Stage Scheduling: A Technique to Reduce the Register Requirements of a Modulo Schedule*

Alexandre E. Eichenberger and Edward S. Davidson

Advanced Computer Architecture Laboratory
EECS Department, University of Michigan
Ann Arbor, MI 48109-2122
alexe,davidson@eecs.umich.edu

Abstract

Modulo scheduling is an efficient technique for exploiting instruction level parallelism in a variety of loops, resulting in high performance code but increased register requirements. We present a set of low computational complexity stage-scheduling heuristics that reduce the register requirements of a given modulo schedule by shifting operations by multiples of II cycles. Measurements on a benchmark suite of 1289 loops from the Perfect Club, SPEC-89, and the Livermore Fortran Kernels shows that our best heuristic achieves on average 99% of the decrease in register requirements obtained by an optimal stage scheduler.

KEYWORDS: Register-sensitive modulo scheduling, software pipelining, loop scheduling, instruction level parallelism, VLIW, superscalar.

1 Introduction

Current research compilers for VLIW and superscalar machines focus on exposing more of the inherent parallelism in an application to obtain higher performance by better utilizing wider issue machines and reducing the schedule length of a code. There is generally insufficient parallelism within individual basic blocks and higher levels of parallelism can be obtained by also exploiting the instruction level parallelism among successive basic blocks. Software pipelining is a technique that exploits the instruction level parallelism present among the iterations of a loop by overlapping the execution of consecutive loop iterations. With sufficient overlap, some machine resources can be fully utilized, resulting in a schedule with maximum steady state throughput.

Modulo scheduling [1][2][3] is a software pipelining technique that results in high performance code while requiring only modest compilation time by restricting the scheduling space: it uses the same schedule for each iteration of a loop and it initiates successive iterations at a constant rate, i.e. one *Initiation Interval* (II clock cycles) apart. This restriction is known as the *modulo scheduling constraint*. As a direct consequence of this constraint, each resource may be used by one iteration at most once within each set of times that are congruent modulo II.

The scope of modulo scheduling has been widened to a large variety of loops. Loops with conditional statements are handled using hierarchical reduction [4] or IF-conversion [5]. Modulo scheduling has also been extended to a large variety of loops with early exits, such as while loops [6][7]. Furthermore, the code expansion due to modulo scheduling can be eliminated by using special hardware, such as rotating register files and support for predicated execution [8].

As modulo scheduling increases execution parallelism, register requirements increase because more values are needed to support more concurrent operations. This effect is inherent to parallelism and will be exacerbated by wider machines and higher latency pipelines [9]. As a result, developing scheduling techniques that exploit instruction level parallelism while containing the register requirements is crucial to the performance of future machines.

Several heuristics have been proposed for register sensitive modulo scheduling. For example, Huff investigated a heuristic based on a bidirectional slack-scheduling method that schedules operations early or late depending on their number of stretchable input and output flow dependences [10]. Llosa *et al* have recently proposed a heuristic that is based on a bidi-

*Appeared in the 28th Annual International Symposium on Microarchitecture, Ann Arbor, pp 338-349, November 1995.

rectional slack-scheduling method with a scheduling priority function tailored to minimize the register requirements [11]. Others have proposed (integer) linear programming solutions [12][13][14][15][16][17]. These approaches are flexible in that they directly generate a modulo schedule, thus potentially generating schedules with lower register requirements. However, several potentially conflicting constraints must be considered at once, such as fulfilling the resource constraints, scheduling operations along critical dependence cycles, maximizing the throughput of the schedule, and minimizing of the schedule length of the critical path.

In this paper, we investigate a two-step approach where a modulo scheduler generates a schedule with high throughput and short schedule length, and is followed by a stage scheduler that can solely focus on decreasing the register requirements. Two-step approaches [15][18][19] are particularly well suited for machines with complex resource requirements, as strategies to maximize the throughput and minimize the schedule length differ significantly from strategies to minimize the register requirements.

We present a set of *stage scheduling* heuristics that reduce the register requirements of a modulo schedule by reassigning some operations to different stages, i.e. by shifting operations by multiples of II cycles. These heuristics address a common shortcoming of current modulo schedulers which tends to result in schedules with increased register requirements for operations not on the critical path.

This paper contributes efficient stage scheduling heuristics with linear or quadratic computational complexity that find a schedule with nearly optimal register requirements. We investigate the performance of these stage scheduling heuristics for a benchmark of 1289 loops from the Perfect Club, SPEC-89, and the Livermore Fortran Kernels. We contrast the performance of these stage scheduling heuristics to the optimal stage scheduler presented in [15]. Our experimental findings show that our best heuristic achieves on average 99% of the decrease in register requirements obtained by an optimal stage scheduler over the entire benchmark suite.

The development of these stage scheduling heuristics was greatly facilitated by our previous work on the optimal stage scheduler, which accurately formalized the feasible space of stage scheduling. While the optimal stage scheduler finds schedules with lower register requirements than the heuristics presented in this paper, its exponential computational complexity makes it impractical for use within a production compiler. Also, our previous work did not present any experimental results.

This work also builds upon a technique presented in [17] that simplifies the dependence graph of a loop iteration. In that work, we explored the entire space of modulo schedules to find a schedule with the highest steady state throughput over all modulo schedules, and the minimum register requirements among such schedules. However, due to the vastly increased complexity of that scheduler, that scheduler would not complete in reasonable time for most of the loops in the benchmark suite used in this paper.

In this paper, we present the impact of stage scheduling on register requirements in Section 2. We derive the information required by the stage scheduling heuristics in Section 3. We describe the stage scheduling heuristics in detail in Section 4. We evaluate the performance of these heuristics for a benchmark suite of 1289 loops in Section 5. Conclusions are presented in Section 6.

2 Register Requirements

In this section, we present the impact of stage scheduling on the register requirements of a modulo scheduled loop. The example *target machine* is a hypothetical processor with three fully-pipelined general-purpose functional units. The memory latency and the `add/sub` latency is 1 cycle, 4 for the `mult`, 8 for the `div`, and 10 for the `sqrt`. We selected these values to obtain concise examples; however, our method works independently of the numbers and types of functional units, resource constraints, and latencies.

Example 1 This example illustrates how to compute the register requirements of a modulo-scheduled loop. This kernel is: `z[i] = 2*x[i] + y[i]`, where the `x[i]` and `y[i]` values are read from memory, `x[i]` is multiplied by 2, incremented by `y[i]`, and stored in `z[i]`, as shown in the dependence graph of Figure 1a.

The vertices of the dependence graph correspond to operations and the edges correspond to virtual registers. The value of each *virtual register* is defined by a unique operation and once its value has been defined, it may be used by several operations. In this paper, a virtual register is reserved in the cycle where its define operation is scheduled, and remains reserved until it becomes free in the cycle following its last-use operation. In our machine model, the additional reserved time beyond the last-use operation cycle is one clock cycle. Other machines may have an additional reserved time which is 0, ≥ 2, or operation dependent. The *lifetime* of a virtual register is the set of cycles during which it is reserved.

