the fast first-level data cache is called a *cache hit* and a load or store from a second-level cache or main memory is called a *cache miss*. The latency of a cache hit and the latency of a cache miss normally differs by a factor of 10 or more.

4.2 Memory Reuse Analysis

Memory reuse analysis is a compiler technique used to predict the cache behavior by analyzing the reuse of data in a program. It predicts whether a memory load or store is a cache hit or a cache miss. Cache memories are based upon the assumption that most data is used multiple times. When data are loaded from the main memory, they are placed in the cache. So if the data is used again before it must be removed from the cache (to make room for other data), the second load of the data is a cache hit. The cache concept, and indeed the entire memory hierarchy, reduces the average memory access time significantly because programs exhibit *locality*.

Programs actually exhibit two types of locality. First, data that have been used "recently" in a program are likely to be reused again soon. Such multiple use of the same data is called *temporal locality* and, likewise, the reuse of previously loaded data is called *temporal reuse*. In addition to temporal locality, programs tend to use data which are "close together" such as contiguous elements of an array. This locality is called *spatial locality* and leads to the term *spatial reuse* when spatial locality yields a cache hit. In terms of cache behavior, temporal reuse occurs when, after data is loaded from the main memory, the same data is loaded again. At the first load, the data will be

moved to the cache; therefore the second load will be a cache hit. Caches make use of spatial reuse in that when one location is used, data in nearby locations are loaded as well. When the first data is loaded, a whole block of data containing the loaded data is loaded into the cache. A block contains a group of adjacent data. So along with the data initially accessed, other data near the initial data are placed in the cache. When a subsequent memory operation accesses data near the initial data (spatial reuse), access will be a cache hit as well.

Let's considering the following loop of matrix multiply. There are four memory references in the innermost loop. The two references of $C[i]$ have temporal locality because the same data, $C[i]$, is accessed every time in the innermost loop. The other two memory references access different data in every iteration. Let's assume the array A and B are stored in row-major order. Then memory access $A[i,j]$ in each iteration has spatial reuse because it is near the previous $A[i,j]$ in the last iteration. Memory access $B[j,i]$ has neither temporal reuse nor spatial reuse because $B[j,i]$ in each iteration is a long distance in memory from the access in the previous iteration.

```
for (i=0; i<M; i++)
    for (j=0; j<M; j++)
    {
            C[i] = C[i] + A[i,j]*B[j,i];
    }
```

Memory reuse analysis examines each memory operation to see if it has spatial reuse, temporal reuse or neither of the two. One relatively simple cache prediction scheme assumes that any static operation that exhibits reuse (as determined by compile-time analysis) will always yield a cache hit and that any static operation that cannot be demonstrated by conservative compile-time analysis to exhibit reuse is always a cache miss. There are currently two popular compile-time analysis models that have been used to identify memory reuse; one is based upon sophisticated dependence analysis[11] and the other uses a linear algebra model [46]. The reader should refer to those resources for algorithm details.

4.3 Removing Hardware Misuse with Memory Reuse Analysis

As discussed previously, the problem of uncertainty latencies is rooted in the deep memory hierarchy of modern microprocessors. However, memory reuse analysis can help solve the

uncertain-latency problem by predicting cache hits and misses. Without using memory reuse analysis, a compiler cannot have the precise knowledge of the latencies of memory operations. When a latency of a memory operation is unknown to a compiler, the compiler must either assume that it is a cache hit or a cache miss. Some compilers assume that all memory loads are cache hits (*all-cache-hit* assumption) and some treat all memory loads as cache misses (*all-cache-miss* assumption). The all-cache-hit assumption under-estimates the latency of a cache miss and the all-cache-miss assumption over-estimates the latency of a cache hit. Both assumptions are imprecise, as the latency difference between cache-hit loads and cache-miss loads can be a factor of ten.

This section will examine both the all-cache-hit and the all-cache-miss assumption used in software pipelining and show that both assumptions can significantly degrade the performance of software pipelining. When memory reuse analysis is used, these degradations can be avoided.

4.3.1 All-Cache-Hit Assumption

Consider an example loop on a machine where a cache hit takes two cycles, a cache miss takes twenty cycles and other operations take two cycles.

```
for (i=0; i<N; i++)
{
        C += A[i];
}
```

Assume the II of the generated software pipeline is 3 cycles and the schedule for one iteration is,

```
Operation       Cycle
x               1               load A[i] => t1
u               2               i + 1 => i
v               4               if (i>=N) goto postlude
y               t               t1 * C => C
```

Note that operation y, scheduled at cycle t, depends on the assumed latency of the load of $A[i]$. If the compiler assumes the load is a cache hit, t will be 3; otherwise, t is 21 assuming the load is a

cache miss. Using an all-cache-hit assumption, where the compiler treats every memory load as a cache hit, operation y will be scheduled at cycle 3. The execution will be:

```
0        load A[i] => a
1        i + 1 => i
2        a * C => C
3        if (i>=N) goto postlude        # load A[i] => a
4                                       # i + 1 => i
5                                       # a * C => C
...
```

If the assumption of compiler is wrong and the load is a cache miss which takes 20 cycles to finish, then a severe speed penalty will occur since the second iteration can not be issued until the cache miss is resolved. The execution of this software pipeline will be 21 cycles per iteration, instead of 3 cycles per iteration, a slowdown of 700%.

4.3.2 All-Cache-Miss Assumption

To avoid the possible severe speed penalty, Rau [33] and Huff [22] chose to assume the worst latency for every memory load, that is, all-cache-miss. For our same example loop in the last section, when the latency of operation x is assumed as 20 cycles, the execution of software pipeline will be:

```
0         load A[i] => a1
1         i + 1 => i
2
3         if (i>=N) goto   # load A[i] => a2
4                          # i + 1 => i
5
6                          # if (i>=N) goto     # load A[i] => a3

         ...                  ...                  ...

20        a1 * C => C
21                         # a2 * C => C
22                                               # a3 * C => C
```

This software pipeline, when at its full speed, completes one iteration every three cycles. Notice the variables $a1$, $a2$, etc are introduced by kernel unrolling because the lifetime of the loaded value is longer than II. The compiler assumes that the load takes 20 cycles, consequently, it reserves 21 cycles of lifetime for the loaded value. This long lifetime needs 7 registers. However, if the load is a hit, only three cycles of lifetime and one register are needed. So the over-estimation costs 6 more registers for this example.

4.3.3 Using Memory Reuse Analysis

As described in the previous sections, when the all-cache-hit assumption under-estimates the latency of cache-miss loads, the speed of the software pipeline is severely degraded; when the all-cache-miss assumption over-estimates the latency of cache-hit loads, the software pipeline uses many more registers than needed. By using memory reuse analysis, software pipelining should neither over-estimate nor under-estimate the latency of memory operations. So the degradation of the software pipeline speed and the unnecessary use of registers can be avoided with memory reuse analysis. In practice, of course, memory reuse analysis cannot guarantee to identify each cache hit and cache miss. However, it should yield improved software pipelining when compared to either an all-cache-hit or an all-cache-miss policy.

Chapter 5

Improving Modulo Scheduling

Modulo scheduling is a effective scheduling technique for software pipelining. Experimental results have shown that it can achieve near-optimal II for most benchmark loops[33] [4]. This chapter discusses two enhancements to modulo scheduling. Previously, modulo scheduling could not eliminate all false recurrences without the hardware support of rotating registers. Section 1 proposes a compiler algorithm that can efficiently eliminate the effect of all false recurrences for conventional architectures. Section 2 describes a faster method of computing RecII than is currently used in practice.

5.1 Eliminating False Recurrences

As described in Chapter 1, anti and output dependences are called false dependences because they can be removed by appropriate variable renaming without changing program semantics. Therefore, the recurrence cycle in a loop that contains anti or output dependences is not a true recurrence. I call it a *false recurrence*. Thus, a *true recurrence* is a loop-carried dependence cycle in which each edge is a true dependence. Various software and hardware methods have been used to eliminate the effect of false dependences and recurrences.

Hardware renaming is one method of eliminating anti-dependences that has been used for a long time[41]. In the realm of hardware renaming, an anti-dependence is known as a WAR (write-after-read) hazard. When the hardware detects a WAR hazard, it can copy the value to a temporary and let the second operation proceed without waiting for the first operation to finish. The hardware then forwards the copied value to the first operation when needed. In this manner, the WAR hazard is eliminated at the cost of an additional location necessary for the temporary.

Compilers have also made attempts to eliminate anti and output dependences by “renaming". In a manner similar to hardware renaming, compilers assign additional temporaries to remove possible anti and output dependences. One popular method of compiler renaming is the use of static single assignment (SSA)[14]. In SSA, each produced value uses a separate variable so that the reuse of variables can be eliminated.

However, when software pipelining, neither traditional hardware or software renaming can completely eliminate false recurrences in a loop. When the execution of different iterations overlaps, the use of a value in a previous iteration may occur long after a define in a later iteration. Thus, the hardware does not even know there is a use in the previous iteration before the define of the current iteration. Traditional software renaming does not rename a value for each iteration. If a value a is defined in a loop body, traditional software renaming does not give a different location to the a produced in each iteration. Therefore, false recurrences may happen due to the reuse of the location of a by multiple iterations.

If a value is defined in a loop, a complete renaming would require a separate variable for each iteration of the loop. This type of renaming was used in vectorizing compilers[24], where the defined variable is expanded into a higher degree of array. However, this type of renaming is too expensive to be practical in conventional machines. One method of special hardware support, namely rotating registers (RR), has been suggested [13] [36]. Rau et al. used RRs for modulo scheduled loops [34]. In a machine with support for RRs, the index of registers are shifted every II cycles. This shifting produces the effect that each II uses a different set of registers and the reuse of the same location can be avoided. Although RRs can eliminate all false recurrences caused by the reuse of variables, it is an expensive hardware feature that is not available on any of today’s machines.

