Lifetime-Sensitive Modulo Scheduling

Richard A. Huff *

Department of Computer Science
Cornell University
Ithaca, NY 14853
(607) 254-8830
huff@cs.cornell.edu

Abstract

This paper shows how to software pipeline a loop for minimal register pressure without sacrificing the loop's minimum execution time. This novel *bidirectional slack-scheduling* method has been implemented in a FORTRAN compiler and tested on many scientific benchmarks. The empirical results—when measured against an absolute lower bound on execution time, and against a novel *schedule-independent* absolute lower bound on register pressure—indicate near-optimal performance.

1 Introduction

Software pipelining increases a loop's throughput by overlapping the loop's iterations; that is, by initiating successive iterations before prior iterations complete. With sufficient overlap, a functional unit can be saturated, at which point the loop initiates iterations at the maximum possible rate.

To find an overlapped schedule, a compiler must represent the complex resource constraints that can arise. Efficiently representing these constraints is especially difficult when adjacent iterations do not follow a common schedule but instead follow a pattern that repeats only after several iterations.

The problem is greatly simplified by restricting the search to a common schedule that initiates successive iterations at a constant rate. This restriction leads to the *modulo constraint*: no resource may be used more than once at the same time modulo the *initiation interval* of **II cycles**. The resulting resource constraints are easily represented in a modulo resource table with II entries: each entry keeps track of the various machine resources that can be reserved during a cycle. The compiler's primary task is to schedule the loop at a minimal II, thereby maximizing the loop's throughput. The overall discipline is called *modulo scheduling* [17]; for a detailed introduction, consult [14].

Prior scheduling research has focussed on achieving minimal execution time, without regard for whether the heuristics unnecessarily inflate register pressure.

- Ever since the studies of the late 1970's [10, 16], most people have advocated using list scheduling, in either a top-down or bottom-up fashion. Consequently, most schedulers consider each cycle in turn, packing it full of operations before considering the next cycle [7, 24, 12].
- For software pipelining loops with recurrences, where a cycle-by-cycle approach is inherently inadequate, compilers have considered each operation in turn, always placing an operation as early as possible in the partial schedule constructed thus far [9, 6].

Although the first approach can avoid excessive pressure at some cost in execution time [8, 3], neither approach schedules for minimal register pressure. In general, unidirectional strategies (top-down or bottom-up) unnecessarily stretch operand lifetimes; for example, by scheduling loads too early or stores too late.

This paper presents a novel *bidirectional* strategy that simultaneously schedules some operations late and other operations early. The resulting *slack-scheduling* framework modulo schedules a loop for minimal register pressure[1] with an increased likelihood of achieving a minimal II. This framework has been integrated into Cydrome's FORTRAN77 compiler [6] and tested on all eligible DO loops in the Lawrence Livermore Loops, the SPEC89 FORTRAN benchmarks, and the Perfect Club codes—a total of 1,525 loops. The scheduler's performance on these loops—when measured against an absolute lower bound on II, and against a novel *schedule-independent* absolute lower bound on register pressure—indicates near-optimal performance.

The next section describes the compiler's target machine. Section 3 defines the schedule-independent lower bounds

* Part of this research was performed as an employee of Hewlett-Packard Laboratories. The work at Cornell was supported by NSF Presidential Young Investigator award CCR-8958543, by NSF grant CCR-9008526, and by ONR grant N00014-93-1-0103.

[1] An infinite supply of registers is assumed, in order to measure register pressure rather than what can be done when spill code is allowed. Besides, no one as yet has a good strategy for spilling registers in a software pipeline.

Pipeline	No.	Operations	Latency
Memory Port	2	load	13
		store	1
Address ALU	2	addr add/sub/mult	1
Adder	1	int add/sub/logical	1
		float add/sub	1
Multiplier	1	int/float multiply	2
Divider	1	int/float div/mod	17
		float sqrt	21
Branch Unit	1	`brtop`	2

Table 1: Functional Unit Latencies

```
  subroutine sample(n, x, y)
  real*4 x(n), y(n)
  do 2 i=3,n
     x(i) = x(i-1) + y(i-2)
     y(i) = y(i-1) + x(i-2)
2 continue
  return
  end
```

Figure 1: A Sample Loop

on II and register pressure. Section 4 presents the slack-scheduling framework. Section 5 presents a bidirectional scheduling heuristic that adds lifetime sensitivity to the slack-scheduling framework. The scheduler's execution time is analyzed in Section 6. Performance measurements are shown in Section 7. Finally, Section 8 offers some comparisons with related work.

2 Target Machine

The target machine is a hypothetical VLIW processor similar to Cydrome's Cydra 5 [20, 2], including architectural support for overlapping loops without using code duplication [5]. Nevertheless, the scheduling techniques shown in this paper can be *directly* applied to conventional RISC machines [14, 23], albeit at the expense of code expansion [19].

2.1 Functional Units

Functional-unit latencies are given in Table 1. The compiler assumes the responsibility for honoring these latencies, scheduling no-ops wherever necessary. All functional units are fully pipelined; except for the divider, which is not pipelined at all. The `brtop` conditional branch conveniently combines several loop-management duties into one instruction [5].

The load latency of 13 cycles was chosen to represent the cost of bypassing a first-level cache and hitting a large off-chip second-level cache—a reasonable choice for a compiler that does not perform any loop-tiling transformations [25]. A memory latency register specifies the load latency that the compiler has chosen to schedule for. The hardware honors this load latency by freezing instruction issue whenever a load does not complete in time.

2.2 Predicated Execution

Every operation has a 1-bit predicate input. If the predicate is false, then the hardware treats the operation as a no-op, else it executes the operation as usual.

Predicated execution often eliminates the need for a conditional branch, as operations from both sides of the conditional may be issued—with the unwanted results being safely no-op'd. This optimization generalizes to if-conversion [1, 13], which transforms unstructured acyclic code into branch-free predicated code. Since modulo scheduling is applicable only to loops with branch-free bodies, if-conversion allows more loops to be modulo scheduled. Finally, predicated execution greatly simplifies code generation after modulo scheduling a loop; see [19] for details.