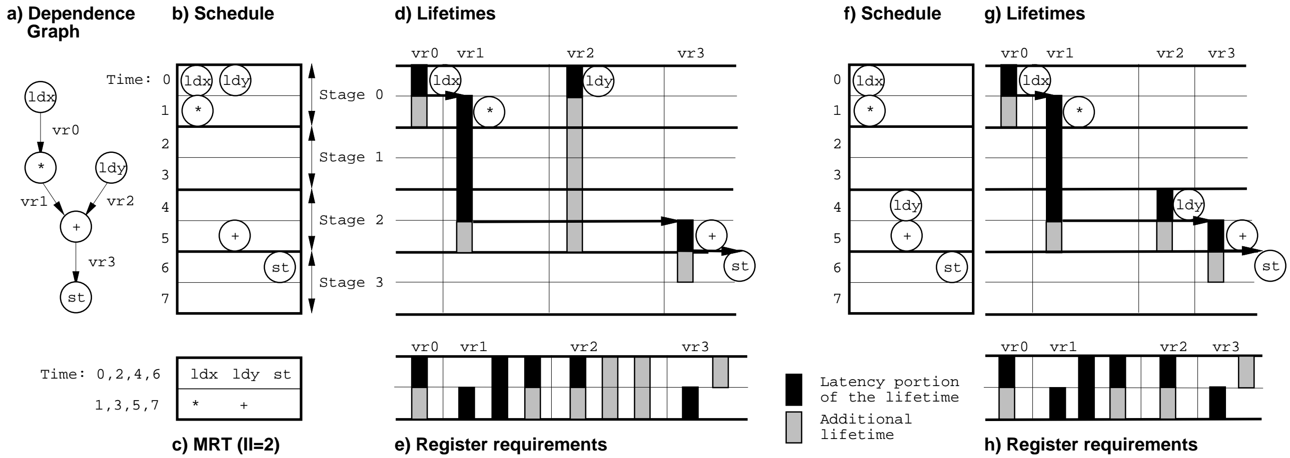


Figure 1: Modulo schedule for the kernel `z[i] = 2*x[i] + y[i]` (Example 1).

A modulo schedule for the kernel of Example 1 on the target machine is illustrated in Figure 1b. In this schedule, the **load-x**, **load-y**, **mult**, **add**, and **store** operations of the iteration starting at time 0 are respectively scheduled at time 0, 0, 1, 5, and 6. A modulo schedule can be logically divided into *stages* of II cycles each. The number of stages in a schedule is referred to as the *Stage Count* [8], and defines the maximum number of concurrent iterations in a modulo schedule. Figure 1b also depicts the 4 stages associated with the schedule of Example 1.

The resource requirements of a modulo schedule are best summarized by using a *Modulo Reservation Table* (MRT), which contains II rows, one per clock cycle, and one column for each resource at which conflicts may occur. The MRT associated with a schedule is obtained by collapsing the schedule for an iteration to a table of II rows, using wraparound. Figure 1c illustrates the MRT associated with the schedule of Figure 1b. The resource constraints of a modulo schedule are satisfied if and only if the packing of the operations within the II rows of the MRT does not exceed the resources of the machine. For our target machine, the resource constraints allow up to 3 operations of any kind to be issued in each row of the MRT.

The virtual register lifetimes associated with this iteration are presented in Figure 1d. The black bars correspond to the initial portion of the lifetime that minimally satisfies the latency of the defining functional unit; the grey bars show the additional lifetime of each virtual register. Ideally, the additional lifetime should not be longer than one cycle; but, because of resource constraints, it is not always possible to schedule each use operation immediately after its input operand latencies expire. Using this figure, the dependence constraints are satisfied if and only if no operation is placed during the portion of the lifetime that overlaps with the black bar of a virtual register that it uses.

The number of live virtual registers in each of the II cycles of a steady-state loop execution is constructed by replicating Figure 1d with a shift of II cycles between successive copies until the pattern in II successive rows repeats indefinitely. Figure 1e then displays these II rows in a compact form and can be constructed quickly by collapsing Figure 1d to II rows, with wraparound. In Figure 1e, we see that exactly seven virtual registers are live in the first row and eight in the second, resulting in a MaxLive of eight.

Formal Problem Definition: Given a modulo schedule, assign each operation to a stage such that operation latencies are satisfied and MaxLive, the maximum number of live values at any single cycle of the loop schedule, is reduced.

Because stage assignment only shifts operations by multiples of II cycles, the resource constraints, as expressed in the MRT, are not affected by the stage scheduler. However, a stage scheduler must ensure that all dependence constraints are satisfied. In Example 1, a stage scheduler can reduce the register requirements associated with the schedule presented in Figure 1b by assigning the **load-y** operation to stage 2 instead of stage 0. The improved schedule, shown in Figure 1f, 1g, and 1h results in a MaxLive of six instead of eight.

3 Input to Stage Scheduling Heuristics

In this section, the information used by the stage scheduling heuristics is derived from the dependence graph and the given modulo schedule of a loop iteration. We then present the general framework for the

stage scheduling heuristics.

We represent a loop by a dependence graph $G = \{V, E_{sched}, E_{reg}\}$, where the set of vertices V represents operations and the sets of edges E_{sched} and E_{reg} correspond, respectively, to the scheduling dependences and the register dependences among operations. A *scheduling edge* enforces a temporal relationship between a pair of dependent operations or between a pair that cannot be freely reordered, such as load and store operations to ambiguous memory locations. A scheduling edge from operation i to operation j, w iterations later, is associated with a latency $l_{i,j}$ and a dependence distance $\omega_{i,j} = w$. A *register edge* corresponds to a data flow dependence carried in a register. In this paper, each register edge is also a scheduling edge. However, there are scheduling edges that have no corresponding register edge, e.g. the dependence between a store and a load to an ambiguous (or the same) memory location results in a scheduling edge but not in a register edge.

