AN IMPLEMENTATION OF SWING MODULO SCHEDULING WITH EXTENSIONS FOR SUPERBLOCKS

BY

TANYA M. LATTNER

B.S., University of Portland, 2000

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2005

Urbana, Illinois

Abstract

This thesis details the implementation of Swing Modulo Scheduling, a Software Pipelining technique, that is both effective and efficient in terms of compile time and generated code. Software Pipelining aims to expose Instruction Level Parallelism in loops which tend to help scientific and graphical applications.

Modulo Scheduling is a category of algorithms that attempt to overlap iterations of single basic block loops and schedule instructions based upon a priority (derived from a set of heuristics). The approach used by Swing Modulo Scheduling is designed to achieve a highly optimized schedule, keeping register pressure low, and does both in a reasonable amount of compile time.

One drawback of Swing Modulo Scheduling, (and all Modulo Scheduling algorithms) is that they are missing opportunities for further Instruction Level Parallelism by only handling single basic block loops. This thesis details extensions to the Swing Modulo Scheduling algorithm to handle multiple basic block loops in the form of a superblock. A superblock is group of basic blocks that have a single entry and multiple exits. Extending Swing Modulo Scheduling to support these types of loops increases the number of loops Swing Modulo Scheduling can be applied to. In addition, it allows Modulo Scheduling to be performed on hot paths (also single entry, multiple exit), found with profile information to be optimized later offline or at runtime.

Our implementation of Swing Modulo Scheduling and extensions to the algorithm for superblock loops were evaluated and found to be both effective and efficient. For the original algorithm, benchmarks were transformed to have performance gains of 10-33%, while the extended algorithm increased benchmark performance from 7-22%.

Acknowledgments

The implementation and writing of this thesis has been challenging, stressful, yet fulfilling and rewarding. While I feel a sense of pride in what I have accomplished, I would not have completed this without the immense love and support from my husband Chris. I have watched him achieve inspirational success in his own educational pursuits and learned a great deal from him. He has always stood by me despite my frequent stress-induced break downs. I can not thank him enough for his patience, understanding, encouragement, and love.

I would also like to thank my parents, Greg and Ursula Brethour. You both have supported my seemingly crazy decision to attempt graduate school. Thank you for your love and support, and for bringing me up with the determination to succeed no matter what the adversity.

I am also very grateful to Jim Ferguson, of NCSA, for allowing me to reduce my appointment and attend graduate school. Without your letter of recommendation, financial support, and understanding, this would not have been possible. I learned a lot from you and my fellow DAST coworkers, and I am extremely grateful for everything.

Special thanks to my advisor, Vikram Adve, who helped me pursue my dreams of writing a thesis. Thank you for your guidance, knowledge, and support.

Lastly, I owe a lot to the friends I have made during my years at UIUC. Thank you for the laughter, fun, memories, and for keeping me sane.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Modern compilers implement several optimizations that extract the parallelism in programs in order to speed up program execution or to utilize multiple processor machines more effectively. Many of these optimizations are applied to loops. Such optimizations include loop distribution, loop interchange, skewing, tiling, loop reversal, and loop bumping [36]. Other techniques unroll loops to increase the size of the loop body to increase potential for scheduling. This produces a more efficient schedule, but ignores the parallelism across loop iterations [31].

Often, these techniques are not successful due to dependences between instructions and across iterations of the loop. A more sophisticated approach, *Software Pipelining*, reconstructs the loop such that each iteration of the loop is executed at a constant interval, producing an optimal schedule. This approach aims to keep the processor's pipeline full and ultimately speed up the execution time of the program.

Software Pipelining has existed for many years and has been proven to be a viable scheduling solution for VLIW (Very Long Instruction Word) and superscalar architectures. As more architectures emerge that provide support for Software Pipelining [20, 24, 37], existing algorithms are being refined, and new techniques are being developed.

In the early days of Software Pipelining, the techniques would schedule instructions from several iterations and look for a pattern [36]. *Modulo Scheduling* is a family of Software Pipelining techniques that uses a "modulo" technique (instead of maximal unrolling) to place instructions in the schedule such that when iterations are overlapped there are no resource or data conflicts. This thesis describes an implementation and extension of one such algorithm, *Swing Modulo Scheduling*.

1.1 Modulo Scheduling

The idea behind Modulo Scheduling can easily be illustrated with the following example. Figure 1.1 shows a sequence of instructions that represent the body of a loop. Assume that this sequence of instructions reads a value from an array, adds some constant to that value, and stores that value. For this simple example, there are only data dependences within one iteration, since subsequent iterations access a different element within the same array.

```
1   load
2   add
3
4   store
```

Figure 1.1: Single Iteration of a Loop

Assume that the add instruction is a single-stage pipelined operation that takes two cycles, while the load and store both take one cycle. As you can see the original loop takes 4 cycles between the initiation of each iteration.

Modulo Scheduling attempts to minimize the time between initiations of loop iterations, and lower the overall execution time of the loop. It does this by modifying the loop such that there are no resource or data conflicts and attempts to keep the processor's pipeline constantly full (i.e. no stalls). Figure 1.2 shows how the code is structured into three sections such that the ramp up stage begins to fill the pipeline, steady state is reached where the pipeline is always full, and finally it ramps down as the pipeline empties and the loop terminates. The goal is to ensure that the majority of the execution time is spent in steady state.

The loop is reconstructed into 3 components [31] related to the pattern described in Figure 1.2:

- Prologue: A sequence of instructions leading to steady state.
- Kernel: Instructions that are executed on every cycle of the loop once its reached steady state.
- Epilogue: Instructions that are executed to finish the loop.

Using the example in Figure 1.1, if we look at 4 iterations of the loop and pick an instruction from each iteration, we can form a kernel.

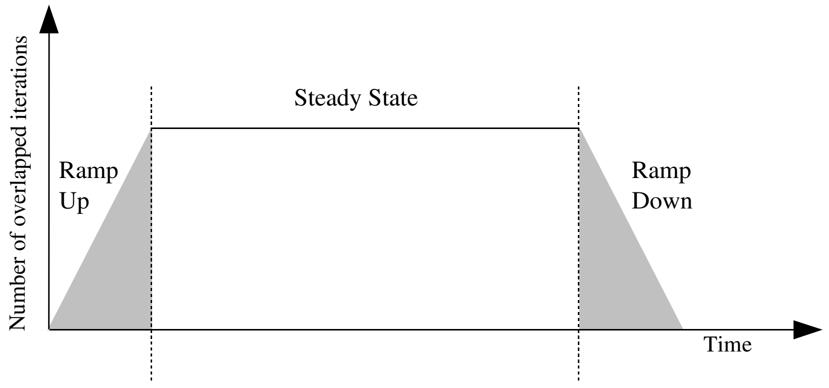
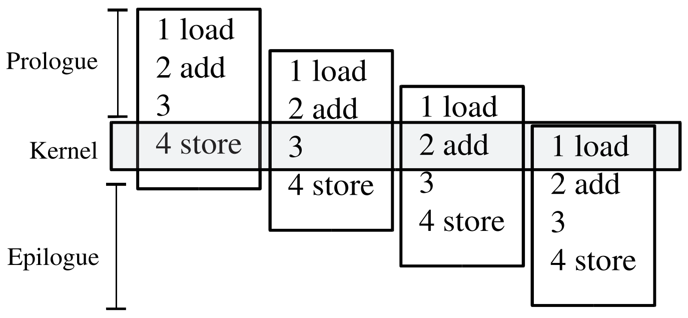


Figure 1.2: Pattern of Software Pipelined Loop

Figure 1.3(a) shows the iterations and instructions selected for the kernel. The prologue for our simple loop is composed of all the instructions above the kernel, and the epilogue are all the instructions below the kernel. With this reconstructed loop, the kernel (Figure 1.3(b)) now initiates iterations of the loop in three cycles instead of the original loop's four cycles. If this loop has a large iteration count, the majority of the time would be spent executing the kernel's optimal schedule and speeding up the overall execution time of the program by 1.3 times.

Modulo Scheduling algorithms traditionally combine a set of heuristics and list scheduling to create this kernel. These techniques are discussed further in Chapter 3. List scheduling and other scheduling terms are explained in Chapter 2.



(a) Software Pipelined Loop

1	store
2	
3	add
4	load

(b) Kernel

Figure 1.3: Software Pipelined Loop with Prologue, Kernel, and Epilogue

1.2 Research Contributions of this Thesis

The first main contribution of this thesis is an implementation of Swing Modulo Scheduling in the LLVM Compiler Infrastructure [26]. The results show that it is an effective and efficient Modulo Scheduling technique. Our implementation shows that Swing Modulo Scheduling successfully achieves an optimal schedule while reducing register pressure (for most cases).

The second main contribution of this thesis is to extend the Swing Modulo Scheduling algorithm to handle more than single basic block loops. We have modified the algorithm to handle superblock loops (single-entry, multiple-exit, multiple basic block loops). These extensions allow the algorithm to be applied at various stages of compilation:

- Statically: Just as the original Swing Modulo Scheduling algorithm was done statically, the extensions simply expand the number of loops acceptable for this transformation.
- Offline: Using profile information, hot paths (subsets of the loop in the form of single entry, multiple exit loops), can be recognized and be transformed by our extensions.
- Runtime: Similar to the profile driven optimization, hot paths are found during program execution, and are transformed dynamically using the extended algorithm.

1.3 Organization of this Thesis

This thesis begins with background information in Chapter 2 on scheduling. In order to understand why Swing Modulo Scheduling was chosen, an overview of Modulo Scheduling and Global Modulo Scheduling approaches, along with related work is presented in Chapter 3. The implementation of Swing Modulo Scheduling is explained in Chapter 4 and extensions to the algorithm are detailed in Chapter 5. The implementation and extensions of this algorithm are evaluated in Chapter 6. Finally, Chapter 7 discuss potential future work, and concludes this work.

Chapter 2

Scheduling Background

Instruction Scheduling aims to rearrange instructions to fill the gap that the delay between dependent instructions creates. If other instructions were not scheduled in this gap, the processor would stall and waste cycles. Instruction scheduling is typically done after machine independent optimizations, and either before or after register allocation. Depending upon the compiler infrastructure, scheduling is done on target machine assembly, or on a low-level representation that closely models assembly.

All Instruction Scheduling techniques aim to produce an *optimal schedule*, a schedule with the shortest length. Schedule length is measured as the total execution time in cycles. In addition, the optimal schedule must be found in a reasonable amount of time.

Instruction Scheduling algorithms must satisfy both dependence and resource constraints when creating a schedule. Dependence constraints are determined by constructing a data dependence graph, a directed graph whose nodes represent instructions, and edges represent a dependence between instructions. A dependence between two instructions is formed when two instructions have a common operand, and one of those instructions defines the operand.

If resource constraints are met, the schedule will not require more resources then the architecture has available. Instruction Scheduling must have a resource usage model that breaks down the resource per pipeline stage for each category of instructions (i.e., loads, integer arithmetic, floating point arithmetic, etc). Using this resource model, the scheduler can populate a *resource reservation table*. A resource reservation table is a matrix in the form of $r \times c$ where, r is the resources, and c is the cycles of the schedule. Each entry in this table is an instruction that uses a resource for that cycle.

While finding the optimal schedule is the main goal, Instruction Scheduling must also be aware of the potential increase of *register pressure* by reordering instructions. Register pressure is a measure of the number of live values at a given point in the program. A *Live Range* is a range from when a value is defined, to its final use in the program. A value is live at a given point if the last use of that value has not occurred. Because architectures have limited registers, the number of live values should not exceed the total number of registers available. If that value has been exceeded, the registers are *spilled* to memory and loaded again when needed. Reading and writing to memory can be quite costly.

Instruction Scheduling can be broken into three categories: *Local Scheduling*, *Global Scheduling*, and *Cyclic Scheduling* [36]. Local Scheduling handles single basic blocks which are regions of straight line code that have a single entry and exit. Global Scheduling can handle multiple basic blocks with acyclic control flow, and Cyclic Scheduling handles single or multiple basic blocks with cyclic control flow. Software Pipelining falls into the latter category.

Local Scheduling typically uses a form of *List Scheduling*. List Scheduling schedules instructions starting at cycle zero, until all instructions have been scheduled. For each cycle it maintains a list of ready instructions (those with no resource or dependence conflicts), and schedules that list in an order based upon some heuristic. Here are a few of the traditional heuristics used:

- Maximum distance (latency) from the node, to a node without successors. This is also known as *height-based* priority.
- Maximum number of children, direct or all descendants.
- Smallest ESTART value, where ESTART is equal to the total latency of the node's predecessors.
- Smallest LSTART value, where LSTART is equal to the total latency of the node's successors.
- Lower *Mobility*, where Mobility is the difference between LSTART and ESTART.
- Nodes on the critical path, which means they have a mobility of zero.

Local Scheduling is limited because it operates only on single basic blocks which are typically not very large. Therefore, while optimal schedules may be found for those small regions, the overall impact on performance could be quite small. Global Scheduling schedules instructions from multiple basic blocks and overlaps the execution of instructions from different basic blocks. There exist many Global Scheduling algorithms [36] such as:

- Trace Scheduling: It identifies frequently executed traces in the program and treats the path as an extended basic block which is scheduled using a list scheduling approach.
- Superblock Scheduling: Superblocks are a subset of traces which have a single entry and multiple exit attributes (therefore they are traces without side exits). List scheduling is typically used to schedule the superblock.
- Hyperblock Scheduling: Excessive control flow can complicate scheduling, so this approach uses a technique called If-Conversion [3] to remove conditional branches. If-Conversion is discussed in Section 3.2.2.

With this background in Instruction Scheduling, Software Pipelining, a form of Cyclic Scheduling, will be discussed in great detail in Chapter 3.

Chapter 3

Previous Work

Software Pipelining [9] is a group of techniques that aim to exploit Instruction Level Parallelism (ILP) by overlapping successive iterations of a loop. Over the years, two main approaches to Software Pipelining have developed: *Move-then-Schedule*, and *Schedule-then-Move*. The Move-then-Schedule techniques [16, 30, 14, 22], which will not be discussed in this thesis, move instructions across the back-edge of the loop in order to achieve a pipelined loop. The Schedule-then-Move algorithms attempt to create a schedule that maximizes performance and constructs a new pipelined loop composed of instructions from current and previous iterations.

The Schedule-then-Move group of techniques is further decomposed into two families. The first is known as *Unroll-based Scheduling*, which use loop unrolling while scheduling to form a software pipelined loop. It repeats this process until the schedule becomes a repetition of an existing schedule. As one can speculate, this type of approach often leads to high time complexity. The second group, *Modulo Scheduling* [27, 33, 12, 21, 4, 16, 28], aims to create a schedule with no resource or dependence conflicts that can be repeated at a constant interval. Since Swing Modulo Scheduling (SMS) falls into the second category, this thesis will briefly describe a few of the other well known algorithms in this category.

Modulo Scheduling is traditionally restricted to single basic block loops without control flow, which can limit the number of candidate loops. Global Software Pipelining techniques have emerged to exploit some of the opportunities for ILP in multiple basic block loops that frequently occur in computation intensive applications. We will explore a few techniques in this area, as it directly relates to the SMS extensions discussed in Chapter 5.

3.1 Modulo Scheduling Approaches

Modulo Scheduling techniques typically use heuristic based approaches to find a near-optimal schedule. While there exist other approaches, such as enumerating all possible solutions and choosing the best one [4], finding the optimal schedule is an NP-complete problem. Therefore, most production compilers [18] implement Modulo Scheduling using heuristic based algorithms.

Modulo Scheduling algorithms exhibit the same pattern when pipelining a loop (Figure 3.1). Each begins by constructing a *Data Dependence Graph* (DDG). Using the DDG, the *Minimum Initiation Interval* (MII), which is the minimum amount of time between the start of successive iterations of a loop, is computed. Modulo Scheduling algorithms aim to create a schedule with an *Initiation Interval* (II) equal to MII, which is the smallest II possible and results in the most optimal schedule. The lower the II, the greater the parallelism.

MII is defined to be the maximum of the resource constrained II (ResMII), and recurrence constrained II (RecMII) of the loop. The exact ResMII may be calculated by using reservation tables, a method of modeling resource usage, but this can lead to exponential complexity [33]. Modulo Scheduling algorithms typically use an approximation by computing the total usage count for each resource and using the most heavily used resource count as ResMII.

Recurrences in the data dependence graph occur when there is a dependence from one instruction to another from a previous iteration. These loop-carried dependences have a distance property which is equal to the number of iterations separating the two instructions involved. Using the data dependence graph, all recurrences are found using any circuit finding algorithm[1]. For each recurrence, II is calculated using the total latencies (L) of all the instructions, the total distance (D), and the following constraint: $L - II * D <= 0$. The recurrence with the highest calculated II sets the RecMII.

```
$\forall b \in$ Single basic block loops without control flow
    $DDG$ = Data dependence graph for $b$
    $MII = \max(RecMII, ResMII)$
    $Schedule(b)$        //Algorithms differ on this step
    $Reconstruct(b)$     //Reconstruct into prologue, kernel, epilogue
```

Figure 3.1: Pseudo Code for General Modulo Scheduling

[1]Circuit finding algorithms find all circuits (a path where the first and last node are identical) where no vertex appears twice.

Using the MII value as their initial II value, the algorithms attempt to schedule each instruction in the loop using some set of heuristics. The set of heuristics used varies widely across implementations of Modulo Scheduling. If an optimal schedule can not be obtained, II is increased, and the algorithm attempts to compute the schedule again. This process is repeated until a schedule is obtained or the algorithm gives up (typically because II has reached a value greater than the original loop's length in cycles).

From this schedule, the loop is then reconstructed into a prologue, a kernel, and an epilogue. The prologue begins the first n iterations. After $n * II$ cycles, a steady state is achieved and a new iteration is initiated every II cycles. The epilogue finishes the last n iterations. Loops with long execution times will spend the majority of their time in the kernel.

A side effect of Modulo Scheduling is that register pressure is inherently increased when overlapping successive iterations. If register pressure increases beyond the available registers, registers must be spilled and the effective II is unintentionally increased. If this situation arises, the Modulo Scheduled loop is typically discarded, and the original loop is used instead.

This thesis will briefly discuss three Modulo Scheduling algorithms which use the pattern mentioned above, and that are similar to SMS.

3.1.1 Iterative Modulo Scheduling

Iterative Modulo Scheduling [33] (IMS) uses simple extensions to the common acyclic list scheduling algorithm and the height-based priority function. IMS begins by constructing a standard data dependence graph, but also includes two pseudo-operations: START and STOP. The START node is made to be the predecessor of all nodes in graph and the STOP node is the successor to all nodes in the graph.

IMS then proceeds to calculate MII, which is the maximum of ResMII and RecMII (Section 3.1). Using MII as the initial II value, IMS schedules all the instructions using a modified acyclic list scheduling algorithm. IMS's list scheduling algorithm differs from traditional list scheduling in the following ways:

- IMS finds the optimal time slot for each instruction instead of scheduling all instructions possible per time slot. Additionally, instructions can be unscheduled and then rescheduled.

- When determining which instruction to schedule next, the instruction with the highest priority is returned based upon a given priority scheme. Instructions may be returned more then once since they may be unscheduled and, later, rescheduled.

- ESTART is a property that represents the earliest time an instruction may be scheduled (based upon the predecessors in the partial schedule). Because an instruction's predecessors can be unscheduled and a node can be rescheduled, ESTART maintains a history of past values and uses either the current ESTART (if it is less than the last ESTART value), or one cycle greater than the last ESTART. This is to prevent instructions from repeatedly causing each other to be unscheduled and rescheduled with no change in the schedule.

- A special version of the schedule reservation table, a *modulo reservation table*, is used in order to adhere to the modulo constraint. Each instruction uses time-slot modulo II when being inserted into the schedule.

- The maximum time slot an instruction may be scheduled is limited to the $minTime + II - 1$, which differs from traditional list scheduling that uses ∞ as its maximum time.

- If a schedule could not be found, the algorithm gives up.

IMS extends the height based priority function to take loop carried dependencies into consideration. An instruction's height is equal to the height of the node in the graph, minus the product of II and the distance from the instruction to its predecessor.

Once a scheduling order has been determined, the range of time slots each instruction may be issued to is determined by predecessors already inserted into the schedule. ESTART is calculated considering those immediate predecessors, which preserves the dependence between the instruction and its predecessors. The dependence between the instruction being scheduled and its successors is preserved by asserting that if any resource or dependence conflict should occur, the instruction's successors are unscheduled. There is no strategy for determining which successor to remove, as IMS removes them all. The displaced instructions will then be rescheduled at a later time.

An extensive study was done on the effectiveness of IMS [10] and other Modulo Scheduling approaches. It determined that IMS has a high register requirement (unlike SMS), but computes a schedule with near optimal II for complex architectures[2].

3.1.2 Slack Modulo Scheduling

Slack Modulo Scheduling [21] (Slack) is a bidirectional Modulo Scheduling strategy that schedules some instructions late, and some early. As with all Modulo Scheduling algorithms, Slack attempts to create sufficient iteration overlap such that the loop initiates iterations at the maximum possible issue rate. The Slack scheduling algorithm takes resource and recurrence constraints, register pressure, and critical paths into consideration when performing instruction scheduling and to limit back-tracking. The term "slack" refers to the amount of freedom the instruction has in its placement in the schedule.

The algorithm begins by constructing a data dependence graph (DDG). Using the DDG, it computes the MII value by examining the resource and recurrence constraints. It uses the same algorithm as IMS to determine ResMII and RecMII, and uses the maximum as the value for MII. Like IMS, it uses two pseudo instructions: START and STOP. START is a predecessor to all nodes in the DDG and is scheduled at a fixed issue slot, cycle 0. STOP is a successor to all nodes in the DDG and is scheduled like any other instruction. The purpose of the START and STOP nodes are to ensure that the ESTART and LSTART are well defined for all instructions (there is never a situation where a node does not have a predecessor or successor in the schedule).

During all stages of the algorithm, the scheduler maintains a minimum distance relation (MINDIST) between all pairs of instructions. Maintaining the MINDIST relation effectively keeps track of the earliest start time (ESTART) and the latest start time (LSTART) bounds for each instruction, given the partial schedule. MINDIST is the minimum number of cycles that a given instruction must be issued before its successor. If there is no path between two instructions, MINDIST is set to $-\infty$. The MINDIST computation is reduced to an all-pairs shortest path algorithm by negating the distances on each dependence. It is important to note that MINDIST must be recalculated for each new II.

The Slack scheduling algorithm is outlined as follows:

1. **Selecting an Instruction to be Scheduled**: Slack chooses instructions with the minimum number of issue slots, conflict free placements, to be scheduled first. This property is called

[2]Complex architectures are those which pipeline simple instructions, while complex ones (fp division, modulo, square root) are not [10].

the dynamic priority of an instruction. The number of issue slots is approximated by the slack value when there is no resource contention. If contention occurs, the number of issue slots is approximated by dividing the slack value in half. If there is a tie for dynamic priority, the instruction with the lowest LSTART is chosen. The slack value is the difference between LSTART and ESTART.

2. **Choosing the Issue Cycle**: A sophisticated heuristic that analyzes flow dependencies is used to determine if the instruction should be placed as early as possible or as late as possible.

3. **Scheduling an Instruction**: If no conflict free issue slot exists for the instruction being scheduled, then one or more instructions must be ejected from the schedule. The instructions to be ejected are those issued after LSTART for the instruction being scheduled. If an instruction is being rescheduled, the maximum of ESTART and one plus the last placement slot, is used as the ESTART value. This prevents instructions from continuously ejecting each other from the schedule. ESTART and LSTART for all instructions are updated after an instruction is successfully scheduled or unscheduled.

4. **Increment II**: If instructions are being ejected too many times, all are removed from the schedule, and II is incremented. The Slack scheduling steps are then repeated until a schedule is found or the algorithm gives up.

According to a study [10] that compared various modulo scheduling approaches, Slack did approximately the same as SMS in reducing register pressure on all types of architectures, but required much more compile time. SMS is usually able to compute a better schedule then Slack for architectures of low[3] and medium[4] complexity and Slack was also beat by IMS on higher complexity architectures.

3.1.3 Integrated Register Sensitive Iterative Software Pipelining

Integrated Register Sensitive Iterative Software [12] (IRIS) Pipelining uses an integrated approach composed of modified heuristics proposed in Stage Scheduling [17] and Rau's iterative method [33].

[3]Low complexity architectures are those with fully pipelined instructions with 3 integer and 3 floating point units, and issue width of 8 instructions [10].

[4]Medium complexity architectures have fully pipelined instructions, but only 2 integer and 2 floating point units and an issue width of 4 instructions [10].

This approach aims to achieve a high initiation rate while maintaining low register requirements. IRIS is a bidirectional strategy and schedules instructions as early or as late as possible.

IRIS is similar to IMS, described in Section 3.1.1, but contains the following modifications:

- Earliest start (ESTART) and latest start (LSTART) are calculated as described by the Slack Modulo Scheduling algorithm [21]. This creates a tighter bound on LSTART.
- Instructions are placed as early or as late as possible in the schedule, which is determined by using modified Stage Scheduling [17] heuristics. The search for an optimal issue slot is done from ESTART to LSTART, or vice-versa.

IRIS is identical to IMS in that it uses the same height-based priority function, the same thresholds to determine when a schedule should be discarded and II increased, and the same technique to eject instructions from the schedule.

This algorithm differs mainly in its use of modified Stage Scheduling [17] heuristics, which are used to determine which direction to search for a conflict-free issue slot. The heuristics are as follows:

1. If the instruction is a source node in the DDG, the partial schedule is searched for any successors. If one or more exist, the algorithm searches from LSTART to ESTART for an issue slot.
2. If the instruction is a sink node in the DDG, and only has predecessors in the schedule, then the search for an issue slot begins from ESTART to LSTART.
3. If this instruction has only successors in the partial schedule, and forms a cut edge[5], then the schedule is scanned from LSTART to ESTART for an open time slot, and vice-versa for predecessors.
4. If an instruction does not fall into any of the categories above, it begins searching for an issue slot from ESTART and ends with LSTART.

[5]A cut edge is an edge whose removal from a graph produces a subgraph with more components than the original graph.