Lam [25] proposed a software solution to this problem called *Modulo Variable Expansion (MVE)*. The advantage of MVE is that it requires no hardware support not found on conventional architectures. MVE identifies all variables that are reproduced at the beginning of each iteration and removes the loop carried dependence between the use and the define of each reproduceable variable. After modulo scheduling, MVE unrolls the kernel as appropriate to avoid any lifetime overlapping itself (see Section 2.4.3).

However, as described in the next section, modulo variable expansion is incomplete because it only applies to a subset of loop-carried anti-dependences. Other loop-carried dependence and loop-independent anti-dependences are ignored. The remaining anti-dependences may still cause false recurrences that cannot be eliminated by MVE.

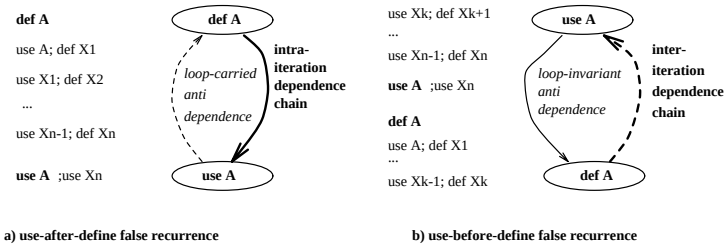


Figure 5.1: False Recurrences

This thesis proposes a renaming technique that eliminates the effect of all false recurrences in modulo scheduled loops on conventional architectures. It is called *complete renaming* because it eliminates the effect of anti-dependences completely at minimum register cost. In the following discussion, we ignore all output dependences since they can easily be eliminated by normal software renaming.

5.1.1 False Recurrence

False recurrences are the loop-carried dependence cycles that have at least one anti-dependence. This section discusses the false recurrences caused by loop-independent and loop-carried anti-dependences. We assume that a false recurrence has only one anti-dependence. This is only for the purpose of simplifying the discussion. All techniques and conclusions are valid for recurrences that include multiple anti-dependences.

Figure 5.1 (a) shows a false recurrence caused by a loop-carried anti-dependence. As shown, there is an intra-iteration true dependence chain from the define of A to its use and a loop-carried anti-dependence that completes a recurrence cycle. The loop-carried anti-dependence is due to the reuse of variable A in loop iterations. Due to the reuse of A, the use of A of the current iteration must precede the define of A of the next iteration. Figure 5.1 (b) shows a false recurrence caused by a loop-independent anti-dependence. There is an inter-loop true dependence arc from the define of A of the current iteration to the use of A of the next iteration which, coupled with the loop-independent anti-dependence, forms a recurrence. The loop-independent anti-dependence is also due to the reuse of variable A in all iterations. Due to the reuse, the use of A must precede the define of A in every iteration. In the following discussion, the reused variable of an anti-dependence means the variable whose reuse causes the anti-dependence. In the above two examples, the reused variable of the both anti-dependences is A.

To eliminate a false recurrence caused by a loop-carried anti-dependence, we must be able to rename the reused variable so that the define of the next iteration can use a different variable than the use of the current iteration and the loop-carried anti-dependence can be eliminated. To remove a false recurrence caused by a loop-independent anti-dependence, the renaming of the reused variable should make the define and the use in each iteration use two different variables so that the loop-independent anti-dependence can be removed.

5.1.2 Modulo Variable Expansion

To eliminate the effect of certain false recurrences, Lam proposed a compiler renaming technique called *Modulo Variable Expansion*[25]. This section gives a detailed description of MVE and shows why MVE is neither complete nor efficient for eliminating all false recurrences. MVE can eliminate the anti-dependence that satisfies,

- it is a loop-carried anti-dependence, and,
- the value defined in the reused variable is reproduceable in each iteration. (A value that is reproduceable in a iteration means that the value does not rely on any value computed by previous iterations.)

Figure 5.2(a) shows the effect of MVE. The renaming of MVE can be seen as two steps. First, before scheduling, MVE prunes the loop-carried anti-dependence. Without the restriction of the loop-carried anti-dependence, the scheduler can schedule the use of A of the current iteration after the define of A of the next iteration. When this happens, the lifetime of A is longer than II and must overlap with itself in the kernel. In the second step, performed after scheduling, MVE renames A into separate registers so that no lifetime of the renamed As can overlap with itself. The second step is also called kernel unrolling because the renaming requires unrolling of the kernel. (The method of kernel unrolling is described in Section 2.4.3.) In Figure 5.2(a), the variable A is renamed into two variables so that the define of A of the next iteration would not affect the use of A of the current iteration. Thus, the loop-carried anti-dependence is removed and so is the false recurrence cycle.

I call the first step of MVE, which is the pruning of the anti-dependence, *logical renaming*, since this step tells the scheduler that the variable can be renamed; I call the second step of MVE, where the variable is physically renamed into separate variables, *physical renaming*. Logical renaming enables scheduling to ignore the anti-dependence, which may lead to lifetimes longer than II; physical renaming avoids the overlapping of the long lifetimes that may result from logical

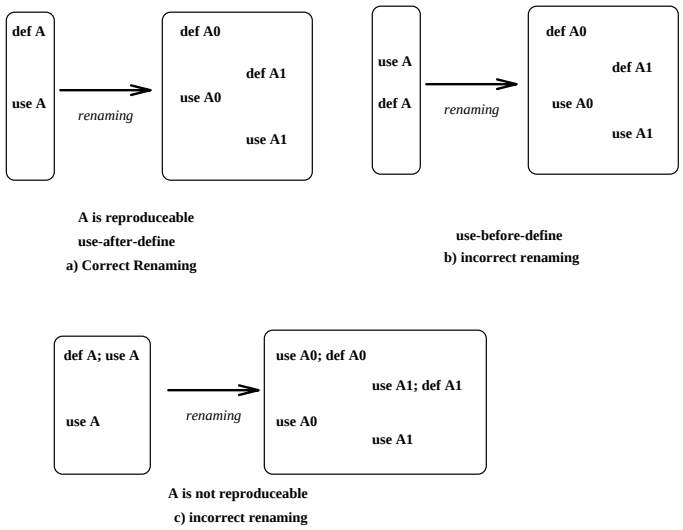


Figure 5.2: Renaming Scheme of MVE

renaming. From now on, we will not refer to MVE as a whole but to logical renaming and physical renaming separately for the purpose of clarity.

Logical and physical renaming are insufficient because, alone, they cannot handle all false recurrences; that is, they cannot eliminate all anti-dependences correctly. The renaming in Figure 5.2(a) is correct only when the anti-dependence is loop-carried and the value of the reused variable is reproduceable in each iteration. As shown in Figure 5.2 (b) and (c), for other anti-dependences, using simple logical and physical renaming can lead to incorrect renaming.

First, logical and physical renaming cannot eliminate loop-independent anti-dependences. A loop-independent anti-dependence constrains the use of A to be scheduled before the define of A in any iteration. If logical renaming took out the loop-independent anti-dependence and allowed the use of A to be scheduled after its define and physical renaming used a different variable for each adjacent iteration, the loop would not be correct after physical renaming since the use of A in one iteration would use a wrong value of A. Figure 5.2 (b) shows that after logical and physical renaming, the use of A uses a wrong value and the loop is not correct.

Second, logical and physical renaming cannot eliminate anti-dependences in which the value of the reused variable is not reproduceable in each iteration. For example, in Figure 5.2 (c), the define of A uses the value of A from the previous iteration; therefore, A is not reproduceable in each iteration. If logical renaming removes the loop-carried anti-dependence and the scheduler schedules the define of A of the next iteration before the use of A of the current iteration, physical renaming will use two different variables for any two adjacent iterations. So A of this iteration

cannot read the A of the previous iteration since the adjacent iterations use two different variables for A. The schedule after logical and physical renaming is incorrect.

Not only are simple logical and physical renaming insufficient, they also are inefficient in terms of register usage because logical renaming blindly prunes all applicable anti-dependences. When the false recurrences caused by an anti-dependence are not the limiting constraint for II, they need not be eliminated. So, preserving such anti-dependence will not affect the performance of software pipelining. However, if we unnecessarily prune the anti-dependence by blind logical renaming, scheduling and physical renaming may unnecessarily use multiple registers for the variable; therefore, software pipelining may use more registers than necessary.

To show the possible over-use of registers by logical renaming, let's consider Figure 5.2 (a) again. If the false recurrence caused by the loop-carried anti-dependence is not more restrictive than other software pipelining constraints, the minimum lifetime of A fits into II cycles. The logical renaming is unnecessary since we can schedule the use of A of the current iteration before the define of A of the next iteration without hurting the value of II. Moreover, the preserving of the anti-dependence would force the scheduler to schedule the A's lifetime to be less than II; therefore, only one register is needed for A. However, if logical renaming blindly prunes this loop-carried anti-dependence, the scheduler may create a lifetime of A that is longer than II. Subsequently, physical renaming would require more than one register for A and unnecessarily increase the register pressure.

5.1.3 Define-Substitution

Simple logical and physical renaming cannot eliminate anti-dependences that either are loop-independent or caused by a reused variable that is not reproduceable. This section proposes a technique that performs substitution to the define of the reused variable so that, after substitutions, logical and physical renaming can eliminate those two types of anti-dependences correctly. I call this substitution technique *define-substitution*. The idea is to substitute the anti-dependences that cannot be handled by logical and physical renaming with anti-dependences that can be eliminated by logical and physical renaming. Define-substitution consists of pre-substitution and post-substitution. Pre-substitution is used on loop-independent anti-dependences; post-substitution is applied to anti-dependences in which the reused variable is not reproduceable. After pre-substitution and post-

substitution, the original anti-dependences are changed to loop-carried dependences where the reused variable is reproduceable [1].