We characterize the initial modulo schedule by its initiation interval II and by the time at which each operation of the first loop iteration is scheduled. We refer to the time at which operation i is scheduled as $time_i$. Using II and $time_i$, we determine the row of the MRT in which operation i is placed, namely $row_i = time_i \bmod II$. We also define the following distance relation between operation i and j [15]:

$rdist_{i,j}$	distance between the row of operation i to the next row of operation j, possibly in the next instance of the MRT.

which may be computed as follows:

$$rdist_{i,j} = (row_j - row_i) \bmod II \qquad (1)$$

Note that since the MRT of a modulo schedule is unchanged by the stage scheduler, row_i and $rdist_{i,j}$ remain constant in this paper. Using these variables, we may define the time difference along a scheduling edge from operation i to operation j as:

$$time_j - time_i = rdist_{i,j} + skip_{i,j} * II \qquad (2)$$

where the first and second term of the right hand side represent, respectively, the fractional part and the integral part of the scheduling distance along the scheduling edge (i, j). We refer to $skip_{i,j}$ as the *skip factor* along edge (i, j). The skip factor corresponds to number of times that the row of operation j is skipped before operation j is scheduled, counting from the schedule time of operation i. In Figure 1c for example, from the time of the `mult` operation (time 1) to the time of the `add` operation (time 5), the row of the `add` operation is skipped twice, at time 1 and 3. The skip factor is within 1 of the *iteration difference* used in [9] [12].

In this paper, we distinguish between two reasons for skipping an operation along a scheduling edge (i, j): $s_{i,j}$, representing the minimum skip factor necessary to hide the latency of the scheduling edge, and $p_{i,j}$, representing an additional (non-negative) skip factor used to postpone operation j further. The $p_{i,j}$ variables are the fundamental variables modified by the stage scheduling heuristics.

Consider a scheduling edge between operation i and operation j, $\omega_{i,j}$ iterations later. Using Equation (2), we may write the dependence constraints, $(time_j + \omega_{i,j} * II) - time_i \geq l_{i,j}$, as:

$$rdist_{i,j} + (\omega_{i,j} + s_{i,j} + p_{i,j}) * II \quad \geq \quad l_{i,j} \qquad (3)$$

Equation (3) states that the distance between the row of operation i to the next row of operation j, increased by skipping $\omega_{i,j} + s_{i,j} + p_{i,j}$ entire instances of the MRT, must be no smaller than the latency $l_{i,j}$ of the scheduling edge (i, j). Since we are interested in the smallest integer value of $s_{i,j}$ that satisfies Equation (3), regardless of the nonnegative value of $p_{i,j}$, we obtain the following value for the minimum skip factor $s_{i,j}$:

$$s_{i,j} = \left\lceil \frac{l_{i,j} - rdist_{i,j}}{II} \right\rceil - \omega_{i,j} \qquad (4)$$

Consequently, all scheduling scheduling edges are satisfied when each use operation is skipped at least $s_{i,j}$ times before being scheduled.

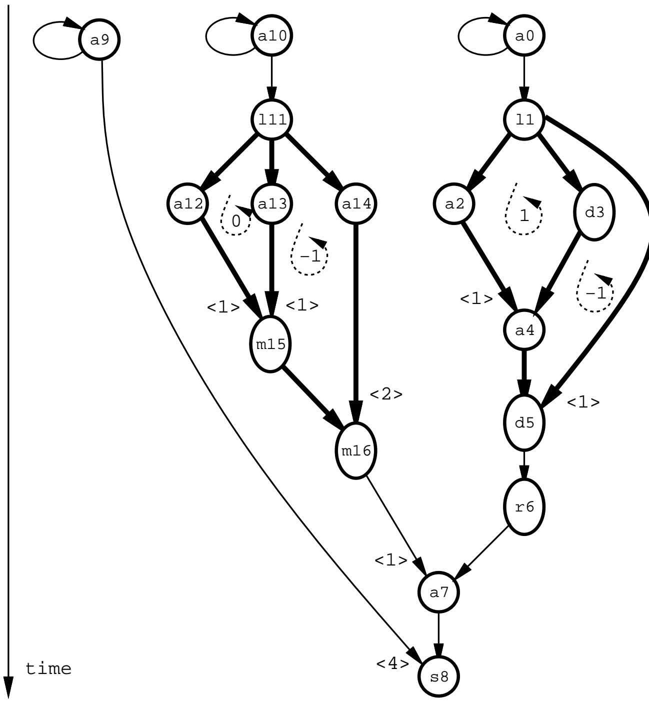
In this paper, we determine the initial $p_{i,j}$ associated with each scheduling edge, given the scheduling time of the operations in the initial schedule, by using:

$$p_{i,j} \quad = \quad \left\lfloor \frac{time_j - time_i - l_{i,j}}{II} \right\rfloor + \omega_{i,j} \qquad (5)$$

where the minimum skip factor $s_{i,j}$ was eliminated from Equation (2) using Equation (4) and $skip_{i,j} = s_{i,j} + p_{i,j}$. The input to the stage scheduling heuristics includes the initial $p_{i,j}$ value, from Equation (5), for each edge of the dependence graph.

Example 2 We will illustrate our techniques using the following kernel as a running example: `z[i] = SQRT(x[i]/ (1/x[i]+x[i]+a)) +(y[i]+a) *(y[i] +b) *(y[i] +c)`, computed as presented in Figure 2a. This example, although constructed to illustrate our heuristics concisely, is representative of several kernels found in our benchmark suite. A modulo schedule with $II = 6$ for our target machine is presented in

a) Dependence graph with additional skip factors



b) Schedule (II=6)

```
Time:  Operations
   0:  a0, a10;
   1:  l1;
   2:  a2, d3, l11;
   3:  a12, a13;
   4:  a14;
   5:  a9;
   *
  10: a4;
  11: d5;
   *
  15: m15;
   *
  19: r6, m16;
   *
  29: a7;
  30: s8;
```

long latency operations
mult: m15, m16
div: d3,d5
sqrt: r6

short latency operations
add: a0,a2,a4,a7, a9,a10,a12, a13,a14
load: l1,l11
store: s8

register and scheduling edge

<x> non-zero additional skip factor

cumulative skip factor along an underlying cycle

Figure 2: Schedule and dependence graph with additional skip factor (Example 2).

Figure 2b. This schedule is a good example of a schedule produced by a register insensitive modulo scheduler: it results in a schedule with high throughput and provides a good stage schedule along the critical path (`add0`, `load1`, `div3`, `add4`, `div5`, `sqrt6`, `add7`, `store8`) as the schedule length of a loop is typically minimized. However, operations not on the critical path tend to be scheduled too early, e.g. operation `add9` is scheduled at time 5 when it could be scheduled as late as at time 29. This effect is due to the fact that current modulo schedulers favor scheduling operations early to prevent increasing II or the schedule length needlessly. Stage scheduling will make a significant difference for schedules in which the number of stages and the number of operations not on the critical path are not too small. Trying different MRTs for a given loop iteration is likely to become increasingly important as II increases.

The dependence graph in Figure 2a has three self-loops on the address arithmetic nodes (`add0`, `add9`, and `add10`) and four underlying cycles[1] in the underlying graph $G = \{V, E_{sched} \cup E_{reg}\}$. The dependence graph for Example 2 is representative of the ones found in our benchmark suite, as on average more than 75% of the operations of a loop iteration belong to an underlying cycle or a self-loop. Non zero $p_{i,j}$ are shown along their respective edges.