According to the comparative study [10], both IRIS and IMS do fairly well, in terms of finding an optimal schedule, on complex architectures since they are both iterative techniques. However, IRIS was least effective in terms of register requirements when compared against SMS for all types of architectures.

3.1.4 Hypernode Reduction Modulo Scheduling

Hypernode Reduction Modulo Scheduling [28] (HRMS) is another bidirectional technique that uses an ordering phase to select the order in which instructions are scheduled. HRMS attempts to shorten the lifetime of loop variants without sacrificing performance.

Like other Modulo Scheduling approaches, HRMS computes the MII from the resource and recurrence constraints and creates a data dependence graph for the program. HRMS is unique in how it orders the instructions to be scheduled. The ordering phase guarantees an instruction will only have predecessors or successors in the partial schedule: The only exception are recurrences, to which have priority. The ordering phase is only performed once, even if II increases.

The ordering phase is an iterative algorithm and for each iteration the neighbors of a *Hypernode* are ordered, and then reduced into a new Hypernode. A Hypernode is a single node that represents a node or subgraph of the DDG. The ordering pass is easily explained for graphs without recurrences. The basic algorithm will be presented first, and then the modifications made for recurrences will be discussed.

For a graph without recurrences, the initial Hypernode may be the first node or any node in the DDG. The predecessors and successors of a Hypernode are alternatively ordered with the following steps:

1. The nodes on all paths between the predecessors/successors are collected.

2. The predecessor/successor nodes from the previous step and the Hypernode are reduced into a new Hypernode.

3. A topological sort is done on the subgraph that the Hypernode represents, and the resulting sorted list is appended to the final ordered list.

4. The steps are repeated until the graph is reduced to a single Hypernode.

Graphs with recurrences are processed first by the ordering phase, and no single node is selected as the initial Hypernode. The recurrences are first sorted according to their RecMII, with the highest RecMII having priority, resulting in a list of sets of nodes (each set is a recurrence). If any recurrence shares the same back edge as another, the sets are merged together into the one with the highest priority. If any node is in more than one set, it is removed from all but the recurrence with the highest RecMII. Instead of ordering predecessors and successors alternatively to this Hypernode, the ordering phase does the following, beginning with the first recurrence in the list:

1. Find all the nodes from the current recurrence to the next in the list. This is done with all back edges removed in order to prevent cycles.

2. Reduce the recurrence, the nodes collected from the previous step, and the current Hypernode (if there is one), into a new Hypernode.

3. Perform a topological sort on the subgraph that the Hypernode represents, and append the nodes to the final ordered list.

4. Repeat the above steps until the graph is reduced to one without recurrences, and then use the algorithm described for graphs without recurrences.

The scheduling phase of HRMS uses the final ordered list and attempts to schedule instructions as close as possible to their predecessors and successors already in the partial schedule. It uses the same calculations for the start and end cycles that SMS does, which will be discussed in Section 4.6. If there are no free slots for the instruction, the schedule is cleared, II is increased, and scheduling begins again.

HRMS is the algorithm that has the most in common with SMS. They both find optimal schedules in a reasonable amount of compile time. However, because they differ in how the nodes are ordered for scheduling (HRMS does not take into consideration the criticality of nodes), HRMS is not as successful as SMS in achieving low register pressure.

3.2 Global Modulo Scheduling

While Modulo Scheduling is an effective technique for scheduling loop intensive programs, it is limited to single basic block loops (without control flow). These restrictions cause many Software Pipelining opportunities on complex loops to be missed. Therefore, a family of techniques that work on complex loops, called *Global Modulo Scheduling* emerged.

As mentioned previously, there are two groups of Schedule-then-Move techniques: Unrolling based, and Modulo Scheduling. There are several unrolling based global Software Pipelining techniques that are able to handle loops with control flow: Perfect Pipelining [2], GURPR* [38], and Enhanced Pipelining [15]. Since the focus of this thesis is on Modulo Scheduling, these techniques will not be discussed.

Global Modulo Scheduling approaches are typically techniques that transform a complex loop into a single basic block of straight line code, and then perform Modulo Scheduling as normal. Code generation is slightly more challenging as the original control flow needs to be reconstructed within the new pipelined loop. There are two well known techniques for transforming complex loops into straight line code: Hierarchical Reduction [25] (described in Section 3.2.1) and If-conversion [3] (described in Section 3.2.2). Last, Enhanced Modulo Scheduling [41] builds off the ideas behind If-conversion and Hierarchical Reduction.

3.2.1 Hierarchical Reduction

Hierarchical Reduction [25] is a technique to transform loops with conditional statements into straight line code which can then be modulo scheduled using any of the techniques previously discussed. The main idea is to represent all the control constructs as a single instruction, and schedule this like any other instruction. Lam modeled her technique after a previous scheduling technique by Wood [42], where conditional statements were modeled as black boxes taking some amount of time, but further refined it so the actual resource constraints would be taken into consideration.

Hierarchical reduction has three main benefits. First, it removes conditional statements as a barrier to Modulo Scheduling. Second, more complex loops are exposed that typically contain a significant amount of parallelism. Finally, it diminishes the penalty for loops that have short execution times because it exposes the opportunity to modulo schedule outer loops that could contain these short inner loops. Hierarchical reduction requires no special hardware support.

This technique schedules the program hierarchically starting with the inner most control constructs. After scheduling the construct, it is reduced to a single node that represents all the resource constraints of its components. A program is successfully scheduled after it is reduced to a single node. The Hierarchical Reduction technique is described in detail in the following steps:

1. The instructions corresponding to the THEN and ELSE branches of a conditional statement are scheduled independently.

2. All scheduling constraints are captured by examining the modulo reservation table and taking the maximum of all the entries for the two branches. A node is created to represent the entire conditional statement.

3. The data dependence graph is updated by replacing all the instructions that are represented by this new node, and the dependences are preserved between what this node represents and other instructions.

4. Finally, the steps are repeated until the whole program has been scheduled and reduced to a single node.

Hierarchical Reduction does require some changes to code generation of the new pipelined loop. For code scheduled in parallel with the conditional statement, that code is duplicated in both branches. It is important to note that while Hierarchical Reduction does successfully represent the recurrence constraints of conditional constructs, it does take the worst case as its value. However, this may not be the path most taken in the loop, and the resulting II may not be truly optimal.

3.2.2 If-Conversion

If-conversion is another technique for transforming loops with conditional statements into straight line code. The idea is to associate the control dependence to a variable which exposes the relationships between instructions in terms of data flow. Essentially, the control dependence is converted to a data dependence [3]. One of the more popular If-conversion algorithms is the RK algorithm [32], and many Modulo Scheduling approaches [34, 13, 41] use it.

If-conversion replaces conditional branches with a compare instruction that sets a flag. Instructions that were dependent upon the conditional branch are now instructions that only execute if the flag is set. If-conversion typically requires hardware support, but some algorithms [41] have made slight modifications to avoid this. Hardware support such as predicated execution, common on VLIW and EPIC architectures (such as IA64) set a conditional flag per instruction and allow instructions to execute only when the conditional flag is true.

While If-conversion does allow loops with conditional statements to be software pipelined, the downside is that both execution path's resources must be summed when determining the resource constraints for the loop. This can lead to a pipelined loop that does not have an optimal II.

3.2.3 Enhanced Modulo Scheduling

Enhanced Modulo Scheduling [41] (EMS) is another Modulo Scheduling technique to modulo schedule loops with conditional branches by translating them into straight line code. If-conversion and Hierarchical Reduction both place restrictions on the scheduling of instructions that may prevent Modulo Scheduling from achieving an optimal II. EMS attempts to avoid these problems by combining the best of both algorithms. It uses If-conversion, without special hardware support, to eliminate prescheduling conditional constructs. It uses the regeneration techniques like Hierarchical Reduction to insert conditional statements back after Modulo Scheduling.

EMS consists of five basic steps: Applying If-conversion, generating the data dependence graph, Modulo Scheduling the loop, applying modulo variable expansion, and finally regenerating the explicit control structure of the code by inserting conditional branches.

If-conversion is performed to transform the loop body into straight line predicated code using the RK algorithm [32]. By using an internal predicated representation similar to the Cydra 5 processors [34], conditional branches are replaced by a predicate definition and are assigned to the appropriate basic blocks. Each basic block is assigned one predicate, which has both a true and a false form. The define instruction sets the predicate to true/false, and clears the false/true predicate if the instruction is true/false.

The data dependence graph is generated by analyzing the anti, true, and output dependencies between all instructions just like previous Modulo Scheduling algorithms discussed. Special rules are used to determine the dependencies when predicates are involved. There are flow dependences between the instruction that defines the predicate and all instructions that belong to the basic block assigned that predicate. Output dependences exist between the predicate define instruction and the predicate merge instruction. The predicate merge instruction is placed in the block that post dominates all of its predecessors. Finally, there is an anti-dependence between all instructions assigned a predicate, and the predicate merge.

EMS uses an iterative Modulo Scheduling algorithm similar to IMS [33], but with some minor enhancements. It uses the same techniques to select instructions based on priority, and schedules instructions at their earliest allowable slot. However, instead of the standard modulo reservation table, the table is extended to allow three entries per slot: empty, no-conflict, and full. No conflict indicates that there is an instruction scheduled for this slot, but other instructions from different paths can be scheduled in the same slot provided there is not control path between them.

Modulo Variable Expansion is a technique first developed by Lam [25] to calculate variable lifetimes of the resulting kernel. The longest variable lifetime is used to determine how many times to unroll the loop in order to not overwrite any values before they are used. After unrolling, the variables are renamed.

Finally, EMS must regenerate the control flow structure in the newly pipelined loop by replacing the predicate define instructions with conditional branches.

While EMS sounds like the ideal Global Modulo Scheduling algorithm, it was only applied to loops without loop carried dependencies resulting from memory instructions. This can seriously limit the number of valid loops to be software pipelined.

3.2.4 Conclusion

Implementing all of the Modulo Scheduling approaches described in this chapter is outside the scope of this thesis. However, a study by Codina et.al [10] performed an in-depth comparison of each approach. They found Swing Modulo Scheduling to generate the most optimal schedules for low and medium complexity architectures. Additionally, for all architectures, SMS was found to be the best at maintaining low register pressure and took the least amount of compile time to find an optimal schedule. For complex architectures, the iterative techniques (IMS and IRIS) both found a more optimal schedule, but had a much higher register pressure. Because SMS is successful at finding an optimal schedule while keeping register pressure low, and does both in an efficient manner, it appears to be the better approach.

Both Global Modulo Scheduling techniques, Hierarchal Reduction and Enhanced Modulo Scheduling handle multiple basic blocks. However, both take the resource and dependence constraints of all paths within the loop into consideration when constructing the schedule. For loops where one path is more frequently executed, this can lead to a less than optimal schedule. The extensions to Swing Modulo Scheduling (Chapter 5) introduce a Modulo Scheduling technique that only considers the most frequently executed (hot) path of the loop.

Chapter 4

Implementing Swing Modulo Scheduling

Swing Modulo Scheduling [27] (SMS) is a Modulo Scheduling approach that considers the criticality of instructions and uses heuristics with a low computational cost. The goal of SMS is to achieve the theoretical Minimum Initiation Interval (MII), as discussed previously in Section 3.1, reduce the number of live values in the schedule (MaxLive) and reduce the Stage Count (SC). The Stage Count is simply the number of iterations live in the resulting kernel. This chapter presents our implementation of SMS in the LLVM Compiler Infrastructure [26].

Unlike other Modulo Scheduling algorithms [21, 33, 12], SMS does no backtracking (unscheduling of instructions), so instructions are only scheduled once. If an instruction can not be scheduled, the whole schedule is cleared, II is increased, and scheduling begins again. SMS is also unique in how it orders instructions for scheduling. It orders instructions by taking the RecMII of the recurrence the instruction belongs to and the criticality of the path (in the Data Dependence Graph) into consideration. This ordering technique aims to reduce the stage count and achieve a schedule of length MII. During scheduling, MaxLive is reduced by only scheduling instructions close to their predecessors and successors.

Swing Modulo Scheduling is composed of three main steps:

1. Computation and Analysis of the Data Dependence Graph (DDG).
2. Node Ordering.
3. Scheduling.

```
for(i = 0; i < 500; ++i)
  A[i] = A[i-1] * 3.4f;
```

(a) C Code

```
%i.0.0 = phi uint [ 0, %entry ], [ %indvar.next, %no_exit ]
%tmp.5 = cast uint %i.0.0 to long
"addrOfGlobal:A1" = getelementptr [500 x float]* %A, long 0
%tmp.6 = getelementptr [500 x float]* "addrOfGlobal:A1", long 0, long %tmp.5
%copyConst = cast uint 4294967295 to uint
%tmp.8 = add uint %i.0.0, %copyConst
%tmp.9 = cast uint %tmp.8 to long
"addrOfGlobal:A2" = getelementptr [500 x float]* %A, long 0
%tmp.10 = getelementptr [500 x float]* "addrOfGlobal:A2", long 0, long %tmp.9
%tmp.11 = load float* %tmp.10
%tmp.12 = mul float %tmp.11, 0x400B333340000000
store float %tmp.12, float* %tmp.6
%indvar.next = add uint %i.0.0, 1
%exitcond = seteq uint %indvar.next, 500
br bool %exitcond, label %loopexit, label %no_exit
```

(b) LLVM Code

Figure 4.1: Simple Loop Example

The first two are computed once, while scheduling is repeated until a schedule has been achieved or the algorithm has reached some maximum II and gives up. Because scheduling is the only part repeated, the computation time is kept reasonable. Like other Modulo Scheduling algorithms, SMS works on all innermost loops without calls and control flow. Loop reconstruction is performed after scheduling is successful, but is not technically part of the SMS algorithm.

Swing Modulo Scheduling was originally chosen over other Modulo Scheduling algorithms mentioned in Chapter 3 because of its ability to keep computation time to a minimum, while still achieving MII and keeping register pressure low. We discuss our experiences with how SMS actually performed in Chapter 6.

4.1 LLVM Compiler Infrastructure

Swing Modulo Scheduling was implemented in the Low Level Virtual Machine (LLVM) Compiler Infrastructure [26]. LLVM is a low-level, RISC-like instruction set and object code representation. It provides type information and data flow information (using SSA [11]), while still being extremely light-weight. The LLVM Compiler Infrastructure provides optimizations that can be applied at compile time, link-time, run-time, and offline profile driven transformations.

SMS was implemented as a static optimization in the SPARC V9 back-end. SMS is performed before register allocation, but after local scheduling. However, nothing in our implementation prevents it from being performed at run-time or offline. The SPARC back-end uses a low-level

```
(n1) sethi %lm(-1), %reg(val 0x100d0eb20)
(n2) sethi %hh(%disp(addr-of-val A)), %reg(val 0x100d31a90)
(n3) add %reg(val 0x100bb0200 i.0.0:PhiCp), %g0, %reg(val 0x100baf6a0 i.0.0)
(n4) sethi %hh(<cp#1>), %reg(val 0x100d18060)
(n5) or %reg(val 0x100d31a90), %hm(%disp(addr-of-val A)), %reg(val 0x100d31b30)
(n6) sethi %lm(%disp(addr-of-val A)), %reg(val 0x100d31c70)
(n7) or %reg(val 0x100d18060), %hm(<cp#1>), %reg(val 0x100d15740)
(n8) or %reg(val 0x100d0eb20), %lo(-1), %reg(val 0x100d0ea80)
(n9) add %reg(val 0x100baf6a0 i.0.0), %reg(val 0x100d0ea80), %reg(val 0x100d0e9e0 maskHi)
(n10) sethi %lm(<cp#1>), %reg(val 0x100d18100)
(n11) sllx %reg(val 0x100d31b30), 32, %reg(val 0x100d31bd0)
(n12) sllx %reg(val 0x100d15740), 32, %reg(val 0x100d157e0)
(n13) or %reg(val 0x100d18100), %reg(val 0x100d157e0), %reg(val 0x100d15880)
(n14) sethi %hh(%disp(addr-of-val A)), %reg(val 0x100d12f60)
(n15) or %reg(val 0x100d31c70), %reg(val 0x100d31bd0), %reg(val 0x100d31d10)
(n16) srl %reg(val 0x100d0e9e0 maskHi), 0, %reg(val 0x100bb9a50 tmp.8)
(n17) or %reg(val 0x100d31d10), %lo(%disp(addr-of-val A)), %reg(val 0x100d319f0)
(n18) sll %reg(val 0x100bb9a50 tmp.8), 2, %reg(val 0x100d31950)
(n19) or %reg(val 0x100d15880), %lo(<cp#1>), %reg(val 0x100d12ec0)
(n20) or %reg(val 0x100d12f60), %hm(%disp(addr-of-val A)), %reg(val 0x100d13000)
(n21) add %reg(val 0x100d319f0), 0, %reg(val 0x100bb73a0 addrOfGlobal:A2)
(n22) ld %reg(val 0x100d12ec0), 0, %reg(val 0x100d17fc0)
(n23) sllx %reg(val 0x100d13000), 32, %reg(val 0x100d10640)
(n24) sethi %lm(%disp(addr-of-val A)), %reg(val 0x100d106e0)
(n25) ld %reg(val 0x100bb73a0 addrOfGlobal:A2), %reg(val 0x100d31950), %reg(val 0x100bb9bf0 tmp.11)
(n26) sll %reg(val 0x100baf6a0 i.0.0), 2, %reg(val 0x100d318b0)
(n27) or %reg(val 0x100d106e0), %reg(val 0x100d10640), %reg(val 0x100d10780)
(n28) add %reg(val 0x100baf6a0 i.0.0), 1, %reg(val 0x100cfb200 maskHi)
(n29) or %reg(val 0x100d10780), %lo(%disp(addr-of-val A)), %reg(val 0x100d33d30)
(n30) srl %reg(val 0x100cfb200 maskHi), 0, %reg(val 0x100bb9e40 indvar.next)
(n31) add %reg(val 0x100bb9e40 indvar.next), %g0, %reg(val 0x100bb0200 i.0.0:PhiCp)
(n32) add %reg(val 0x100d33d30), 0, %reg(val 0x100bb7460 addrOfGlobal:A1)
(n33) subcc %reg(val 0x100bb9e40 indvar.next), 500, %g0, %ccreg(val 0x100d343f0)
(n34) fmuls %reg(val 0x100bb9bf0 tmp.11), %reg(val 0x100d17fc0), %reg(val 0x100bb9c70 tmp.12)
(n35) st %reg(val 0x100bb9c70 tmp.12), %reg(val 0x100bb7460 addrOfGlobal:A1), %reg(val 0x100d318b0)
(n36) be %ccreg(val 0x100d343f0), %disp(label loopexit)
(n37) ba %disp(label no_exit)
```

Figure 4.2: LLVM Machine Code for a Simple Loop

representation that closely models the SPARC V9 assembly [1]. Each instruction has an opcode and a list of operands. For SMS in the SPARC back-end, we only deal with operands of the following types:

- **Machine Register**: This is a representation of a physical register for the SPARC architecture.
- **Virtual Register**: These are LLVM values, which is the base representation for all values computed by the program that may be used as operands to other values.
- **Condition Code Register**: The register that stores the results of a compare operation.
- **PC Relative Displacement**: A displacement that is added to the program counter (PC). This is used for specifying code addresses in control transfer instructions (i.e. branches).

- **Global Address**: The address for a global variable.

Throughout this Chapter, we illustrate the phases of Swing Modulo Scheduling on a simple example. Figure 4.1 shows a C for-loop that sets elements of a floating point array to the previous element multiplied by some constant. It also shows the LLVM representation for the loop. This loop is perfect for SMS since floating point computations typically have a high latency and it is ideal to overlap their execution with other instructions. Figure 4.2 shows the LLVM code translated to a machine code representation that closely models the SPARC V9 instruction set [1]. SMS is performed on this low-level representation. Lastly, Figure 4.3 shows the LLVM instructions for our simple loop example and their corresponding machine instructions.

4.1.1 Architecture Resource Description

The LLVM Compiler Infrastructure provides a *SchedInfo API* to access information about the architecture resources that are crucial for Scheduling of any kind, including Swing Modulo Scheduling. The SchedInfo API provides information such as the following:

- Instruction Resource Usage: The resources an instruction uses during each stage of the pipeline.
- Resources Available: The resources and number of each resource.
- Issue Slots: Total number of issue slots.
- Total Latency: The associated latency for each instruction (or class of instructions) which is the time (in cycles) for how long it takes from the time the instruction starts until its dependents can use its results.

For our implementation we have written a SchedInfo description for the SPARC IIIi architecture which is described in Section 6.1.

```
%i.0.0 = phi uint [ 0, %entry ], [ %indvar.next, %no_exit ]
  (n31) add %reg(val 0x100bb9e40 indvar.next), %g0, %reg(val 0x100bb0200 i.0.0:PhiCp)
  (n3) add %reg(val 0x100bb0200 i.0.0:PhiCp), %g0, %reg(val 0x100baf6a0 i.0.0)
%tmp.5 = cast uint %i.0.0 to long
"addrOfGlobal:A1" = getelementptr [500 x float]* %A, long 0
  (n14) sethi %hh(%disp(addr-of-val A)), %reg(val 0x100d12f60)
  (n20) or %reg(val 0x100d12f60), %hm(%disp(addr-of-val A)), %reg(val 0x100d13000)
  (n23) sllx %reg(val 0x100d13000), 32, %reg(val 0x100d10640)
  (n24) sethi %lm(%disp(addr-of-val A)), %reg(val 0x100d106e0)
  (n27) or %reg(val 0x100d106e0), %reg(val 0x100d10640), %reg(val 0x100d10780)
  (n29) or %reg(val 0x100d10780), %lo(%disp(addr-of-val A)), %reg(val 0x100d33d30)
  (n32) add %reg(val 0x100d33d30), 0, %reg(val 0x100bb7460 addrOfGlobal:A1)
%tmp.6 = getelementptr [500 x float]* "addrOfGlobal:A1", long 0, long %tmp.5
  (n26) sll %reg(val 0x100baf6a0 i.0.0), 2, %reg(val 0x100d318b0)
%copyConst = cast uint 4294967295 to uint
  (n1) sethi %lm(-1), %reg(val 0x100d0eb20)
  (n8) or %reg(val 0x100d0eb20), %lo(-1), %reg(val 0x100d0ea80)
%tmp.8 = add uint %i.0.0, %copyConst
  (n9) add %reg(val 0x100baf6a0 i.0.0), %reg(val 0x100d0ea80), %reg(val 0x100d0e9e0 maskHi)
  (n16) srl %reg(val 0x100d0e9e0 maskHi), 0, %reg(val 0x100bb9a50 tmp.8)
%tmp.9 = cast uint %tmp.8 to long
"addrOfGlobal:A2" = getelementptr [500 x float]* %A, long 0
  (n2) sethi %hh(%disp(addr-of-val A)), %reg(val 0x100d31a90)
  (n5) or %reg(val 0x100d31a90), %hm(%disp(addr-of-val A)), %reg(val 0x100d31b30)
  (n11) sllx %reg(val 0x100d31b30), 32, %reg(val 0x100d31bd0)
  (n6) sethi %lm(%disp(addr-of-val A)), %reg(val 0x100d31c70)
  (n15) or %reg(val 0x100d31c70), %reg(val 0x100d31bd0), %reg(val 0x100d31d10)
  (n17) or %reg(val 0x100d31d10), %lo(%disp(addr-of-val A)), %reg(val 0x100d319f0)
  (n21) add %reg(val 0x100d319f0), 0, %reg(val 0x100bb73a0 addrOfGlobal:A2)
%tmp.10 = getelementptr [500 x float]* "addrOfGlobal:A2", long 0, long %tmp.9
  (n18) sll %reg(val 0x100bb9a50 tmp.8), 2, %reg(val 0x100d31950)
%tmp.11 = load float* %tmp.10
  (n25) ld %reg(val 0x100bb73a0 addrOfGlobal:A2), %reg(val 0x100d31950), %reg(val 0x100bb9bf0 tmp.11)
%tmp.12 = mul float %tmp.11, 0x400B333340000000
  (n4) sethi %hh(¡cp#1¿), %reg(val 0x100d18060)
  (n7) or %reg(val 0x100d18060), %hm(¡cp#1¿), %reg(val 0x100d15740)
  (n10) sethi %lm(¡cp#1¿), %reg(val 0x100d18100)
  (n12) sllx %reg(val 0x100d15740), 32, %reg(val 0x100d157e0)
  (n13) or %reg(val 0x100d18100), %reg(val 0x100d157e0), %reg(val 0x100d15880)
  (n19) or %reg(val 0x100d15880), %lo(¡cp#1¿), %reg(val 0x100d12ec0)
  (n22) ld %reg(val 0x100d12ec0), 0, %reg(val 0x100d17fc0)
  (n34) fmuls %reg(val 0x100bb9bf0 tmp.11), %reg(val 0x100d17fc0), %reg(val 0x100bb9c70 tmp.12)
store float %tmp.12, float* %tmp.6
  (n35) st %reg(val 0x100bb9c70 tmp.12), %reg(val 0x100bb7460 addrOfGlobal:A1), %reg(val 0x100d318b0)
%indvar.next = add uint %i.0.0, 1
  (n28) add %reg(val 0x100baf6a0 i.0.0), 1, %reg(val 0x100cfb200 maskHi)
  (n30) srl %reg(val 0x100cfb200 maskHi), 0, %reg(val 0x100bb9e40 indvar.next)
%exitcond = seteq uint %indvar.next, 500
  (n33) subcc %reg(val 0x100bb9e40 indvar.next), 500, %g0, %ccreg(val 0x100d343f0)
br bool %exitcond, label %loopexit, label %no_exit
  (n36) be %ccreg(val 0x100d343f0), %disp(label loopexit)
  (n37) ba %disp(label no_exit)
```

Figure 4.3: LLVM Instructions and Corresponding Machine Instructions

4.2 Data Dependence Graph Construction

Swing Modulo Scheduling begins by constructing the Data Dependence Graph (DDG) for a single basic block without control flow or calls. The DDG consists of nodes that represent instructions[1] and their corresponding properties, and edges that represent the true, anti, and output dependencies in the loop.

Our implementation only constructs a DDG for the instructions deemed to be the loop body. This excludes instructions related to the iteration count and branch. The SMS algorithm (or any Modulo Scheduling algorithm) has no mechanism to ensure that these excluded instructions are from the current iteration, stage 0, in the resulting kernel. Keeping these instructions in the current iteration is critical to proper execution of the loop. These excluded instructions are reinserted during the loop reconstruction phase described in Section 4.7.