To address the problem of modulo scheduling loops-with-branches on machines without predicated execution, two extensions to modulo scheduling have been developed; namely, hierarchical reduction [9], and enhanced modulo scheduling [23]. In essence, each approach reduces the problem to scheduling branch-free loop bodies, at the cost of code expansion.

2.3 Rotating Register Files

When modulo scheduling a loop, it is quite common for an operation's result to be live for more than II cycles, thus preventing the operation from targeting the same register in adjacent iterations. For example, consider the loop shown in Figure 1. Each iteration generates a pair of values ($x(i)$ and $y(i)$) that are used two iterations later. Dataflow analysis can detect that the values may be passed from iteration to iteration within registers rather than main memory. Performing this load/store elimination [15, 6] leaves the resulting registers live for more than II cycles, as will be illustrated shortly.

The compiler must ensure that the successive outputs of an operation can be kept in distinct registers. In the absence of hardware support, the loop may be unrolled and the duplicate register specifiers renamed appropriately [9]. However, this *modulo variable expansion* technique can result in a large amount of code expansion [18].

A rotating register file can solve this problem without duplicating code. Consider saving the series of values generated by an operation in its own infinite pushdown stack. Old values can be read out of anywhere in the stack, and new values can be pushed on top, but a value cannot be modified once it has been pushed onto the stack; that is, the stack enforces a *dynamic single assignment* discipline [15]. Since each value pushed onto the stack has the same lifetime, only a constant portion of the stack is live at any given moment. In par-

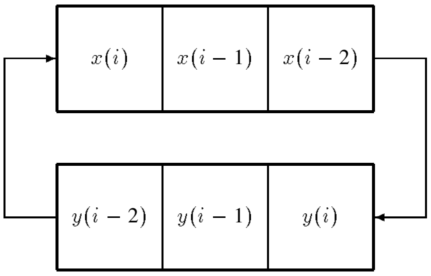

Figure 2: Concatenating Shifters

ticular, once the software pipeline reaches its steady state, each stack acts like a finite shifter that shifts its values once every II cycles. A rotating register file can be thought of as a concatenation of these shifters—end to end—into a finite circular queue, as illustrated in Figure 2.

The implementation of a rotating register file is quite simple. Consider a file of 2^n registers. The illusion of rotation is created by adding (modulo 2^n) each n-bit register specifier to a dedicated n-bit *iteration control pointer* (**ICP**). The ICP is decremented every II cycles by the loop's `brtop` conditional branch instruction. Thus rotation is simply a stylized mechanism for indirectly addressing a register file.

As an illustration of rotation at work, consider Figure 3, which shows a naive allocation of the values for $x(i)$ and $y(i)$ within a rotating file of six registers[2]. (Other loop variants have been omitted for clarity.) The values generated by the first iteration are shown in bold face. Lifetimes are outlined with ovals. When the figure says that $x(i)$ has a lifetime of [0,5), it means that the first iteration of the loop reserves a register for $x(3)$ at cycle 0, and that this register may not be overwritten until its last use at cycle 5. At each cycle in the space-time diagram, the ICP points to the register directly above the solid bar. At cycle 0, the rotating register file contains the initial live-in values for each recurrence.

Unlike the Cydra 5, the target machine has only three register files, two of which rotate: the **ICR** file contains rotating predicates, used for iteration control and if-converted code; the **RR** file contains rotating addresses, ints, and floats; and the **GPR** file contains loop-invariant addresses, ints, and floats. In order to measure register pressure, the compiler assumes that each register file is infinite.

3 Absolute Lower Bounds

3.1 Lower Bounds on II

A loop's minimum II is bounded below by two factors: resource contention and recurrence circuits.

Consider resource contention. If each iteration of the loop requires N units of a resource (e.g., integer adders), and the machine can supply at most R units of the resource per cycle, then II must be at least $\lceil N/R \rceil$. Therefore the resource with the maximum such ratio determines a lower bound on II—call it **ResMII**.

Consider recurrence circuits. A recurrence circuit from an operation to an instance of itself **Omega** (Ω) iterations later, must have a total latency L of no more than the $\Omega \times$ II cycles that separate the two operations. Hence a feasible schedule must have II $\geq \lceil L/\Omega \rceil$. Therefore the elementary recurrence circuit with the maximum such ratio determines a lower bound on II—call it **RecMII**.

To enable the scheduler to compute the Ω for a recurrence circuit, the dependence analyzer labels each dependence arc with its **omega** (ω), which is the minimum number of iterations that must separate the operations of the dependence. The compiler's front-end performs important code optimizations, such as load/store elimination, when a dependence's ω is the *exact* number of iterations spanned by the dependence[3]. Nevertheless, even conservative lower bounds on ω can lessen scheduling constraints significantly.

Although a graph can contain exponentially many elementary recurrence circuits, most loop bodies have very few. So the compiler computes RecMII by simply scanning each circuit [21]. In any case, RecMII can be computed in $O(V \times E \times \log V)$ time by indirectly finding a circuit with the minimum cost-to-time ratio, where a dependence arc is viewed as having a "cost" of $-$latency, and a "time" of ω [11]. If R is the minimum cost-to-time ratio, then RecMII $= \lceil -R \rceil$.

The ResMII and RecMII lower bounds are defined using the ceiling function because it does not make sense to talk about a non-integral II. However, if a compiler performs loop unrolling, then it can take advantage of fractional lower bounds. For instance, if a loop had an exact minimum II of $3/2$, then the compiler could unroll the loop once and attempt to schedule for an II of 3. Unfortunately, the current compiler does not perform any such loop transformations.

In practice, almost all loops can achieve their absolute lower bound of **MII** $= \max(\text{ResMII}, \text{RecMII})$. But for some loops, the minimum feasible II is more than MII.

3.2 Lower Bounds on Register Pressure

First a few miscellaneous details:

- Some machines need two registers to hold a 64-bit scalar; other machines need only one. To normalize the results in this paper, all programs were compiled so that a scalar used only one register.
- This paper concentrates on register pressure arising from loop variants. So "registers" will henceforth refer solely to the RR file.
- Operations that execute under mutually exclusive predicates may use the same destination register without interfering with each other. Unfortunately, the compiler does

[2] An optimal allocation would use only four rotating registers.