Because we describe a stage schedule using time-differences associated with edges instead of absolute times associated with operations, we must ensure that, when there are several paths between two operations, the sums of the time-differences between these two operations are the same regardless of the path. The additional skip factor is one term in the time-differences, as shown in Equation (2), and the other terms may not be the same, because of different latencies and different numbers of operations. Once the other terms are computed, the sum of the time-differences along two such paths are set to be equal, thus obtaining an equation with additional skip factors and a constant term that may not necessarily be equal to 0.

In general, there is one such equation for each pair of distinct paths between two operations in the underlying dependence graph. As shown in [15], it is sufficient to have one equation per elementary underlying cycle, as multiple paths between operations are composed from elementary underlying cycles in the underlying dependence graph. These equations are referred to as *underlying cycle constraints* and precisely define the space of feasible stage schedules. Since we are

[1] An underlying cycle corresponds to a cycle in the underlying graph, which is formed by ignoring the direction of each arc.

given a valid modulo schedule, the constant term can be quickly computed as the sum of the $p_{i,j}$ from the given modulo schedule, where $p_{i,j}$ is taken as negative when the edges are traversed in the reverse direction. These constant terms are shown in Figure 2a inside the curved dashed arrows.

4 Stage Scheduling Heuristics

This section presents the stage scheduling heuristics investigated in this paper. First, we describe the three basic transforms modifying the additional skip factors ($p_{i,j}$) considered by our heuristics. Then we present the four heuristics considered in this paper. We conclude by illustrating the effect of these heuristics on our running example.

4.1 Stage Scheduling Transforms

The first transform modifies the additional skip factors associated with cut edges, i.e. edges that belong to acyclic parts of the underlying dependence graph. We proved in [15] that the additional skip factors of a dependence graph without underlying cycles can be individually minimized by setting their values to zero. We apply this theorem here to the parts of the underlying dependence graph that are acyclic.

Theorem 1 (Acyclic Transform) *Setting to zero the additional skip factor of a cut edge of the underlying dependence graph results in a valid stage schedule.*

Proof: To guarantee that the acyclic transform results in valid stage schedules, we must show that each $p_{i,j}$ remains nonnegative and that the sum of the $p_{i,j}$ along each underlying cycle of the dependence graph remains unchanged. By definition, a cut edge cannot be part of any underlying cycle as, otherwise, the removal of that edge would not create a partition of the underlying graph. As a result, this transform does not modify the value of the additional skip factors along any underlying cycles. Consequently, the sum of the $p_{i,j}$ along each underlying cycle remains constant and each $p_{i,j}$ remains nonnegative. □

The second transform shifts some of the additional skip factors associated with the output edges of a vertex to the input edges of that vertex. Figure 3 illustrates this transform, where operation i is scheduled three stages later, i.e. three units of additional skip factor are propagated up, from the output edges to the input edges of operation i.

Theorem 2 (Up-Propagation Transform) *Reducing the additional skip factor of each output edge of a vertex by the minimum additional skip factor among these edges and increasing the additional skip factor of each input edge by the same amount results in a valid stage schedule.*

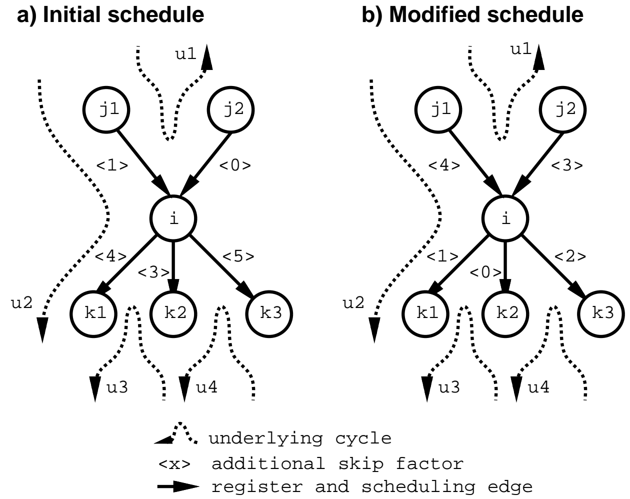


Figure 3: Up-propagation transform.

Proof: By decreasing the $p_{i,j}$ of each output edge of vertex i by the minimum $p_{i,j}$ among these edges, the $p_{i,j}$ values are guaranteed to remain nonnegative. Since the $p_{i,j}$ of each input edge is increased by the same amount as the output edges decrease, the sum of the $p_{i,j}$ along each underlying cycle is guaranteed to remain constant. □

The similar transform, *down-propagation*, shifts the additional skip factor down instead of up, and is obtained by interchanging the words input and output in the description of the up-propagation transform.

4.2 Acyclic Heuristic (AC)

The first heuristic reduces the register requirements associated with edges that belong to acyclic parts of the underlying dependence graph. This heuristic has a linear computational complexity in the number of edges and results in valid stage schedules with no larger register requirements.

First, the heuristic detects all cut edges, using a linear-time algorithm that enumerates the biconnected components of an undirected graph [20, pp. 179–187], i.e. the underlying cycles of the dependence graph. Cut edges are then simply the edges linking vertices from distinct biconnected components. Note that self-loop edges can safely be ignored during the stage scheduling process, because they do not introduce any scheduling constraints for the stage scheduler.[2]

The heuristic then performs the constant-time acyclic transform on each of the cut edges. Since

[2] The self-loop scheduling constraints only affect the choice of II, which is unchanged by the stage scheduling heuristics.

acyclic transforms only decrease the $p_{i,j}$, and since the register requirements associated with a stage schedule is the sum of the skip factors of the last-use operations plus other unrelated terms [15], the register requirements cannot increase.

This heuristic highlights the advantage of describing a stage schedule using skip factors associated with edges instead of absolute scheduling times associated with operations. In Figure 2 for example, setting $p_{16,7}$, the additional skip factor between operations `mult16` and `add7`, to 0 effectively delays `mult16` and all its predecessors by II cycles. This simple local change thus reschedules operations `add10`, `load11`, `add12`, `add13`, `add14`, `mult15`, and `mult16` one stage later.

Because the additional skip factor of a cut edge can always be set to 0, without affecting any of the underlying cycles and regardless of the rest of the dependence graph, the acyclic heuristic will be used first, and subsequent heuristics will ignore both cut edges and self-loop edges. The next heuristics will thus operate only on the non-trivial cyclic parts of the underlying graph. In Figure 2a, there are two non-trivial biconnected components, one containing the operations `load1`, `add2`, `div3`, `add4`, and `div5`, and one containing the operations `load11`, `add12`, `add13`, `add14`, `mult15`, and `mult16`. The edges of these components are shown in bold in Figure 2a.