The DDG construction examines each instruction of the loop body and determines its relationship between all other instructions. Each edge in the DDG represents a data dependence between the two instructions. There are three types of dependencies in the DDG:

- True Dependence: If the first instruction writes to a value, and a second instruction reads the same value, there is a true dependence from the first instruction to the second.
- Anti Dependence: If the first instruction reads a value, and a second instruction writes a value, then there is an anti dependence from the first instruction to the second.
- Output Dependence: If two instructions both write to the same value, then there is an output dependence between them.

Each dependence has a distance associated with it, called the iteration difference. If the distance is zero, this means the dependence is a loop-independent dependence, in other words a dependence within one iteration. If the distance is greater than zero, there is a dependence across iterations, a loop-carried dependence. The value of the distance for loop-carried dependences is one, unless further analysis can prove the actual number of iterations between the instructions.

Dependences are generated for all machine registers, memory instructions, and LLVM values (both described in Section 4.1). Loop-carried dependences exist for memory and any LLVM values

[1]All references to instruction mean the low-level Machine Instruction representation in the SPARC back-end.

or machine registers that are live across iterations. LLVM values that are live across iterations were originally represented as ϕ[2] instructions at the LLVM level. Because SMS is operating on a lower-level representation, it must determine which machine instructions were generated by the LLVM ϕ instruction. Two copy instructions located at the start and end of the basic block are generated for each ϕ instruction. There is a true-dependence from the last instruction related to the ϕ, and the first instruction with a distance of one. Memory instructions have loop-carried dependences of distance one, and the dependence type is determined by the types of the two instructions involved (load or store). Simple dependence analysis is used to eliminate many of these dependences, by examining the memory pointers and determining if the same memory is being accessed. A more sophisticated dependence analysis (Section 4.2.1) is used to determine the actual distance for the dependencies.

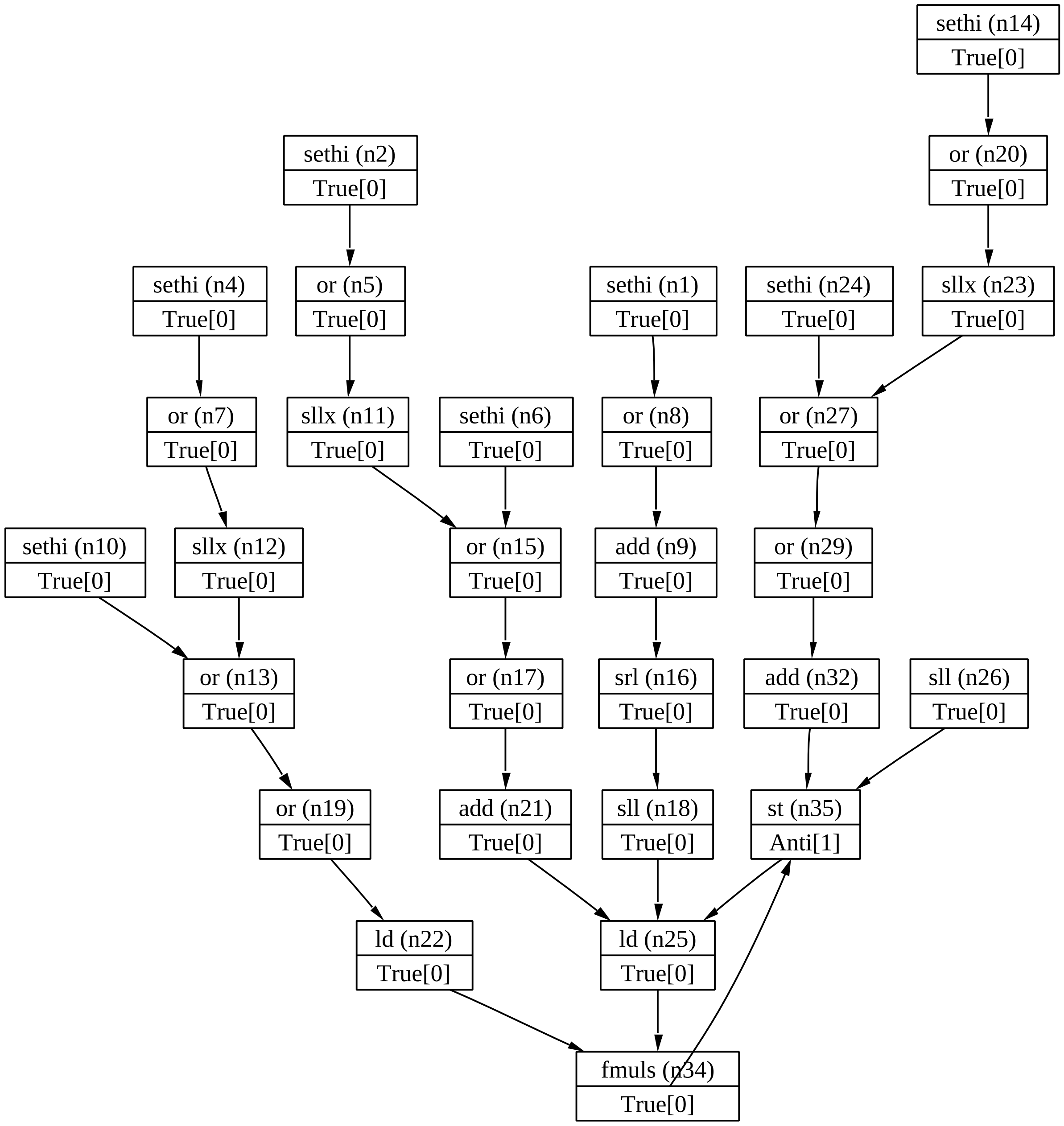
Figure 4.4 is the dependence graph for the example loop. Each node in the graph corresponds to an instruction in the loop body. The edges between the nodes are the dependences between the instructions. Each dependence is marked with its type, and the distance (if greater than zero). The dependence graph for our example mostly has true dependences because our code is in SSA[11] and no value is defined more than once. Additionally, there are loop-carried dependences (with a distance of one) between all loads and stores that were proven to access the same memory.

4.2.1 Dependence Analysis

Simple dependence analysis eliminates many dependencies between loads and stores but can not determine the actual distance for loop-carried dependencies. Assuming a distance of one is conservative and results in many missed opportunities for parallelism. Our implementation uses a more sophisticated analysis to calculate the actual distances for loop-carried dependences.

We implemented a Dependence Analyzer that uses two analyses, *Alias Analysis* (AA) and *Scalar Evolution* (SE), which are both provided by the LLVM compiler infrastructure. Alias Analysis provides information about the points-to relationship for references in the program, which it may decide is a may/must/no relation. The Scalar Evolution analysis uses Chains of Recurrences (CRs) as a way of representing the fixed relation between two memory references [7, 40, 39, 43, 6].

[2]A ϕ instruction [11] is a merge point for a variable, where it has as many operands as there are versions of the variable.



Dependence Graph

Figure 4.4: Dependence Graph After Dependence Analysis

```
getDependenceInfo(Instruction inst1, Instruction inst2, bool srcBeforeDest)
Deps = EmptyList        //Empty list of dependences
if (inst1 == inst2)
    return Deps              //No dependences to the same node (self loops)
if( !isLoad(inst1) || !isStore(inst1) || !isLoad(inst2) || !isStore(inst2))
    return Deps//Only deal with memory instructions
if (isLoopInvariant(inst1) && isLoopInvariant(inst2))
    aliasResult = AA.alias(inst1, inst2)
    if (aliasResult != NoAlias)
        Deps = createDep(inst1, inst2, srcBeforeDest, 0)
        return Deps
    else
        return Deps
else
    inst1B = base pointer of memory reference for inst1
    inst2B = base pointer of memory reference for inst2
    aliasResult = AA.alias(inst1B, inst2B)
    if (aliasResult == MustAlias)
        advDepAnalysis(inst1, inst2, srcBeforeDest)
    else if(aliasResult == MayAlias)
        Deps = createDep(inst1, inst2, srcBeforeDest, 0)
return Deps

advDepAnalysis(Instruction inst1, Instruction inst2, bool srcBeforeDest)

if (!isSingleDimensional(inst1) || !isSingleDimensional(inst2))
    return createDep(inst1, inst2, srcBeforeDest, 0)
SCEV1 = SE.getSCEV(inst1)
SCEV2 = SE.getSCEV(inst2)
if ( isAffine(SCEV1) && isAffine(SCEV2))
                              //Affine means A + B*x form
    if (SCEV1.B != SCEV2.B)
        return createDep(inst1, inst2, srcBeforeDest, 0)
    if (SCEV1.A == SCEV2.A)
        return createDep(inst1, inst2, srcBeforeDest, 0)
    dist = SCEV1.A - SCEV2.A
    if (dist > 0)
        return createDep(inst1, inst2, srcBeforeDest, dist)

createDep(Instruction inst1, Instruction inst2, bool srcBeforeDest, int dist)
if (!srcBeforeDest && dist==0)
    dist = 1
if (isLoad(inst1) && isStore(inst2)))
    if(srcBeforeDest)
        return Anti-Dependence with a distance of dist
    else
        return True-Dependence with a distance of dist
else if (isStore(inst1) && isLoad(inst2))
    if(srcBeforeDest)
        return True-Dependence with a distance of dist
    else
        return Anti-Dependence with a distance of dist
else if (isStore(inst1) && isStore(inst2))
    return Output-Dependence with a distance of dist
```

Figure 4.5: Pseudo Code for Dependence Analyzer

Figure 4.5 shows the algorithm used by our dependence analyzer. The **getDependenceInfo** function takes two instructions and a boolean that indicates if the first instruction is executed before the second and returns the list of dependences between them. If the two instructions are the same, there is no dependence because an instruction only occurs once in the final loop. The two instructions are then checked to ensure that they are both memory operations (load or a store).

Once the dependence analyzer is confident that two memory operations are being analyzed, it examines each memory reference and determines if the memory addresses accessed are loop invariant. If the addresses are loop invariant, then Alias Analysis alone can be used to determine if there is a dependence between them. If the two addresses are not loop invariant, then Alias Analysis is used to compare the base pointers for each memory reference. If AA can prove there is a No-Alias relation, then no dependence is created. If AA can only prove that the two base pointers May-Alias, then a dependence is created. Lastly, if AA can prove that the two addresses Must-Alias, then further dependence analysis is need.

The **createDep** procedure creates the dependence between the two instructions. The distance of the dependence is almost always determined by the callee. However, if the first instruction occurs after the second (in execution order), and the distance has defaulted to zero (this means the true distance could not be found), the distance is set to one. This means that a conservative assumption is taken that the instructions have a dependence across one iteration.

If further analysis is needed, the **advDepAnalysis** function is called. It begins by determining if the memory access is to a single dimensional array. Our dependence analyzer only handles single dimensional arrays (as they are most common), but there is nothing preventing it from being extended to handle multi dimensional arrays. Using Scalar Evolution analysis, the memory reference is transformed into a uniform representation: $A + B * x$, where A is the offset, and B*x is some constant times the base pointer. Our dependence analyzer has already used Alias Analysis to determine the relationship between base pointers (Must-Alias). The B values are compared, and if they are not equal a dependence is created between the instructions. Lastly, the offsets (A value) are compared. If they are equal, the same element is being accessed and a dependence is created. If they are not equal, the difference between the two values is the distance of the dependence, and a dependence is created.

4.3 Calculating the Minimum Initiation Interval

The Minimum Initiation Interval (MII) is the minimum number of cycles between initiations of two iterations of the loop. The value is constrained by resources or dependences in the Data Dependence Graph. If there are not enough resources available, instructions will be delayed from issuing until the needed resources are free. If there are dependence constraints, instructions can not complete until all its operand values are available. SMS uses the MII as a starting value for II when generating a schedule, which is the lowest value that can be achieve given resource and dependence constraints.

4.3.1 Resource II

The Resource Minimum Initiation Interval (ResMII) is calculated by summing the resource usage requirements for one iteration of the loop. A reservation table [33] represents the resource usage patterns for each cycle during one iteration of a loop. By performing a bin-packing of the reservation table for all instructions, the exact ResMII is found. However, this process can be time consuming (bin-packing is an NP complete problem), so an approximation for ResMII is computed.

To calculate an approximation for ResMII, each instruction is examined for its resource usages. The most heavily used resource sets the ResMII. Figure 4.2 shows all the instructions for our example loop. Examining these instructions will show that the most heavily used resource is the integer unit. A total of 33 instructions use this resource, and there are 2 integer units, which sets the ResMII for this loop at 17.

4.3.2 Recurrence II

Recurrences may be found in the the DDG if instructions have dependences across iterations of the loop. Memory operations (load/store) are most likely the cause of a recurrence in the DDG. Recurrences are also known as circuits or cycles.

In order to compute the Recurrence Minimum Initiation Interval (RecMII), all recurrences in the DDG must be found. In our implementation of SMS, the algorithm proposed by Donald Johnson [23] is used to find all elementary circuits in the DDG. If no vertex except the first and last appear twice, then a circuit is termed elementary. Johnson's algorithm is extremely efficient

```
**findAllCircuits**(DDG $G$)
empty *stack*
$s = 1$
while ($s < n$)
   $Ak$ = { Adjacency structure of a strong component K with least vertex
      in subgraph of G induced by $\{s, s+1, ..., n\}$ }
   if ($Ak \neq \emptyset$)
      $s$ = {Least vertex in $Ak$}
      $\forall\ i \in Ak$
         $blocked(i)$ = false
         $B(i) = \emptyset$
      **circuit**($s$)
      $s = s + 1$
   else
      $s = n$

**circuit**(int $v$)
$f$ = false
stack $v$
$blocked(v)$ = true
$\forall w \in Ak(v)$
   if($w = s$)
      output circuit composed of stack followed by s
      $f$ = true
   else if ($\neg blocked(w)$)
      if (circuit($w$))
         $f$ = true
if ($f$)
   **unblock**($v$)
else
   $\forall w \in Ak$
      if ($v \ni B(w)$)
         put $v$ on $B(w)$
unstack v
return $f$

**unblock**(int $u$)
$blocked(u)$ = false
$\forall w \in B(u)$
   delete $w$ from $B(u)$
   if ($blocked(w)$)
      **unblock**($w$)
```

Figure 4.6: Pseudo Code for Circuit Finding Algorithm

(compared to all other existing circuit finding algorithms) and finds all circuits in a graph in $O((n+e)(c+1))$, where c is the total number of circuits, n is the total number of nodes, and e is the total number of edges in the DDG.

Figure 4.6 shows Johnson's circuit finding algorithm. It begins, by ordering all the nodes in the graph. It finds the Strongly Connected Component[3] (SCC) with the least vertex, and finds all recurrences within this SCC. Recurrences are built by building elementary paths from the least vertex. The **circuit()** procedure is responsible for appending a node to the path, determining if a recurrence is found, and unblocking the node once it exits. Nodes are *blocked* whenever they are added to the path in order to guarantee that a node can never be used twice on the same path. The process of unblocking a node is delayed as long as possible, usually until a recurrence is found. It repeats the process for each SCC in the graph, in the order set by how the nodes are ordered.

Node	Latency	Node	Latency
sethi (n1)	1	or (n17)	1
sethi (n2)	1	sll (n18)	1
sethi (n4)	1	or (n19)	1
or (n5)	1	or (n20)	1
sethi (n6)	1	add (n21)	1
or (n7)	1	ld (n22)	3
or (n8)	1	sllx (n23)	1
add (n9)	1	sethi (n24)	1
sethi (n10)	1	ld (n25)	3
sllx (n11)	1	sll (n26)	1
sllx (n12)	1	or (n27)	3
or (n13)	1	or (n29)	1
sethi (n14)	1	add (n32)	1
or (n15)	1	fmuls (n34)	4
srl (n16)	1	st (n35)	0

Table 4.1: Node Latencies for Simple Loop Example

Each recurrence in the graph imposes a constraint on the interval between each instruction in this recurrence and the same instruction, Distance(c) iterations later. This constraint must be at least the total latency of the recurrence, Latency(c). However, the distance is really $II * Distance(c)$, which means the constraint is defined as follows [33]: $RecMII = Latency(c) / Distance(c)$. Table 4.1 shows the latencies for all the nodes in the dependence graph.

Using the Data Dependence Graph in Figure 4.4, the circuit finding algorithm finds one recurrence consisting of the following nodes: ld (n25), fmuls (n34), and st (n35). The total latency of

[3]A strongly connected component is a component in which every vertex is reachable from every other vertex.

this circuit is 7, and the total distance is 1. Therefore, the RecMII for the loop is 7. Because this is smaller than our ResMII, the ResMII is used as the starting II value.

4.4 Node Properties

After the Data Dependence Graph has been constructed, a few node properties need to be calculated in order to properly order and schedule the instructions. Because a graph may contain cycles, one back edge (doesn't matter which one) is ignored for each recurrence. For the following node property calculations, λ is the latency of a instruction, δ is the distance of the dependence edge between two nodes, $Pred()$ is the set of predecessors for a node, $Succ()$ is the set of successors for a node, and MII is minimum initiation interval described in Section 4.3.

Node	ASAP	ALAP	MOB	Depth	Height	Latency
sethi (n1)	0	1	1	0	12	1
sethi (n2)	0	0	0	0	13	1
sethi (n4)	0	1	1	0	12	1
or (n5)	1	1	0	1	12	1
sethi (n6)	0	2	2	0	11	1
or (n7)	1	2	1	1	11	1
or (n8)	1	2	1	1	11	1
add (n9)	2	3	1	2	10	1
sethi (n10)	0	3	3	0	10	1
sllx (n11)	2	2	0	2	11	1
sllx (n12)	2	3	1	2	10	1
or (n13)	3	4	1	3	9	1
sethi (n14)	0	7	7	0	6	1
or (n15)	3	3	0	3	10	1
srl (n16)	3	4	1	3	9	1
or (n17)	4	4	0	4	9	1
sll (n18)	4	5	1	4	8	1
or (n19)	4	5	1	4	8	1
or (n20)	1	8	7	1	5	1
add (n21)	5	5	0	5	8	1
ld (n22)	5	6	1	5	7	3
sllx (n23)	2	9	7	2	4	1
sethi (n24)	0	9	9	0	4	1
ld (n25)	6	6	0	6	7	3
sll (n26)	0	12	12	0	1	1
or (n27)	3	10	7	3	3	1
or (n29)	4	11	7	4	2	1
add (n32)	5	12	7	5	1	1
fmuls (n34)	9	9	0	9	4	4
st (n35)	13	13	0	13	0	0

Table 4.2: Node Attributes for Simple Loop Example

- $ASAP_u$: The As Soon As Possible attribute indicates the earliest time that the instruction may be scheduled. It is computed as follows:

$If\ Pred(u) = \{\ \}$

$\quad ASAP_u = 0$

$else$

$\quad ASAP_u = max_{\forall v \epsilon Pred(u)}\ (ASAP_v + \lambda_v + \delta_{v,u} * MII)$

- $ALAP_u$: The As Late As Possible attribute determines the latest cycle an instruction may be scheduled. It is computed using the following:

$If\ Succ(u) = \{\ \}$

$\quad ALAP_u = max_{\forall v \epsilon V}\ ASAP_v$

$else$

$\quad ALAP_u = min_{\forall v \epsilon Succ(u)}\ (ALAP_v - \lambda_u + \delta_{u,v} * MII)$

- MOB_u: The Mobility of an instruction is the number of time slots that an instruction may be scheduled in. The lower the value, the more critical the node and a MOB of zero indicates the most critical path. This attribute is calculated as follows:

$MOB_u = ALAP_u - ASAP_u$

- D_u: The Depth of a node is the number of nodes or maximum distance between this node and a node with no predecessors. It is computed as follows:

$If\ Pred(u) = \{\ \}$

$\quad D_u = 0$

$else$

$\quad D_u = max_{\forall v \epsilon Pred(u)}\ (D_v + \lambda_v)$

- H_u: The Height of a node is the maximum distance between this node and a node without successors. It is calculated as follows:

$If\ Succ(u) = \{\ \}$

$H_u = 0$

$else$

$H_u = max_{\forall v \epsilon Succ(u)}\ (H_v + \lambda_u)$

Table 4.2 shows the calculated properties for all the nodes in the data dependence graph. Looking at this table, we can see that there are a few long latency instructions (fmuls (n35), ld (n25), ld (n22)) which should ideally be overlapped with other instructions. The nodes with a low Mobility and high Height are those instructions that are considered on the critical path. The ASAP and ALAP values give some indication as to how early or how late instructions can be scheduled based upon latencies of predecessors and successors.

4.5 Node Ordering

The node ordering step is a sophisticated algorithm that uses the data dependence graph and the node attributes to create a scheduling order. The ordering algorithm is used to give priority to instructions that are on the most critical paths, while keeping register pressure low. It accomplishes the first by using heuristics to schedule instructions with the highest mobility last. The second is achieved by ordering instructions such that no instruction is scheduled after both its predecessors and successors. By keeping an instruction close to its predecessors and successors, live value ranges are decreased. The only exception is for recurrences, where one instruction is scheduled after its predecessors and successors (which can not be avoided).

```
1  $P$ = Empty ordered list of sets of nodes
2  while ($Recc_{List} \neq \emptyset$)
3     $Recc$ = {Recurrence with highest RecMII}
4     if ($P = \emptyset$)
5        $P = P \mid Recc$
6     else
7        $\forall$ $v$, where $v$ are nodes connecting $Recc$ to any set in $P$
8           $Recc = Recc \mid v$
9        $P = P \mid Recc$
10 $NodesLeft$ = {All nodes not in P}
11 $\forall$ connected components, $C$, $\in NodesLeft$
12    $P = P \mid C$
```

Figure 4.7: Pseudo Code for Partial Node Ordering Algorithm

The ordering algorithm begins by calculating a partial order, a list of sets of nodes. Figure 4.7 describe the partial node ordering algorithm, where | denotes the list append operation. For a graph with recurrences, the first set in the partial order list is the recurrence with the highest RecMII. The next highest RecMII recurrence set is appended to the partial list including any nodes that connect it to any recurrence already in the partial order, and removing any nodes already in the partial order. This is repeated until all recurrences have been added. If there are nodes not in the partial order or the graph has no recurrences, nodes are grouped into connected components, a set of nodes that are connected, and the set is appended to the partial order.

Figure 4.8 shows the partial order for our simple loop example. The partial order is an ordered list of sets. The first set consists of nodes from the lone recurrence in the dependence graph. The other sets represent the connected components in the graph (minus the recurrence). There is no order in which the connected components are added.

Once the partial order has been computed, the final node ordering algorithm produces a list of nodes that is sent to the scheduler. The algorithm shown in Figure 4.9 traverses each subgraph of the set of nodes in the partial order. In the case of a connected dependence graph with no recurrences, it traverses the whole graph.

The algorithm begins with the node at the bottom of the most critical path and visits all the ancestors according to their depth, traveling bottom-up. If the ancestors have equal depth, priority is given to nodes with less mobility. Once all the ancestors are visited, the descendants of the node are visited in order of height, traversing top-down. This upward and downward traversal is repeated until all nodes have been placed in the final order and the entire graph has been traversed.

```
Set #1: ld (n25), fmuls (n34), st (n35)
Set #2: sethi (n2), or (n5), sllx (n11), or (n15), or (n17), add (n21), sethi (n6)
Set #3: sethi (n10), sethi (n4), or (n7), sllx (n12), or (n13), or (n19), ld (n22)
Set #4: sethi (n1), or (n8), add (n9), srl (n16), sll (n18),
Set #5: sethi (n14), or (n20), sllx (n23), or (n27), or (n29), add (n32), sethi (n24)
Set #7: sll (n26)
```

Figure 4.8: Simple Loop Example Partial Order

The final node ordering algorithm shown in Figure 4.9, uses | to denote the list append operation, and Succ_L(O) and Pred_L(O) are defined as follows:

Pred_L(O) = $\{v \mid \exists\ u \in O \text{ where } v \in Pred(u) \text{ and } v \ni O\}$

Succ_L(O) = $\{v \mid \exists\, u \in O \text{ where } v \in Succ(u) \text{ and } v \ni O\}$

```
O = Empty_List
foreach $S$   //Each set in the partial order in decreasing priority
  if ((Pred_L($O$) $\cap$ $S$) $\neq \emptyset$)
    $R$ = Pred_L($O$) $\cap$ $S$
    $order$ = bottom-up
  else if((Succ_L($O$) $\cap$ $S$) $\neq \emptyset$)
    $R$ = Succ_L($O$) $\cap$ $S$
    $order$ = top-down
  else
    $R$ = {Node with the highest ASAP in S, pick any if more then one}
    $order$ = bottom-up
while ($R \neq \emptyset$)
  if ($order$ = top-down)
    while ($R \neq \emptyset$)
      $V$ = {Element of R with highest Height. Use highest MOB to break ties}
      $O = O \mid V$
      $R$ = ($R-$ $V$) $\cup$ (Succ(V) $\cap$ $S$)
    $order$ = bottom-up
    $R$ = Pred_L($O$) $\cap$ $S$
  else
    while ($R \neq \emptyset$)
      $V$ = {Element of R with highest Depth. Use lowest MOB to break ties}
      $O = O \mid V$
      $R$ = ($R-$ $V$) $\cup$ (Pred(V) $\cap$ $S$)
    $order$ = top-down
    $R$ = Succ_L($O$) $\cap$ $S$
```

Figure 4.9: Pseudo Code for Final Node Ordering Algorithm

For the loop example, the Final Node ordering algorithm processes each set in the partial order and determines the final node ordering to be the following:

O = {st (n35), fmuls (n34), ld (n25), sll (n18), srl (n16), add (n9), or (n8), sethi (n1), add (n21), or (n15), sllx (n11), or (n5), sethi (n2), ld (n22), or (19), or (n13), sllx (n12), or (n7), sethi (n4), sethi (n6), sethi (n10), add (n32), or (n29), or (n27), sllx (n23), or (n20), sethi (n14), sethi (n24), sll (n26)}

4.6 Scheduling

The scheduling phase of Swing Modulo Scheduling schedules the nodes in the order determined by the node ordering algorithm. Conceptually a schedule is a table where the rows represent cycles, and columns are issue slots [4]. Scheduling an instruction reserves an issue slot for a specific cycle. The combination of instructions that can be grouped together in the issue slots is dependent upon

[4]Our implementation (for the SPARC IIIi) has 4 issue slots.

the architecture and resources. If all instructions have not been scheduled, the table is called a partial schedule.