[3] The literature on vectorizing and parallelizing compilers would call an exact ω the dependence's *distance*.

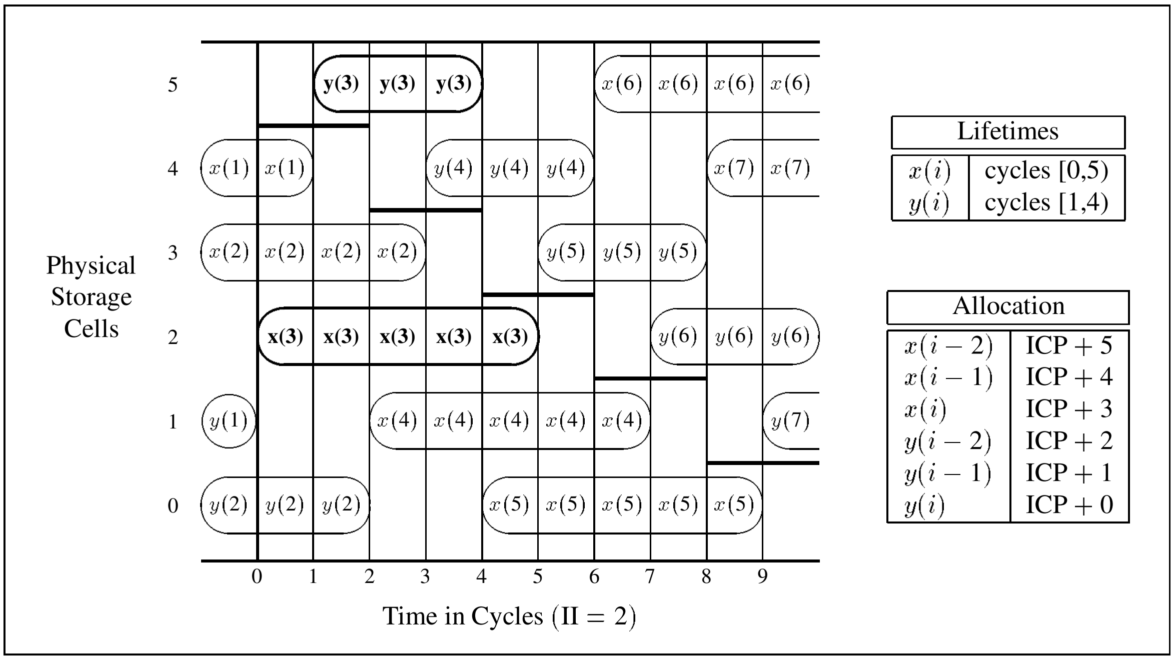

Figure 3: A Naive Allocation for the Sample Loop

Overlapping Lifetimes

			$y1$	$y2$	$y3$			
					y1	**y2**	**y3**	
$x0$	$x1$	$x2$	$x3$	$x4$				
		$x0$	$x1$	$x2$	$x3$	$x4$		
				x0	**x1**	**x2**	**x3**	**x4**
Time Modulo II:				0	1	2	3	...

LiveVector $= \langle 4, 4\rangle$

Figure 4: Computing the Sample LiveVector

not perform the requisite analysis. Therefore the compiler allocates registers, and computes lower bounds, as if all predicates may be true.

Once a loop has been scheduled, an absolute lower bound on the schedule's register pressure can be found by computing the maximum number of values that are live at any cycle of the schedule. Figure 4 illustrates this computation for the sample loop of Figure 1. (Values for address arithmetic have been omitted for clarity.) The lifetimes generated by one iteration are shown in bold face. Since $II = 2$, lifetimes from adjacent iterations overlap each other as shown in the diagram. Summing the number of live values in each of the II columns gives a loop's *LiveVector*. Due to the modulo constraint, a faster algorithm is possible: simply wrap the lifetimes generated by the first iteration around a vector of length II. In any case, the LiveVector's maximum, **MaxLive**, is the desired lower bound.

Allocating registers for a modulo-scheduled loop is beyond the scope of this paper. For an extensive discussion of the problem, including heuristic solutions and empirical results, consult [18]. One of the most remarkable results reported in that paper is the ability of their allocation strategies to almost always achieve the MaxLive lower bound on a schedule's register pressure[4]. Due to that result, this paper approximates a schedule's **register pressure** with its MaxLive lower bound.

Three observations lead to a novel *schedule-independent* lower bound on a loop's final register pressure:

1. MaxLive is usually very close to the LiveVector's average, *AvgLive*, especially when a long schedule is wrapped around a small II[5].

2. AvgLive can be expressed as the total length of all lifetimes divided by II.

3. Given a fixed II, a schedule-independent lower bound on the length of a value's lifetime, *MinLT*, can be calculated. (Section 5.1 shows how.)

Therefore **MinAvg** $= \lceil \sum_v \text{MinLT}(v)/\text{II} \rceil$ is a useful lower bound. In particular, MaxLive - MinAvg provides an absolute measure of how well the scheduler minimizes register pressure.

[4] In particular, the wands-only strategy using end-fit with adjacency ordering never needed more than MaxLive$+$ 1 registers, and the blades-allocation strategy using best-fit with adjacency and start-time orderings never needed more than MaxLive $+$ 5 registers. The paper did not explicitly state these facts. Instead, I had the privilege of directly examining the paper's raw data.

[5] Over 97% of the time, MaxLive is no more than AvgLive $+$ 5.

4 Slack Scheduling

Slack scheduling focuses on each operation's scheduling freedom—or *slack*, which is precisely defined in the next subsection, after a little motivation.

To schedule straight-line code, (instead of a loop), a list-scheduling compiler can simply consider each cycle in turn, packing it full of operations before considering the next cycle. After all, an operation that cannot fit in the current cycle can always find room in a later cycle.