4.3 Sink/Source Heuristic (SS)

The second heuristic addresses single vertices that are scheduled too early or too late. It considers only sink vertices, i.e. vertices with no output edges, and source vertices, i.e. vertices with no input edges. This heuristic has a linear computational complexity in the number of edges and results in valid stage schedules with no larger register requirements.

First, the heuristic detects all sink and source vertices, ignoring cut edges and self-loop edges. Then, the heuristic applies the up-propagation transform to all source vertices, visiting each edge at most once. Since source vertices have no input edges, the up-propagation transform only decreases the additional skip factor. Thus, the register requirements cannot increase. Similarly, the heuristic applies the down-propagation to all the sink vertices.

4.4 Up Heuristic (UP)

We now present a technique that enhances the impact of the sink/source heuristic by propagating the additional skip factors upward in the dependence graph. This heuristic has a linear computational complexity in the number of edges, and results in valid stage schedules. Note that this heuristic may increase the register requirements of a schedule, although that rarely occurs in our benchmark.[3]

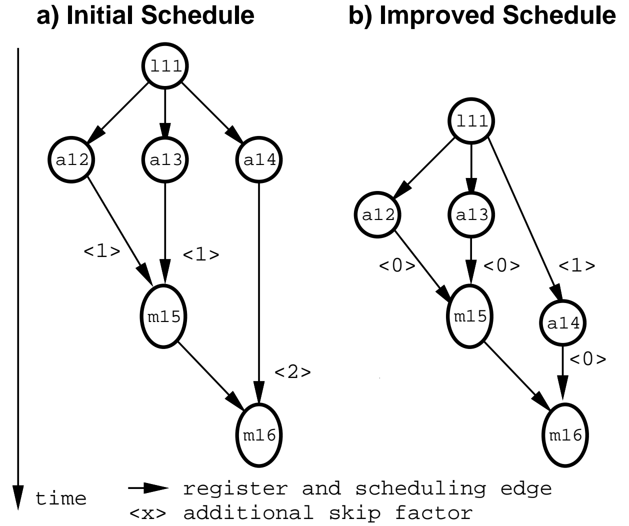


Figure 4: Up propagation heuristic.

We illustrate this heuristic by investigating one of the biconnected components of our running example, shown in Figure 4a. Notice that the sink/source heuristic cannot reduce the additional skip factors as both the minimum $p_{i,j}$ among the output edges of the source operation `load11` and the minimum $p_{i,j}$ among the input edges of the sink operation `mult16` are 0. However, $p_{12,15}$, $p_{13,15}$, and $p_{14,16}$ can each be reduced by 1 without modifying the sum of the $p_{i,j}$ along the two underlying cycles and without resulting in negative $p_{i,j}$ values.

This heuristic proceeds by applying the up-propagation transform to the vertices of the dependence graph, in reverse topological order. In Figure 4a, the transform can be applied to operations `add12`, `add13`, and `add14`, and then `load11`. Figure 4b shows the additional skip factors after completion of this heuristic. For strongly connected components, it may be advantageous to use this heuristic several times. Also, a similar heuristic may be constructed using the down-propagation transform with the normal topological order.

4.5 Register Sink/Source Heuristic (RSS)

This last heuristic is based on removing redundant register edges in the dependence graph, which helps

[3]In our benchmark suite of 1289 loops, the up-heuristic increased the register requirements of only three loops, and those by no more than 2.

make improvements of a stage schedule more apparent. This heuristic has a quadratic computational complexity in the number of edges and results in valid stage schedules with no larger register requirements.

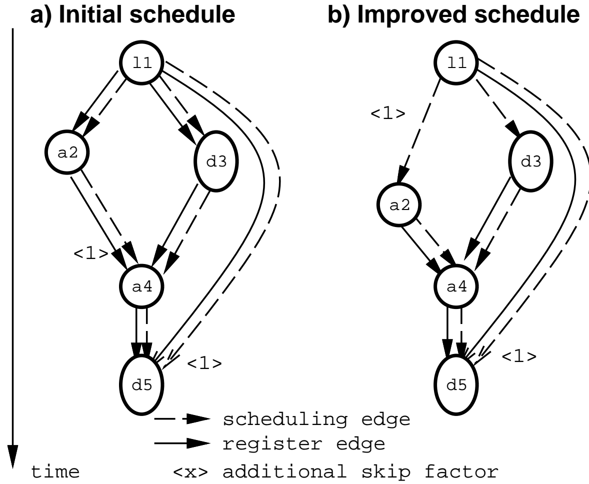


Figure 5: Register source heuristic example.

Figure 5 illustrates the importance of removing redundant register edges, using the second biconnected component of Figure 2a. Figure 5a illustrates the dependence graph for this component before the elimination of redundant register edges. In this graph, the sum of the additional skip factors along the leftmost underlying cycle must be equal to 1. For example, either $p_{1,2}$ or $p_{2,4}$ may be 1, with the other $p_{i,j} = 0$. Without careful inspection, one may conclude that either solution results in the same register requirements, as reducing one lifetime increases the other, since both $p_{1,2}$ and $p_{2,4}$ are associated with scheduling and register edges.

However, by using the redundant edge removal technique presented in [17], we discover that the register edges (`load1`, `add2`) and (`load1`, `div3`) are redundant, as the register requirements generated by the register edge (`load1`, `div5`) cover the register requirements of these two edges. It is then apparent that minimizing $p_{2,4}$ instead of $p_{1,2}$ results in lower register requirements, since $p_{1,2}$ is no longer associated with a register edge, as shown in Figure 5b.

First, the heuristic removes all redundant edges, which has a quadratic computational complexity in the number of edges[17], if the MinDist relation is provided. Computing the MinDist relation [10] has a cubic computational complexity in the number of operations. However, this relation may be readily available, as some modulo schedulers use this relation to determine the minimum initiation interval.

Then, the heuristic applies the up-propagation transform to each reg-source vertex, i.e. vertices without input register edges. Although the up-propagation transform may increase the additional skip factor along the input edges, we know that none of these edges contributes to the overall register requirements. Thus, the register requirements cannot increase. We then apply the down-propagation transform to the reg-sink vertices, i.e. those without output register edges.

4.6 Improved Stage Schedule

To summarize the benefits of the stage scheduling heuristics presented in this paper, we consider the schedules of Example 2 presented in Figures 2b and 6b. The new stage schedule in Figure 6 delays one operation by 4 stages (`add9`), one operation by 3 stages (`add14`), four operations by 2 stages (`add10`, `load11`, `add12`, `add13`), and three operations by 1 stage (`add2`, `mult15`, `mult16`), reducing the register requirements by 9, i.e. a reduction of 4 for `vr9`, 2 for `vr14`, 1 for `vr2`, `vr12`, `vr13`, and `vr16`, and an increase of 1 for `vr11`. Figure 6a also shows which stage scheduling heuristics produce each modification in the dependence graph.