When scheduling instructions, SMS attempts to place instructions as close to their predecessors or successors in the partial schedule. By placing instructions close to their neighbors, register pressure is reduced.

```
$\forall n \in O$
  if ((Succ($n$) $\in PS$) && (Pred($n$) $\in PS$))
    $EStart = max_{v \in PSP(u)}(t_v + \lambda_v - \delta_{v,u} * II)$
    $LStart = max_{v \in PSS(u)}(t_v - \lambda_u - \delta_{u,v} * II)$
    Schedule node in free slot starting from $EStart$ until min($LStart$, $EStart + II - 1$
  else if (Pred($n$) $\in PS$)
    $EStart = max_{v \in PSP(u)}(t_v + \lambda_v - \delta_{v,u} * II)$
    Schedule node in free slot starting from $EStart$ until $EStart + II - 1$
  else if (Succ($n$) $\in PS$)
    $LStart = max_{v \in PSS(u)}(t_v - \lambda_u - \delta_{u,v} * II)$
    Schedule node in free slot starting from $LStart$ until $LStart - II + 1$
  else
    $EStart = ASAP_u$
    Schedule node in free slot starting from $EStart$ until $EStart + II - 1$
  if (!scheduled)
    $II = II + 1$
    Clear schedule and restart
```

Figure 4.10: Pseudo Code for Scheduling Algorithm

Figure 4.10 shows the SMS scheduling algorithm, where PS stands for the partial schedule, PSP means the predecessors in the partial schedule, and PSS is the successors in the partial schedule. Each instruction is scheduled from a start-cycle to an end-cycle, which creates a window of time that the instruction can be legally scheduled. The start and end cycles are calculated based upon what is already in the partial schedule. The schedule is scanned forwards (if the start-cycle is earlier than the end-cycle) or backwards (if the start-cycle is later than the end-cycle). Instructions are scheduled according to the following rules:

- For instructions that have no successors or predecessors in the partial schedule, the instruction is scheduled from ESTART until ESTART $+ II - 1$, where ESTART $= ASAP_u$.
- If the instruction only has predecessors in the partial schedule, the instruction is scheduled from ESTART until ESTART $+ II - 1$, where ESTART $= max_{v \in PSP(u)}(t_v + \lambda_v - \delta_{v,u} * II)$.
- If the instruction only has successors in the partial schedule, the instruction is scheduled from LSTART until LSTART $- II + 1$, where LSTART $= min_{v \in PSS(u)}(t_v - \lambda_u - \delta_{u,v} * II)$.

- For instructions that have both successors and predecessors (which only happens once per recurrence), the instruction is scheduled from ESTART until min(LSTART, ESTART $+II-1$). ESTART and LSTART are defined the same as the previous two situations.

If no free slot exists for an instruction, the entire schedule is cleared and II is increased. Scheduling resumes and this pattern repeats until a schedule is found or the maximum II has been reached. In our implementation maximum II is set to the total latency of the original loop.

Cycle	Issue1	Issue2	Issue3	Issue4
0	sethi(n2)	sethi(n6)		
1	sethi(n1)	or(n5)		
2	or(n8)	sllx(n11)		
3	add(n9)	or(n15)		
4	srl(n16)	or(n17)		
5	sll(n18)	add(n21)		
6	ld(n25)			
7				
8				
9	sll(n26)			
10	sethi(n14)	sethi(n24)		
11	sethi(n5)	or(n20)		
12	or(n7)	sllx(n23)		
13	sethi(n10)	sllx(n12)		
14	or(n13)	or(n27)		
15	or(n19)	or(n29)		
16	ld(n22)	add(n32)		
17				
18				
19			fmuls(n34)	
20				
21				
22				
23		st(n35)		

Table 4.3: Schedule for a Single Iteration of the Loop Example

Using this schedule, the kernel is constructed by taking all instructions scheduled at a cycle greater than II, finds what stage they are from, and what cycle in the kernel it should be scheduled. The stage is found by dividing the cycle by II (and rounding down). The kernel cycle is equal to the instruction's scheduled cycle *modulo* II. Additionally, the instructions related to the induction variable and branch (not considered during previous phases) are reinserted at their proper location (preserving dependencies and placing the branch at the end) in the kernel. During the scheduling process kernel conflicts, resource conflicts with instructions from another stage,were checked before an instruction was assigned an issue slot.

Table 4.3 shows the schedule for a single iteration and the kernel for the loop we have been using as an example throughout the chapter. The SPARC IIIi architecture can issue 4 instructions per cycle. The combination of instructions that can be issued depends on what resources they use during each stage of the pipeline. For simplicity, the schedule in Table 4.3 only shows the issue slots, but the scheduling algorithm checks both that there is an available issue slot, and all resources are available.

In the schedule, all instructions before cycle 17 belong to stage 0 (the current iteration of the loop), while all instructions after belong to stage 1. The scheduling algorithm has managed to generate a schedule of length 17, which was our MII. This is an optimal schedule. The instructions have been scheduled such that many of the single cycle instructions can be overlapped with the floating point multiply (n34) which takes 4 cycles. Table 4.4 shows the kernel for the modulo scheduled loop. The number enclosed in brackets indicates which stage the instruction is from. The fmuls (n34) instruction is from stage 1, which means that the instruction is from a previous iteration.

Cycle	Issue1	Issue2	Issue3	Issue4
0	sethi(n2)	sethi(n6)		
1	sethi(n1)	or(n5)		
2	or(n8)	sllx(n11)	fmuls(n34)[1]	
3	add(n9)	or(n15)		
4	srl(n16)	or(n17)		
5	sll(n18)	add(n21)		
6	ld(n25)	st(n35)[1]		
7				
8				
9	sll(n26)			
10	sethi(n14)	sethi(n24)		
11	sethi(n5)	or(n20)		
12	or(n7)	sllx(n23)		
13	sethi(n10)	sllx(n12)		
14	or(n13)	or(n27)		
15	or(n19)	or(n29)		
16	ld(n22)	add(n32)		

Table 4.4: Kernel for Loop Example

4.7 Loop Reconstruction

The loop reconstruction phase is responsible for generating the prologues, epilogues, kernel, and fixing the control flow of the original program to branch to the modulo scheduled loop. Figure 4.11 shows the loop reconstruction algorithm.

The kernel constructed by the scheduling phase consists of instructions from multiple stages. Instructions from a stage greater than zero are a part of a previous iteration. Prior to entering the kernel, the previous iterations must be initiated in the prologues. Lines 6-14 in Figure 4.11 illustrate how the prologue is constructed. There are as many basic blocks in the prologue as there are stages in the kernel, minus one. For example, our sample loop kernel (Table 4.4) has two stages, and a max stage of one. This results in a prologue with one basic block, which consists of all instructions from the original basic block (in original execution order) that are from stage 0 in the kernel. If an instruction's operand is used in an instruction from a greater stage, a copy of that value is made to save the value. Figure 4.12 shows the generated prologue for our sample loop. Notice the extra *or* and *fmovs* instructions that save values that are used in the kernel, these are the inserted copies.

The epilogue exists to finish iterations that were initiated in either a prologue or the kernel, but have not completed. Lines 18-23 show the steps to create the epilogue. For each stage greater than zero in the kernel, there is a basic block in the epilogue.

The kernel construction is detailed in Lines 24-29 in Figure 4.11. For any instruction that defines a value that is used by an instruction from a later stage, that value must be saved. Instructions from stages greater than zero are then updated to use the correct version of the value. Figure 4.13 shows the kernel for our example loop.

Finally, the branches need to be corrected to branch to the proper basic block. For each basic block in the prologue, the branch must be updated to either branch to the next basic block in the prologue (or kernel if its the last basic block) or to the corresponding basic block in the epilogue. The kernel branch is updated to branch to itself or to the first epilogue. Epilogue branches are changed to unconditional branches to the next basic block in the epilogue or the original loop exit point. Lastly, the branch to the original loop in our program must be updated to branch to the prologue.

Once the prologue, epilogue, and kernel have been generated, the loop has been successfully

```
$maxStage$ = maximum stage in kernel
$Prologue$ = list of prologue basic blocks
$Epilogue$ = list of epilogue basic blocks
$kernelBB$ = new kernel basic block

for($i = 0$; $i <= maxStage$; $++i$)       //Create Prologue
  $BB$ = new basic block
  for($j = i$; $j >= 0$; $- - j$)
    $\forall n$ instructions in original basic block
      if ($n \in$ kernel at stage $j$)
        BB.add(n)
        if ($n$ defines value used in kernel at later stage)
          BB.add(copy value instruction)
  Prologue.add(BB)

for($i = maxStage - 1$; $i >= 0$; –$i$)    //Create Epilogue
  $BB$ = new basic block
  for($j = maxStage$; $j > i$; –$j$)
    $\forall n$ instructions in original basic block
      if ($n \in$ kernel at stage $j$)
        update n to use correct operand values
        BB.add(n)
  Epilogue.add(BB)

$\forall n$ instructions $\in$ kernel                //Create Kernel
  if ($n \in$ kernel at $stage > 0$)
    update n to use correct operand values
  if ($n$ defines value used in kernel at later stage)
    BB.add(copy value instruction)

$\forall b \in Prologue$                             //Update Prologue Branches
  if ($b$ not last $\in Prologue$)
    update branch to branch to correct bb in the epilogue/prologue
  else
    update branch to branch to kernel/epilogue

$\forall b \in Epilogue$                             //Update Epilogue Branches
  if ($b$ not last $\in Epilogue$)
    change branch to unconditional branch to next basic block
  else
    change branch to unconditional branch to original loop exit

Update kernel branch to branch to kernel/epilogue
Update program's branch to original loop to branch to the prologue
```

Figure 4.11: Pseudo Code for Loop Reconstruction Algorithm

modulo scheduled and the Swing Modulo Scheduling algorithm has completed. SMS is applied to each single basic block loop in the program.

```
Prologue:
sethi %lm(-1), %reg(val 0x100d0eb20)
sethi %hh(%disp(addr-of-val A)), %reg(val 0x100d31a90)
add %reg(val 0x100bb0200 i.0.0:PhiCp), %g0, %reg(val 0x100baf6a0 i.0.0)
sethi %hh(<cp#1>), %reg(val 0x100d18060)
or %reg(val 0x100d31a90), %hm(%disp(addr-of-val A)), %reg(val 0x100d31b30)
sethi %lm(%disp(addr-of-val A)), %reg(val 0x100d31c70)
or %reg(val 0x100d18060), %hm(<cp#1>), %reg(val 0x100d15740)
or %reg(val 0x100d0eb20), %lo(-1), %reg(val 0x100d0ea80)
add %reg(val 0x100baf6a0 i.0.0), %reg(val 0x100d0ea80), %reg(val 0x100d0e9e0 maskHi)
sethi %lm(<cp#1>), %reg(val 0x100d18100)
sllx %reg(val 0x100d31b30), 32, %reg(val 0x100d31bd0)
sllx %reg(val 0x100d15740), 32, %reg(val 0x100d157e0)
or %reg(val 0x100d18100), %reg(val 0x100d157e0), %reg(val 0x100d15880)
sethi %hh(%disp(addr-of-val A)), %reg(val 0x100d12f60)
or %reg(val 0x100d31c70), %reg(val 0x100d31bd0), %reg(val 0x100d31d10)
srl %reg(val 0x100d0e9e0 maskHi), 0, %reg(val 0x100bb9a50 tmp.8)
or %reg(val 0x100d31d10), %lo(%disp(addr-of-val A)), %reg(val 0x100d319f0)
sll %reg(val 0x100bb9a50 tmp.8), 2, %reg(val 0x100d31950)
or %reg(val 0x100d15880), %lo(<cp#1>), %reg(val 0x100d12ec0)
or %reg(val 0x100d12f60), %hm(%disp(addr-of-val A)), %reg(val 0x100d13000)
add %reg(val 0x100d319f0), 0, %reg(val 0x100bb73a0 addrOfGlobal:A2)
ld %reg(val 0x100d12ec0), 0, %reg(val 0x100d17fc0)
fmovs %reg(val 0x100d17fc0), %reg(val 0x100d43f20)
sllx %reg(val 0x100d13000), 32, %reg(val 0x100d10640)
sethi %lm(%disp(addr-of-val A)), %reg(val 0x100d106e0)
ld %reg(val 0x100bb73a0 addrOfGlobal:A2), %reg(val 0x100d31950), %reg(val 0x100bb9bf0 tmp.11)
fmovs %reg(val 0x100bb9bf0 tmp.11), %reg(val 0x100d43fc0)
sll %reg(val 0x100baf6a0 i.0.0), 2, %reg(val 0x100d318b0)
or %reg(val 0x100d318b0), 0, %reg(val 0x100d44060)
or %reg(val 0x100d106e0), %reg(val 0x100d10640), %reg(val 0x100d10780)
add %reg(val 0x100baf6a0 i.0.0), 1, %reg(val 0x100cfb200 maskHi)
or %reg(val 0x100d10780), %lo(%disp(addr-of-val A)), %reg(val 0x100d33d30)
srl %reg(val 0x100cfb200 maskHi), 0, %reg(val 0x100bb9e40 indvar.next)
add %reg(val 0x100bb9e40 indvar.next), %g0, %reg(val 0x100bb0200 i.0.0:PhiCp)
add %reg(val 0x100d33d30), 0, %reg(val 0x100bb7460 addrOfGlobal:A1)
or %reg(val 0x100bb7460 addrOfGlobal:A1), 0, %reg(val 0x100d44100)
subcc %reg(val 0x100bb9e40 indvar.next), 500, %g0, %ccreg(val 0x100d343f0)
or %reg(val 0x100d44060), 0, %reg(val 0x100cff6e0)
fmovs %reg(val 0x100d43f20), %reg(val 0x100cff780)
fmovs %reg(val 0x100d43fc0), %reg(val 0x100cff820)
or %reg(val 0x100d44100), 0, %reg(val 0x100cfce60)
fmovs %reg(val 0x100d43fc0), %reg(val 0x100cfcf00)
fmovs %reg(val 0x100d43f20), %reg(val 0x100cf01d0)
or %reg(val 0x100d44100), 0, %reg(val 0x100cf0270)
or %reg(val 0x100d44060), 0, %reg(val 0x100cf0310)
be %ccreg(val 0x100d343f0), %disp(label Epilogue)
nop
ba %disp(label Kernel)
nop
```

Figure 4.12: Modulo Scheduled Loop for our Example Loop (Prologue)

```
Kernel:
or %reg(val 0x100cff6e0), 0, %reg(val 0x100d05750)
fmovs %reg(val 0x100cff780), %reg(val 0x100d44220)
fmovs %reg(val 0x100cff820), %reg(val 0x100d4ace0)
or %reg(val 0x100cfce60), 0, %reg(val 0x100d056b0)
sethi %hh(%disp(addr-of-val A)), %reg(val 0x100d31a90)
add %reg(val 0x100bb0200 i.0.0:PhiCp), %g0, %reg(val 0x100baf6a0 i.0.0)
sethi %lm(%disp(addr-of-val A)), %reg(val 0x100d31c70)
sethi %lm(-1), %reg(val 0x100d0eb20)
or %reg(val 0x100d31a90), %hm(%disp(addr-of-val A)), %reg(val 0x100d31b30)
or %reg(val 0x100d0eb20), %lo(-1), %reg(val 0x100d0ea80)
sllx %reg(val 0x100d31b30), 32, %reg(val 0x100d31bd0)
fmuls %reg(val 0x100d4ace0), %reg(val 0x100d44220), %reg(val 0x100bb9c70 tmp.12)
add %reg(val 0x100baf6a0 i.0.0), %reg(val 0x100d0ea80), %reg(val 0x100d0e9e0 maskHi)
or %reg(val 0x100d31c70), %reg(val 0x100d31bd0), %reg(val 0x100d31d10)
srl %reg(val 0x100d0e9e0 maskHi), 0, %reg(val 0x100bb9a50 tmp.8)
or %reg(val 0x100d31d10), %lo(%disp(addr-of-val A)), %reg(val 0x100d319f0)
sll %reg(val 0x100bb9a50 tmp.8), 2, %reg(val 0x100d31950)
add %reg(val 0x100d319f0), 0, %reg(val 0x100bb73a0 addrOfGlobal:A2)
ld %reg(val 0x100bb73a0 addrOfGlobal:A2), %reg(val 0x100d31950), %reg(val 0x100bb9bf0 tmp.11)
fmovs %reg(val 0x100bb9bf0 tmp.11), %reg(val 0x100d05610)
st %reg(val 0x100bb9c70 tmp.12), %reg(val 0x100d056b0), %reg(val 0x100d05750)
sll %reg(val 0x100baf6a0 i.0.0), 2, %reg(val 0x100d318b0)
or %reg(val 0x100d318b0), 0, %reg(val 0x100d2c020)
sethi %hh(%disp(addr-of-val A)), %reg(val 0x100d12f60)
sethi %lm(%disp(addr-of-val A)), %reg(val 0x100d106e0)
sethi %hh(<cp#1>), %reg(val 0x100d18060)
or %reg(val 0x100d12f60), %hm(%disp(addr-of-val A)), %reg(val 0x100d13000)
or %reg(val 0x100d18060), %hm(<cp#1>), %reg(val 0x100d15740)
sllx %reg(val 0x100d13000), 32, %reg(val 0x100d10640)
sethi %lm(<cp#1>), %reg(val 0x100d18100)
sllx %reg(val 0x100d15740), 32, %reg(val 0x100d157e0)
or %reg(val 0x100d18100), %reg(val 0x100d157e0), %reg(val 0x100d15880)
or %reg(val 0x100d106e0), %reg(val 0x100d10640), %reg(val 0x100d10780)
or %reg(val 0x100d15880), %lo(<cp#1>), %reg(val 0x100d12ec0)
add %reg(val 0x100baf6a0 i.0.0), 1, %reg(val 0x100cfb200 maskHi)
or %reg(val 0x100d10780), %lo(%disp(addr-of-val A)), %reg(val 0x100d33d30)
ld %reg(val 0x100d12ec0), 0, %reg(val 0x100d17fc0)
fmovs %reg(val 0x100d17fc0), %reg(val 0x100d2c0c0)
srl %reg(val 0x100cfb200 maskHi), 0, %reg(val 0x100bb9e40 indvar.next)
add %reg(val 0x100bb9e40 indvar.next), %g0, %reg(val 0x100bb0200 i.0.0:PhiCp)
add %reg(val 0x100d33d30), 0, %reg(val 0x100bb7460 addrOfGlobal:A1)
or %reg(val 0x100bb7460 addrOfGlobal:A1), 0, %reg(val 0x100d2c160)
subcc %reg(val 0x100bb9e40 indvar.next), 500, %g0, %ccreg(val 0x100d343f0)
or %reg(val 0x100d2c020), 0, %reg(val 0x100cff6e0)
fmovs %reg(val 0x100d2c0c0), %reg(val 0x100cff780)
fmovs %reg(val 0x100d05610), %reg(val 0x100cff820)
or %reg(val 0x100d2c160), 0, %reg(val 0x100cfce60)
fmovs %reg(val 0x100d05610), %reg(val 0x100cfcf00)
fmovs %reg(val 0x100d2c0c0), %reg(val 0x100cf01d0)
or %reg(val 0x100d2c160), 0, %reg(val 0x100cf0270)
or %reg(val 0x100d2c020), 0, %reg(val 0x100cf0310)
be %ccreg(val 0x100d343f0), %disp(label Epilogue)
nop
ba %disp(label Kernel)
nop

Epilogue:
fmovs %reg(val 0x100cfcf00), %reg(val 0x100d2c280)
fmovs %reg(val 0x100cf01d0), %reg(val 0x100d2c320)
fmuls %reg(val 0x100d2c280), %reg(val 0x100d2c320), %reg(val 0x100bb9c70 tmp.12)
or %reg(val 0x100cf0270), 0, %reg(val 0x100d442c0)
or %reg(val 0x100cf0310), 0, %reg(val 0x100d2c4d0)
st %reg(val 0x100bb9c70 tmp.12), %reg(val 0x100d442c0), %reg(val 0x100d2c4d0)
ba %disp(label loopexit)
nop
```

Figure 4.13: Modulo Scheduled Loop for our Example Loop (Kernel and Epilogue)

Chapter 5

Extending Swing Modulo Scheduling for Superblocks

On many programs, Swing Modulo Scheduling is limited by only handling single basic block loops. The potential for parallelism is increased if instructions can be moved across basic block boundaries, which means that instructions are moved across conditional branches. However, moving instructions above or below a conditional branch can alter the programs behavior if not done safely.

Traditional Modulo Scheduling techniques only transform single basic block loops without control flow, resulting in many missed opportunities for parallelism. However, very few Modulo Scheduling techniques can handle multiple basic block loops. These techniques, called Global Modulo Scheduling, were discussed in Section 3.2. All Global Modulo Scheduling algorithms [25, 41] schedule *all paths* within the loop, which involves taking resource usage and dependence constraints for each execution path. However, one execution path may be taken more often than another. In these situations, Modulo Scheduling should aim to decrease the execution time for the most frequently executed path even though this could increase the execution time of the less frequent path. Overall, the performance of the program will be increased.

Trace Scheduling is a technique for general instruction scheduling (not Software Pipelining) that schedules frequently executed paths, called traces. Traces are a sequence of basic blocks that may have exits out of the middle (called side exits), and transitions from other traces into the middle (called side entrances). These multiple-entry multiple-exit groups of basic blocks are scheduled ignoring the side exits and entrances, but extra bookkeeping is done to ensure the program is correct regardless of which path is taken. This bookkeeping increases the complexity of scheduling.

Removing the side entrances forms a superblock, which decreases the scheduling complexity.

We extended Swing Modulo Scheduling to support superblock loops in order to take advantage of the parallelism of multiple basic block loops. This chapter will discuss the details of what changes were made to the algorithm described in Chapter 4.

While these extensions were implemented as a static optimization in the SPARC V9 backend of the LLVM Compiler Infrastructure, it can seamlessly be applied to superblock loops found at runtime or offline using profile information.

5.1 Overview

We extended Swing Modulo Scheduling to handle superblock loops, which are single-entry, multiple-exit, multiple basic block loops. These extensions allow instructions to be moved above conditional branches (upward code motion) or below conditional branches (downward code motion).

Downward code motion occurs when an instruction is moved below a conditional branch. It is fairly straight forward and only requires that the branch is not dependent upon the instruction that is being moved. A copy of that instruction is placed in the side exit in the event that the branch was actually taken, ensuring that the programs behavior is unaltered.

Upward code motion is the process of moving an instruction above a conditional branch. The execution of this instruction is termed "speculative execution", because the execution of the instruction occurs before it would have (or not have) in original program order. This instruction originally was executed only if the branch was not taken. Upward code motion is useful for hiding the latency of load instructions or other high latency instructions. In order for an instruction to be a candidate for upward code motion, two restrictions must be met [8, 29]:

1. **The destination of the instruction must not be used before it is redefined when the branch is taken (exited from superblock).**

2. **The instruction must never cause an exception that may halt the programs execution when the branch is taken.**

The first restriction ensures that if an instruction is moved above a branch and that instruction defines a value, then if the branch is taken, all instructions after the branch will not use that value.

By moving the instruction above the branch, we are performing a computation that may not have been executed in the original program. Therefore, if the branch was miss-predicted, we need to guarantee that instructions after the branch are using the right value (as in the original program).

Because the LLVM Compiler Infrastructure Intermediate representation and the SPARC V9 backend intermediate representation are in SSA form, there is no risk of any value being defined twice. However, because Swing Modulo Scheduling is attempting to overlap iterations of the loop, and loops can redefine dynamic values, it is important to not redefine the value before the outcome of the side exit is known.

The second restriction guards against exceptions from halting the program due to moving an instruction above a branch. Because the instruction moved above the branch is being executed before it would have in the original program it may never have been executed. This exception is not one that would have occurred if the instruction had not been moved.

The second restriction can be relaxed depending upon which architecture SMS is implemented for. For some architectures, such as the SPARC V9, a subset of instructions can potential trap or cause exceptions(such as floating point arithmetic). These instructions can not be moved upward because the programs execution behavior could be altered (an exception could cause the program to abnormally halt). If the architecture provides non-trapping versions of these instructions or general support for predication, those can be used instead and the non-trapping instruction can safely be moved above the branch. Unfortunately, this is not an option for the SPARC V9.

The best type of hardware support is one that provides *Predicated Execution*, such as IA64 [20]. Predicated Execution allows instructions to be nullified during their execution in the pipeline. So if an instruction is speculatively moved above a branch and the branch is taken, even if the instruction is already in the processor's pipeline, the instruction can be nullified. Therefore the programs behavior is not altered and values are not incorrectly redefined.

5.2 Changes to Dependence Graph

In order to successfully Modulo Schedule a superblock loop and maintain correct execution of the program, some changes need to be made to the Data Dependence Graph. These changes ensure that the restrictions mention in Section 5.1 for code motion are met.

Recall from Section 4.2 that a Data Dependence Graph consists of nodes for each instruction, and edges represent the dependence between instructions. There are three main changes made to the Data Dependence Graph:

- A predicate node is introduced to represent the instructions related to the inner branches of the superblock.
- Edges are created between the predicate node and all trapping instructions.
- Edges are created between the predicate node and all instructions that define values that are live if a side exit branch is taken.

The first change introduces a new node called the *predicate node*. This node represents the instruction that computes the condition (for a conditional branch) and the branch itself. This node allows those instructions to be treated as one instruction and will be scheduled for the same stage. Having the condition and branch instructions in the same stage is crucial for proper execution of the superblock loop.

Data dependences between the predicate node and other instructions are created as described in Section 4.2, knowing which values the instructions (that the predicate node represents) uses and defines. Additionally, dependences are created between a predicate node and any other predicate nodes after it. This is to ensure that the side exits of the superblock are preserved in order.