To software pipeline a loop, a scheduler must handle cyclic data dependencies, which arise from the loop's *non-trivial* recurrence circuits. A trivial recurrence circuit, which is a dependence arc from an operation to itself, imposes no scheduling constraints, as the compiler previously ensured that II $\geq$ RecMII. Henceforth, "recurrence circuit" implicitly signifies "non-trivial", unless explicitly stated otherwise.

As mentioned in Section 3.1, a scheduler must not stretch a recurrence circuit's length beyond $\Omega \times$ II cycles. In addition, placing an operation at a cycle t commits resources for cycles $t + k \times$ II, for all k. So an operation that cannot fit in one cycle might not fit in any later cycles. Therefore a list-scheduling compiler is not likely to find a feasible schedule at MII when recurrence circuits are present.

Slack scheduling solves this problem by integrating recurrence constraints and critical-path considerations into an operation-driven framework with limited backtracking.

4.1 Computing Estart and Lstart Bounds

When an operation is placed into a partial schedule, it will in general have an earliest start time (**Estart**) and a latest start time (**Lstart**), due to predecessors and successors that have already been placed. For operations on recurrence circuits, these constraints cannot be avoided. The difference between these bounds is the operation's **Slack**.

To ensure that Estart and Lstart are well defined for all operations, the compiler adds two pseudo-operations to the loop body: **Start** and **Stop**. Start is a predecessor of each operation; Stop is a successor of each operation. Start is fixed at cycle 0; Stop is scheduled just like any other operation.

The scheduler maintains Estart and Lstart bounds with the aid of a *minimum distance* relation. For each pair of operations x and y,

> **MinDist**(x, y) is the minimum number of cycles (possibly negative) by which x must precede y in any feasible schedule, or $-\infty$ if there is no path in the dependency graph from x to y.

Computing MinDist is an all-pairs longest-paths problem. Simply assign each dependence arc a cost of latency $- \omega \times$ II. Since II $\geq$ RecMII, all cycles have non-positive costs. Hence the problem can be reduced to a fast all-pairs shortest-paths calculation by negating each arc's weight. Finally, set $\text{MinDist}(x, x) = 0$ for each operation x. Although MinDist must be recomputed for each attempted II, the overhead is reasonable since most loops achieve MII.

Given the MinDist relation, the scheduler initializes

$$\begin{aligned} \text{Estart}(x) &= \text{MinDist}(\text{Start}, x), \text{ and} \\ \text{Lstart}(x) &= \text{Lstart}(\text{Stop}) - \text{MinDist}(x, \text{Stop}), \end{aligned}$$

where $\text{Lstart}(\text{Stop})$ is judiciously set to equal or exceed $\text{Estart}(\text{Stop})$. When an operation y is placed at a time t, the scheduler updates the bounds to:

$$\begin{aligned} \text{Estart}(x) &= \max\left(\text{Estart}(x), t + \text{MinDist}(y, x)\right), \text{ and} \\ \text{Lstart}(x) &= \min\left(\text{Lstart}(x), t - \text{MinDist}(x, y)\right). \end{aligned}$$

4.2 An Operation-Driven Framework

Once a compiler can compute both Estart and Lstart bounds, it can add critical-path considerations by controlling $\text{Lstart}(\text{Stop})$. If a loop has no resource contention ($\text{ResMII} = 1$), then it can always be scheduled to meet its critical path. Otherwise the scheduler sets

$$\text{Lstart}(\text{Stop}) = \lceil \text{Estart}(\text{Stop}) / \text{II} \rceil \times \text{II}.$$

This provision of extra slack lessens the overall amount of backtracking and improves the final schedule. Once set, $\text{Lstart}(\text{Stop})$ is reset only when $\text{Estart}(\text{Stop})$ is pushed out beyond either $\text{Lstart}(\text{Stop})$ or Stop's current placement in the schedule.

The scheduler places operations one by one until either a feasible schedule is found or the heuristics give up. The central loop comprises the following 6 steps:

1. Choose a good operation to place into the current partial schedule. (Section 4.3)
2. Search for a good issue cycle within the operation's Estart and Lstart bounds. (Section 5.2)
3. If no conflict-free issue cycle exists, then create room for the operation by *ejecting* one or more operations from the schedule. (Section 4.4)
4. Place the operation, and update the modulo resource table.
5. Update the Estart and Lstart bounds for all unplaced operations to reflect the new partial schedule. (Section 4.1)
6. If operations are ejected too many times, then remove all operations from the schedule, increment II, and start all over again.

In practice, almost all loops succeed at MII. Even so, in Step 6 the compiler increments II by $\max\left(\lfloor 0.04 \times \text{II} \rfloor, 1\right)$, rather than by 1, in order to avoid spending an excessive amount of time compiling large complex loops[6].

[6]Incrementing II by 1 lowered the total II by 45 at the expense of 29% more time spent in the scheduler.

4.3 Choosing a Good Operation to Place

Slack scheduling is characterized by its attempt to always choose an operation with the minimum number of *issue slots* available to it. An issue slot for an operation is a conflict-free potential placement of the operation into the current partial schedule. The number of issue slots for an operation can be approximated by its slack, which is an upper bound on the number of *issue cycles* available to it. Each issue cycle may hold multiple issue slots for the operation; for instance, the hypothetical target machine can issue two address-add operations per cycle. Unfortunately, the compiler assigns operations to functional units before scheduling commences, thereby restricting an operation to one issue slot per cycle.

If a loop has no resource contention (ResMII $= 1$), then an operation has precisely as many issue cycles as its slack predicts. Otherwise the scheduler tries to estimate resource contention by dividing a *critical* operation's slack value in half. An operation is **critical** if it uses a critical resource. A resource is critical if one iteration uses the resource for at least $0.90 \times$ II cycles. Critical operations are marked just before attempting each new value of II. Furthermore, divisions and square roots tend to have very few issue slots, due to the complex non-pipelined resource patterns they employ. The compiler primitively takes this into account by halving an operation's slack value (yet again) if it uses the divider. The final slack value is the operation's *dynamic priority*.

At each iteration of its central loop, the scheduler chooses an operation with minimum dynamic priority, which determines a unique operation 48% of the time. Ties are broken by choosing the operation with the smallest Lstart, as this top-down bias interacts well with the scheduler's backtracking policy. In practice, this dynamic priority scheme cleanly integrates recurrence constraints and critical-path considerations. Section 8 gives an intuitive explanation for this empirical result.