In order to substitute loop-independent anti-dependences with desirable loop-carried anti-dependences, pre-substitution copies the value of the reused variable before any of its use in each iteration. Figure 5.3 (a) shows the effect of pre-substitution on a loop-independent anti-dependence. Pre-substitution copies the value of the reused variable, A, to a substituting variable A' and changes the use of A to the use of A'. After substitution, the anti-dependence from the use of A to the define of A is changed to a loop-carried anti-dependence from the use of A' to the define of A'. A' is reproduceable in each iteration since the recurrence existing between the define of A and the define of A' ensures that A' has a correct value in each iteration. Because the new anti-dependence is loop-carried and A' is reproduceable, logical and physical renaming can be performed on A' and the false recurrence can be eliminated. In pre-substitution, the copying should be done before any use of A; only the uses of A that occur before the define of A can be changed to use A'. The placement of a copy operation in pre-substitution is the same location where software renaming, SSA, would put a ϕ node. This ϕ node merges of the value coming in from the outside of the loop and the value generated inside the loop. If the ϕ node is implemented by a copy operation, then SSA can have the effect of pre-substitution.

Post-substitution is used on anti-dependences that are caused by the reuse variable that is not reproduceable. As shown in the example in Figure 5.3 (b), post-substitution copies the reused variable, A, to a substituting variable A' after the define of A, and changes the use of A to A'. After substitution, the anti-dependence between the use of A' and the define of A' is loop-carried, and A' is reproduceable. So logical and physical renaming can be performed on A' and the anti-dependence can be removed. In post-substitution for a loop-carried dependence, the copying of A to A' should be done after the define of A but before any of its uses; only the uses of A that are after the define of A may can be changed to use A'. Post-substitution for any loop-independent anti-dependence is unnecessary since after pre-substitution, the reused variable must be reproduceable.

As define-substitution inserts a copy of the reused variable, a true and an anti-dependence are also inserted into the DDG. We must not allow these dependences to be inserted into any existing recurrences because they will increase the length of the host recurrence. Let's consider Figure 5.1 (a) for an example. If A is not reproduceable, post-substitution copies it to A' and changes $use\ A;\ def\ X1$ to use A'. This change lengthens the dependence chain from the define

[1] Here we expand the concept of reproduceable to include those values produced in remaining recurrences cycles after logical renaming. They are reproduceable in each iteration due to the constraint imposed by the recurrences.

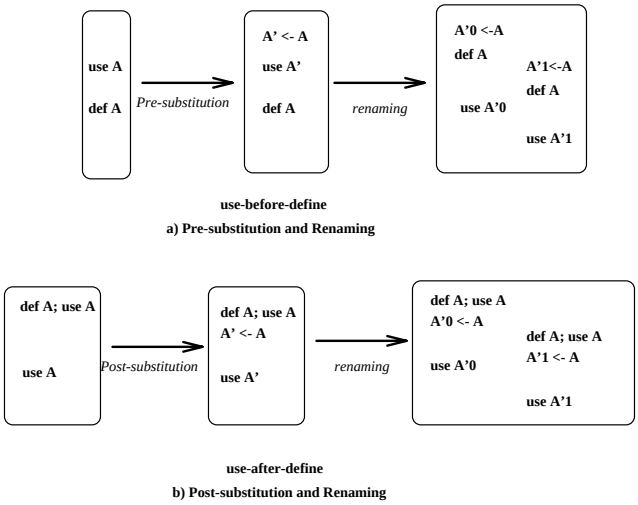


Figure 5.3: Pre and Post-Substitution

of A to the use of A. If the define and the use are in a true recurrence cycle, this change would lengthen the true recurrence cycle as well. To avoid this result, we must prohibit substituting a use of A if the use is in a true recurrence (called host recurrence) with the define of A. Although this restriction preserves the anti-dependence, it does not affect the effect of define-substitution because the preserved anti-dependence cannot cause false recurrences longer than the host true recurrence. However, a practical concern arises in the implementation of this restriction: how do we know if the define and the use of a reused variable is in a true recurrence cycle? The actual problem is more complicated because, for the purpose of efficiency, we may preserve some false recurrences in the loop and we must consider those false recurrences as well as all true recurrences. This problem can be solved; but we delay the answer to the next section. With the restriction described above, define-substitution will cause no additional recurrence to the loop body.

As shown in the above discussion, define-substitution changes the anti-dependences that cannot be handled by logical and physical renaming so that after define-substitution, all anti-dependences can be eliminated by logical and physical renaming. Although define-substitution inserts copies into the loop body, it does not create higher recurrences. Therefore, with define-substitution, the effect of all false recurrences can be eliminated for modulo scheduling.

5.1.4 Restrictive Renaming

The elimination of false recurrences increases register demand since both define-substitution and renaming require additional registers. To minimize the register cost of eliminating false re-

currences, we must avoid any unnecessary uses of define-substitution and renaming. This section describes a technique that can identify those false recurrences that need to be eliminated. By applying define-substitution and renaming only to those necessary false recurrences, unnecessary register cost can be avoided. The technique is called *restrictive renaming* since it restricts the define-substitution and renaming. Similarly we call the blind logical and physical renaming used by MVE *unrestrictive renaming*.

To minimize the use of define-substitution and renaming, we want to eliminate only false recurrences that are more restrictive than other software pipelining constraints, namely constraints due to true recurrences and constraints of limited resources in the target machine. I call the lowerbound on II due to true recurrences *TrueII*. Thus, the larger of the TrueII and ResII, *TrueMII*, is the lowerbound II due to everything but false recurrences. The purpose of restrictive renaming is to achieve TrueMII while eliminating a minimum number of false recurrences. Therefore, only false recurrences with a length greater than TrueMII need be eliminated. The problem then, becomes, finding all false recurrences and determining the length of each false recurrence.

I will categorize false recurrences by the number of anti-dependences in the dependence cycle. False recurrences that contain only one anti-dependence are denoted as α^1-recurrences. Similarly, false recurrences that contain n anti-dependences are denoted as α^n-recurrences. An α^1-recurrence cycle consists of a true dependence chain with a single anti-dependence. Let's assume that we know the length of the longest true dependence chain between any two DDG nodes. The length of each α^1-recurrence can be obtained by checking every anti-dependence: if an anti-dependence is from node x to y, the longest length of the α^1-recurrence caused by this anti-dependence is the length of the longest true dependence chain from y to x plus the timing of the anti-dependence.

The length of the longest true dependence chain between any two nodes can be found by computing the longest path problem and considering only true dependences in the DDG. (The timing of loop-carried dependences in the DDG is computed using II equal to TrueMII.) The computing of the longest path problem can be done by changing the sign of edges and computing the shortest path problem.

Using TrueMII and considering only true dependences, we can compute the longest path between any two nodes and represent it with an adjacency matrix. We call this matrix $TrueDist^0$. The longest true dependence chain from node x to y is the value in $TrueDist^0[x, y]$. If there is no path from x to y, $TrueDist^0[x, y]$ is $-\infty$. Thus, if an anti-dependence a is from node y to x, the

length of the longest α^1-recurrence caused by a is

$$timing_{anti} + (TrueDist^0[x, y] + iteration\_difference(a) * TrueMII),$$

where $iteration\_difference(a)$ is either 0 (loop-independent) or 1 (loop-carried). If the length of the α^1-recurrence is greater than 0, this false recurrence needs to be eliminated.

After finding all α^1-recurrences that need to be eliminated and the necessary anti-dependences that need to be removed, we can proceed to find all α^2 false recurrences that have a length longer than II. This can be done by computing the longest path between any two nodes that has at most one anti-dependence edge in the path. We start from $TrueDist^0$ and extend it to include all anti-dependences that are not removed in the elimination of α^1-recurrences. If a is such an anti-dependence from node x to y and $TrueDist^0[x, y]$ is $-\infty$, we change the entry to the timing of the anti-dependence. Assuming the new matrix is $Dist$, then $TrueDist^1[x, y]$ is the result of $Dist^3$. Now the length of longest path containing at most one anti-dependence between every two nodes is included in $TrueDist^1$. In similar fashion, we can find all α^2 recurrences that have a length greater than TrueMII.

We then can find α^k-recurrences $(k > 2)$ similarly. The maximum k we need to consider is the number of DDG nodes N. However, the method can stop earlier if the number of anti-dependences left is less than k when we are checking for α^k-recurrences.

Restrictive renaming is computationally expensive. The computing of $TrueDist^0$ at most can use time of $O(log_2 TrueMII * log_2 N * N^3)$. The checking of α^k false recurrences takes $O(log_2 k * N^3)$. The checking of all false recurrence can be $O(N^4)$. However, it is noticeable that restrictive renaming can find all false recurrences longer than TrueMII using only polynomial time even though there can be an exponential number of false recurrence cycles in the loop body.

Restrictive renaming, though expensive, is promising because it identifies those false recurrences that can cause undesirable recurrence constraints; by eliminating only those false recurrences, restrictive renaming minimizes the cost of define-substitution and renaming.

The following discussion answers the question raised in the last section. It is placed in this section only because the answer relies on the definition of $TrueDist^N$. The question is how to tell if two DDG nodes are in a true recurrence. Before answering the question, we need to expand the consideration to include the effect of restrictive renaming. After restrictive renaming, there are not only true recurrence cycles but also false recurrence cycles in the DDG. Therefore, define-substitution should avoid lengthening not only true recurrences but also false recurrences. Therefore, define-substitution should avoid substituting a use if the use and the define is in either a

true or a false recurrence. Then the question is how to tell if a use of A and the define of A is in any recurrence cycle in the DDG. To check if two DDG nodes x and y are in a recurrence cycle, we can check if the sum of $TrueDist^N[y,x] + TrueDist^N[x_i,y]$ is $-\infty$ or not. If it is not, they are in the same recurrence cycle; otherwise, they are not in any recurrence cycle.