Using both the up-propagation and the register sink/source transform is advantageous, as they address distinct features of underlying cycles. However, the sink/source transform is fully covered by the register sink/source transform; thus it need not be employed if the register sink/source transform is used. Finally, one could consider using the up-propagation transform several times, as the propagation of the $p_{i,j}$ values may require more than one pass before being reduced for a dependence graph with strongly connected components.

5 Measurements

We investigate the register requirements of the integer and floating point register files for a benchmark of loops obtained from the Perfect Club [21], SPEC-89 [22], and the Livermore Fortran Kernels [23]. Our input loops consist exclusively of innermost loops with no early exits, no procedure calls, and fewer than 30 basic blocks, as compiled by the Cydra 5 Fortran77 compiler [24]. The input to our scheduling algorithms consists of the Fortran77 compiler intermediate representation after load-store elimination, recurrence back-substitution, and IF-conversion. Our benchmark suite consists of 1289 out of 1327 loops successfully modulo scheduled by the Cydra 5 Fortran77 compiler, with 1002 from the Perfect Club, 298 from SPEC, and 27 from the Livermore Fortran Kernels.

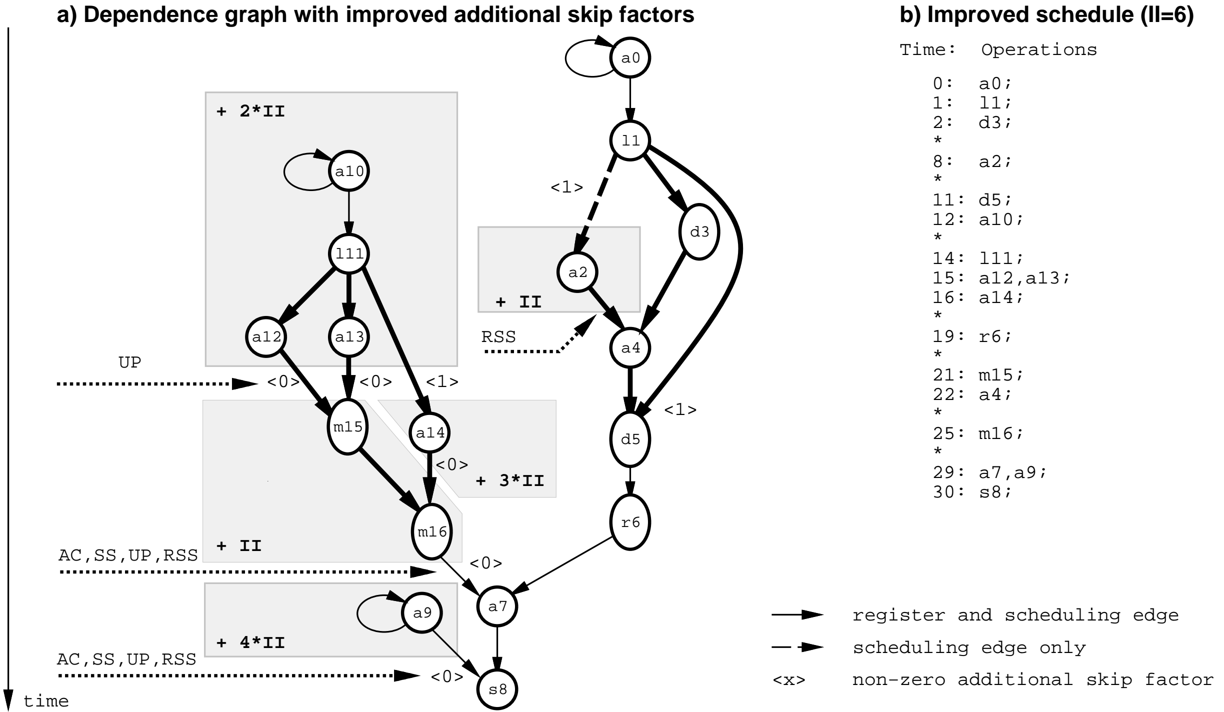


Figure 6: Improved schedule (Example 2).

	Ops.	Ops. in cycles	Underlying cycles	Reg. edges	Sched. edges
min.	2	1	0	1	1
avg.	17.3	13.1	2.8	22.0	25.0
max.	161	160	66	220	345

Table 1: Characteristics of the benchmark suite.

Table 1 summarizes the principal characteristics of the benchmark suite. On average, more than 75% of the operations in a loop iteration belong to an underlying cycle or a self-loop in the underlying dependence graph.

The machine model used in these experiments corresponds to the Cydra 5 machine. This choice was motivated by the availability of quality code for this machine. Also, the resource requirements of the Cydra 5 machine are complex [25], thus stressing the importance of good and robust scheduling algorithms. In particular, the machine configuration is the one used in [3], with 7 functional units (2 memory port units, 2 address generation units, 1 FP adder unit, 1 FP multiplier unit, and 1 branch unit).

Before presenting the performance of the stage scheduling heuristics, we characterize the register requirements associated with this benchmark by investigating the performance of three modulo/stage schedulers. We are unable to provide a comparison with Huff's scheduling algorithm since his machine model differs slightly from ours and his latest scheduler, presented in [10], was not available to us.

MinReg Stage Scheduler[15]: This stage scheduler minimizes MaxLive, the maximum number of live values at any single time in the loop schedule, over all valid modulo schedules that share a given MRT. The resulting schedule has the lowest achievable register requirements for the given machine, loop, and MRT. For dependence graphs with underlying cycles, the complexity of this scheduler is exponential.

MinBuf Stage Scheduler [15]: This stage scheduler minimizes the integer MaxLive over all valid modulo schedules that share a given MRT. The resulting schedule has the lowest achievable buffer requirements for the given machine, loop, and MRT. Buffers must be reserved for a time interval that is a multiple of II, whereas registers may be reserved for arbitrary time periods. We present here the register requirements associated with these schedules in our comparisons. For dependence graphs with underlying cycles, the complexity of this scheduler is polynomial.

Iterative Modulo Scheduler[3]: This modulo scheduler has been designed to deal efficiently with realistic machine models while producing schedules with near optimal steady state throughput. Experimental findings show that this algorithm requires the scheduling of only 59% more operations than does acyclic list

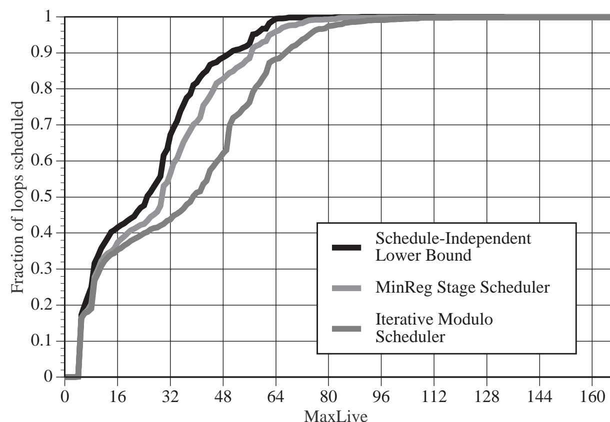


Figure 7: Register requirements.