4.4 Limited Backtracking

Ejecting operations out of the schedule is a form of backtracking, which must be controlled. To that end, when the scheduler cannot place an operation x due to a lack of conflict-free issue slots, it forces the operation into cycle max $\left(\text{Estart}(x), 1 + \text{last placement of } x\right)$ by ejecting all[7] operations that conflict with its resource needs. This heuristic avoids livelock by forcing x into successively later cycles.

If operation x is forced into a cycle beyond Lstart(x), then other operations may conflict with x due to dependency constraints. These operations need not be immediate successors of x, as MinDist reflects the transitive closure of the successor relation. Nevertheless, ejecting all of these operations—rather than just the immediate successors—tends to reduce the overall amount of backtracking and improve the final schedule.

Ejecting an operation may loosen the Lstart bounds of ancestors and the Estart bounds of descendants. The scheduler recursively marks these bounds as invalid, pruning its depth-first search whenever it encounters a placed operation. If p operations are placed and u operations are unplaced, then the bounds for all unplaced operations can be recomputed in $O(p \times u)$ time. In practice, only a couple operations get ejected at a time, hence few of the bounds need to be recomputed. So the update takes only linear time on average.

[7] With one exception: The loop's `brtop` conditional branch cannot be ejected, as its placement determines the schedule's II. Hence an operation must search successive cycles to avoid conflicts with `brtop`.

5 Lifetime Sensitivity

Prior scheduling algorithms have always placed an operation as early as possible within the partial schedule constructed thus far. This unidirectional strategy unnecessarily stretches operand lifetimes; for example, by scheduling loads too early. In general, unidirectional approaches are a legacy from an over-reliance on the methods and intuition underlying list scheduling.

In contrast, slack scheduling can accommodate a novel bidirectional approach that attempts to place an operation either as early as possible or as late as possible, depending on a sophisticated heuristic. The heuristic's primary goal is to minimize each value's lifetime, in the hope that this will minimize the overall peak register pressure.

5.1 Computing MinLT

The compiler eases the scheduler's task by putting the loop body into static single assignment (SSA) form [4], thereby giving each value a unique defining operation and a precise set of flow dependencies. In that setting, a value's lifetime is merely the length of its longest flow dependence in the final schedule.

A novel schedule-independent lower bound, *MinLT*, on the length of a lifetime can be calculated with the aid of the MinDist relation. Suppose a value v has a defining operation d and a set S of flow dependencies. For each flow dependence $f \in S$, let u denote the corresponding operation that uses v. Then $\textbf{MinLT}(v) = \max_{f \in S} \left(\omega \times \text{II} + \text{MinDist}(d, u)\right)$. This lower bound plays a key role in the following bidirectional heuristic.

5.2 Choosing a Good Issue Cycle

To find an issue cycle for an operation, the scheduler linearly scans between the operation's Estart and Lstart bounds; the scanning stops as soon as an available issue cycle is found. Due to the modulo constraint, at most II consecutive cycles need to be scanned. 46% of the time, the operation has no slack. Otherwise the key decision is whether to scan from Estart to Lstart, or from Lstart to Estart. Empirically, the following heuristics tend to favor an early placement twice as often as a late placement.

When deciding whether to favor placing the operation early or late, the scheduler considers only flow dependencies whose lengths can be stretched. Thus the following are ignored:

- loop invariants, which are stored in the GPR file.
- duplicate inputs, in order to not count a lifetime twice.
- self-recurrences, as their lengths are fixed.

Henceforth, "inputs" and "outputs" refer to stretchable flow dependencies.

If an operation has neither inputs nor outputs, (for example, an accumulator that is not referenced until the loop exits), then it should be placed early in the schedule to minimize the overall schedule length. Otherwise, the scheduler tries to estimate—given the current partial schedule—how many of the operation's inputs and outputs are stretchable

Since the loop body is in SSA form, placing the operation early will stretch its outputs. But placing the operation late might not stretch an input, since some other operation may stretch the input farther than this operation could. In particular, suppose that an operation d defines a value v that is used ω iterations later by an operation u. If $\text{Estart}(d) + \text{MinLT}(v) \geq \omega \times \text{II} + \text{Lstart}(u)$, then operation u cannot stretch v's lifetime.

Empirically, the operation has more stretchable inputs than outputs 30% of the time, and fewer stretchable inputs than outputs 4% of the time, with ties occurring the remaining 20% of the time. In the first case, the operation should be placed early in the schedule; in the second case, it should be placed late. In case of a time, the operation's placement cannot affect the final register pressure, but it might affect the likelihood of finding a feasible schedule. To minimize backtracking, the scheduler attempts to place the operation near whichever group of operations is less likely to be ejected: the operation's immediate predecessors or successors. The scheduler predicts that whichever group has a larger fraction of operations placed is less likely to be ejected. In case of a tie, the scheduler places an operation early if and only if no predecessor or successor has yet been placed.

6 Compilation Time

The bidirectional slack-scheduling framework has been integrated into Cydrome's FORTRAN77 compiler. The compiler is capable of modulo scheduling arbitrary DO loops with unstructured bodies, so long as the loop bodies are acyclic, have no assigned or computed gotos, and make no procedure calls. Even loops with early exits can be modulo scheduled [22], although that experimental feature was not employed for these experiments. Nevertheless, the compiler does not attempt to modulo schedule loops with less than 5 iterations, or more than 30 basic blocks before if-conversion, or which have ResMII > 500, as the benefits would be minimal.

The scheduler has been tested on all eligible DO loops in the Lawrence Livermore Loops, the SPEC89 FORTRAN benchmarks, and the Perfect Club codes—a total of 1,525 loops. Table 2 characterizes the complexity of these loops.