The second change upholds the second code motion restriction. On the SPARC V9, loads, stores, integer divide, and all floating point arithmetic potentially trap. Therefore, a dependence is created between those instructions and the predicate node for the branch that the instruction would be moved above (if allowed). Because of the dependences between predicate nodes, it is not necessary to add edges between all trapping instructions and all predicate nodes. It is only necessary to add them between the trapping instruction and the predicate node for the predecessor basic block.

Dependences between the predicate node and load instructions can be eliminated if it can be proven that accessing the memory is safe. This occurs when the load is from global or stack memory and the index is within the legal bounds of the memory allocated.

The last change is to prevent a value from being redefined before it is used. This situation can occur if the predicate node is scheduled in the kernel from a previous iteration (stage > 0). It is possible that instructions that are before the branch in the original program, are executed before the branch is determined to be taken or not. For values that are live if the branch is taken, this will produce incorrect results. Therefore loop-carried dependences between the predicate node and all instructions that define values that are live outside the trace must be created. This will ensure that all the instructions that are live if the side exit is taken are from the same iteration in the kernel.

```
$LiveOutside$ = Empty list of instructions
$\forall n$ instructions $\in$ superblock
  $\forall u \in uses(n)$
  if (!inSuperBlock($u$))
    $LiveOutside.add(n)$
```

Figure 5.1: Pseudo Code for Determining Values Live Outside the Trace

To determine which values are live outside the trace, a simple version of live variable analysis is performed. Figure 5.1 illustrates the simple approach used by our implementation. The uses of each instruction are examined. Each using instruction belongs to a basic block (its parent). For each of the uses of the value, is basic block is tested to see if it belongs to the set of basic blocks that make up the superblock. If a user is not in the superblock, the value is determine to be live outside the trace.

5.3 Changes to Loop Reconstruction

Section 4.7 described the steps taken to reconstruct the modulo scheduled loop into a prologue, kernel, and epilogue. The Swing Modulo Scheduling extensions for superblock loops must also reconstruct the loop into a prologue, kernel, and epilogue. The main difference between the extensions and the standard algorithm is that the prologue, epilogue, and kernel are all superblocks, and consist of multiple basic blocks with side exits. Changes must be made to make sure the side exits are handled properly.

Figure 5.2 shows the loop reconstruction algorithm for superblock loops. Lines 1-30 are identical to the standard implementation, but the prologue, epilogue, and the kernel are one or more

```
$maxStage$ = maximum stage in kernel
$Prologue$ = list of prologue superblocks
$Epilogue$ = list of epilogue superblocks
$kernelBB$ = new kernel superblock

for($i = 0$; $i <= maxStage$; $++i$)      //Create Prologue
    $BB$ = new superblock
    for($j = i$; $j >= 0$; $--j$)
        $\forall n$ instructions in the superblock
            if ($n \in$ kernel at stage $j$)
                BB.add(n)
                if ($n$ defines value used in kernel at later stage)
                    BB.add(copy value instruction)
    Prologue.add(BB)

for($i = maxStage - 1$; $i >= 0$; $-i$)    //Create Epilogue
    $BB$ = new superblock
    for($j = maxStage$; $j > i$; $-j$)
        $\forall n$ instructions in the superblock
            if ($n \in$ kernel at stage $j$)
                update n to use correct operand values
                BB.add(n)
    Epilogue.add(BB)

$\forall n$ instructions $\in$ kernel                //Create Kernel
    if ($n \in$ kernel at $stage > 0$)
        update n to use correct operand values
    if ($n$ defines value used in kernel at later stage)
        BB.add(copy value instruction)

$\forall sideExits \in$ the superblock
    $sideExitBlock$ = new basic block
    $\forall n$ instructions moved below this side exit
        if ($n \in$ kernel at stage 0)
            sideExitBlock.add(n)
    $sideEpilogue$ = clone epilogue
    update last branch in epilogue to branch to $sideExitBlock$
    update $sideExitBlock$ to branch to original loop exit

$\forall b \in Prologue$                           //Update Prologue Branches
    if ($b$ is not a side exit)
        update branch to correct superblock in cloned epilogue for this side exit
    else
        if ($b$ not last $\in Prologue$)
            update branch to branch to correct superblock in epilogue/prologue
        else
            update branch to branch to kernel/epilogue

$\forall b \in Epilogue$                           //Update Epilogue Branches
    if ($b$ not last $\in Epilogue$)
        change branch to unconditional branch to next superblock in epilogue
    else
        change branch to unconditional branch to original loop exit

Update kernel branch to branch to kernel/epilogue
Update program's branch to original loop to branch to prologue
```

Figure 5.2: Pseudo Code for Loop Reconstruction Algorithm for Superblocks

superblocks. Lines 31-37 are the first steps for handling side exits. For each side exit in the original superblock, a new *Side Exit Block* is created. Instructions moved below this side exit are placed into the new basic block. Because the epilogue finishes any iterations that are in flight, the side exit block only includes instructions from the current iteration (stage 0). Second, the epilogue is cloned and the last superblock's branch in the epilogue is updated to branch to the new side exit block. An unconditional branch is added to the side exit block to branch to the original loop exit in the program.

Once the side exit and corresponding side exit epilogues have been created, the prologue branches must be updated. Lines 39-48, update the side exit branches in the prologue to branch to the corresponding epilogue for each side exit. The last branch of each superblock in the prologue either branches to the next superblock in the prologue or the kernel. Other branch updates (Lines 49-56) require no changes from the original algorithm.

Once the superblock loop has been reconstructed, the Swing Modulo Scheduling for superblocks pass is complete.

5.4 Superblock Loop Example

To understand the changes made to the original Swing Modulo Scheduling algorithm, we detail the steps for a simple superblock loop example. Figure 5.4 shows the C and LLVM code for a simple superblock loop. The loop computes and store values for two arrays. It has a side exit in the loop if one of the previous array values is less than some value. Note that this loop consists of two basic blocks, a single entry (aside from the back edge), and one side exit. Figure 5.4 shows the LLVM Machine code for our example superblock loop. Recall that this machine code closely resembles the SPARC V9 assembly.

The first step is to construct the Data Dependence Graph for the instructions that make up the body of the loop. Figure 5.5 is the DDG for our example superblock loop. The first thing to notice is that the conditional branch instructions (n13, n14, n15) have been represented by a PredNode node for the no_exit basic block. The dependences between the instructions it represents and other instructions in the loop body have been preserved.

While the majority of the dependencies between the instructions are created as normal, a few

```
for(i = 1; i < 500; ++i) {
  B[i] = B[i-1] * 3.2f;
  if(A[i-1] < 4.5f)
   break;
  A[i] = A[i-1] * 3.4f;
}
```

(a) C Code

```
no_exit:
    %indvar = phi uint [ 0, %entry ], [ %i.0.0, %endif ]
    %i.0.0 = add uint %indvar, 1
    %tmp.6 = cast uint %i.0.0 to long
    %tmp.7 = getelementptr [500 x float]* %TMP2, long 0, long %tmp.6
    %tmp.10 = cast uint %indvar to long
    %tmp.11 = getelementptr [500 x float]* %TMP2, long 0, long %tmp.10
    %tmp.12 = load float* %tmp.11
    %tmp.13 = mul float %tmp.12, %FPVAL2
    store float %tmp.13, float* %tmp.7
    %tmp.17 = getelementptr [500 x float]* %TMP, long 0, long %tmp.10
    %tmp.18 = load float* %tmp.17
    %tmp.19 = setlt float %tmp.18, %FPVAL3
    br bool %tmp.19, label %loopexit, label %endif

endif:
    %tmp.23 = getelementptr [500 x float]* %TMP, long 0, long %tmp.6
    %tmp.29 = mul float %tmp.18, %FPVAL
    store float %tmp.29, float* %tmp.23
    %inc = add uint %indvar, 2
    %tmp.3 = setlt uint %inc, 500
    br bool %tmp.3, label %no_exit, label %loopexit
```

(b) LLVM Code

Figure 5.3: Simple Superblock Loop Example

are added to control upward and downward code motion. Because there are no instructions that are live when the side exit is taken, no loop-carried dependencies between the PredNode and those instructions are created. Second, you will notice that there are dependencies created between the PredNode and the store (n22), and the PredNode and the fmuls (n19). Those two instructions could potentially cause an exception and alter the original behavior of the program. Therefore, the dependencies are created to prevent the instructions from being moved above the branch (upward code motion).

Once the DDG has been created, the MII value must be determined by computing the ResMII and RecMII as described in Section 4.3. For our superblock loop example there are four recurrences, with the highest RecMII value of 8. This is from the recurrence consisting of the st (n11), ld (n7), and fmuls (n10) and also from the recurrence consisting of PredNode, ld (n12), fmuls (n19), and st (n22). Because these recurrences have the same RecMII, it does not matter which one is chosen as the highest RecMII. Each recurrence has a total latency of 8 and a distance of 1, which results in a RecMII of 8. Table 5.1 shows the latencies for each instruction. Please note that the PredNode has a latency of 3 because that is the total latency of all the instructions is represents.

The resource usage is totaled for all instructions in the loop body. The most heavily used

```
no_exit:
  (n1)  add %reg(val 0x100c4eae0 indvar:PhiCp), %g0, %reg(val 0x100c4df10 indvar)
  (n2)  add %reg(val 0x100c4df10 indvar), 1, %reg(val 0x100dd7480 maskHi)
  (n3)  add %g0, %reg(val 0x100c4df10 indvar), %reg(val 0x100c585b0 tmp.10)
  (n4)  sll %reg(val 0x100c585b0 tmp.10), 2, %reg(val 0x100d9a020)
  (n5)  sll %reg(val 0x100c585b0 tmp.10), 2, %reg(val 0x100dd75c0)
  (n6)  srl %reg(val 0x100dd7480 maskHi), 0, %reg(val 0x100c58ab0 i.0.0)
  (n7)  ld %reg(val 0x100c547e0 TMP2), %reg(val 0x100dd75c0), %reg(val 0x100c58dc0 tmp.12)
  (n8)  add %g0, %reg(val 0x100c58ab0 i.0.0), %reg(val 0x100c58b50 tmp.6)
  (n9)  sll %reg(val 0x100c58b50 tmp.6), 2, %reg(val 0x100dd7520)
  (n10) fmuls %reg(val 0x100c58dc0 tmp.12), %reg(val 0x100c586b0 FPVAL2), %reg(val 0x100c58e40 tmp.13)
  (n11) st %reg(val 0x100c58e40 tmp.13), %reg(val 0x100c547e0 TMP2), %reg(val 0x100dd7520)
  (n12) ld %reg(val 0x100c58530 TMP), %reg(val 0x100d9a020), %reg(val 0x100c591a0 tmp.18)
  (n13) %ccreg(val 0x100dd7c90) = fcmps %reg(val 0x100c591a0 tmp.18), %reg(val 0x100c58730 FPVAL3)
  (n14) fbl %ccreg(val 0x100dd7c90), %disp(label loopexit)
  (n15) ba %disp(label endif)

endif
  (n16) add %reg(val 0x100c58ab0 i.0.0), %g0, %reg(val 0x100c4eae0 indvar:PhiCp)
  (n17) sll %reg(val 0x100c58b50 tmp.6), 2, %reg(val 0x100dd7d30)
  (n18) add %reg(val 0x100c4df10 indvar), 2, %reg(val 0x100d9a0c0 maskHi)
  (n19) fmuls %reg(val 0x100c591a0 tmp.18), %reg(val 0x100c58630 FPVAL), %reg(val 0x100c59490 tmp.29)
  (n20) srl %reg(val 0x100d9a0c0 maskHi), 0, %reg(val 0x100c595d0 inc)
  (n21) subcc %reg(val 0x100c595d0 inc), 500, %g0, %ccreg(val 0x100dd7dd0)
  (n22) st %reg(val 0x100c59490 tmp.29), %reg(val 0x100c58530 TMP), %reg(val 0x100dd7d30)
  (n23) bcs %ccreg(val 0x100dd7dd0), %disp(label no_exit)
  (n24) ba %disp(label loopexit)
```

Figure 5.4: LLVM Machine Code for a Superblock Loop

resource is the integer unit (15 uses). Because there are two of this resource, the ResMII is computed to be 8. The maximum of ResMII (8) and RecMII (8) is used for MII, which means that MII is set to 8.

The next phase of the extended Modulo Scheduling algorithm is to calculate the various properties for each node. These properties are used to order the nodes for scheduling. The ASAP, ALAP, MOB, Height, and Depth (described in Section 4.3) are computed for each node in the DDG. For these calculations one back edge (doesn't matter which one) is ignored in order to avoid endlessly cycling in the graph. Table 5.2 shows the node attributes for this example. The PredNode is treated like any other node for these calculations.

Node	Latency	Node	Latency
PredNode (no_exit)	1	add (n3)	3
sll (n4)	1	sll (n5)	1
ld (n7)	3	add (n8)	1
sll (n9)	1	fmuls (n10)	4
st (n11)	1	ld (n12)	3
sll (n17)	1	fmuls (n19)	4
st (n22)	1		

Table 5.1: Node Latencies for Simple Loop Example

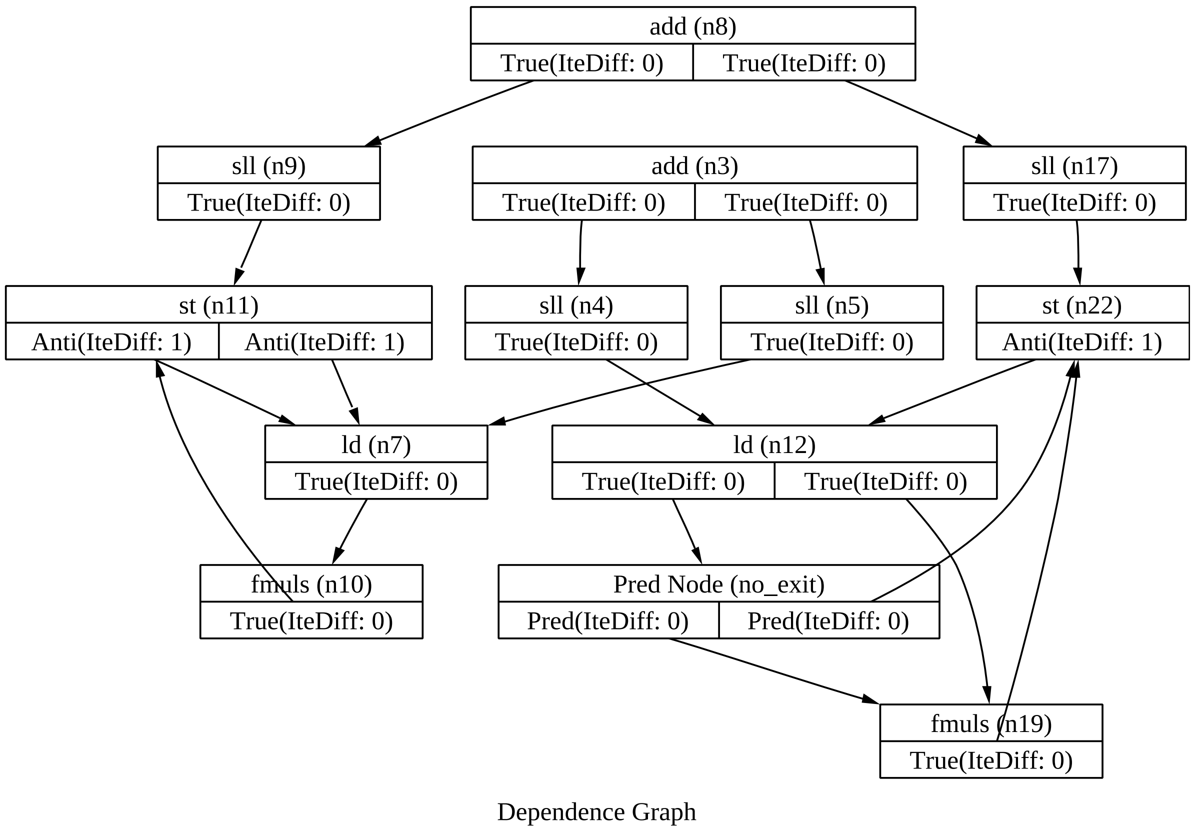


Figure 5.5: Dependence Graph After Dependence Analysis

Node	ASAP	ALAP	MOB	Depth	Height	Latency
PredNode (no_exit)	5	5	0	5	7	3
add (n3)	0	0	0	0	12	1
sll (n4)	1	1	0	1	11	1
sll (n5)	1	4	3	1	8	1
ld (n7)	2	5	3	2	7	3
add (n8)	0	10	10	0	2	1
sll (n9)	1	11	10	1	1	1
fmuls (n10)	5	8	3	5	4	4
st (n11)	9	12	3	9	0	1
ld (n12)	2	2	0	2	10	3
sll (n17)	1	11	10	1	1	1
fmuls (n19)	8	8	0	8	4	4
st (n22)	12	12	0	12	0	1

Table 5.2: Node Attributes for Simple Loop Example

Our extended Swing Modulo Scheduling algorithm performs node ordering exactly as the original algorithm. Using the node attributes and the DDG, it determines the best ordering possible in order to achieve the most optimal schedule. It begins by determining the partial order which is a list of sets of nodes. Figure 5.6 shows the partial order for our example. Recall from Section 4.5 that the partial order lists the recurrences from the highest RecMII to the lowest, with the remaining sets consisting of the other connected components of the graph. Since we have two recurrences with the same RecMII, those two occur first in the partial order (Set #1 and Set #2). Sets #3 and #4 represent the remaining two connected components of the DDG once the two recurrences are removed.

```
Set #1: PredNode (no_exit), ld (n12), fmuls (n19), st (n22)
Set #2: ld (n7), fmuls (n10), st (n11)
Set #3: add (n3), sll (n4), sll (n5)
Set #4: add (n8), sll (n9), sll (n17)
```

Figure 5.6: Superblock Loop Example Partial Order

Using this partial order, the next phase of our extended SMS algorithm (like the original) is to determine the final node ordering. The final node ordering algorithm (described in Figure 4.9) traverses each subgraph that each set represents in the partial order. It begins with Set #1 and determines that the st (n22) is the most critical node and places it first on the final order list. Each ancestor to the st (n22) is visited according to their depth and added to the final node order. This means that fmuls (n19) with a depth of 8 (greatest depth out of Set #1) is added first, followed by PredNode, and finally the ld (n12). This process is repeated for each set in the partial order (traversing bottom-up or top-down) until all nodes have been added to the final order. The list below is the final node ordering for our example:

O = { st (n22), fmuls (n19), PredNode (no_exit), ld (n12), st(n11), fmuls (n10), ld (n7), sll (n4), sll (n5), add (n3), sll (n9), sll (n17), add (n8) }

Using the final node ordering, each instruction is placed in the final schedule. Table 5.3 shows the schedule for a single iteration of our superblock loop example. The SPARC IIIi architecture can issue 4 instructions per cycle. The combination of instructions that can be issued depends on what resources each instruction uses during each stage of the pipeline. This information is acquired using the SchedInfo API discussed in Section 4.1.1. For simplicity the schedule show in Table 5.3 only

Cycle	Issue1	Issue2	Issue3	Issue4
0	add(n3)			
1	sll(n5)			
2	ld(n7)			
3				
4	sll(n4)			
5	ld(n12)			
6				
7				
8	fmuls(n10)	PredNode(no_exit)		
9				
10	add(n8)			
11	sll(n9)	fmuls(n19)		
12	st(n11)			
13				
14	sll(n17)			
15	st(n22)			

Table 5.3: Schedule for a Single Iteration of the Loop Example

shows the issue slots, but the scheduling algorithm checks that both an issue slot and all resources are available.

Our achieved II value for this example was 10 cycles, which is greater than our calculated MII (8 cycles). This means that all instructions from cycle 0 to cycle 10 are from the current iteration in the kernel, and all instructions scheduled after cycle 10 belong to another stage in the kernel. Table 5.4 shows the kernel for our Modulo Scheduled loop.

Cycle	Issue1	Issue2	Issue3	Issue4
0	add(n3)	sll(n9)[1]	fmuls(n19)[1]	
1	sll(n5)	st(n11[1])		
2	ld(n7)			
3	sll(n17)[1]			
4	sll(n4)	st(n22)[1]		
5	ld(n12)			
6				
7				
8	fmuls(n10)	PredNode(no_exit)		
9				
10	add(n8)			

Table 5.4: Kernel for Loop Example

Because we have instructions in the kernel that are from a stage greater than zero (which means that they complete a previous iteration), we will have a prologue and an epilogue. This is identical to the original Swing Modulo Scheduling algorithm, however we must handle side exits properly. The new reconstructed algorithm described in Figure 5.2 transforms our superblock loop into a prologue, kernel, epilogue, side exit, and side epilogue. Figure 5.7, Figure 5.8 and Figure 5.9 show the final LLVM machine code after the reconstruction. The key difference to note is that the side exit in the prologue and kernel both branch to a side epilogue (to complete the iterations in flight) and than to a side exit (to execute any instructions moved below the conditional branch) before exiting the loop.

```
PROLOGUE:
add %reg(val 0x100c4eae0 indvar:PhiCp), %g0, %reg(val 0x100c4df10 indvar)
add %reg(val 0x100c4df10 indvar), 1, %reg(val 0x100dd7480 maskHi)
add %g0, %reg(val 0x100c4df10 indvar), %reg(val 0x100c585b0 tmp.10)
sll %reg(val 0x100c585b0 tmp.10), 2, %reg(val 0x100d9a020)
sll %reg(val 0x100c585b0 tmp.10), 2, %reg(val 0x100dd75c0)
srl %reg(val 0x100dd7480 maskHi), 0, %reg(val 0x100c58ab0 i.0.0)
ld %reg(val 0x100c547e0 TMP2), %reg(val 0x100dd75c0), %reg(val 0x100c58dc0 tmp.12)
add %g0, %reg(val 0x100c58ab0 i.0.0), %reg(val 0x100c58b50 tmp.6)
or %reg(val 0x100c58b50 tmp.6), 0, %reg(val 0x100ddeb10)
fmuls %reg(val 0x100c58dc0 tmp.12), %reg(val 0x100c586b0 FPVAL2), %reg(val 0x100c58e40 tmp.13)<
def>
fmovs %reg(val 0x100c58e40 tmp.13), %reg(val 0x100ddcba0)
ld %reg(val 0x100c58530 TMP), %reg(val 0x100d9a020), %reg(val 0x100c591a0 tmp.18)
fmovs %reg(val 0x100c591a0 tmp.18), %reg(val 0x100ddcc40)
%ccreg(val 0x100dd7c90) = fcmps %reg(val 0x100c591a0 tmp.18), %reg(val 0x100c58730 FPVAL3)
fmovs %reg(val 0x100ddcc40), %reg(val 0x100dcf940)
fmovs %reg(val 0x100ddcba0), %reg(val 0x100dcf9e0)
or %reg(val 0x100ddeb10), 0, %reg(val 0x100dcfa80)
or %reg(val 0x100ddeb10), 0, %reg(val 0x100dcfb20)
fmovs %reg(val 0x100ddcba0), %reg(val 0x100ddbe70)
or %reg(val 0x100ddeb10), 0, %reg(val 0x100ddbfa0)
fmovs %reg(val 0x100ddcc40), %reg(val 0x100ddc040)
fbl %ccreg(val 0x100dd7c90), %disp(label SideEpilogue)
nop
ba %disp(label PROLOGUE2)
nop

PROLOGUE2
add %reg(val 0x100c58ab0 i.0.0), %g0, %reg(val 0x100c4eae0 indvar:PhiCp)
add %reg(val 0x100c4df10 indvar), 2, %reg(val 0x100d9a0c0 maskHi)
srl %reg(val 0x100d9a0c0 maskHi), 0, %reg(val 0x100c595d0 inc)
subcc %reg(val 0x100c595d0 inc), 500, %g0, %ccreg(val 0x100dd7dd0)
bcs %ccreg(val 0x100dd7dd0), %disp(label Kernel)
nop
ba %disp(label EPILOGUE)
nop
```

Figure 5.7: Modulo Scheduled Loop for our Superblock Loop (Prologue)

```
Kernel:
fmovs %reg(val 0x100dcf940), %reg(val 0x100ddcf00)
fmovs %reg(val 0x100dcf9e0), %reg(val 0x100ddcfa0)
or %reg(val 0x100dcfa80), 0, %reg(val 0x100ddce60)
add %reg(val 0x100c4eae0 indvar:PhiCp), %g0, %reg(val 0x100c4df10 indvar)
add %reg(val 0x100c4df10 indvar), 1, %reg(val 0x100dd7480 maskHi)
add %g0, %reg(val 0x100c4df10 indvar), %reg(val 0x100c585b0 tmp.10)
sll %reg(val 0x100ddce60), 2, %reg(val 0x100dd7520)
fmuls %reg(val 0x100ddcf00), %reg(val 0x100c58630 FPVAL), %reg(val 0x100c59490 tmp.29)
sll %reg(val 0x100c585b0 tmp.10), 2, %reg(val 0x100dd75c0)
st %reg(val 0x100ddcfa0), %reg(val 0x100c547e0 TMP2), %reg(val 0x100dd7520)
srl %reg(val 0x100dd7480 maskHi), 0, %reg(val 0x100c58ab0 i.0.0)
ld %reg(val 0x100c547e0 TMP2), %reg(val 0x100dd75c0), %reg(val 0x100c58dc0 tmp.12)
sll %reg(val 0x100ddce60), 2, %reg(val 0x100dd7d30)
sll %reg(val 0x100c585b0 tmp.10), 2, %reg(val 0x100d9a020)
st %reg(val 0x100c59490 tmp.29), %reg(val 0x100c58530 TMP), %reg(val 0x100dd7d30)
ld %reg(val 0x100c58530 TMP), %reg(val 0x100d9a020), %reg(val 0x100c591a0 tmp.18)
fmovs %reg(val 0x100c591a0 tmp.18), %reg(val 0x100ddd040)
fmuls %reg(val 0x100c58dc0 tmp.12), %reg(val 0x100c586b0 FPVAL2), %reg(val 0x100c58e40 tmp.13)
fmovs %reg(val 0x100c58e40 tmp.13), %reg(val 0x100ddd0e0)
%ccreg(val 0x100dd7c90) = fcmps %reg(val 0x100c591a0 tmp.18), %reg(val 0x100c58730 FPVAL3)
fmovs %reg(val 0x100ddd040), %reg(val 0x100dcf940)
fmovs %reg(val 0x100ddd0e0), %reg(val 0x100dcf9e0)
fmovs %reg(val 0x100ddd0e0), %reg(val 0x100ddbe70)
fmovs %reg(val 0x100ddd040), %reg(val 0x100ddc040)
fbl %ccreg(val 0x100dd7c90), %disp(label SideEpilogue)
nop
ba %disp(label Kernel2)
nop

Kernel2:
add %g0, %reg(val 0x100c58ab0 i.0.0), %reg(val 0x100c58b50 tmp.6)
or %reg(val 0x100c58b50 tmp.6), 0, %reg(val 0x100ddd180)
add %reg(val 0x100c58ab0 i.0.0), %g0, %reg(val 0x100c4eae0 indvar:PhiCp)
add %reg(val 0x100c4df10 indvar), 2, %reg(val 0x100d9a0c0 maskHi)
srl %reg(val 0x100d9a0c0 maskHi), 0, %reg(val 0x100c595d0 inc)
subcc %reg(val 0x100c595d0 inc), 500, %g0, %ccreg(val 0x100dd7dd0)
or %reg(val 0x100ddd180), 0, %reg(val 0x100dcfa80)
or %reg(val 0x100ddd180), 0, %reg(val 0x100dcfb20)
or %reg(val 0x100ddd180), 0, %reg(val 0x100ddbfa0)
bcs %ccreg(val 0x100dd7dd0), %disp(label Kernel)
nop
ba %disp(label EPILOGUE)
nop
```