5.2 Improved Algorithm for Computing RecII

The computing of RecII is the most time consuming step in modulo scheduling. The current popular method proposed by Huff takes time as high as $O(log_2N * N^3)$ in checking if a trial RecII is valid[22]. However, Huff's method is not sensitive to the complexity of the DDG; i.e. it uses same amount time in checking a trial RecII for DDGs of different complexity. This section proposes two changes to Huff's checking method so that the time required for checking a trial RecII is faster than Huff's method for DDGs that are not "completely" connected. The simpler the DDG, the faster the checking of each trial RecII. In the worst case, the new method should take only slightly longer time than Huff's method.

From the Equation 2.1, RecII can be computed if all loop-carried dependence cycles are known. However, finding all elementary cycles in a graph is generally difficult and computationally expensive because a graph can have an exponential number of cycles. Huff [22] formulated the problem as a minimal cost-time ratio problem [27]. Each dependence edge can be viewed as having a cost and a time. The cost is the negative of its timing constraint; and the time is its iteration difference. The longest recurrence cycle, which determines RecII, is the one having the smallest cost per time value.

Huff computed RecII by finding the smallest valid RecII [22] [33]. A valid RecII means that the RecII is higher than or equal to the length of any recurrence cycle in the DDG. A binary search can be used to find the smallest valid RecII to minimize the number of trials of RecII.

Huff's method for checking validity of each trial RecII is:

- Step 1, initialize matrix Dist. Given a trial RecII, initialize an adjacency matrix, $Dist$ such that $Dist[i,j]$ is the largest timing constraint from node i to node j. If there is no direct dependence from i to j, $Dist[i,j]$ is $-\infty$.
- Step 2, change the sign of all elements in $Dist$. Thus, the problem of computing the longest-path is changed to the problem of finding the shortest-path.

- Step 3, compute $Dist^N$, where N is the number of nodes in DDG. $Dist[i,j]$ now is the length of the shortest path from node i to node j.
- Step 4, change sign again and denote the result matrix as MinDist. Now $MinDist[i,j]$ is the length of the longest path from node i to node j.
- Step 5, check whether this trial RecII is valid. If, for any i, $MinDist[i,i]$ is greater than 0, then this trial RecII is not valid because it is impossible to schedule node i after node i. Otherwise, the trial RecII is valid.

In Huff's algorithm, the time needed for the checking of both a valid and an invalid trial RecII is $N^3 log_2 N$.

5.2.1 Faster Checking of Trial RecII

This section proposes a new checking method that can finish faster than Huff's method for DDGs of less complexity. The most time consuming step of Huff's method is Step 3 in which the computing of $Dist^N$ takes $O(N^3 log_2 N)$. To shorten the computing time of RecII, two changes can be made to the Step 4 and 5 so that the checking of the trial RecII can be done before finishing the computing of $Dist^N$.

First, checking an invalid trial RecII can be done before the computing of $Dist^N$. After computing of each $Dist^i$, check diagonal elements of $Dist^i$. If there is any value less than 0, this trial RecII is invalid. Actually, if element $[i,i]$ is negative in $Dist^i$, this element will be always be positive in MinDist. Therefore, if there is a negative element in a $Dist^i$, then this trial RecII is invalid. If the number of nodes in the longest recurrence cycle is M, then this new checking method takes $N^3 \lceil log_2 M \rceil$.

Second, checking a valid RecII can also be done before the computing of $Dist^N$. After the computing of $Dist^i$, check if $Dist^i$ is the same as the previous product, $Dist^{i/2}$. If it is, the trial RecII is valid. In fact, when a matrix W satisfies $W^2 = W$, then W^k is equal to W for any k. If, for any two nodes x and y, the number of nodes in the longest path from node x to node y is M, the new test method determines the validity of RecII in less than $N^3(\lceil log_2 M \rceil + 1)$.

The first new checking method takes N comparisons and the second new checking method takes time N^2 comparisons for each validation. These factors are small because the comparisons are fast operations and they can be done in parallel.

These two new checking methods are faster than Huff's method in practice because the number of nodes in the recurrence cycles and the number of nodes consists of the longest path from one node to another are much smaller than the total number of nodes in the DDG.

Eliminating Negative Cycles

One additional step is needed for the checking of valid RecII to be possible. The negative cycles need to be eliminated in initializing $Dist$ in order to ensure the convergence of $Dist$. The method for eliminating all self-negative cycles is, for any i, if $Dist[i,i]$ is less than 0, let $Dist[i,i]$ equal to 0. This additional step will remove all negative cycles in the DDG.

5.3 Chapter Summary

This chapter presented two improvements to modulo scheduling algorithm. First, an efficient software renaming technique was described for eliminating the effect of all false recurrences for modulo scheduling. Second, a modification was made to the algorithm of computing RecII in modulo scheduling so that the time needed by the computation of RecII will be sensitive to the complexity of DDGs.

To eliminate the effect of all kinds of anti-dependences, this chapter proposed a software pipelining technique called define-substitution. Previously, not all false recurrences could be eliminated by modulo variable expansion because two kinds of anti-dependences cannot be eliminated: loop-independent anti-dependences and anti-dependences in which the reused variable is not reproduceable. Define-substitution transforms these two kinds of anti-dependences so that modulo variable expansion can be correctly applied to all anti-dependences and all false recurrences can be eliminated. Define-substitution consists of pre-substitution and post-substitution. Pre-substitution changes loop-independent anti-dependences to loop-carried dependences in which the reused variable is reproduceable. Post-substitution transforms anti-dependences in which the reused variable is non-reproduceable to anti-dependences with a reproduceable reused variable.

To minimize the register cost by the renaming in the elimination of false recurrences, this chapter proposed a technique called restrictive renaming. Restrictive renaming minimizes register usage by applying define-substitution and modulo variable expansion only to those false recurrences that cause higher recurrence constraints than other software pipelining constraints. Restrictive renaming checks false recurrences containing one or more anti-dependences and finds

the anti-dependences that can cause false recurrence cycles with a length longer than other software pipelining constraints. Restrictive renaming minimizes the use of define-substitution and modulo variable expansion; therefore the additional register cost is kept to a minimum.

The second improvement proposed in this chapter is a modification to Huff's algorithm of computing RecII. The original algorithm checks each trial RecII in a fixed time for all DDGs. The time needed by the modified checking method depends on the complexity of a DDG. The complexity of a DDG here is measured by the maximum number of nodes in a recurrence and the maximum number of nodes in a path between any two nodes. The less the complexity of the DDG, the less the time needed for computing RecII by the modified method. This modified method should be faster in practice than Huff's method because most DDGs are of low complexity.

Chapter 6

Evaluation

This chapter describes two experiments performed to test the hypothesis that software pipelining can be improved by unroll-and-jam and memory reuse analysis. Section 1 describes the implementation issues of the iterative modulo scheduling method. Section 2 presents the results of combining unroll-and-jam with software pipelining. The decrease both on resource and recurrence constraints are measured. Section 3 describes the memory reuse experiment in which software pipelining was performed using each of three assumptions about memory latencies: all-cache-hit, memory reuse analysis, and all-cache-miss. The focus of that experiment is measuring the decrease in register pressure available when using memory reuse analysis compared to assuming all-cache-miss.

6.1 General Experimental Setup

The iterative modulo scheduling algorithm presented by Rau [33] is implemented in Rocket. Rocket is a retargetable optimizing compiler[40]. It performs traditional optimizations and program analyses. Rocket currently has two front ends, one for C and one for Fortran programs. The C front-end generates Rocket intermediate code directly. The Fortran front-end takes the intermediate code, ILoc, generated by ParaScope [6], and converts it to Rocket intermediate code. All Rocket backend optimizations, including software pipelining, use Rocket intermediate code. The following subsections describe implementation related issues and tradeoffs in this research.

Loop Selection

The current implementation only handles single-basic-block loops. Both C and Fortran front-ends may insert additional branches to a straight-line loop body and turn a single-block loop into a multi-block loop. Therefore, the implementation first examines each innermost loop body to see if it can be converted into straight-line code. If so, the compiler eliminates the unnecessary branches and puts the loop body into one basic block.

For each straight-line loop body, Rocket next finds the index variable and the bound variable. These two variables determine the number of iterations in the loop. Our current software pipelining implementation requires that the number of iterations be known at the starting point of the execution of the loop [1], that is, both the bound variable and the index increment must be loop invariant.

Generating the DDG

After the loop selection, the loop body is one single basic block. The DDG for this basic block is generated normally using Rocket. Then, the following changes are needed to the DDG. First, loop-carried dependences are added. Rocket computes the loop-carried scalar dependences by itself and uses the dependence analysis result of Parascope for array dependences.

Next, redundant scalar loop-carried dependences are pruned as described in Section 2.3.1.

Iterative Modulo Scheduling

Our iterative modulo scheduling implementation follows Rau's method [33] except that a modified method is used in computing RecII to make the computing faster. (See Section 5.2)

Our implementation of modulo variable expansion uses the method of computing the unroll amount that restricts the degree of unrolling but does not minimize register usage [25]. The reason is to limit the size of the kernel generated. In computing reproduceable variables, we count variables that are defined only by loop-independents, loaded values and the index variable.