Metric	Min	50%	90%	Max
# Basic Blocks	1	1	1	15
# Operations	2	11	48	556
# Critical Ops at MII	0	2	16	323
# Ops on Recurrences	0	0	8	198
# Div/Mod/Sqrt Ops	0	0	1	11
RecMII	1	1	17	206
ResMII	1	3	19	323
MII	1	3	31	323
MinAvg at MII	1	13	28	57
# GPRs	1	2	8	65

Table 2: Measurements from all 1525 Loops

Scheduling the loops consumed 3.96 minutes of the compilation time on an HP 9000/730 workstation. For 889 of the loops, totaling 7,278 operations, no backtracking was required. For the other 636 loops, the scheduler placed 23,603 operations in 306,860 iterations of its central loop (see Section 4.2). Step 3 had to be invoked 157,694 times, thereby ejecting 282,130 operations. Step 6 was invoked a mere 139 times. Overall, backtracking consumed 65% of the scheduler's execution time. In contrast, computing RecMII took only 6% of the time, and computing MinDist took 10%.

Cydrome's scheduler took $6.5\times$ longer to schedule the loops, primarily because it backtracked $3.7\times$ as much.

7 Performance

The scheduler achieved optimal execution time ($\text{II} = \text{MII}$) for 96% of the loops. Overall, the loops would execute in $1.01\times$ their minimum time, which represents a $1.11\times$ speedup over Cydrome's scheduler. Tables 3 and 4 characterize this performance[8].

The final figures compare the register pressure generated by the bidirectional slack scheduler (denoted "New Scheduler") and Cydrome's scheduler (denoted "Old Scheduler").

- Figure 5 graphs the MaxLive $-$ MinAvg metric. Notice how close MaxLive can get to MinAvg:
  - 46% of the loops achieve optimality.
  - 93% of the loops are within 10 RRs of ideal.
- Figure 6 shows the overall pressure for the RR file. Notice that modulo scheduling does not require excessively many rotating registers:
  - 92% of the loops use no more than 32 RRs.
  - Only 5 loops use more than 64 RRs.

[8]Cydrome's scheduler failed to pipeline 14 of the loops. Each failure is represented in Table 4 by the last II that was attempted.

Loop Class	Opt	All	%	II	MII	Ratio
Has Conditional	56	59	95	1,450	1,447	1.00
Has Recurrence	305	344	89	7,873	7,775	1.01
Has Both	43	54	80	1,664	1,585	1.05
Has Neither	1,059	1,068	99	6,530	6,501	1.00
All Loops	1,463	1,525	96	17,517	17,308	1.01

For the 62 Loops with II > MII				
Metric	Min	50%	90%	Max
II	5	21	64	248
MII	4	19	62	206
II − MII	1	1	6	42
II/MII	1.02	1.06	1.25	1.55

Table 3: Slack Scheduling Performance

Loop Class	Opt	All	%	II	MII	Ratio
Has Conditional	47	59	80	1,515	1,447	1.05
Has Recurrence	272	344	78	8,896	7,775	1.14
Has Both	32	54	59	2,403	1,585	1.52
Has Neither	1,042	1,068	98	6,579	6,501	1.01
All Loops	1,393	1,525	91	19,393	17,308	1.12

For the 132 Loops with II > MII				
Metric	Min	50%	90%	Max
II	3	29	104	618
MII	2	20	68	323
II − MII	1	2	45	412
II/MII	1.02	1.11	3.22	4.5

Table 4: Cydrome's Scheduling Performance

- Figure 7 displays the number of loop invariants kept in the GPR file. It is possible to trade off GPRs for RRs under certain circumstances, so their combined pressure is also of interest.
  - 97% of the loops use no more than 16 GPRs.
  - Only 3 loops use more than 32 GPRs.
  - 82% of the loops keep RRs + GPRs $\leq$ 32.
  - Only 16 loops have RRs + GPRs $>$ 64.
- Figure 8 shows why ICR pressure is of no real concern: only one loop uses more than 32 ICR predicates. (The schedulers generate very similar ICR pressure.)

This performance is due to the bidirectional heuristics of Section 5.2; without them, the slack scheduler generates nearly the same register pressure as Cydrome's scheduler.

In summary, these measurements show that the absolute lower bounds on II and register pressure can usually be achieved by the bidirectional slack-scheduling framework. In addition, the scheduler appears quite robust, as other experiments with different latencies for the functional units give very similar performance results and compilation times.

8 Related Work

By using a dynamic priority scheme, slack scheduling provides a novel integration of recurrence constraints and critical-path considerations. The intuition underlying this integration is that the operations on a recurrence circuit can have a lot of slack until one of them gets placed, at which point the slack can sharply converge nearly to zero. In some sense, the recurrence gets *fixed* at this point, as the other operations now have an anchor around which they must be placed. The dynamic-priority scheme can detect this transition because the scheduler maintains *precise* Estart and Lstart bounds for all operations at all times. Even in the absence of recurrence circuits, the dynamic-priority scheme is an important refinement to the critical-path method [10].

Cydrome's scheduler has a similar backtracking operation-driven framework, with very different heuristics [6]. In particular, it does not employ a dynamic priority scheme; instead, it relies on a static priority that favors those operations whose initial slack is minimal. Thus the scheduler cannot detect when a recurrence circuit becomes fixed. To be safe, the scheduler places all operations that are on recurrence circuits, before placing any other operations.

In order to dispense with backtracking altogether, the Warp compiler special-cases recurrence circuits within a list-scheduling framework [9]. In essence, the compiler fixes the relative timing of the operations on a recurrence circuit before scheduling the overall loop body. By thus reducing each recurrence circuit to a complex pseudo-operation, only acyclic dependencies remain, which are easily dealt with.

However, neither of the two prior approaches is totally satisfactory because the early placement of all operations from a recurrence circuit can be an unnecessary constraint on the scheduler; after all, the minimum schedule length of a recurrence circuit need not be anywhere near its limit of $\Omega \times$ II cycles. The empirical results in [9] and Section 7 support this intuition.

The hardware timing constraints that arise when locally compacting microcode are similar to recurrence constraints. Prior work in this area refers to an operation's Estart and Lstart bounds as its *absolute timing*, while the MinDist relation encodes the operations' *extended timings* [24]. The slack scheduling framework handles these timing constraints more naturally than prior methods.