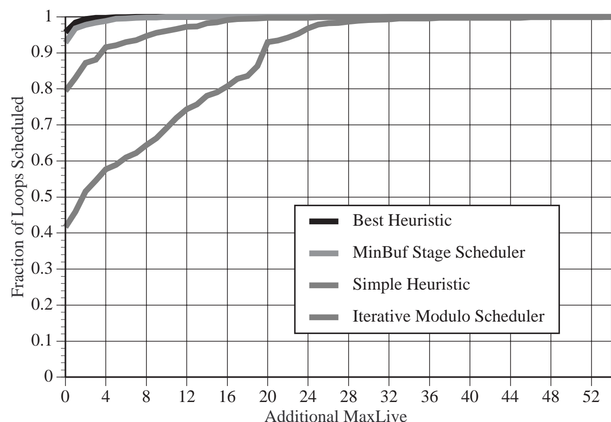


Figure 8: Additional register requirements.

scheduling while resulting in schedules that are optimal in II for 96% of loops in their benchmark [3]. In its current form, it does not attempt to minimize the register requirements of its schedules; however, we will show that the register requirements of its schedules may be reduced significantly by the simple heuristics of this paper.

In this section, the Iterative Modulo Scheduler was used to generate the required MRT input for the MinReg and MinBuf Stage Schedulers as well as the initial modulo schedule for the stage scheduling heuristics.

We first investigated the register requirements of the MinReg Stage Scheduler which results in the lowest register requirements over all valid modulo schedules that share a common MRT. Figure 7 presents the fraction of the loops in the complete benchmark that can be scheduled for a machine with any given number of registers without spilling and without increasing II. In this graph, the X-axis represents MaxLive and the Y-axis represents the fraction of loops scheduled.

The "Schedule Independent Lower Bound" curve corresponds to [10] and is representative of the register requirements of a machine without any resource conflicts. There is a significant gap between the MinReg Stage Scheduler curve and the Lower Bound curve which we believe is caused by two factors. First, the MinReg Stage Scheduler searches for the schedule with the lowest register requirements only among the modulo schedules that share a given MRT. Therefore, when the MRT given to the stage scheduler is suboptimal, the stage scheduler will find a local minimum that may be significantly larger than the absolute minimum.[4] The second factor is that the schedule-independent lower bound may be significantly too optimistic due to the complexity of some of the loops in the benchmark suite and the fact that this lower bound ignores the resource constraints of the machine. The third curve presents the register requirements of the Iterative Modulo Scheduler, which presently does not attempt to minimize the register requirements. The gap between these last two curves indicates the degree of improvement that can be achieved by stage scheduling heuristics.

Figure 8 illustrates the number of additional registers needed when scheduling algorithms other than the MinReg Stage Scheduler are used. From top to bottom, the curves present the additional registers needed by the MinBuf Stage Scheduler, the Best Stage Scheduler Heuristic,[5] a Simple Stage Scheduler Heuristic,[6] and the Iterative Modulo Scheduler. We first notice that both the best heuristic and the MinBuf Stage Scheduler, whose curves nearly overlap, find a stage schedule with no additional register requirements in 96% and 93% of the loops, respectively. The MinBuf Stage Scheduler schedules 95% of the loops with no more than 1 additional register. The Simple Stage Schedule Heuristic finds a stage schedule with no additional register requirements in 80% of the loops, and schedules 95% of the loops with no more than 10 additional registers. The Iterative Modulo Scheduler results in a stage schedule with no additional register requirements in 40% of the loops, a surprisingly high number for a scheduler that does not even attempt to minimize the register requirements. This result may be partially explained by the fact that this scheduler

[4] Experimental evidence on a subset of this benchmark suite, consisting of all the loops with no more than 12 operations and II up to 5, for which an optimal modulo schedule was sought over all MRTs, showed however that the difference between the local and absolute minimum was surprisingly small[17]. However, this result may not hold for larger loops.

[5] The best heuristic uses the up-propagation transform three times followed by the register sink/source transform.

[6] The simple heuristic uses the sink/source transform.

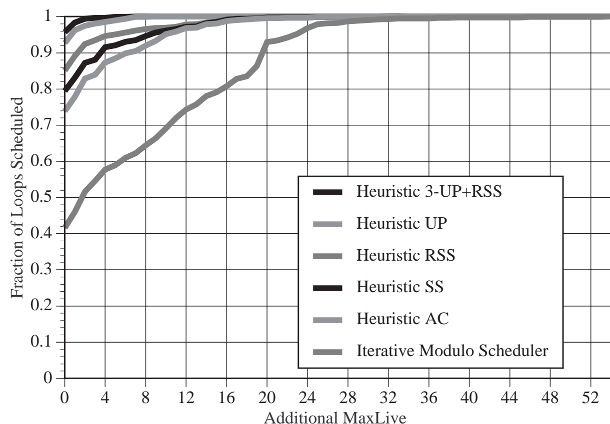


Figure 9: Additional register requirements of the stage scheduling heuristics.

does attempt to minimize the length of a schedule, which generally results in a stage schedule with low register requirements along the critical path of that schedule.

Figure 9 illustrates the additional register requirements of several combinations of stage scheduling heuristics. To facilitate the reading of this graph, the heuristics are listed in the legend in order of decreasing performance.

The bottom curve of Figure 9 corresponds to the performance of the Iterative Modulo Scheduler. The next curve corresponds to the AC heuristic, which finds a stage schedule with no additional register requirements in 74% of the loops, and schedules 95% of the loops with no more than 10 additional registers. The significant decrease in register requirements due to the AC heuristic is uniquely obtained by rescheduling entire biconnected components of the underlying dependence graph as close to one another as possible. To decrease the register requirements of the modulo scheduled loops further, the schedule within each biconnected component must also be considered. The next curve corresponds to the SS heuristic, which reschedules those vertices in the dependence graph that have no input or no output edges. The SS heuristic results in lower register requirements than the AC heuristic in the range from 1 to 10 additional registers. The next curve corresponds to the RSS heuristic, which performs noticeably better than the two previous heuristics in the range from 1 to 10 additional registers. This heuristic requires the removal of redundant register edges, which is accomplished in quadratic time if the MinDist relation is available in a lookup table. We notice that all heuristics presented so far schedule approximately 95% of the loops with no more than 10 additional registers. To improve the register requirements further, we must address more aggressively the biconnected components using the up-propagation heuristic.