6.1.1 Target Machine

The testing of the performance of software pipelining depends on choices provide by both Rocket's front-end and back-end. The front-end compiles both Fortran and C loops and the back-end

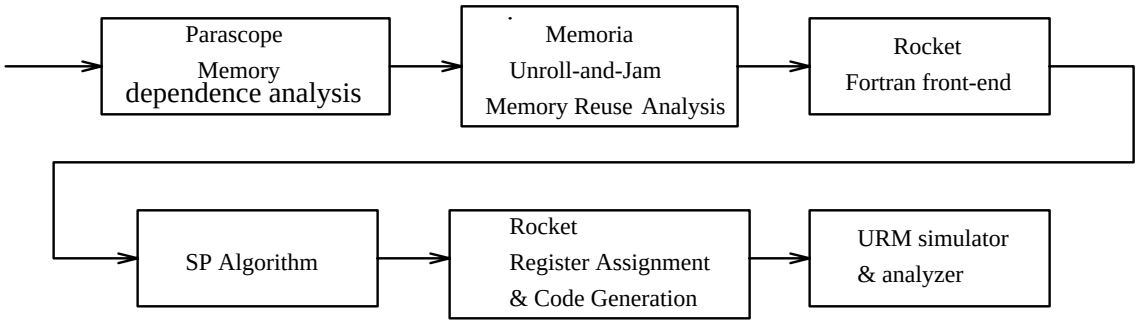
[1] It is not necessary to know the number of iterations at compile time

generates code for a collection of architectures including a URM (Unlimited Resource Machine) machine [30]. This research described here evaluates the performance of Fortran programs on the URM machine.

6.1.2 Instrumentation

The experiment uses the implementation of unroll-and-jam and memory reuse analysis in Memoria, the implementation of array analysis in Parascope[6], the software pipelining in Rocket, and the URM simulator [30]. The general experimental method is graphically described in Figure 6.1. Memoria takes Fortran programs, performs unroll-and-jam and memory reuse analysis and generates the transformed Fortran program with comments indicating the reuse information. Parascope compiles the Fortran program into Iloc intermediate code[6]. Parascope performs various global code optimizations, as well as array analysis. The result of array analyses, along with the result of memory reuse analysis generated by Memoria, are added as comments in the generated Iloc code.

Rocket's front-end converts an Iloc program into Rocket intermediate code and uses the result of array analysis and memory reuse analysis. Rocket performs several conventional compiler optimizations and generates a DDG for each basic block. The software pipelining algorithm identifies loops that can be software pipelined and converts them into single basic block loop bodies. The array analysis results passed by Parascope are used to identify and add both loop-independent and loop-carried dependences for the loop body. Loop-carried scalar dependences are computed and added by Rocket's software pipelining algorithm. Certain loop-carried anti dependences are removed due to the logical renaming of modulo variable expansion. After software pipelining, Rocket performs instruction scheduling on non-loop codes. The register assignment method in Rocket is based on the commonly-used graph-coloring paradigm. After register assignment, URM assembly code is generated. Performance results can be obtained by using the URM simulator.



Experimental set up

Figure 6.1: Experimental Method

6.2 Improvement Obtained by Using Unroll-and-Jam

This section presents the result of combining unroll-and-jam with software pipelining. A similar experiment has been done previously [8]. The results of the previous experiment were not complete for the following reasons. First, the array analysis result was added into the compiler by hand, so it is not completely accurate. Second, the implementation of modulo variable expansion is not as sophisticated as the current implementation; fewer false recurrences were eliminated in the previous experiment. Moreover, in the previous experiment, we did not perform register assignment and code generation. These shortcomings have been addressed in the latest experiment. Due to the above reasons, the results obtained in this experiment are more complete than previous results described in [8].

6.2.1 Machine Model

For our target architecture we chose a machine with four integer units and two floating-point units. Only one of the integer units can be used for memory operations. Each integer operation has a latency of two cycles while each floating-point operation has a latency of four cycles. We assume that the machine has an unlimited number of registers when performing software pipelining and collect the result of register usage both with and without using unroll-and-jam. The unroll-and-jam configuration assumes a machine with 64 registers and limits its unrolling accordingly. The reason that we chose a disproportionate number of integer functional units is to compensate for the lack of addressing-mode optimization in Rocket.

6.2.2 Test Cases

Our experimental test suite includes the Perfect, SPEC and RiCEPS benchmark suites. Each benchmark was examined for loops with nesting depth of two or more. We then applied our transformation system to those loops to determine the improvement in the software pipelining initiation interval available when using unroll-and-jam. We investigated 64 nested loops in the benchmark suites to which unroll-and-jam can be applied. Of those loops, unroll-and-jam unrolled 26. The remaining 38 loops were not unrolled because Memoria's heuristics for loop unrolling suggested that no benefit would accrue from unrolling the loop. Memoria uses two simple heuristics to determine when to unroll-and-jam. If Memoria recognizes that the loop balance is greater than the target machine balance, it will unroll in an attempt to lower the loop balance. Additionally if Memoria detects an inner-loop recurrence, it will unroll the outer loop N times where N represents the target machine's floating point pipe depth. This attempts to remove any pipeline interlock within the inner loop. In the 38 loops Memoria failed to unroll, no inner-loop recurrence was found. In addition, one (or both) of two conditions led Memoria to conclude that improved loop balance was not possible through unroll-and-jam. Thirty-one loops contain no outer-loop-carried dependences so no improvement of the balance was possible. Eighteen loops were already determined to be in balance and so no unrolling was deemed necessary to achieve peak efficiency. Of the 26 unrolled loops three contained intrinsic function calls and were therefore not considered in this study since our measurement of RecII is not accurate in the presence of function calls. The software pipelining implementation successfully scheduled all but one of the remaining loops. In addition to the benchmark loops, 4 loops extracted from Fortran kernels are also included. Therefore, a total of 26 loops were tested.

6.2.3 Test Results

Of the 26 loops which we both unrolled and software pipelined, all 26 showed schedule improvements due to the unroll-and-jam procedure. Table 6.1, lists the 22 benchmark loops as well as the 4 kernel loops along with their improvements in initiation interval due to unroll-and-jam.

The most noteworthy result from Table 6.1 is that, not only did all 26 unrolled loops show schedule improvements by software pipelining when unroll-and-jam was applied first, but that the amount of improvement is quite significant, ranging from a low of 29.0% to a high of 94.2%. Performing an unweighted average on the % Improvement column shows that, on average, the 26 loops showed an improvement of 56.9%. Of the 26 loops unrolled, 18 showed improvement greater

Program	SubProgram/ Loop Number	Before Unrolling RecII	ResII	II	After Unrolling times	RecII	ResII	II	% Improvement
RiCEPS									
simple	conduct1	7	4	7	17	7	52	52	56.3
	conduct2	8	6	9	51	8	131	132	71.2
SPEC									
dnasa7	btrix1	12	17	18	12	12	104	104	51.9
	btrix2	11	22	22	5	11	52	52	52.7
	btrix3	12	41	41	2	12	58	58	29.3
	btrix4	11	22	22	5	11	51	51	53.6
	gmtry3	4	8	8	49	4	99	99	74.7
	vpenta1	8	23	23	3	8	49	49	29.0
	vpenta2	9	18	18	3	9	30	30	44.4
PERFECT									
adm	radb2	12	26	26	10	13	94	94	63.8
	radbg1	7	3	7	50	7	52	52	85.1
flo52	collc	7	5	7	4	7	17	17	39.3
	dflux2	7	6	7	25	7	102	102	41.7
	dflux3	7	5	7	25	7	77	77	56.0
	dflux5	7	5	7	3	11	9	11	47.6
	dflux6	7	5	7	4	7	14	14	50.0
	dflux7	7	4	7	25	7	76	76	56.6
	dfluxc1	7	6	7	8	8	26	26	53.6
	dfluxc3	7	6	7	16	7	60	60	46.4
	dfluxc4	7	7	8	17	7	87	87	36.0
	dfluxc5	7	4	7	25	7	52	52	70.3
	dfluxc6	7	4	7	25	7	76	76	56.6
KERNELS									
	dmxpy	10	4	10	49	8	51	53	89.2
	fold	10	4	10	19	4	11	11	94.2
	mmjik	10	4	10	4	4	10	10	75.0
	sor1k	22	11	22	18	8	179	179	54.8
AVERAGE									56.9

Table 6.1: Software Pipelining Improvement with Unroll-and-Jam

than or equal to 50%, corresponding to a speed-up of 2 or better; 3 loops showed an improvement over 80%, a speed-up of more than 5. These significant improvements come from the following three sources.

First, unroll-and-jam decreases the unit ResII (ResII divided by unroll factor) by reducing the memory resource demand in the innermost loop. Before unroll-and-jam, all loops have high memory resource demand. The machine can issue only one memory operation per cycle. As a result, the memory bottleneck creates a high ResII. Other resources are not fully utilized due to the memory bottleneck. Unroll-and-jam reduces the number of memory operations and subsequently reduces the ResII. The average decrease in unit ResII of all loops tested is 42.7%.

Second, unroll-and-jam reduces the effect of recurrences in the innermost loop because it can be used to change any recurrence-bounded loop into a resource-bounded loop. After unroll-and-jam, the RecII of the innermost loop remains the same or less for 23 out of 26 loops. Four loops showed a decrease on RecII. The reason is that before unrolling, the loops have memory load and store in the longest recurrence while after unroll-and-jam, the load and store are eliminated by scalar replacement; so the length of the recurrence cycle is shortened. Of the 3 loops with an increased RecII, 2 of them have an increase of 1 cycle and 1 increases 4 cycles. These small increases for the three loops could be due to some minor false recurrences created by unroll-and-jam. These false recurrences can be eliminated either by using hardware support of rotating registers or by the software techniques presented in Chapter 5.