Prior efforts at lifetime-sensitive scheduling have been in the context of straight-line code for conventional RISC processors [8, 3]. This work has advocated Integrated Prepass Scheduling (IPS) within a list-scheduling framework. IPS

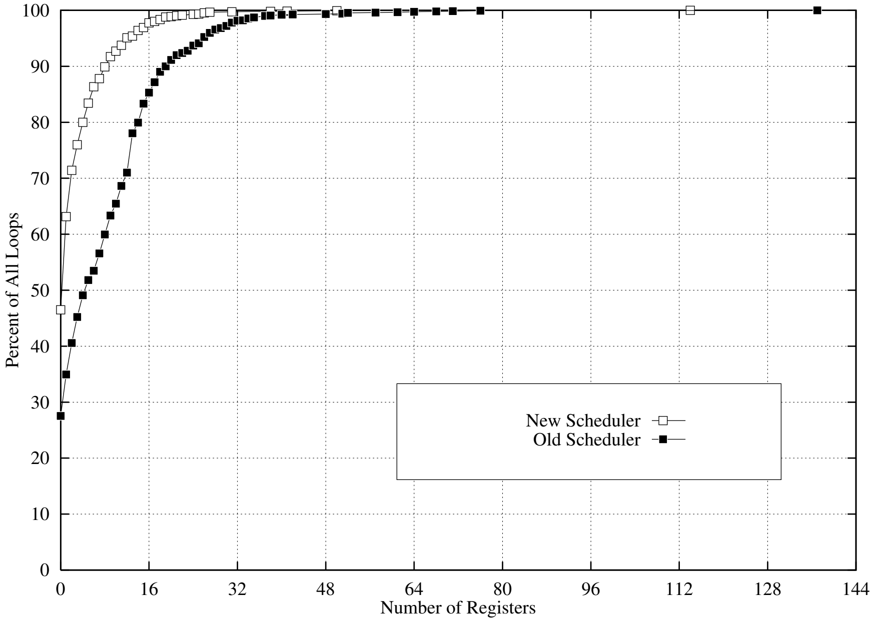

Figure 5: MaxLive - MinAvg

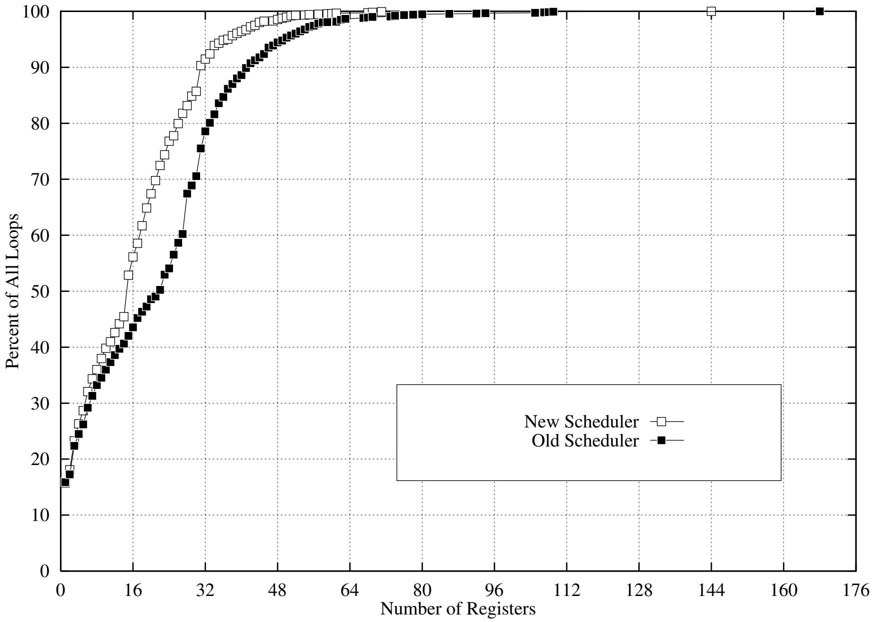

Figure 6: MaxLive

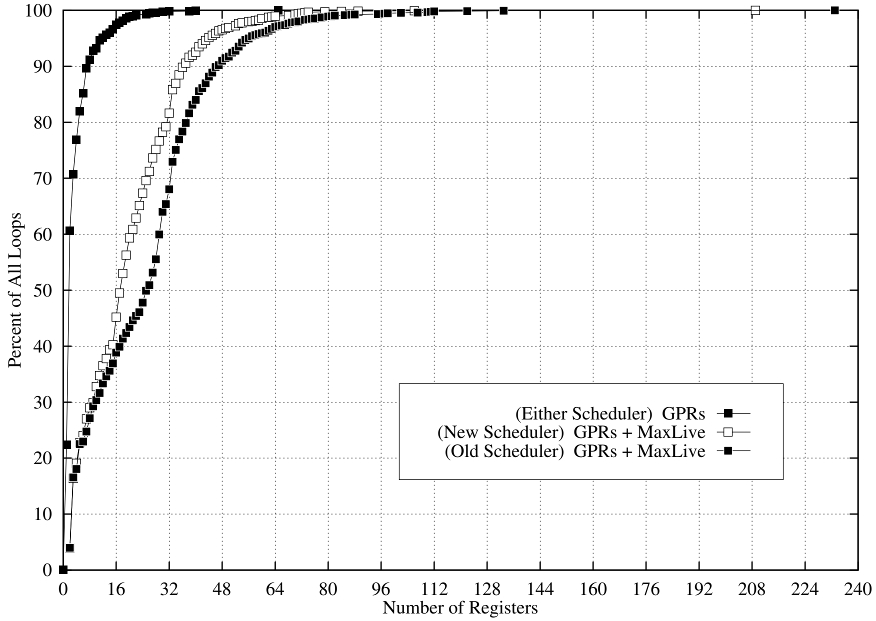

Figure 7: GPRs + MaxLive

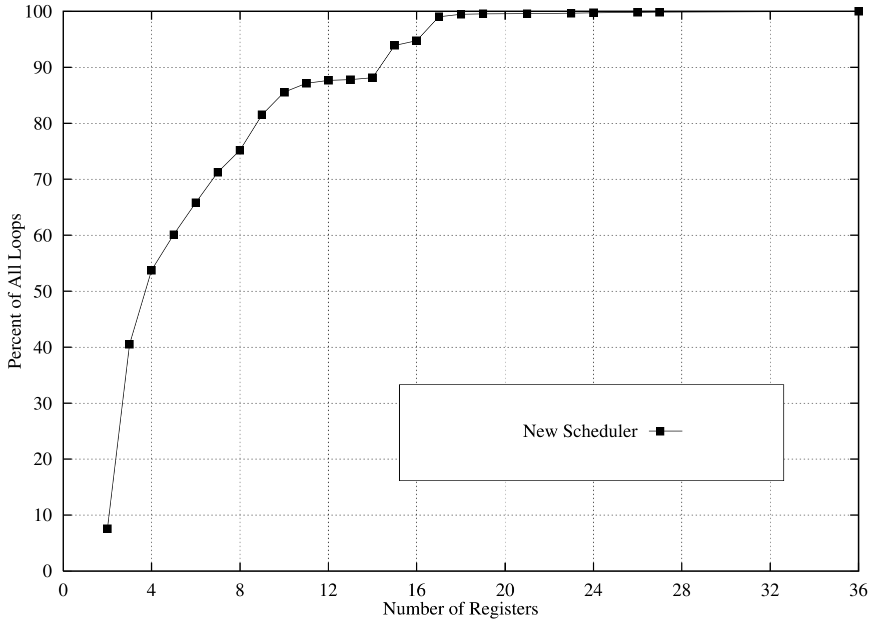

Figure 8: ICR Predicate Usage

switches between a heuristic for avoiding pipeline interlock and a heuristic for reducing register pressure, based on how close the partial schedule is to a register pressure limit. Yet the heuristic for avoiding interlock does not attempt to take register pressure into account—it can squander registers just as freely as previous schedulers. In contrast, the bidirectional slack-scheduling framework, which can be applied to straight-line code as well as loops, attempts to integrate lifetime sensitivity into the placement of each operation. Future experimentation may assess how well slack-scheduling would work in the context where IPS has been studied.

9 Acknowledgments

My advisor, Keshav Pingali, for his patience. Bob Rau, Mike Schlansker, and Vinod Kathail, for giving me: an interesting problem, Cydrome's source code, and many discussions during my stay at HP Labs. Mayan Moudgill, for proofreading an early draft of this paper. Paul Stodghill, Wei Li, Mark Charney, and anonymous referees, for their comments.

References

[1] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Programming Languages*, pages 177–189, Jan. 1983.

[2] G. R. Beck, D. W. L. Yen, and T. L. Anderson. The Cydra-5 mini-supercomputer: Architecture and implementation. *Journal of Supercomputing*, 7(1/2), Jan. 1993.

[3] D. G. Bradlee, S. J. Eggers, and R. R. Henry. Integrating register allocation and instruction scheduling for RISCs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 122–131, Apr. 1991.

[4] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 25–35, Jan. 1989.

[5] J. C. Dehnert, P. Y.-T. Hsu, and J. P. Bratt. Overlapped loop support in the Cydra 5. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 26–38, Apr. 1989.

[6] J. C. Dehnert and R. A. Towle. Compiling for the Cydra 5. *Journal of Supercomputing*, 7(1/2), Jan. 1993.

[7] P. B. Gibbons and S. S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, pages 11–16, 1986.

[8] J. R. Goodman and W.-C. Hsu. Code scheduling and register allocation in large basic blocks. In *Proceedings of the 1988 International Conference on Supercomputing*, pages 442–452, June 1988.