The last two mostly overlapped curves of Figure 9 correspond to the performance of the UP and 3-UP+RSS heuristics. These two heuristics perform significantly better, finding an optimal stage schedule for 93% and 96% of the loops and scheduling 98% of the loops with no more than 3 and 1 additional registers, respectively. Although these two curves mostly overlap, the 3-UP+RSS heuristic does perform slightly better, with additional register requirements of 95 registers instead of 240 registers summed over the 1289 loops of the benchmark suite. These two heuristics are clearly the best stage schedule heuristics and should definitely be implemented when compiling for machines in which the register requirements of modulo scheduled loops are critical.

	SS	3-UP+RSS	MinBuf	MinReg
avg.	.064 ms	.358 ms	99.8 ms	11.2 sec
max.	.900 ms	4.94 ms	17.5 sec	56.6 min
std. dev.	.083 ms	.463 ms	800. ms	130. sec

Table 2: Computation time on a SPARC-20.

To conclude, the average, the maximum, and the standard deviation of the computation time for some of the stage scheduling heuristics and the optimum (MinReg) stage scheduler are given in Table 2.

6 Conclusion

Modulo scheduling is an efficient technique for exploiting instruction level parallelism, resulting in high performance code, but increased the register requirements. This paper contributes efficient stage scheduling heuristics that reduce the register requirements of a modulo schedule by reassigning some operations to different stages, i.e. by shifting operations by multiples of II cycles. These heuristics address a common shortcoming of current modulo schedulers which tends to result in schedules with increased register requirements due to poor placement of the operations not on the critical path.

We investigated the performance of several stage scheduling heuristics for a benchmark of 1289 loops from the Perfect Club, SPEC-89, and the Livermore Fortran Kernels. Our empirical findings show that the register requirements decrease by 24.9% on average for an optimal stage scheduler with respect to a register-insensitive modulo scheduler. Our best stage scheduling heuristic (3-UP+RSS) decreases the register requirements by 24.6%, achieving on average 99% of the decrease in register requirements obtained by

an optimal stage scheduler over the entire benchmark suite. Our best linear-time stage scheduling heuristic (UP) decreases the register requirements by 24.1%, achieving on average 97% of the decrease in register requirements obtained by an optimal stage scheduler over the entire benchmark suite. Even a simple linear-time stage scheduling heuristic (SS) decreases the register requirements by 19.7%, achieving on average 82% of the optimal decrease in register requirements.

The computational complexity of all these heuristics is vastly less than the exponential complexity of the optimal stage scheduler. The complexity of both the SS and UP stage scheduling heuristics is linear in the number of edges in the dependence graph. Even our best stage scheduling (3-UP+RSS) heuristic is only quadratic in the number of edges in the dependence graph, if provided the MinDist relation.

Acknowledgements

This work was supported in part by the Office of Naval Research under grant number N00014-93-1-0163 and by Hewlett-Packard. The authors would like to thank Santosh Abraham, Sadun Anik, Richard Johnson, Vinod Kathail, Michael Schlansker, and Bob Rau for their many useful suggestions and for providing the input data set.

References

[1] B. R. Rau, C. D. Glaeser, and R. L. Picard. Efficient code generation for horizontal architectures: Compiler techniques and architecture support. *ISCA*, pages 131–139, 1982.

[2] P. Y. Hsu. *Highly Concurrent Scalar Processing*. PhD thesis, U. of Illinois at Urbana-Champaign, 1986.

[3] B. R. Rau. Iterative Modulo Scheduling: An algorithm for software pipelining loops. *MICRO*, pages 63–74, Nov. 1994.

[4] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. *PLDI*, pages 318–328, June 1988.

[5] N. J. Warter, G. E. Haab, and J. W. Bockhaus. Enhanced Modulo Scheduling for loops with conditional branches. *MICRO*, pages 170–179, Dec. 1992.

[6] M. S. Schlansker, V. Kathail, and S. Anik. Height reduction of control recurrences for ILP processors. *MICRO*, pages 40–51, Nov. 1994.

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

[8] B. R. Rau, M. Lee, P. P. Tirumalai, and M. S. Schlansker. Register allocation for software pipelined loops. *PLDI*, pages 283–299, June 1992.

[9] W. Mangione-Smith, S. G. Abraham, and E. S. Davidson. Register requirements of pipelined processors. *ICS*, pages 260–271, July 1992.

[10] R. A. Huff. Lifetime-sensitive modulo scheduling. *PLDI*, pages 258–267, June 1993.

[11] J. Llosa, M. Valero, and E. Ayguade. Bidirectional scheduling to minimize register requirements. *Fifth Workshop on Compilers for Parallel Computers*, pages 534–554, June 1995.

[12] Q. Ning and G. R. Gao. A novel framework of register allocation for software pipelining. *POPL*, pages 29–42, 1993.

[13] Dupont de Dinechin. Simplex scheduling: More than lifetime-sensitive instruction scheduling. *PACT*, 1994.

[14] R. Govindarajan, E. R. Altman, and G. R. Gao. Minimizing register requirements under resource-constrained rate-optimal software pipelining. *MICRO*, pages 85–94, Nov. 1994.

[15] A. E. Eichenberger, E. S. Davidson, and S. G. Abraham. Minimum register requirements for a modulo schedule. *MICRO*, pages 75–84, Nov. 1994.

[16] J. Wang, A. Krall, and M.A. Ertl. Decomposed software pipelining with reduced register requirement. In *PACT*, June 1995.

[17] A. E. Eichenberger, E. S. Davidson, and S. G. Abraham. Optimum modulo schedules for minimum register requirements. *ICS*, pages 31–40, July 1995.

[18] C. Eisenbeis and D. Windheiser. Optimal software pipelining in presence of resource constraints. *PACT*, Aug. 1993.

[19] J. Wang, C. Eisenbeis, M. Jourdan, and B. Su. Decomposed software pipelining: A new perspective and a new approach. In *Int. J. of Parallel Programming*, volume 22, pages 357–379, 1994.

[20] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

[21] M. Berry et al. The Perfect Club Benchmarks: Effective performance evaluation of supercomputers. *The Int. J. of Supercomputer Appl.*, 3(3):5–40, Fall 1989.

[22] J. Uniejewski. SPEC Benchmark Suite: Designed for today's advanced system. *SPEC Newsletter*, Fall 1989.

[23] F. H. McMahon. The Livermore Fortran Kernels: A computer test of the numerical performance range. TR UCRL-53745, Lawrence Livermore Nat. Lab., Livermore, California, 1986.

[24] J. C. Dehnert and R. A. Towle. Compiling for the Cydra 5. In *The J. of Supercomputing*, volume 7, pages 181–227, 1993.

[25] G. R. Beck et al. The Cydra 5 mini-supercomputer: Architecture and implementation. In *The J. of Supercomputing*, volume 7, pages 143–180, 1993.