Third, unroll-and-jam eliminates the effect of the interaction between recurrence and resource constraints. The interaction between recurrence and resource constraints is triggered by the high recurrence in the innermost loop, i.e. RecII is close or high than ResII. As unroll-and-jam decreases unit RecII much faster than it does unit ResII, the RecII after unroll-and-jam is much smaller than ResII. When RecII is much lower than ResII, the interaction between recurrence and resource constraints is very unlikely to happen. In the experiment, software pipelining could not achieve lower-bound II for 3 out of 28 loops before unroll-and-jam, but all of the three loops achieved lower-bound II after unroll-and-jam. We attribute this result to unroll-and-jam converting a loop so that RecII is much smaller than ResII and thus reducing the effect of interaction between recurrence and resource constraints. In total, there are two loops that could not achieve lower-bound II after unroll-and-jam although the RecII of both loops are much smaller than the ResII. The reason is not quite clear. Examination of the two loops showed that both loops are very balanced and all machine resources are heavily used. This heavy resource usage may create difficulties for the scheduler to schedule all available resources while observing all timing constraints. Because the

scheduler we used is a heuristic-based method, it does not guarantee an optimal solution. Therefore, the reason of why these two loops does not achieve lower-bound may not due to the interaction between recurrence and resource constraints but due to the inability of the scheduler to examine all possible schedules.

Both the second and third effect of unroll-and-jam on software pipelining are due to the fact that unroll-and-jam can exploit cross-loop parallelism and the increased parallelism can reduce the effect of recurrence constraints and the interaction between recurrence and resource constraints. In this experiment, the second and the third effect are the sources of the rest of the improvement other than that caused by the decrease of the resource constraint, which is the part between the decrease of 42.7% on ResII and 56.9% on overall II.

The experiment also measures the change on the register pressure after unroll-and-jam. Table 6.2 gives the register pressure before and after unroll-and-jam for the 25 out of the 26 loops tested [2]. The I-Reg and F-Reg column shows the number of integer and floating-point registers used. The register estimation algorithm in the unroll-and-jam used in this research considers only floating-point registers and does not consider the effect of the overlapping execution in software pipelining. In the experiment, unroll-and-jam assumes a machine which has 64 floating-point registers. However, the software pipeline of 7 out of 26 loops used more than 60 floating-point registers after unroll-and-jam; two of them used 81 floating-point registers. This shows that the effect of overlapping execution can be significant on register pressure.

The major source of additional floating-point registers needed is the kernel unrolling of modulo variable expansion, in which one variable can be expanded into several variables. Unroll-and-jam does not consider the effect of kernel unrolling. As discussed in Section 3.4, using MinDist may help unroll-and-jam to estimate the effect of kernel unrolling.

The integer register pressure of the loops tested tends to be much higher than floating-point registers. Eighteen out of 26 loops had a higher integer register pressure than its floating-point register pressure. So predicting integer register pressure can be more important to achieving a feasible software pipeline schedule, at least for this test suite. However, predicting integer register pressure is even more difficult than predicting floating-point register pressure because it involves the consideration of not only the computation in the loop but also the address computation, control structure, etc.

[2]The register requirement of simple/conduct2 is not collected because the compiler ran out of memory in code generation.

Program	SubProgram/ Loop Number	Before Unrolling			After Unrolling			
		II	I-Reg	F-Reg	times	II	I-Reg	F-Reg
RiCEPS								
simple	conduct1	7	19	5	17	52	118	85
SPEC								
dnasa7	btrix1	18	37	14	12	104	73	81
	btrix2	12	45	15	5	52	72	36
	btrix3	41	98	104	2	58	98	97
	btrix4	22	46	15	5	51	74	44
	gmtry3	8	18	9	49	99	141	153
	vpenta1	23	59	37	3	49	81	51
	vpenta2	19	46	28	3	30	62	53
PERFECT								
adm	radb2	26	42	15	10	94	114	64
	radbg1	7	16	3	50	52	94	145
flo52	collc	7	20	3	4	17	36	14
	dflux2	7	22	7	25	102	150	71
	dflux3	7	20	3	25	77	114	36
	dflux5	7	22	11	3	11	37	17
	dflux6	7	22	6	4	14	40	17
	dflux7	7	19	7	25	76	113	68
	dfluxc1	7	28	11	8	26	50	23
	dfluxc3	7	24	7	16	60	137	51
	dfluxc4	8	27	9	17	87	151	81
	dfluxc5	7	20	5	25	52	152	45
	dfluxc6	7	20	7	25	76	112	68
KERNELS								
	dmxpy	10	14	5	49	53	79	67
	fold	10	13	5	19	11	8	24
	mmjik	10	15	5	4	10	12	16
	sor1k	22	18	5	18	179	57	22

Table 6.2: Register Pressure Change with Unroll-and-Jam

Overall, this experiment certainly supports the hypothesis that unroll-and-jam can improve software pipelining's ability to generate efficient code. First, unroll-and-jam can remove memory bottlenecks and reduce resource constraint by 42.7%. Second, unroll-and-jam can increase the amount of parallelism in the innermost loop by exploiting cross-loop parallelism. The increased parallelism can reduce the effect of recurrence constraints and the interaction between resource and recurrence constraints. Combined with the first factor, unroll-and-jam achieves an average of 56.9% improvement on II of the 26 benchmark and kernel loops tested, corresponding to a speedup of more than 2 on kernels.

6.3 Improvement Obtained by Using Memory-Reuse Analysis

6.3.1 Machine Model

For testing the effect of memory reuse analysis, we chose a URM machine with two integer units and two floating-point units. Either of the integer units can be used for memory operations. Each integer operation has a latency of two cycles while each floating-point operation has a latency of four cycles. A cache hit takes two cycles and a cache miss takes an additional 25 cycles. Because we are interested in knowing the number of registers needed for software pipelining, the machine has an unlimited number of registers.

6.3.2 Test Cases

Table 6.3 gives the programs tested using all-cache-hit, all-cache-miss assumption and memory reuse analysis. The first column is the name of the program in each benchmarks. The second and third column gives the number of lines in the program and the number of loops software pipelined and tested. Not all loops in the programs are tested. Loops containing conditional structures and function calls are not software pipelined. Only the innermost loops which successfully compiled and software pipelined are listed.

6.3.3 Results

Table 6.4 gives software pipelining results when assuming all memory operations are cache hits. This assumption uses the smallest number of registers. However, when a cache miss happens, the software pipeline stalls until the cache miss is resolved. In our machine model, the stall takes 25 cycles. So if there is a cache miss in the innermost loop, the actual II in execution is

Program	No. of Lines	No. of Loops Tested
SPEC		
hydro2d	4461	84
su2cor	2413	30
swm256	523	14
OTHERS		
kernels	180	12
TOTAL		140

Table 6.3: Test Loops for Memory Reuse Analysis

Program	ResII	RecII	II	MinLT	MaxLive	Reg
SPEC						
hydro2d	5.05	6.60	7.83	45.61	17.78	18.71
swm256	18.57	21.71	27.14	194.43	35.79	37.93
su2cor	11.23	11.00	15.07	115.24	25.37	27.33
OTHERS						
kernels	4.67	7.42	8.25	43.50	18.25	18.58

Table 6.4: Result using All-Cache-Hit Assumption

25 cycles more than that shown, which is a severe speed penalty to software pipelining. We could not run our test cases on a real machine, therefore, we do not have statistical results on the speed penalty when using the all-cache-hit assumption.

Table 6.5 shows the results when using memory reuse analysis. For memory accesses that Parascope array analysis can not predict, we assume they are all cache misses to avoid any possible speed penalty. The result shows a modest increase in RecII, II and register usage.

The increase of RecII is due to the fact that a long cache miss latency may introduce higher false recurrences. The length of true recurrences is not increased since when a load or store is in a

Program	ResII	RecII	II	MinLT	MaxLive	Reg
SPEC						
hydro2d	5.05	7.38	8.56	58.29	19.33	20.71
swm256	18.57	44.79	50.07	383.86	40.79	43.57
su2cor	11.23	17.27	20.37	157.64	26.90	28.17
OTHERS						
kernels	4.67	7.42	8.25	45.58	18.67	19.34

Table 6.5: Result using Memory Reuse

Program	ResII	RecII	II	MinLT	MaxLive	Reg
SPEC						
hydro2d	5.05	9.77	10.79	116.94	24.05	26.37
swm256	18.57	50.21	55.36	572.78	50.21	55.86
su2cor	11.23	30.30	32.57	338.10	29.24	31.40
OTHERS						
kernels	4.67	13.67	14.50	99.75	22.00	24.00

Table 6.6: Result using All-Cache-Miss Assumption

true recurrence cycle, memory reuse analysis can recognize that it must be a cache hit. As described in Chapter 5, false recurrences are formed by a long dependence chain and an anti-dependence. Although modulo variable expansion is used, it does not eliminate all false recurrences. So RecII is increased due to long cache-miss latencies. However, since false recurrences can be eliminated by using either the substitution technique described in Chapter 5 or hardware support of rotating registers, the increase in RecII can be avoided if those two techniques are used. The increase in register usage is a natural consequence since some memory operations are cache misses and the corresponding lifetimes are longer than those in the all-cache-hit assumption.

Table 6.6 lists the results of assuming all memory operations are cache misses. The results show a significant increase in RecII, II, and, register demand.

There two reasons for the increase in RecII. First, as when using memory reuse analysis, long cache-miss latencies introduce higher recurrences. Because the number of long cache-miss latencies in all-cache-miss assumption is larger than the number of long latencies when using memory reuse analysis, the effect of false recurrences on RecII can be higher. All-cache-miss assumption can also lengthen true recurrences when a load or store is in a true recurrence cycle. However, the increase on the length of the true recurrence cycles could be avoided if we performed scalar replacement before software pipelining.