[9] M. S. Lam. *A Systolic Array Optimizing Compiler*. Kluwer Academic Publishers, 1989.

[10] D. Landskov, S. Davidson, B. Shriver, and P. W. Mallet. Local microcode compaction techniques. *ACM Computing Surveys*, pages 261–294, Sept. 1980.

[11] E. L. Lawler. *Combinatorial Optimization: Networks and Matroids*. Saunders College Publishing, 1976.

[12] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donnell, and J. C. Ruttenberg. The Multiflow trace scheduling compiler. *Journal of Supercomputing*, 7(1/2), Jan. 1993.

[13] J. C. H. Park and M. S. Schlansker. On predicated execution. Technical Report HPL-91-58, Hewlett-Packard Laboratories, May 1991.

[14] S. Ramakrishnan. Software pipelining in PA-RISC compilers. *Hewlett-Packard Journal*, pages 39–45, June 1992.

[15] B. R. Rau. Data flow and dependence analysis for instruction level parallelism. In *Fourth Workshop on Languages and Compilers for Parallel Computing*, pages 236–250, Aug. 1991.

[16] B. R. Rau and J. A. Fisher. Instruction-level parallel processing: History, overview and perspective. *Journal of Supercomputing*, 7(1/2), Jan. 1993.

[17] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proceedings of the 14th Annual Microprogramming Workshop*, pages 183–197, Oct. 1981.

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

[19] B. R. Rau, M. S. Schlansker, and P. Tirumalai. Code generation schemas for modulo scheduled loops. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 158–169, Dec. 1992.

[20] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle. The Cydra 5 departmental supercomputer: Design philosophies, decisions, and trade-offs. *IEEE Computer*, pages 12–35, Jan. 1989.

[21] J. C. Tiernan. An efficient search algorithm to find the elementary circuits of a graph. *Communications of the ACM*, pages 722–726, Dec. 1970.

[22] P. Tirumalai, M. Lee, and M. S. Schlansker. Parallelization of loops with exits on pipelined architectures. In *IEEE Proceedings of Supercomputing '90*, pages 200–212, Nov. 1990.

[23] N. J. Warter, J. W. Bockhaus, G. E. Haab, and K. Subramanian. Enhanced modulo scheduling for loops with conditional branches. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 170–179, Dec. 1992.

[24] P. Wijaya and V. H. Allan. Incremental foresighted local compaction. In *Proceedings of the 22nd Annual International Symposium on Microarchitecture*, pages 163–171, Aug. 1989.

[25] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 30–44, June 1991.