Figure 5.8: Modulo Scheduled Loop for our Example Loop (Kernel and Epilogue)

```
SideExit:
add %reg(val 0x100c4eae0 indvar:PhiCp), %g0, %reg(val 0x100c4df10 indvar)
add %reg(val 0x100c4df10 indvar), 1, %reg(val 0x100dd7480 maskHi)
add %g0, %reg(val 0x100c4df10 indvar), %reg(val 0x100c585b0 tmp.10)
sll %reg(val 0x100c585b0 tmp.10), 2, %reg(val 0x100dd75c0)
srl %reg(val 0x100dd7480 maskHi), 0, %reg(val 0x100c58ab0 i.0.0)
ld %reg(val 0x100c547e0 TMP2), %reg(val 0x100dd75c0), %reg(val 0x100c58dc0 tmp.12)
sll %reg(val 0x100c585b0 tmp.10), 2, %reg(val 0x100d9a020)
ld %reg(val 0x100c58530 TMP), %reg(val 0x100d9a020), %reg(val 0x100c591a0 tmp.18)
fmuls %reg(val 0x100c58dc0 tmp.12), %reg(val 0x100c586b0 FPVAL2), %reg(val 0x100c58e40 tmp.13)
%ccreg(val 0x100dd7c90) = fcmps %reg(val 0x100c591a0 tmp.18), %reg(val 0x100c58730 FPVAL3)
add %g0, %reg(val 0x100c58ab0 i.0.0), %reg(val 0x100c58b50 tmp.6)
ba %disp(label loopexit)
nop

SideEpilogue:
or %reg(val 0x100ddbfa0), 0, %reg(val 0x100ddc760)
sll %reg(val 0x100ddc760), 2, %reg(val 0x100dd7d30)
fmovs %reg(val 0x100ddc040), %reg(val 0x100ddc800)
fmuls %reg(val 0x100ddc800), %reg(val 0x100c58630 FPVAL), %reg(val 0x100c59490 tmp.29)
st %reg(val 0x100c59490 tmp.29), %reg(val 0x100c58530 TMP), %reg(val 0x100dd7d30)
ba %disp(label SideExit)
nop
```

Figure 5.9: Modulo Scheduled Loop for our Example Loop (Side Exit and Side Epilogue)

Chapter 6

Results

In this chapter, the Swing Modulo Scheduling algorithm and the extensions for superblock loops are evaluated on the following key issues: efficiency in terms of compile time, how close to optimal the achieved schedule is, and the overall performance impacts of the transformation taking into consideration register spills and execution time.

First, we provide some background information about the SPARC Architecture in Section 6.1, then the results for the SMS algorithm are discussed in Section 6.2 and finally, the results for the superblock extensions are discussed in Section 6.3.

6.1 Ultra SPARC IIIi Architecture

We implemented Swing Modulo Scheduling in the LLVM Compiler Infrastructure [26] as a static optimization in the SPARC V9 backend (Section 4.1). We wrote a scheduling description, described in Section 4.1.1, for the Ultra SPARC IIIi to describe the resources and other scheduling restrictions imposed by the architecture.

The Ultra SPARC IIIi processor, developed by Sun Microsystems, is 4-way super-scalar processor with a 14 stage pipeline. It implements the 64-bit SPARC V9 architecture and can issue up to 4 instructions per clock cycle (given the right mix of instructions).

The scheduling description for the Ultra SPARC IIIi processor describes for each instruction the latency (in cycles), blocking properties, pipeline resource usages, and the grouping rules. The execution units described below give a broad overview of the latencies for each type of instruction based upon the execution unit utilized. Full latency details are available in the Ultra SPARC IIIi manual [1].

Blocking is when the processor halts the dispatch of another group of instructions for a set number of cycles. Instructions such as floating point divide, integer divide, integer multiply, and floating point square root block anywhere from 5-70 cycles. Some instructions, such as floating point divide, only block other floating point operations from dispatching for some number of cycles. This blocking property limits the amount of instruction level parallelism that Swing Modulo Scheduling can take advantage of.

The Ultra SPARC IIIi has 6 parallel execution pipelines, referred to here as resources:

- 2 integer Arithmetic and Logic Units (ALU) pipelines: Handles all integer addition, subtraction, logic operations, and shifts. Each operation takes 1 cycle.
- 1 Branch pipeline: Handles all branch instructions, resolving one branch per cycle.
- 1 Load/Store pipeline: Handles load and store instructions. Loads take between 2 and 3 cycles (on an L1 hit) and stores utilize a store buffer and have an effective latency of 0 cycles. Additionally this pipeline handles integer multiplication and division. Integer multiplication has a latency of 6 to 9 cycle (depending upon value of the operands), and integer division has a latency of 40 to 70 cycles.
- 1 Floating-Point Multiply pipeline: Handles floating point multiplication in single or double precision. It has 4 cycles of latency, but is fully pipelined. Floating point division is iterative (not pipelined) and can take between 17 (single precision) and 20 (double precision) cycles.
- 1 Floating-Point Addition pipeline: Handles all floating point addition operations. It is fully pipelined and has 4 cycles of latency.

The mix of instructions that can be dispatched is controlled by the resources used by each instruction. For example, 2 SHIFTs and a floating point ADD can be issued together because there are 2 ALU units and 1 floating point addition pipeline, but 2 SHIFTs and 1 integer ADD instruction can not be issued together because this group needs 3 ALUs when only 2 are available.

Instruction grouping rules restrict which instructions may be issued together. Grouping rules are used so that instructions are maintained in execution order, each pipeline only runs a subset of instructions, and because some multi-cycle instructions require helpers (NOPs) to maintain the pipelines. Instructions, depending upon their type, may abide by one of the following grouping rules:

- Break Group After: The instruction must be the last in the group.
- Break Group Before: The instruction must be the first in the group.
- Single Issue Group: The instruction must be issued by itself.

It is important to abide by an instruction's grouping rules when constructing the schedule otherwise instructions will be unnecessarily stalled.

As discussed in Section 5.1, the SPARC V9 architecture does not have alternative instructions for those that cause exceptions, or support for speculative execution. The SPARC V9 architecture does have an instruction which can fetch up to 2KB of data which can potentially reduce load misses in the cache. This does not help with speculative execution of code for Global Swing Modulo Scheduling, but it potentially reduces the number of load misses in the cache. However in our implementation, the scheduling description uses a higher latency for loads, which places the load at a higher priority in the scheduler. This latency padding increases the changes that the load will complete (if missed in the cache or not) before instructions use the result,reducing the amount of time the processor is stalled.

6.2 Swing Modulo Scheduling Results

This section presents an evaluation of our implementation of the Swing Modulo Scheduling algorithm for the SPARC V9 architecture.

6.2.1 Methodology and Benchmarks

Swing Modulo Scheduling was implemented as a static optimization in the SPARC V9 backend of the LLVM Compiler Infrastructure. Each benchmark is compiled using llvmgcc, all the normal LLVM optimizations are run, instructions are selected, local scheduling occurs, and SMS is applied. After SMS, register allocation is done and SPARC V9 assembly code is generated. GCC 3.4.2 is used to assemble and link the executable. Each executable is run three times, and the minimum time (user+system) is used as the final execution time.

All benchmarks were compiled and executed on an 1GHz Ultra SPARC IIIi processor system. For this work, we selected benchmarks with a large number of single basic block loops from the SPECINT 2000, SPECFP 2000, SPECINT 95, and SPECFP 95 benchmarks, the PtrDist suite [5], the FreeBench suite [35], and the Prolangs-C suite. Additionally, some miscellaneous programs such as fpgrowth [19], sgefa, make_dparser, hexxagon, optimizer-eval, and bigfib were included. Benchmarks elided from these results were excluded because of a lack of single basic blocks loops or minor bugs in the implementation.

6.2.2 Loop Statistics

Swing Modulo Scheduling (SMS) transforms single basic block (SBB) loops without control flow into a prologue, kernel, and epilogue. Table 6.1 shows the loop statistics for a variety of benchmarks. The columns of the table are as follows:

- Program: Name of the benchmark.
- LOC: Number of lines of code in the benchmark.
- SBB Loops: Total number of single basic block loops.
- Calls: Number of SBB loops that have calls.
- Cond Mov: Number of SBB loops that have conditional move instructions. In the SPARC V9 backend conditional move instructions violate SSA, and thus can not be handled by our SMS implementation.
- Large: Number of SBB loops with more than 100 instructions.
- Invalid: Number of SBB loops that are invalid for reasons such as the loop's trip count is not loop invariant.
- Valid: The number of SBB loops that can be scheduled by SMS.
- Percentage: The percentage of SBB loops that are valid.

Program	LOC	SBB Loops	Calls	Cond Mov	Large	Invalid	Valid	Percentage
175.vpr	17728	122	36	2	0	9	57	46.72
197.parser	11391	283	102	6	3	65	42	14.84
171.swim	435	16	2	0	8	0	6	37.5
172.mgrid	489	29	7	0	4	2	16	55.17
168.wupwise	2184	138	9	0	0	0	129	93.48
130.li	7598	79	46	0	0	1	3	3.80
102.swim	429	15	1	0	8	0	6	40.0
101.tomcatv	190	7	1	0	2	0	4	57.14
107.mgrid	484	27	6	0	4	2	15	55.55
104.hydro2d	4292	68	2	0	10	0	56	82.35
fpgrowth	634	19	5	0	0	2	11	57.89
sgefa	1220	19	5	0	0	2	11	57.89
make_dparser	19114	78	24	0	0	4	10	12.82
hexxagon	1867	95	3	0	0	9	13	13.68
optimizer-eval	1641	60	17	0	0	11	26	43.33
anagram	650	6	2	0	0	1	3	50.0
bc	7297	97	13	13	1	2	14	14.43
ft	1803	7	2	0	0	1	4	57.14
ks	782	13	3	0	0	6	4	30.76
pcompress2	903	22	2	0	0	0	13	59.09
analyzer	923	11	3	0	0	0	6	54.54
neural	785	9	0	4	0	0	5	66.67
agrep	3968	175	21	3	0	18	80	45.71
football	2258	29	6	0	1	0	9	31.03
simulator	4476	41	0	0	0	0	25	60.98
toast	6031	33	1	0	0	1	13	39.39
mpeg2decode	9832	56	24	4	1	0	23	41.07
bigfib	311	96	1	0	0	8	14	14.58

Table 6.1: Loop Statistics for the Benchmarks

The loop statistics give an idea of the opportunities that are available for the SMS transformation. The number of single basic blocks in the benchmarks range from 175 in **agrep** to 7 in **101.tomcatv** and **ft**. Table 6.1 shows that a large number of loops are rejected due to function calls. In **197.parser** and **130.li** over half of the loops are rejected because they contain a call. Unfortunately, handling calls is not something any Modulo Scheduling algorithm supports and is not a problem unique to Swing Modulo Scheduling.

There are very few loops that contain conditional moves. The most are found in **bc** and **104.hydro2d**. The reason loops with conditional moves can not be handled is because a value is redefined in the SPARC V9 backend and SSA is violated. Our implementation assumes that the basic blocks are in SSA form, and it is not a limitation of SMS that loops with this property can not be scheduled. With enough time, the correct changes to our implementation of the SMS algorithm and the LLVM SPARC V9 backend could be made to allow loops with conditional moves to be processed.

The Large column denotes loops that contain over 100 instructions. Because Modulo Scheduling can significantly increase register pressure, loops with many instructions have a higher potential for a large number of live values regardless of the heuristics used to schedule instructions. Swing Modulo Scheduling, as discussed in Chapter 3 performs well at keeping register pressure low, but it can not prevent spills from happening. In many production compilers, if Modulo Scheduling generates spills, the original loop is used instead of the modulo scheduled loop. Because SMS is run before register allocation and no live variable analysis is available, there is no way to predict or know when spills have been generated until after the SMS pass has completed. At that point, it is very difficult to undo Modulo Scheduling. Based upon our experiments, we have found that loops with greater than 100 instructions are extremely likely to generate spills and degrade performance substantially. Therefore, our implementation rejects any loop that has greater than 100 instructions. This actually occurs infrequently in practice as shown in Table 6.1. The most large loops rejected is 10 from **104.hydro2d**, **171.swim** and **102.swim** are close behind with 8. Most benchmarks reject at most one large loop.

The Invalid column represents loops that are rejected for other reasons, but by far the most common is that the loop's trip count is not loop invariant. This means that the number of times the loop executes is dependent upon some value that is being computed in the loop. Swing Modulo Scheduling, as well as all Modulo Scheduling algorithms, must know the number of times the loop iterates, or must prove at compile time that the loop iterates based upon some value computed before the loop is entered. Because instructions are reordered with the kernel, and can exist from previous iterations, the number of times the loop executes must not be dependent upon some value one of those instructions computes.

Table 6.1 shows that most benchmarks have loops in the correct form, while programs such as **197.parser**, **hexxagon**, **optimizer-eval**, and **agrep** have a large number of loops that are not in the correct form. Most likely these loops are *while* loops in the original program.

The last two columns in Table 6.1 show the number of single basic block loops that are valid (no calls, conditional moves, are small, and have a loop-invariant induction variable) and the percentage of loops that are valid. This ranges anywhere from 93.48% for **168.wupwise** to 3.8% for **130.li**. A larger number of valid loops does not guarantee that a benchmark will have increased performance.

6.2.3 Compile Time

Swing Modulo Scheduling differentiates itself from other Modulo Scheduling techniques because it never backtracks when scheduling instructions. Because of this, Swing Modulo Scheduling is very efficient in the amount of time it takes to compute a schedule.

Table 6.2 shows the breakdown of compile time for Swing Modulo Scheduling for programs that range from 190-19114 lines of code, where each column is the following:

- Program: Name of benchmark.
- Valid: The number of SBB valid loops that are available to be modulo scheduled.
- MII: Time to calculate the RecMII and ResMII values for all SBB loops.
- NodeAttr: Time to compute all the node attributes for all SBB loops.
- Order: Time to order the nodes for all SBB loops.
- Sched: Time to compute the schedule and kernel for all SBB loops.
- Recon: Time to construct the loop into a prologue, epilogue, kernel, and stitch it back into the original program for all SBB loops.
- SMS: Total time for the Swing Modulo Scheduling algorithm to process all SBB loops.
- Total: Total time to compile the benchmark.
- Ratio: Ratio of the total compile time spent on Modulo Scheduling all the SBB loops to the total compile time.

On average, Swing Modulo Scheduling has a very low compile time percentage of 1% for most programs, with all but one under 14%. There is one program that drastically increased compile time. **175.vpr** has a 37% compile time percentage, and most of that is from the time to calculate the RecMII and ResMII. This increase in time is primarily due to the circuit finding algorithm. Figure 6.1 shows the break down of the compile times of the phases of SMS as a bar chart.

Program	Valid	MII	NodeAttr	Order	Sched	Recon	SMS	Total	Ratio
175.vpr	57	94.5799	0.0600	7.2100	0.2200	0.0300	103.6000	278.7899	0.37
197.parser	42	0.0898	0.0100	0.0299	0.0399	0.0099	0.3199	90.3099	0.00
171.swim	6	0.0500	0.0299	0.0398	0.0999	0.0400	0.2999	8.2200	0.04
172.mgrid	16	0.1700	0.0399	0.0399	0.2599	0.0399	0.5799	11.4699	0.05
168.wupwise	129	2.9199	0.4599	0.6099	4.9300	0.1099	9.5400	66.3700	0.14
130.li	3	0.0099	0.0000	0.0000	0.0000	0.0000	0.1099	48.5400	0.00
102.swim	6	0.0699	0.0500	0.0500	0.2099	0.0299	0.4299	9.3500	0.05
101.tomcatv	4	0.0298	0.0300	0.0300	0.0400	0.0199	0.1500	3.5800	0.04
107.mgrid	15	0.1100	0.0399	0.0498	0.1799	0.0300	0.4700	11.1300	0.04
104.hydro2d	56	1.4199	0.1899	0.2799	0.5700	0.1000	2.6899	44.3499	0.06
fpgrowth	11	0.3199	0.0000	0.0700	0.0199	0.0000	0.4199	4.3800	0.10
sgefa	11	0.0099	0.0000	0.0000	0.0399	0.0300	0.1100	5.7500	0.02
make_dparser	10	0.0499	0.0100	0.0100	0.0000	0.0000	0.1699	119.8199	0.00
hexxagon	13	0.0300	0.0100	0.0100	0.0500	0.0099	0.1900	81.1600	0.00
optimizer-eval	26	0.0299	0.0099	0.0099	0.0199	0.0100	0.1599	10.0900	0.02
anagram	3	0.0199	0.0000	0.0000	0.0299	0.0000	0.0599	3.0299	0.02
bc	14	0.0399	0.0100	0.0100	0.0400	0.0300	0.1599	68.5599	0.00
ft	4	0.0000	0.0000	0.0099	0.0100	0.0000	0.0199	3.3000	0.01
ks	4	0.0099	0.0000	0.0000	0.0199	0.0000	0.0399	5.3400	0.01
pcompress2	13	0.0399	0.0099	0.0198	0.0599	0.0100	0.1699	4.9700	0.03
analyzer	6	0.0299	0.0100	0.0100	0.0099	0.0000	0.0600	4.0300	0.01
neural	5	0.0200	0.0099	0.0099	0.0200	0.0000	0.0600	3.2899	0.02
agrep	80	0.1998	0.0199	0.0199	0.1899	0.0800	0.7000	48.5600	0.01
football	9	0.0799	0.0300	0.0499	0.0500	0.0199	0.2399	47.7299	0.01
simulator	25	0.0798	0.0100	0.0199	0.0199	0.0400	0.2000	30.8499	0.01
toast	13	0.0499	0.0099	0.0099	0.0499	0.0700	0.2199	47.1699	0.00
mpeg2decode	23	0.0899	0.0000	0.0200	0.0600	0.0300	0.2499	53.4700	0.00
bigfib	14	0.0499	0.0100	0.0100	0.2100	0.0000	0.3599	78.7400	0.00

Table 6.2: Compile Time Breakdown for Benchmarks

For graphs with strongly connect components (SCCs) with many edges (almost N^2 edges), the exponential nature of the circuit finding algorithm explodes. A solution is to use the SCC as the recurrence in the graph for SCCs with an excessive number of edges. This does not impact the correctness of SMS, but may not estimate the minimal RecMII. If an SCC is used, it represents all the recurrences within it. The RecMII for this SCC is calculated by dividing the total latency of all the nodes in the SCC by the sum of all the dependence distances.

Our experiments have shown that SCCs with more than 100 edges cause the circuit finding algorithm to exhibit exponential behavior. Therefore, for our experiments SCCs were used instead of finding all recurrences when the number of edges exceeded 100. However, **175.vpr** may need to have a lower threshold and a more sophisticated heuristic is needed.

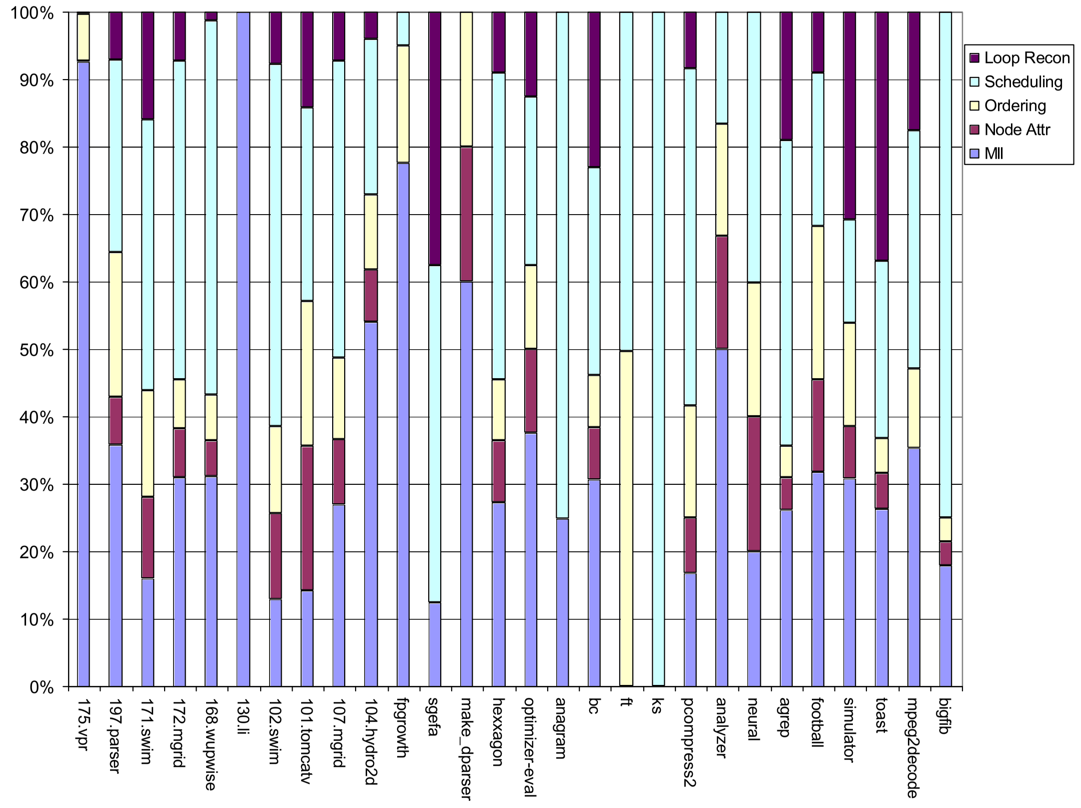


Figure 6.1: Compile Times for the Phases of SMS

6.2.4 Static Measurements

Traditionally, Modulo Scheduling algorithms have been evaluated on how close to theoretical Initiation Interval (MII) the actual schedule achieves. Minimum II (MII) is the maximum of the resource and recurrence II values. In theory, this value represents the maximum number of cycles need to schedule all the instructions.

Program	Valid-Loops	Sched-Loops	Stage0	RecCon	ResCon	MII-Sum	II-Sum	II-Ratio
175.vpr	57	16	7	0	57	166	171	0.97
197.parser	42	22	18	0	42	218	218	1.00
171.swim	6	6	1	0	6	193	193	1.00
172.mgrid	16	13	3	0	16	189	253	0.75
168.wupwise	129	25	0	0	129	570	570	1.00
130.li	3	2	1	0	3	21	21	1.00
102.swim	6	6	1	0	6	334	334	1.00
101.tomcatv	4	4	0	0	4	107	107	1.00
107.mgrid	15	12	3	0	15	177	218	0.81
104.hydro2d	56	46	10	0	56	788	815	0.97
fpgrowth	11	1	1	0	10	8	8	1.00
sgefa	11	10	0	0	11	67	104	0.64
make_dparser	10	3	0	0	10	35	35	1.00
hexxagon	13	5	2	0	13	41	41	1.00
optimizer-eval	26	2	0	0	26	16	16	1.00
anagram	3	2	1	0	3	20	20	1.00
bc	14	7	0	0	14	205	205	1.00
ft	4	1	1	0	4	9	9	1.00
ks	4	3	2	0	4	21	22	0.95
pcompress2	13	9	4	0	13	120	120	1.00
analyzer	6	3	1	0	6	48	48	1.00
neural	5	2	0	0	5	15	19	0.95
agrep	80	49	18	0	80	577	577	1.00
football	9	4	2	0	9	70	70	1.00
simulator	25	24	19	0	25	290	290	1.00
toast	13	11	0	0	13	170	172	0.99
mpeg2decode	23	13	1	0	23	139	139	1.00
bigfib	14	3	1	0	14	27	27	1.00

Table 6.3: Static Measurements for the Benchmarks

Table 6.3 shows the static measurements for the loops that were scheduled. Each column lists the following:

- Program: Benchmark name.
- Valid-Loops: The number of valid single basic block loops.
- Sched-Loops: The number of loops successfully scheduled.
- Stage0: The number of loops whose schedule does not overlap iterations.

- RecCon: The number of loops whose MII is constrained by recurrences.
- ResCon: The number of loops whose MII is constrained by resources.
- MII-Sum: The sum of MII for all loops.
- II-Sum: The sum of achieved II for all loops.
- II-Ratio: The ratio of actual II to theoretical II .

The Sched-Loops column shows the number of loops that were successfully scheduled. This means that Swing Modulo Scheduling was able to compute a schedule without any resource or recurrence conflicts which a length (in cycles) less than the total latency of all instructions in the loop. It is possible for a loop to not be scheduled if there are not enough machine resources to be able to schedule instructions without stalling the processor. Once Swing Modulo Scheduling reaches the maximum allowable value for II, the algorithm gives up and the loop is left unchanged.