Table 6.7 shows the decrease on II and register usage using memory reuse analysis over assuming all-cache-miss. Not only does using memory reuse analysis decrease the registers used by 10% to 22%, it also decreases the II by 10% to 43%. The decrease in II shows a speedup of 1.1 to 1.8 over the II using all-cache-miss assumption. However, as discussed before, the decrease on II would not be present if scalar replacement is used and all false recurrences are eliminated. However, if scalar replacement and false-recurrence elimination were included, we would expect the register usage to rise even more dramatically.

The decrease in registers used shows the degree to which assuming all-cache-miss over-use registers. However, the comparison on the registers used is not on an equal basis. Due to the higher II in all-cache-miss assumption than in memory reuse analysis, the misuse of registers by assuming all-cache-miss can be much more. The dramatically higher II of the all-cache-miss policy helps to relieve the register pressure in the software pipelines because the software pipeline length is increased. In a lengthened software pipeline schedule, the lifetimes can be scattered and decrease the value of MaxLive. A longer software pipeline schedule also gives each register longer available time, which increases the chances of register reuse. As a result, both MaxLive and the number of registers used is reduced by the higher II.

Because MaxLive and the number of register used do not accurately reflect the decrease on register usage because of their dependence on II, we want to use another measure not dependent on II. The lifetime of variables is not directly dependent on II, therefore it is more accurate assessment of the effect on register of the different latency assumptions. The lower-bound on the sum of the lifetime of all variables, MinLT, can be computed accurately using Huff's method [22]. MinLT is also independent of any particular software pipeline schedule. The result in the third column shows a more dramatic decrease, ranging from 33% to 54%, on MinLT when using memory reuse analysis over assuming all-cache-miss. The decrease in MinLT shows the decrease in register usage when the machine has unlimited resources and no resource conflict ever occurs.

However, MinLT is not accurate in practice. The hardware constraints restrict the schedule and lengthen the lifetime of variables. To measure the increase on register usage on a given machine, we can measure the sum of the actual length of variable lifetimes. Unfortunately, we did not measure the actual lifetimes in our experiment. However, the product of II and the number of register used can give us a good estimate of the sum of the actual variable lifetimes. If registers are considered as a resource used to hold the lifetimes, the product of the number of registers and II shows how many resources were used in the software pipeline. If the register assigner does a good job in assigning lifetimes to registers, the product of the number of register used and II should be a close estimate of the total length of actual lifetimes. The fifth column in Table 6.7 shows the decrease on the product of registers used and II, which ranges from 29% to 54%.

From the decrease on MinLT and the product of II and the number of registers used, it is quite possible that the decrease on register pressure when using memory reuse analysis over assuming all-cache-miss ranges from 20% to 50% if II remains the same in both assumptions.

Program	decrease on II	decrease on MinLT	decrease on Reg	decrease on Reg X II
SPEC				
hydro2d	22%	50%	21%	39%
swm256	10%	33%	22%	29%
su2cor	37%	53%	10%	44%
OTHERS				
kernels	43%	54%	19%	54%

Table 6.7: Decrease on II and Register Pressure: Memory Reuse vs. All-Cache-Miss

Chapter 7

Conclusion

The high performance of today's microprocessors is achieved mainly by fast and multiple issuing hardware and optimizing compilers that together exploit instruction-level parallelism (ILP) in programs. Software pipelining is a very effective ILP compiler technique because it can exploit inter-iteration parallelism of loops. However, four difficulties have, to date, prevented software pipelining from achieving better performance. First, the parallelism in the innermost loop may not be sufficient. Although outer loops in nested loops may have sufficient parallelism, they are typically ignored in software pipelining. Second, if the loop has a high demand on the memory resources and the machine cannot satisfy that demand by providing substantial parallelism in memory accesses, the resulting memory bottleneck will severely degrade overall machine utilization. Third, the cache structure in modern microprocessors generates uncertain latencies on memory operations. These uncertain memory latencies can cause significant hardware misuse in software pipelining. Finally, the software pipelining algorithm may not be able to exploit all available parallelism in the innermost loop. In particular, the popular iterative modulo scheduling technique [25] [33] does not eliminate all the false recurrences which, in turn, hurts performance for most ILP machines. These four difficulties cause over-restrictive software pipelining constraints which unnecessarily limit the performance of software pipelining.

This research investigated these four problems, found solutions to each of them, and examined the effectiveness of some of those solutions.

7.1 Contributions

First, this research verifies, both in theory and in experiment, that unroll-and-jam does exploit cross-loop parallelism for software pipelining. In theory, this research shows that the new recurrences created by unroll-and-jam are either false recurrences or recurrences that can be eliminated by simple renaming. Therefore, RecII remains the same before and after unroll-and-jam and the unit RecII decreases in proportion to the degree of unrolling. Experimental evidence showed that, of all loops unroll-and-jammed and software pipelined, no cross-loop recurrence other than false recurrences occurred in the tested loops. So, using unroll-and-jam, the problem of insufficient parallelism in the innermost loop can be relieved for nested loops.

Second, this research measured the benefit of removing the memory bottleneck by unroll-and-jam for software pipelining. For a collection of benchmark loops, unroll-and-jam on average decreased the resource constraint by 40%. This is equivalent to a speedup in excess of 1.6 and demonstrates the benefit of using unroll-and-jam to remove memory bottlenecks to software pipelining in nested loops.

However, this research also found that register pressure can restrict the application of unroll-and-jam. Large degrees of unrolling caused dramatic increases in register pressure, especially integer register pressure. Current algorithms for predicting register pressure for unroll-and-jam are not sufficient because they consider neither integer register pressure nor the overlapping effect of software pipelining. Experimental evidence showed the unrolling degree picked by the current unroll-and-jam heuristic was too high and that register requirements of the transformed loop often exceeded 80 integer registers and 40 floating-point registers.

Summarizing the experimental results then, leads to the conclusion that unroll-and-jam can solve the problems of insufficient parallelism and the memory bottleneck for software pipelining as long as the transformed loop does not lead to exploding register pressure. If the unroll-and-jam algorithm can effectively control the register pressure of the transformed loop, unroll-and-jam will be a very effective technique for improving software pipelining on nested loops with either insufficient innermost-loop parallelism or high demand of memory operations. As shown in the experiments, the potential improvement can be very dramatic. For machines with a high degree of ILP or a low proportion of memory resources, unroll-and-jam should be applied because the performance of software pipelining is more likely to be restricted by insufficient parallelism and the memory bottleneck. If the machine has a large number of registers, e.g. partitioned register file, the benefit of unroll-and-jam should be maximally exploited. However, if the target machine

has limited hardware ILP, relatively sufficient memory resources and a limited number of registers, unroll-and-jam may not achieve a significant improvement for software pipelining.

Another contribution of this research is the study of memory reuse analysis to solve the hardware misuse problem caused by uncertain memory latencies. Analyses showed that assuming all-cache-hit can cause severe speed penalties because when a cache miss occurs, the software pipeline may stall until the cache miss is finished. To avoid this speed penalty, many researchers adopt an all-cache-miss assumption in software pipelining [34] [22] [17] [16]. However, assuming all-cache-miss leads to much higher register requirement than necessary. This research compares assuming all-cache-miss with using memory reuse analysis. For over a hundred benchmark loops tested, using memory reuse analysis decreased II from 10% to 43% and register pressure from 10% to 22%. Chapter 6 outlines how the higher II caused when assuming all-cache-miss over memory reuse analysis can be eliminated with more sophisticated software or hardware techniques. However, eliminating the difference in II for all-cache-miss vs. memory reuse analysis, the all-cache-miss assumption will use even more registers. Thus, I conclude that the memory reuse analysis can achieve much more efficient hardware usage for software pipelining than the alternative strategies; therefore, memory reuse analysis should always be used with software pipelining.

The final contribution of this research is a new method to efficiently eliminate the effect of all false recurrences in modulo scheduling. These new techniques, define-substitution and restrictive renaming, represent an improvement over the current method of modulo scheduling. Modulo variable expansion[25] is shown to be incomplete and inefficient for this purpose. Define-substitution can be used to eliminate all kinds of false recurrences; restrictive renaming eliminates a minimum number of false recurrences so that the register cost can be kept in minimum. By eliminating the effect of all false recurrences using define-substitution and restrictive renaming, modulo scheduling can exploit all available parallelism in loops without special hardware support.

7.2 Future Work

The implementation of define-substitution and restrictive renaming should be done in order to experimentally evaluate their effectiveness and efficiency. They should be implemented as a separate step after the DDG of the loop body has been obtained and ResII has been computed but before logical renaming and scheduling. In theory, define-substitution can eliminate all false recurrences for conventional machines without rotating registers; but modulo variable expansion cannot. Restrictive renaming uses fewer registers than blind logical renaming. However, we

do not know how much improvement define-substitution and restrictive renaming will provide compared to modulo variable expansion and blind logical renaming. After the implementation of define-substitution and restrictive renaming, we can perform software pipelining on large number of benchmark loops and kernels. By comparing RecII, II and register usage of software pipelining between define-substitution and modulo variable expansion and between restrictive renaming and unrestrictive renaming, the effectiveness of define-substitution and restrictive renaming on real programs can be evaluated.

Using unroll-and-jam in conjunction with software pipelining also requires further study. Unroll-and-jam has been shown to be a very effective technique in improving software pipelining for nested loops. To fully exploit the benefit of unroll-and-jam, however, the degree of unrolling must be controlled so that the register pressure after software pipelining does not exceed the number of registers on the machine. To more accurately measure the register pressure of unroll-and-jam on software pipelining, we need to consider the overlapping effect of software pipelining. As suggested in Chapter 3, the overlapping effect may be estimated by using a $MinDist$ matrix.