Swing Modulo Scheduling attempts to overlap iterations of the loop. However, if there are insufficient resources or too many loop-carried dependencies, it may be impossible for any ILP to be exposed. Therefore, schedules may be obtained that have a maximum stage of 0, which means that no iterations were overlapped. While this situation is not ideal, the schedule may still be optimal and be executed in less cycles than the original schedule. This is because instructions are scheduled to avoid processor stalls caused by resource or dependence conflicts and are scheduled close to the predecessors and successors reducing register pressure. Our experiments actually show register spills to decrease for some benchmarks and is discussed in the next section.

The RecCon and ResCon columns show how many loops are constrained by resources and which are constrained by dependences. From our description of the SPARC V9 architecture in Section 6.1 it is no surprise that all the benchmarks are constrained by resources. This is because the SPARC has very strict grouping rules and blocking properties for many of its instructions. This means that while on some architectures Swing Modulo Scheduling can execute other instructions while long latency instructions (such as integer divide) are executing, the SPARC blocks all instructions from issuing. This reduces the possibility of using ILP. There are very few instructions with long latency that do not block instructions. This is the primary reason that the benchmark's MII is constrained by resources.

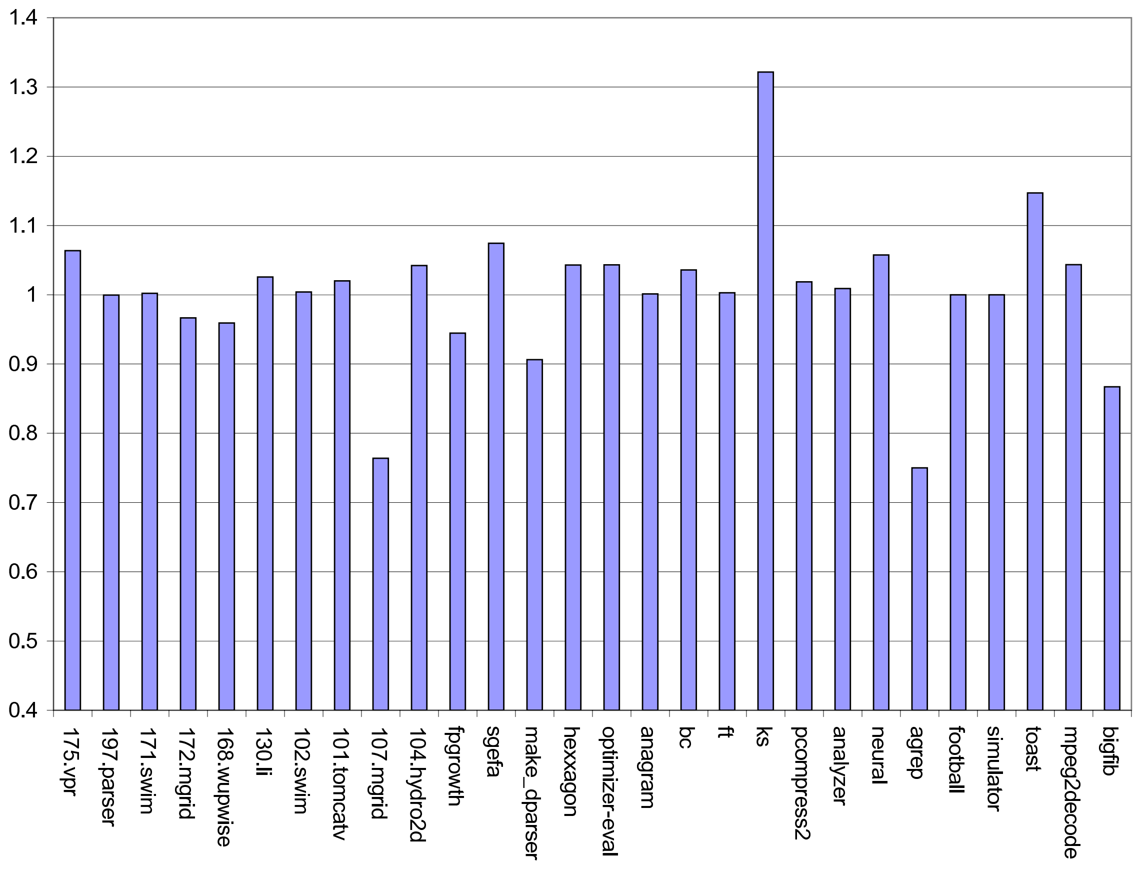


Figure 6.2: Theoretical II to Achieved II Ratio

The last three columns in Table 6.3 list the theoretical II and actual II values achieved, and their ratio. The goal of Swing Modulo Scheduling is to achieve an optimal schedule with an II as close to MII as possible.

Figure 6.2 charts the II ratio for each benchmark. Almost all benchmarks have a ratio of 1.0, meaning theoretical II was achieved. Achieving theoretical II indicates that SMS found the most optimal schedule possible given resource and recurrence constraints. For **172.mgrid**, **sgefa**, and **neural** the actual II achieved is 25-55% higher. This can be attributed to two sources. The first is that the ordering of the nodes may not be optimal. The order of the instructions affects the window of time that the instruction may be scheduled. Recall in Section 4.6 that instructions scan the schedule from some starting cycle to some end cycle which is determined by what nodes are already in the partial schedule. The second thing that can affect the actual II is how RecMII is calculated. In the previous section we mentioned that in some cases where the circuit finding algorithm takes exponential time, so the SCC is used to approximate RecMII. This approximate may be drastically different than what the scheduler can actually achieve.

Overall, SMS performs very well at finding optimal schedules for almost all benchmarks.

6.2.5 Performance Results

Performance gains for Swing Modulo Scheduled programs are very dependent upon the architecture targeted and the dynamic loop iteration counts. There can be a significant performance cost for restructuring the loop into a prologue (ramp up), kernel (steady state), and epilogue (ramp down). For loops that do not execute for a long period of time, this cost can actually reduce performance. Additionally, the architecture has a key role in how much ILP can be achieved as found in the previous section. Increasing register spills can also reduce performance.

Table 6.4 shows some statistics for the performance results of our benchmarks, with the columns as follows:

- Program: Name of benchmark.
- LOC: Lines of code in the original benchmark.
- Sched-Loops: Number of scheduled loops.

Program	LOC	Sched-Loops	Spills	MS Time	No-MS Time	Runtime Ratio
175.vpr	17728	16	-27	38.57	36.26	1.06
197.parser	11391	22	-48	24.80	24.81	1.00
171.swim	435	6	62	60.63	60.50	1.00
172.mgrid	489	13	50	136.61	141.33	0.97
168.wupwise	2184	25	35	141.66	147.69	0.96
130.li	7598	2	1	0.40	0.39	1.03
102.swim	429	6	100	2.48	2.47	1.00
101.tomcatv	190	4	12	24.94	24.45	1.02
107.mgrid	484	12	67	30.38	39.77	0.76
104.hydro2d	4292	46	31	31.14	29.88	1.04
fpgrowth	634	1	-7	67.25	71.19	0.95
sgefa	1220	10	10	37.39	34.80	1.07
make_dparser	19114	3	-38	0.29	0.32	0.91
hexxagon	1867	5	-2	65.95	63.23	1.04
optimizer-eval	1641	2	-7	104.35	100.03	1.04
anagram	650	2	0	8.06	8.05	1.00
bc	7297	7	-7	6.64	6.41	1.04
ft	1803	1	0	3.50	3.49	1.00
ks	782	3	0	13.97	10.57	1.32
pcompress2	903	9	12	1.09	1.07	1.02
analyzer	923	3	9	1.13	1.12	1.01
neural	785	2	-1	2.76	2.61	1.06
agrep	3968	49	159	0.03	0.04	0.75
football	2258	4	-35	0.01	0.01	1.00
simulator	4476	24	8	0.01	0.01	1.00
toast	6031	11	141	0.39	0.34	1.15
mpeg2decode	9832	13	48	0.24	0.23	1.04
bigfib	311	3	110	7.44	8.58	0.87

Table 6.4: Performance Results for the Benchmarks

- Spills: The net gain/loss of register spills after Modulo Scheduling.
- MS Time: Execution time of the program transformed by SMS.
- No-MS Time: Execution time of the program without SMS.
- Runtime Ratio: The runtime ratio of the benchmark.

Because Swing Modulo Scheduling aims to reduces register pressure, on many cases the number of register spills is actually decreased by performing SMS. For example,**175.vpr, 197.parser, fpgrowth, make_dparser, hexxagon, optimizer-eval, bc, neural**, and **football** have up to 48 spills eliminated. However, SMS can also increase register pressure, which is demonstrated by the fair number of benchmarks that have more register spills. This increase in register spills can slow down the program.

Figure 6.3 charts the runtime ratio results of Swing Modulo Scheduling. Overall, most bench-

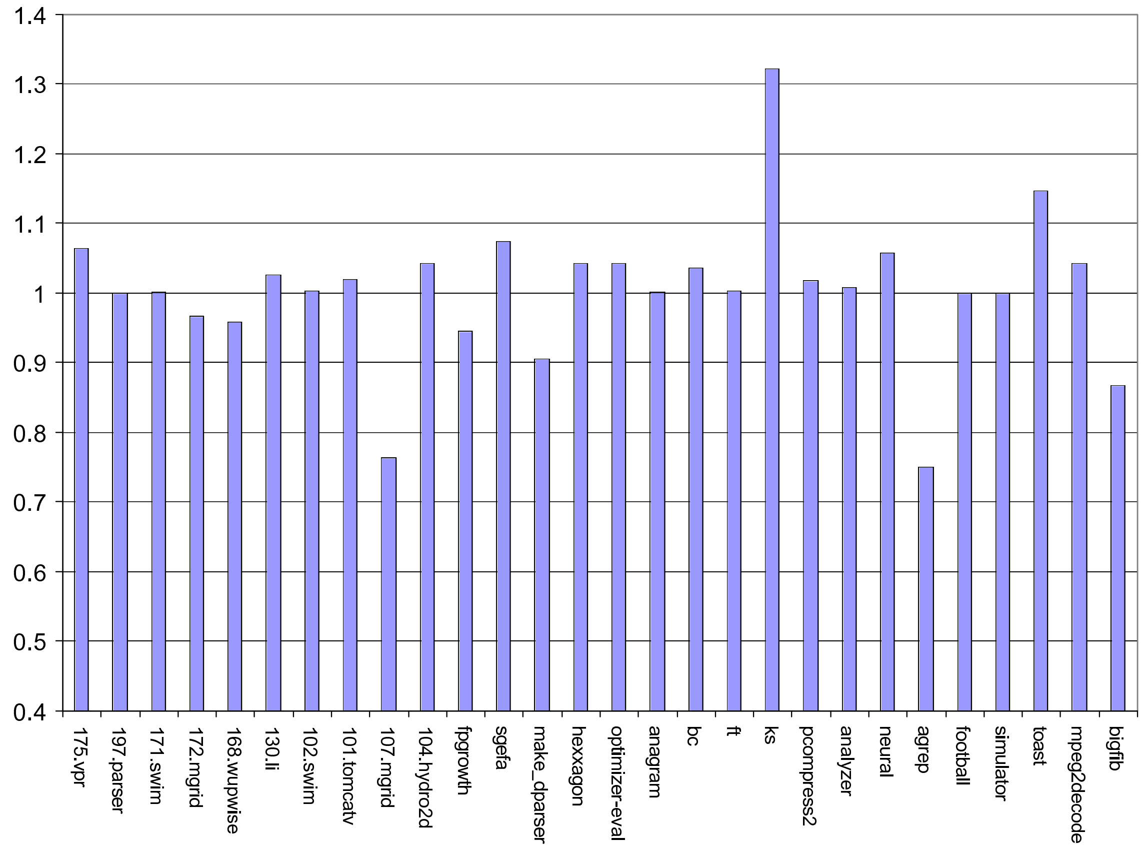


Figure 6.3: Runtime Ratio Results

marks have no significant performance gains or losses from Modulo Scheduling. Most likely, this is because there was virtually no ILP available due to resource constraints imposed by the architecture (no benchmarks were constrained by recurrences). Another issue is that while some loops may be sped up by SMS, other shorter loops are offsetting the speedup by their performance cost from the prologue and epilogues. Ideally, heuristics, such as iteration count threshold, need to be used to determine which loops to Modulo Schedule and which to not. This can be easily done for loops whose iteration counts are known at compile time, but we did not experiment with this heuristic in our implementation. Overall, we feel that the architecture's resource constraints played the biggest role in the lack of performance gains.

There are a couple of benchmarks that show significant performance gains (10-33%), **107.mgrid, bigfib, and make_dparser**, while a few others, **172.mgrid, 168.wupwise, fpgrowth**, show small speedups (1-9%). **agrep** runs for such a short amount of time its execution time is likely to be noise. **107.mgrid** speeds up by 33%, despite increasing the number of register spills by 50 and schedules a modest number of loops. Its speedup can be attributed to long running loops that overcome the startup cost of all the modulo scheduled loops. In addition, several of the loops scheduled in 107.mgrid have floating point instructions that do not block other instructions from issuing. These are 4-7 cycle latency instructions which provide a small amount of potential ILP.

While the performance results are not uniformly positive, it does give some indication that given a different architecture, Swing Modulo Scheduling would have more freedom to schedule instructions, overlap latencies, and ultimately decrease execution times.

6.3 Superblock Swing Modulo Scheduling Results

This section presents an evaluation of our implementation of Swing Modulo Scheduling for Superblocks for the SPARC V9 architecture.

6.3.1 Methodology and Benchmarks

The extensions to the Swing Modulo Scheduling algorithm for superblocks was implemented as a static optimization in the SPARC V9 backend of the LLVM Compiler Infrastructure. Each benchmark is compiled with llvmgcc, all the normal LLVM optimizations are applied, instructions are selected, local scheduling is performed, and finally SMS for superblocks is applied. Superblocks are found using the LLVM Loop Analysis, where each loop is checked to see if it meets the criteria of a superblock (single entry, multiple exit). After SMS for superblocks has completed, registers are allocated, and SPARC V9 assembly code is generated. GCC 3.4.2 is used to assembly and link the executable. For the final execution time, the executable is run three times and the minimum (user + system) is used as the result.

All benchmarks were compiled and executed on a 1 GHz Ultra SPARC IIIi processor system. Benchmarks with superblock loops were selected from the PtrDist suite [5], the MediaBench suite, MallocBench suite, the Prolangs-C suite, the Prolangs-C++ suite, and the Shootout-C++ suite. Additionally, some miscellaneous programs such as make_dparser, hexxagon, and spiff were included. Benchmarks elided from these results were primarily because of a lack of superblock loops and a few due to bugs in the implementation.

6.3.2 Superblock Statistics

Swing Modulo Scheduling for superblocks transforms superblock loops without control flow into a prologue, kernel, epilogue, side epilogues, and side exits. Table 6.5 shows the superblock loop statistics for the benchmarks selected. The columns of the table are as follows:

- Program: Name of the benchmark.
- LOC: Number of lines of code in the benchmark[1].
- Loops: Total number of loops.
- SB: Total number of superblock loops.
- Calls: Number of superblock loops that have calls.
- CondMov: Number of superblock loops that have conditional move instructions. In the SPARC V9 backend conditional move instructions violate SSA, and thus can not be handled by our extended SMS implementation.

[1]The number of lines of code do not include libraries. Therefore, it is not unusual to see a large number of loops for a small programs because they are found in the libraries.

Program	LOC	Loops	SB	Calls	CondMov	Invalid	Valid	Percentage
make_dparser	19111	206	17	4	1	1	11	64.71
hexxagon	1867	146	9	4	0	1	4	44.44
spiff	5441	185	4	1	0	0	3	75.00
anagram	650	10	2	1	0	0	1	50.00
bc	7297	77	3	1	0	0	2	66.67
gs	23423	177	8	4	0	0	4	50.00
timberwolfmc	24951	428	4	1	0	0	3	75.00
assembler	3177	67	2	0	0	0	2	100.00
unix-tbl	2829	50	4	0	1	1	2	50.00
city	923	124	4	0	0	1	3	75.00
deriv1	195	119	4	0	0	1	3	75.00
employ	1024	119	4	0	0	1	3	75.00
office	215	117	4	0	0	1	3	75.00
shapes	245	120	4	0	0	1	3	75.00
encode	1578	10	2	0	0	0	2	100.00
ackermann	17	121	4	0	0	1	3	75.00
ary	23	121	4	0	0	1	3	75.00
ary2	43	121	4	0	0	1	3	75.00
ary3	26	123	4	0	0	1	3	75.00
except	69	118	4	0	0	1	3	75.00
fibo	22	118	7	3	0	1	3	42.85
hash	36	123	5	1	0	1	3	60.00
hash2	35	125	5	1	0	1	3	60.00
heapsort	72	117	4	0	0	1	3	75.00
hello	12	117	4	0	0	1	3	75.00
lists	58	116	4	0	0	1	3	75.00
lists1	80	132	8	0	0	1	7	87.50
matrix	66	124	4	0	0	1	3	75.00
methcall	65	119	4	0	0	1	3	75.00
moments	86	9	1	0	0	0	1	100.00
nestedloop	24	116	4	0	0	1	3	75.00
objinst	67	119	4	0	0	1	3	75.00
random	33	116	4	0	0	1	3	75.00
reversefile	26	122	4	0	0	1	3	75.00
sieve	44	119	4	0	0	1	3	75.00
spellcheck	52	133	10	6	0	1	3	30.00
strcat	29	113	4	0	0	1	3	75.00
sumcol	25	116	4	0	0	1	3	75.00
wc	40	118	4	0	0	1	3	75.00
wordfreq	98	24	3	2	0	0	1	33.33

Table 6.5: Superblock Loop Statistics for the Benchmarks

- Invalid: Number of superblock loops that are invalid for reasons such as the loop's trip count is not loop invariant.
- Valid: The number of superblock loops that can be scheduled by our extended SMS algorithm.
- Percentage: The percentage of superblock loops that are valid.

The superblock loop statistics represent the opportunities that are available for our extended SMS algorithm. The number of superblock loops in the benchmarks range from 17 in **make_dparser** to 1 in **moments**. The reason these numbers are low is because superblocks are not commonly found in typical programs and more likely formed by using profile information. However, the goal of this thesis is to prove the potential of the extensions, despite the reduced number of superblocks.

Table 6.5 shows that the majority of the superblock loops are rejected because they contain calls. **spellcheck** is at the top with 6 superblock loops rejected, followed closely by **make_dparser**, **hexxagon**, and **gs** with 4, **fibo** with 3, **wordfreq** with 2, and a few other benchmarks with 1 superblock loop rejected. As mentioned previously, all Modulo Scheduling algorithms (including Swing Modulo Scheduling) can not handle loops with calls, and our extensions to SMS do not change that restriction.

There are very few superblock loops that contain conditional moves. The most are 1 in **make_dparser** and **unix-tbl**. The reason that superblock loops with conditional moves can not be transformed is because SSA is violated by the SPARC V9 backend and our implementation assumes that the loop is in SSA form. The invalid column represents the number of superblock loops that were rejected for other reasons such as the loop's trip count is not loop invariant. This was explained in the previous section (Section 6.2.2). Several benchmarks have superblock loops rejected for this reason.

The last two columns in Table 6.5 show the number of superblock loops that are valid (no calls, no conditional moves, and have loop-invariant induction variable) and the percentage of loops that are valid. This ranges anywhere from 100% (such as **assembler**, **encode**, and **moments**) to 30% (**spellcheck**).

6.3.3 Compile Time

As mentioned previously, Swing Modulo Scheduling is more efficient than many Modulo Scheduling algorithms because it never backtracks when scheduling instructions. Our extensions to SMS for superblocks does not change this property, and only adds slightly more complexity to the reconstruction phase to handle side exits.

Program	Valid	MII	NodeAttr	Order	Sched	Recon	SMS Total	Total	Ratio
make_dparser	11	0.0199	0.0100	0.0000	0.0100	0.0300	0.1100	120.7599	0.0009
hexxagon	4	0.0000	0.0000	0.0099	0.0200	0.0099	0.0700	81.0399	0.0009
spiff	3	0.0100	0.0000	0.0000	0.0199	0.0000	0.0499	33.6899	0.0015
anagram	1	0.0100	0.0000	0.0000	0.0200	0.0000	0.04	2.9299	0.0133
bc	2	0.0000	0.0000	0.0000	0.0000	0.0099	0.0299	67.8800	0.0004
gs	8	0.0798	0.0000	0.0100	0.0300	0.0100	0.11	120.8199	0.0018
timberwolfmc	3	0.0000	0.0000	0.0000	0.0299	0.0000	0.0299	282.4299	0.0004
assembler	2	0.0000	0.0000	0.0000	0.0000	0.0099	0.0499	14.4400	0.0020
unix-tbl	2	0.0099	0.0000	0.0000	0.0099	0.0000	0.0399	54.6600	0.0009
city	3	0.0000	0.0000	0.0000	0.0000	0.0300	0.07	77.7499	0.0005
deriv1	3	0.0099	0.0000	0.0000	0.0000	0.0200	0.05	71.0300	0.0010
employ	3	0.0200	0.0000	0.0000	0.0000	0.0000	0.0399	65.8900	0.0006
office	3	0.0000	0.0000	0.0100	0.0199	0.0000	0.0199	70.1399	0.0007
shapes	3	0.0099	0.0000	0.0000	0.0099	0.0000	0.7999	62.6199	0.0006
encode	2	0.0000	0.0000	0.0000	0.0000	0.0099	0.0199	4.2699	0.0047
ackermann	3	0.7399	0.0000	0.0198	0.0000	0.0000	0.7999	71.9099	0.0111
ary	3	0.0100	0.0000	0.0000	0.0000	0.0000	0.0700	70.8699	0.0010
ary2	3	0.0000	0.0000	0.0000	0.0199	0.0000	0.0499	70.5399	0.0007
ary3	3	0.0064	0.0007	0.0015	0.0047	0.0072	0.0700	71.3900	0.0010
except	3	0.0099	0.0000	0.0099	0.0000	0.0099	0.0799	71.1899	0.0011
fibo	3	0.0000	0.0099	0.0000	0.0100	0.0000	0.0700	69.5499	0.0010
hash	3	0.0000	0.0000	0.0099	0.0099	0.0000	0.0499	71.0399	0.0007
hash2	3	0.0199	0.0000	0.0000	0.0000	0.0100	0.0900	72.6099	0.0012
heapsort	3	0.0000	0.0000	0.0000	0.0099	0.0000	0.0400	68.5900	0.0006
hello	3	0.0199	0.0000	0.0100	0.0000	0.0000	0.0700	69.7400	0.0010
lists	3	0.0099	0.0000	0.0000	0.0099	0.0000	0.0600	70.8599	0.0008
lists1	7	0.0299	0.0000	0.0000	0.0000	0.0000	0.0900	73.7599	0.0012
matrix	3	0.0000	0.0000	0.0100	0.0099	0.0000	0.0599	71.4600	0.0008
methcall	3	0.0099	0.0000	0.0000	0.0099	0.0099	0.0499	70.1699	0.0007
moments	1	0.0000	0.0000	0.0000	0.0099	0.0000	0.0199	4.5699	0.0044
nestedloop	3	0.0000	0.0000	0.0000	0.0000	0.0200	0.0600	70.2000	0.0009
objinst	3	0.0099	0.0000	0.0000	0.0000	0.0099	0.0399	70.5699	0.0006
random	3	0.0000	0.0000	0.0000	0.0099	0.0099	0.0499	69.9299	0.0007
reversefile	3	0.0199	0.0000	0.0000	0.0000	0.0099	0.0500	72.5100	0.0007
sieve	3	0.0099	0.0000	0.0000	0.0000	0.0099	0.0599	71.1399	0.0008
spellcheck	3	0.3500	0.0099	0.0000	0.0000	0.0099	0.4000	70.4400	0.0057
strcat	3	0.0000	0.0000	0.0000	0.0000	0.0099	0.0799	68.9199	0.0012
sumcol	3	0.0199	0.0000	0.0000	0.0000	0.0000	0.0600	72.0800	0.0008
wc	3	0.0099	0.0000	0.0000	0.0000	0.0099	0.0699	71.0900	0.0010
wordfreq	1	0.0099	0.0000	0.0000	0.0000	0.0000	0.0099	6.2700	0.0016

Table 6.6: Compile Time Breakdown for Benchmarks

Table 6.6 shows the breakdown of compile time for our extended Swing Modulo Scheduling algorithm for superblocks on benchmarks that range from 12-24951 lines of code, where each column is the following:

- Program: Name of benchmark.
- Valid Loops: The number of valid superblock loops that are available to be modulo scheduled.
- MII: Time to calculate the RecMII and ResMII values for all superblock loops.
- NodeAttr: Time to compute all the node attributes for all superblock loops.
- Order: Time to order the nodes for all superblock loops.
- Sched: Time to compute the schedule and kernel for all superblock loops.
- Recon: Time to construct the loop into a prologue, epilogue, kernel, side epilogues, side exits, and stitch it back into the original program for all superblock loops.
- SMS Total: Total time for the Swing Modulo Scheduling algorithm to process all superblock loops.
- Total: Total time to compile the benchmark.
- Ratio: Ratio of the total compile time spent on Modulo Scheduling all the superblock loops to the total compile time.

Overall, our extended Swing Modulo Scheduling algorithm has a very low compile time of less than 1% of the total compile time and some are faster than we can measure due to the resolution of our timer. Even **make_dparser** which has the most superblock loops with 11 has a very low compile time percentage of 0.01%. While the number of superblock loops per benchmark is small, these numbers provide a solid basis for the conclusion that our extensions to the Swing Modulo Scheduling algorithm for superblocks has not drastically increased the time complexity.