In addition, to measure the benefit of memory reuse analysis over the all-cache-hit assumption, we need to either apply software pipelining on a real machine or simulate software pipelines on an analyzer that can simulate cache behavior. To accurately measure the decrease of register pressure of software pipelining between using memory reuse analysis and assuming all-cache-miss, we need to eliminate all false recurrences. This can be done after the implementation of define-substitution and restrictive renaming.

Finally, the implementation of software pipelining should be generalized to general loops. If-conversion should be used to convert a multiple basic-block loop-body into a single basic-block loop body. The data dependence analysis technique should be changed to avoid adding loop-independent dependence between two operations of disjoint execution paths. The algorithm of modulo variable expansion, define-substitution and restrictive renaming should consider the effect of multiple execution paths as well. The current code generation scheme is designed with the assumption that the loop body is a single basic block. Reverse if-conversion needs to be implemented if there is no hardware support for predicate execution[44]. For loops in which the index variable and the bound variable are not loop invariant, we need to allow the software pipeline to finish on any iteration by either assuming a control dependence from any early exit to any later operation or by using special hardware support of *speculative code motion* [28][42] to relax the control dependence. For loops with multiple exits where the loop count is known prior to loop

entry, code generation schemes such as multiple-postlude can be used as described by Rau et al[35]. For loops with calls to small functions, we can inline the called function and then perform software pipelining.

References

[1] A. Aiken and A. Nicolau. Perfect Pipelining: A New Loop Optimization Technique. In *Proceedings of the 1988 European Symposium on Programming, Springer Verlag Lecture Notes in Computer Science, #300*, pages 221–235, Atlanta, GA, March 1988.

[2] A. Aiken and A. Nicolau. A development environment for horizontal microcode. *IEEE Transactions on Software Engineering*, 14(5):584–594, May 1988.

[3] V. H. Allan, U. R. Shah, and K. M. Reddy. Petri net versus modulo scheduling for software pipelining. In *Proceedings of the 28th International Symposium on Microarchitecture (MICRO-28)*, pages 125–133, Ann Arbor, MI, December 1995.

[4] E.R. Altman, R. Govindarajan, and G.R. Gao. Scheduling and Mapping: Software Pipelining in the Presence of Structural Hazards. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation"*, pages 139–150, La Jolla, CA, June 1995.

[5] D. Callahan, S. Carr, and Ken Kennedy. Improving register allocation for subscripted variables. In *PLDI90*, pages 53–65, White Plains, NY, June 1990.

[6] D. Callahan, K. Cooper, R. Hood, K. Kennedy, and L. Torczon. Parascope: A parallel programming environment. In *Proceedings of the First International Conference on Supercomputing*, Athens, Greese, 1987.

[7] D. Callahan and K. Kennedy. Estimating interlock and improving balance for pipelined machines. *Journal of Parallel and Distributed Computing*, 5:334–358, 1988.

[8] S. Carr, C. Ding, and P. Sweany. Improving Software Pipelining With Unroll-and-Jam. In *28th Hawaii International Conference on System Sciences*, 1996.

[9] S. Carr and K. Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions on Programming Languages and Systems*, 16(6):1768–1810, 1994.

[10] S. Carr and K. Kennedy. Scalar replacement in the presence of conditional control flow. *Software Practice and Experience*, 24(1):51–77, January 1994.

[11] S. Carr, K.S. McKinley, and C.W. Tseng. Compiler optimizations for improving data locality. In *ACM SIGPLAN NOTICES*, volume 29, pages 252–262, Nov 1994.

[12] S. Carr and Q. Wu. An analysis of unroll-and-jam on the HP 715/50. Technical Report CS-95-03, Department of Computer Science, Michigan Technological University, Hougton, MI, 1995.

[13] A.E. Charlesworth. An approach to scientific array processing: The architectural design of the AP_120B/FPS_164 family. *Computer*, pages 18–27, Sept. 1981.

[14] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck. An efficient method of computing static single assignment form. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, 1989.

[15] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. A practical data flow framework for array reference analysis and its use in optimizations. *SIGPLAN Notices*, 28(6):68–77, June 1993. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*.

[16] A. E. Eichenberger and E. S. Davidson. Stage scheduling: A technique to reduce the register requirement of a modulo schedule. In *Proceedings of the 28th International Symposium on Microarchitecture (MICRO-28)*, pages 338–349, Ann Arbor, MI, December 1995.

[17] A. E. Eichenberger, E. S. Davidson, and S. G. Abraham. Minimum register requirements for a modulo schedule. In *Proceedings of the 27th International Symposium on Microarchitecture (MICRO-27)*, pages 63–74, San Jose, CA, December 1994.

[18] M. Flynn. Very high-speed computing systems. *Proc. IEEE*, pages 1901–1909, 1966.

[19] G.R. Gao, W-B. Wong, and Q. Ning. A Timed Petri-Net Model for Fine-Grain Loop Scheduling. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 204–218, June 26-28 1991.

[20] Gina Goff, Ken Kennedy, and Chau-Wen Tseng. Practical dependence testing. In *PLDI*, pages 15–29, June 1991.

[21] R. Govindarajan, E. R. Altman, and G. R. Gao. Minimizing Register Requirements under Resource-Constrained Rate-Optimal Software Pipelining. In *Proceedings of the 27th Microprogramming Workshop (MICRO-27)*, pages 85–94, San Jose, CA, November 1994.

[22] R. A. Huff. Lifetime-Sensitive Modulo Scheduling. In *Conference Record of SIGPLAN Programming Language and Design Implementation*, June 1993.

[23] D. Kuck. *The Structure of Computers and Computations Volume 1*. John Wiley and Sons, New York, 1978.

[24] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 207–218, January 1981.

[25] M.S. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 318–328, Atlanta, GA, June 1988.

[26] D. M. Lavery and W. W. Hwu. Unrolling-Based Optimizations for Modulo Scheduling. In *Proceedings of the 28th International Symposium on Microarchitecture (MICRO-28)*, pages 327–337, Ann Arbor, MI, December 1995.

[27] E. L. Lawler, H. Rinehart, and Winston Pub. *Combinatorial Optimization: Networks and Matroids*. 1976.

[28] Scott A. Mahlke, William Y. Chen, Wen-Mei W. Hwu, B. Ramak rishna Rau, and Michael S. Schlansker. Sentinel scheduling for VLIW and superscalar processors. In *ASPLOS5*, volume 27, pages 238–247, Boston, MA, Oct 1992.

[29] A. Nicolau and J.A. Fisher. Measuring the parallelism available for very long instruction word architectures. *IEEE Transactions on Computers*, 33(11):968–976, Nov 1984.

[30] David Poplawski. The unlimited resource machine (URM). Technical Report CS-TR 95-1, Department of Computer Science, Michigan Technological University, Hougton, MI, February 1995.

[31] M. Rajagopalan. *A New Model for Software Pipelining Using Petri Nets*. Master's thesis, Department of Computer Science, Utah State University, Logan, UT, July 1993.

[32] M. Rajagopalan and V. H. Allan. Specification of Software Pipelining using Petri Nets. *International Journal on Parallel Processing*, 3(22):279–307, 1994.

[33] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the 27th International Symposium on Microarchitecture (MICRO-27)*, pages 63–74, San Jose, CA, December 1994.

[34] B. R. Rau, M. Lee, P.P. Tirumalai, and M. S. Schlansker. Register Allocation for Software Pipelined Loops. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation"*, pages 283–229, San Francisco, CA, June 1992.

[35] B. R. Rau, M. S. Schlansker, and P.P. Tirumalai. Code Generation Schema for Modulo Scheduled Loops. In *Proceedings of Micro-25, The 25th Annual International Symposium on Microarchitecture*, December 1992.

[36] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle. The Cydra 5 Departmental Supercomputer: Design Philosophies, Decisions, and Trade-offs. *The IEEE* Computer, pages 12–25, January 1989.

[37] B.R. Rau and J.A. Fisher. Instruction-level parallel processing: History, overview and perspecitve. *Journal of Supercomputing*, 7(1/2):9–50, 1993.

[38] M. Schlansker, V. Kathail, and S. Anik. Height reduction of control recurrences for ILP processors. In *Proceedings of the 27th International Symposium on Microarchitecture (MICRO-27)*, pages 40–51, December 1994.

[39] B. Su, S. Ding, J. Wang, and J. Xia. GURPR - A Method for Global Software Pipelining. In *Proceedings of the 20th Microprogramming Workshop (MICRO-20)*, pages 97–105, Colorado Springs, CO, December 1987.

[40] Philip H. Sweany and Steven J. Beaty. Overview of the ROCKET retargetable C compiler. Technical Report CS-94-01, Department of Computer Science, Michigan Technological University, Hougton, MI, January 1994.

[41] J. E. Thornton. Parallel operation in the control data 6600. In *Proc. AFIPS Fall Joint Computer Conference*, pages 33–40, 1964.

[42] P. Tirumalai, M. Lee, and M.S. Schlansker. Parallelization of Loops with exits on pipelined architectures. In *Proceedings of SuperComputing '90*, pages 200–212, November 1990.

[43] G.S. Tjaden and M.J. Flynn. Detection and parallel execution of parallel instructions. *IEEE Transaction on Computers*, 19(10):889–895, oct 1970.

[44] N.J. Warter, G.E. Haab, and J.W. Bockhaus. Enhanced Modulo Scheduling for Loops with Conditional Branches. In *Proceedings of the 25th Annual International Symposium on Microarchitecture (MICRO-25)*, pages 170–179, Portland, OR, December 1-4 1992.

[45] N.J. Warter, Scott A. Mahlke, Wen-Mei W. Hwu, and B. R. Rau. Reverse If-Conversion. In *PLDI*, pages 290–299, Albuquerque, NM, June 1993.

[46] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. 26(6):30–44, June 1991.