Figure 6.4 shows the break down of compile times as a bar chart. This chart illustrates that there is not one phase that is dominant in regards to the greatest compile time. It varies depending upon the benchmark. For example, **wordfreq** and **ary** spend almost all their time calculating MII, while **encode** and **objinst** spend all their time reconstructing the loop. Without benchmarks

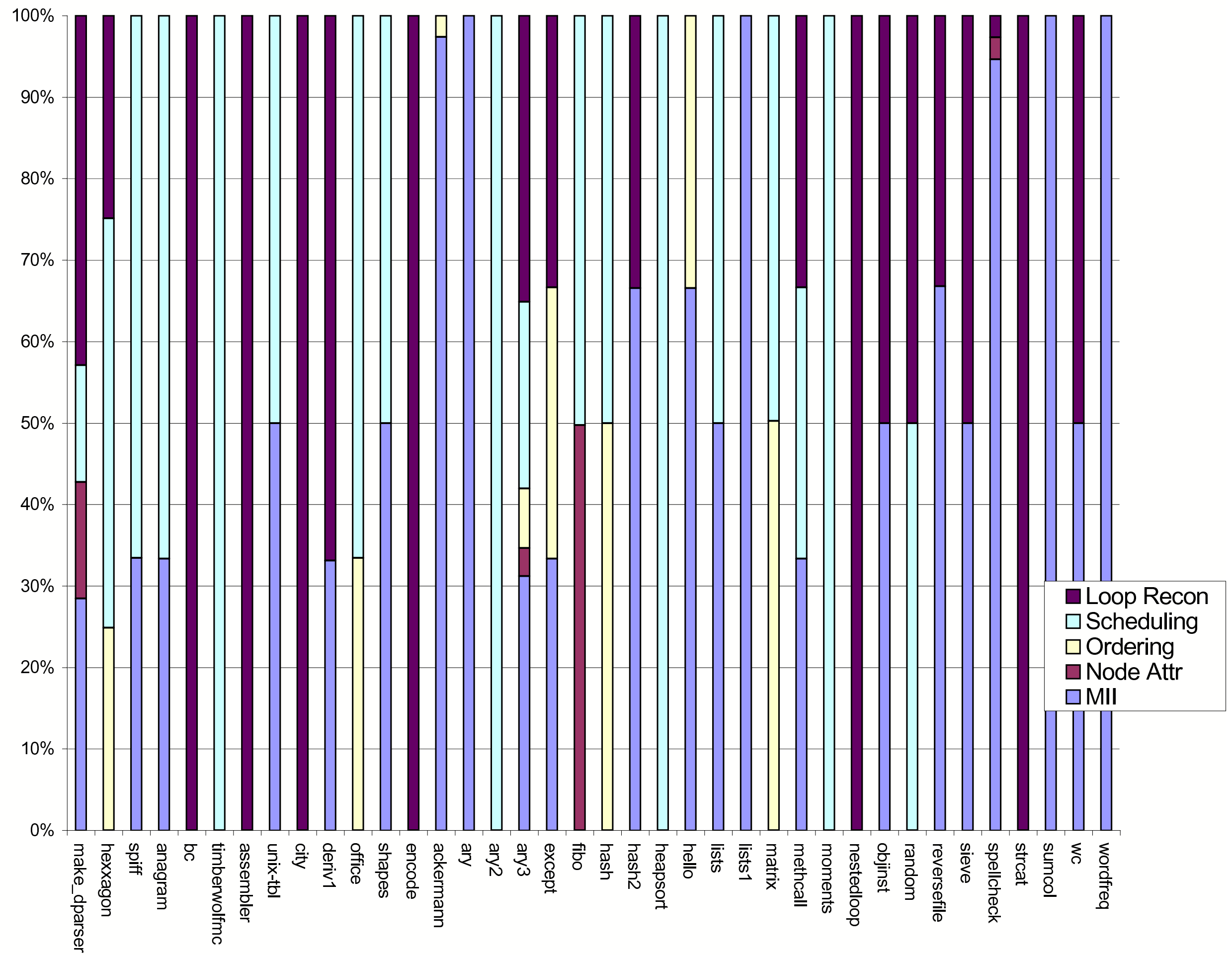


Figure 6.4: Compile Times for the Phases of Extended SMS

with a large number of superblocks, its not clear which phase dominates the others. However, we can speculate that given our extensions to the Swing Modulo Scheduling algorithm, the majority of compile time will still be spent calculating MII (due to the circuit finding algorithm and the increase in dependences).

6.3.4 Static Measurement Results

Modulo Scheduling algorithms traditionally are evaluated on how close the schedule is to achieving the theoretical Initiation Interval (MII). Minimum II (MII) is the maximum of the resource and recurrence II values. This value represents the theoretical maximum number of cycles needed to complete one iteration of the loop.

Table 6.7 shows the static measurements for the superblock loops that were scheduled. Each column is the following:

- Program: Benchmark name.
- Valid: The number of valid superblock loops.
- Sched: The number of superblock loops successfully scheduled.
- Stage0: The number of superblock loops whose schedule does not overlap iterations.
- RecCon: The number of superblock loops whose MII is constrained by recurrences.
- ResCon: The number of superblock loops whose MII is constrained by resources.
- MII-Sum: The sum of MII for all superblock loops.
- II-Sum: The sum of achieved II for all superblock loops.
- II-Ratio: The ratio of actual II to theoretical II

The number of superblock loops successfully scheduled is denoted by the Sched-Loops column. This means that the loops were scheduled without any resource or recurrence conflicts with a length (in cycles) less than the total latency of all instructions in the loop. Just as the original SMS algorithm, it is possible for no schedule without conflicts to be found. This means that no

Program	Valid	Sched	Stage0	RecCon	ResCon	MII-Sum	II-Sum	II-Ratio
make_dparser	11	9	3	0	9	69	69	1.00
hexxagon	4	4	1	0	4	30	30	1.00
spiff	3	3	0	0	3	25	25	1.00
anagram	1	1	0	0	1	26	26	1.00
bc	2	2	2	0	2	12	12	1.00
gs	8	5	5	0	8	54	54	1.00
timberwolfmc	3	1	1	0	3	7	7	1.00
assembler	2	2	0	0	2	12	12	1.00
unix-tbl	2	2	1	0	2	36	36	1.00
city	3	3	1	0	3	21	21	1.00
deriv1	3	3	1	0	3	21	21	1.00
employ	3	3	1	0	3	21	21	1.00
office	3	3	1	0	3	21	21	1.00
shapes	3	3	1	0	3	21	21	1.00
encode	2	2	2	0	2	22	22	1.00
ackermann	3	3	1	0	3	21	21	1.00
ary	3	3	1	0	3	21	21	1.00
ary2	3	3	1	0	3	21	21	1.00
ary3	3	3	1	0	3	21	21	1.00
except	3	3	1	0	3	21	21	1.00
fibo	3	3	1	0	3	21	21	1.00
hash	3	3	1	0	3	21	21	1.00
hash2	3	3	1	0	3	21	21	1.00
heapsort	3	3	1	0	3	21	21	1.00
hello	3	3	1	0	3	21	21	1.00
lists	3	3	1	0	3	21	21	1.00
lists1	7	5	1	0	5	30	30	1.00
matrix	3	3	1	0	3	21	21	1.00
methcall	3	3	1	0	3	21	21	1.00
moments	1	1	1	0	1	6	6	1.00
nestedloop	3	3	1	0	3	21	21	1.00
random	3	3	1	0	3	21	21	1.00
reversefile	3	3	1	0	3	21	21	1.00
sieve	3	3	1	0	3	21	21	1.00
spellcheck	3	3	1	0	3	21	21	1.00
strcat	3	3	1	0	3	21	21	1.00
sumcol	3	3	1	0	3	21	21	1.00
wc	3	3	1	0	3	21	21	1.00
wordfreq	1	1	1	0	1	6	6	1.00

Table 6.7: Static Measurements for the Benchmarks

schedule could be created that did not stall the processor. Our extensions to the SMS algorithm gives up on scheduling once the length of the proposed schedule is longer than the total latency of all instructions in the loop.

Insufficient resources or a large number of loop-carried dependences can make it impossible to overlap iterations of the loop. Therefore, schedules may be created that have a maximum stage of zero which means that no iterations were overlapped. While this situation is not ideal, the schedule may still be optimal and execute in less cycles than the original schedule. In particular, because we are moving instructions above or below conditional branches, this may have a positive effect on overall performance despite not exploiting ILP. Additionally, reordering instructions may also reduce register pressure, and our experiments have shown the number of register spills do decrease after extended SMS was applied.

The RecCon and ResCon columns show how many superblock loops were constrained by recurrences and resources. Similar to the original SMS algorithm, all our benchmarks are constrained by resources. This is due to the SPARC V9 architecture's strict grouping rules and blocking properties of many instructions. Because there are very few instructions with long latencies that do not block other instructions from executing, the amount of ILP is extremely limited. This is the primary reason that all the benchmarks are constrained by resources.

Surprisingly all the benchmarks selected achieve theoretical II. This means that given the MII value, a schedule of length MII was successfully created. This can be attributed to the original SMS algorithm's consistency of achieving MII, and the relatively low number of superblocks found. As the number of superblocks increase, we should see more variation in the achieved II value.

6.3.5 Performance Results

The performance gains for our extended Swing Modulo Scheduling algorithm are directly tied to the architecture targeted and the dynamic loop iteration counts. Restructuring the superblock loop into a prologue (ramp up), kernel (steady state), and epilogue (ramp down) can have a significant performance cost if the loop does not execute for a long period of time. In addition, the architecture directly controls how much ILP can be achieved. Because the SPARC V9 IIIi architecture does not support speculative execution (Section 5.2), and most long latency instruction block all others from executing, the architecture is limiting what instructions can be moved above a conditional branch, and overlapped with other instructions. Increasing register spills can also reduce performance.

Program	LOC	Sched-Loops	Spills	MS Time	No-MS Time	Runtime Ratio
make_dparser	19111	9	112	0.31	0.32	0.97
hexxagon	1867	4	-24	67.96	72.68	0.94
spiff	5441	3	9	0.01	0.01	1.00
anagram	650	1	0	8.17	8.13	1.00
bc	7297	2	-77	6.48	6.34	1.02
gs	23423	5	27	0.01	0.01	1.00
timberwolfmc	24951	1	164	0.01	0.01	1.00
assembler	3177	2	19	0.01	0.01	1.00
unix-tbl	2829	2	4	0.01	0.01	1.00
city	923	3	-17	0.10	0.09	1.11
deriv1	195	3	-9	0.01	0.00	1.00
employ	1024	3	-63	0.01	0.00	1.00
office	215	3	17	0.01	0.02	0.50
shapes	245	3	63	0.02	0.02	1.00
encode	1578	2	-2	0.59	0.58	1.02
ackermann	17	3	23	326.83	325.06	1.01
ary	23	3	92	0.73	0.73	1.00
ary2	43	3	-64	0.68	0.69	0.99
ary3	26	3	20	67.06	66.52	1.01
except	69	3	61	0.07	0.08	0.88
fibo	22	3	-37	15.45	15.46	1.00
hash	36	3	52	4.05	4.00	1.01
hash2	35	3	30	29.15	31.48	0.93
heapsort	72	3	19	19.84	20.16	0.99
hello	12	3	31	0.01	0.01	1.00
lists	58	3	-38	27.43	27.28	1.01
lists1	80	5	-23	1.89	1.93	0.98
matrix	66	3	56	69.27	66.73	1.04
methcall	65	3	-14	85.96	87.96	0.98
moments	86	1	0	1.51	1.51	1.00
nestedloop	24	3	-42	42.31	51.76	0.82
objinst	67	3	65	41.62	41.91	0.99
random	33	3	-4	35.65	36.00	0.99
reversefile	26	3	40	0.01	0.01	1.00
sieve	44	3	-63	24.28	24.54	0.99
spellcheck	52	3	-184	0.01	0.01	1.00
strcat	29	3	-143	1.76	1.70	1.04
sumcol	25	3	-59	0.01	0.01	1.00
wc	40	3	18	0.01	0.01	1.00
wordfreq	98	1	-11	0.01	0.01	1.00

Table 6.8: Performance Results for the Benchmarks

Table 6.8 shows the performance statistics for our selected benchmarks, where the columns are as follows:

- Program: Name of benchmark.
- LOC: Lines of code in the original benchmark.

- Sched-Loops: Number of scheduled superblock loops.
- Spills: The net gain/loss of static register spills after our extended SMS.
- MS Time: Execution time of the program transformed by our extended SMS.
- No-MS Time: Execution time of the program without our extended SMS.
- Runtime Ratio: The runtime ratio of the benchmark.

Our extensions to the Swing Modulo Scheduling algorithm for superblock loops do not alter the original algorithm's goal of keeping register pressure low. In many cases, the number of register spills actually decreased by performing SMS on the superblock loops. For example, **spellcheck** has a reduction of 184 spills, **strct** removes 143 spills, and **bc** eliminates 77 spills. However, in many cases the number of register spills are increased. **timberwolfmc** adds 164 spills, **make_dparser** increases spills by 112, and **ary** adds 92 spills. Despite the increase in spills, many of the benchmarks still see performance gains.

Figure 6.5 charts the runtime ratio results of our extensions to the Swing Modulo Scheduling algorithm[2]. Overall, most benchmarks have no significant performance gains or losses from Modulo Scheduling the superblock loops. This is most likely because of increased register spills offsetting the optimal schedule and because there was not much ILP due to resource constraints imposed by the architecture (no benchmarks were constrained by recurrences).

There are a few benchmarks that show significant performance gains (7-22%), **hexxagon**, **hash2**, **except**, **nestedloop**, while many others, **make_dparser**, **ary2**, **heapsort**, **lists1**, **methcall**, **objinst**, **random**, and **sieve** show smaller speedups (1-3%). Please note that the benchmarks that have a 2 times speedup are attributed to noise and were only included in this chart for consistency reasons.

nestedloop has the largest speedup of 22% which can attributed to a reduction in 42 spills, and successfully Modulo Scheduling 3 superblock loops (all but one with overlapped iterations). Following close behind with a 14% speedup is **except** which Modulo Schedules 3 superblock loops, and actually increases register spills by 61. The speedup is attributed to the long loop iterations of

[2]Runtime ratio was compared to the original program execution time without the original SMS transformation.

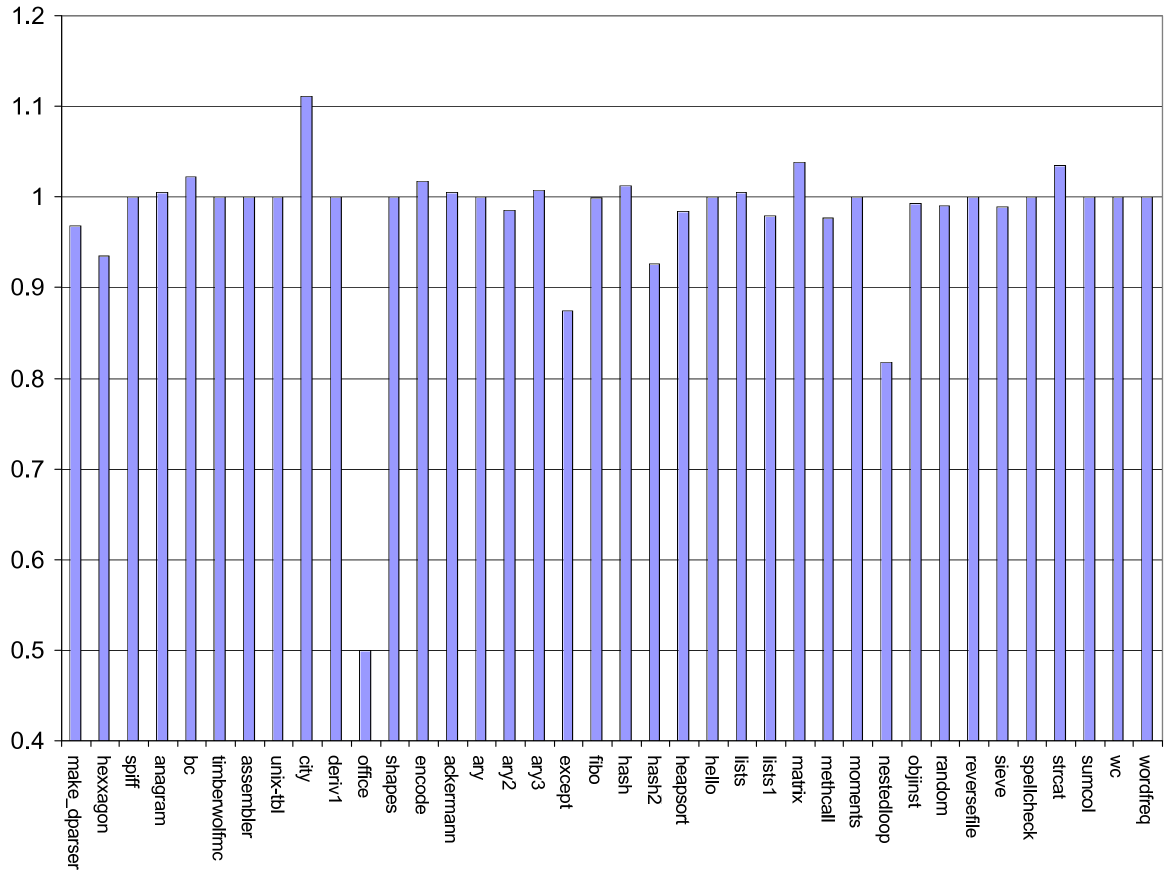


Figure 6.5: Runtime Ratio Results

the superblocks that were Modulo Scheduled and took advantage of ILP (2 out of the 3). For all the benchmarks that had a 1% speedup, this is attributed to superblock loops found in the C++ library that are being Modulo Scheduled and have overlapped iterations.

Given an architecture with speculative execution and with less strict grouping and blocking properties, these performance gains from applying Swing Modulo Scheduling for superblock loops can only get better.

Chapter 7

Conclusion

This thesis describes our implementation of the Swing Modulo Scheduling algorithm and our extensions to support superblock loops. We have described in detail the steps of each algorithm, and provided results to support our claim that SMS and our extended SMS algorithm is effective in terms of code generation and efficient in regards to compile time. For the original algorithm, benchmarks were transformed to have performance gains of 10-33%, and SMS only adds 1% of compile time on average. The extended SMS algorithm increased benchmark performance from 7-22% and adds relatively little time to the total compile time (0.01% on average).

While we have accomplished our research goals, there are still many areas left to investigate. We would like to implement Swing Modulo Scheduling for other architectures, in particular IA64. IA64 is an ideal platform because it has a large number of registers (reducing the worry of increasing register spills), many long latency instructions (that do not block other instructions from dispatching), and has speculative execution (for our extended SMS algorithm to reduce the number of added dependencies). Using this type of architecture would allow SMS to have more freedom to exploit ILP and most likely achieve better performance gains.

We also would like to explore using our extended Swing Modulo Scheduling algorithm for superblock loops as a runtime or profile guided optimization. Ideally, superblocks would be formed using dynamic feedback to find the most executed path in a loop. This would increase the number of superblocks available to be Modulo Scheduled and increase the probability of performance gains since the loop only contains the most frequently executed path. Additionally, we have proven that Swing Modulo Scheduling requires very little compile time on average, which makes it cost effective to be performed at runtime.

In conclusion, the Swing Modulo Scheduling algorithm has been shown to be a valuable optimization. We feel that its unique ordering technique, and sophisticated heuristics put it among the best of the Modulo Scheduling techniques.

References

[1] *UltraSPARC IIIi Processor User's Manual*. Sun Microsystems, June 2003.

[2] Alexander Aiken and Alexandru Nicolau. Optimal loop parallelization. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 308–317, New York, NY, USA, 1988. ACM Press.

[3] J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of control dependence to data dependence. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 177–189, New York, NY, USA, 1983. ACM Press.

[4] Erik R. Altman and Guang R. Gao. Optimal modulo scheduling through enumeration. *International Journal of Parallel Programming*, 26(3):313–344, 1998.

[5] Todd Austin, et al. The Pointer-intensive Benchmark Suite. `www.cs.wisc.edu/~austin/ptr-dist.html`, Sept 1995.

[6] Olaf Bachmann, Paul S. Wang, and Eugene V. Zima. Chains of recurrences-a method to expedite the evaluation of closed-form functions. In *ISSAC '94: Proceedings of the International Symposium on Symbolic and Algebraic Computation*, pages 242–249, New York, NY, USA, 1994. ACM Press.

[7] Johnnie L. Birch. Using the chains of recurrences algebra for data dependence testing and induction variable substitution. Master's thesis, Florida State University, 2002.

[8] Pohua P. Chang, Nancy J. Water, Scott A. Mahlke, William Y. Chen, and Wen mei W. Hwu. Three superblock scheduling models for superscalar and superpipelined processors. Technical report, University of Illinois at Urbana-Champaign, 1991.

[9] Alan Charlesworth. An approach to scientific array processing: The architectural design of the ap120b/fps-164 family. In *Computer*, volume 14, pages 18–27, 1981.

[10] Josep M. Codina, Josep Llosa, and Antonio González. A comparative study of modulo scheduling techniques. In *ICS '02: Proceedings of the 16th International Conference on Supercomputing*, pages 97–106, New York, NY, USA, 2002. ACM Press.

[11] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, pages 13(4):451–490, October 1991.

[12] Amod K. Dani. Register-sensitive software pipelining. In *IPPS '98: Proceedings of the 12th International Parallel Processing Symposium*, page 194, Washington, DC, USA, 1998. IEEE Computer Society.

[13] James C. Dehnert, Peter Y.-T. Hsu, and Joseph P. Bratt. Overlapped loop support in the cydra 5. In *ASPLOS-III: Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 26–38, New York, NY, USA, 1989. ACM Press.

[14] K. Ebcioğlu and Toshio Nakatani. A new compilation technique for parallelizing loops with unpredictable branches on a vliw architecture. In *Selected Papers of the Second Workshop on Languages and Compilers for Parallel Computing*, pages 213–229, London, UK, UK, 1990. Pitman Publishing.

[15] K. Ebcioğlu and Toshio Nakatani. A new compilation technique for parallelizing loops with unpredictable branches on a vliw architecture. In *Selected Papers of the Second Workshop on Languages and Compilers for Parallel Computing*, pages 213–229, London, UK, UK, 1990. Pitman Publishing.

[16] Kemal Ebcioğlu. A compilation technique for software pipelining of loops with conditional jumps. In *MICRO 20: Proceedings of the 20th Annual Workshop on Microprogramming*, pages 69–79, 1987.

[17] Alexandre E. Eichenberger and Edward S. Davidson. Stage scheduling: A technique to reduce the register requirements of a modulo schedule. In *MICRO 28: Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 338–349, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press.

[18] Mostafa Hagog. Swing modulo scheduling for gcc. In *Proceedings of the GCC Developer's Summit*, pages 55–65, Ottawa, Ontario, Canada, 2004. GCC.

[19] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. pages 1–12, 2000.

[20] Jerry Huck, Dale Morris, Jonathan Ross, Allan Knies, Hans Mulder, and Rumi Zahir. Introducing the ia-64 architecture. *IEEE Micro*, 20(5):12–23, 2000.

[21] Richard A. Huff. Lifetime-sensitive modulo scheduling. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pages 258–267, New York, NY, USA, 1993. ACM Press.

[22] Suneel Jain. Circular scheduling: A new technique to perform software pipelining. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 Conference on Programming language Design and Implementation*, pages 219–228, New York, NY, USA, 1991. ACM Press.

[23] Donald B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing*, 4(1):77–84, March 1975.

[24] Rakesh Krishnaiyer, Dattatraya Kulkarni, Daniel Lavery, Wei Li, Chu cheow Lim, John Ng, and David Sehr. An advanced optimizer for the ia-64 architecture. *IEEE Micro*, 20(6):60–68, 2000.

[25] Monica Lam. Software pipelining: An effective scheduling technique for vliw machines. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 318–328, New York, NY, USA, 1988. ACM Press.

[26] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program

Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[27] Josep Llosa. Swing modulo scheduling: A lifetime-sensitive approach. In *PACT '96: Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques*, page 80, Washington, DC, USA, 1996. IEEE Computer Society.

[28] Josep Llosa, Mateo Valero, Eduard Ayguade, and Antonio González. Hypernode reduction modulo scheduling. In *MICRO 28: Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 350–360, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press.

[29] Wen mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringman, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. The superblock: An effective technique for vliw and superscalar compilation. *Journal of Supercomputing*, pages 229–248, 1993.

[30] Soo-Mook Moon and Kemal Ebcioğlu. An efficient resource-constrained global scheduling technique for superscalar and vliw processors. In *MICRO 25: Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 55–71, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[31] S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1997.

[32] J. C. H. Park and M. Schlansker. On predicated execution. In *Tech. Rep. HPL-91-58.* Hewlett Packard Software Systems Laboratory, 1991.

[33] B. Ramakrishna Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *MICRO 27: Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 63–74, New York, NY, USA, 1994. ACM Press.

[34] B. Ramakrishna Rau, David W. L. Yen, Wei Yen, and Ross A. Towie. The cydra 5 departmental supercomputer: Design philosophies, decisions, and trade-offs. *Computer*, 22(1):12–26, 28–30, 32–35, 1989.

[35] Peter Rundberg and Fredrik Warg. The FreeBench v1.0 Benchmark Suite. `http://www.freebench.org`, Jan 2002.

[36] Y. N. Srikant and Priti Shankar, editors. *The Compiler Design Handbook: Optimizations and Machine Code Generation*. CRC Press, 2002.

[37] Eric Stotzer and Ernst Leiss. Modulo scheduling for the tms320c6x vliw dsp architecture. In *LCTES '99: Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 28–34, New York, NY, USA, 1999. ACM Press.

[38] Bogong Su and Jian Wang. Gurpr*: A new global software pipelining algorithm. In *MICRO 24: Proceedings of the 24th Annual International Symposium on Microarchitecture*, pages 212–216, New York, NY, USA, 1991. ACM Press.

[39] R. van Engelen. Symbolic evaluation of chains of recurrences for loop optimization, 2000.

[40] Robert van Engelen. Efficient symbolic analysis for optimizing compilers. In *CC '01: Proceedings of the 10th International Conference on Compiler Construction*, pages 118–132, London, UK, 2001. Springer-Verlag.

[41] Nancy J. Warter, Grant E. Haab, Krishna Subramanian, and John W. Bockhaus. Enhanced modulo scheduling for loops with conditional branches. In *MICRO 25: Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 170–179, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[42] Graham Wood. Global optimization of microprograms through modular control constructs. In *MICRO 12: Proceedings of the 12th Annual Workshop on Microprogramming*, pages 1–6, Piscataway, NJ, USA, 1979. IEEE Press.

[43] Eugene V. Zima. On computational properties of chains of recurrences. In *ISSAC '01: Proceedings of the 2001 International Symposium on Symbolic and Algebraic Computation*, page 345, New York, NY, USA, 2001. ACM Press.