Unrolling-Based Optimizations for Modulo Scheduling

Daniel M. Lavery Wen-mei W. Hwu

Center for Reliable and High-Performance Computing
University of Illinois
Urbana-Champaign, IL 61801

Abstract

Modulo scheduling is a method for overlapping successive iterations of a loop in order to find sufficient instruction-level parallelism to fully utilize high-issue-rate processors. The achieved throughput modulo scheduled loop depends on the resource requirements, the dependence pattern, and the register requirements of the computation in the loop body. Traditionally, unrolling followed by acyclic scheduling of the unrolled body has been an alternative to modulo scheduling. However, there are benefits to unrolling even if the loop is to be modulo scheduled. Unrolling can improve the throughput by allowing a smaller non-integral effective initiation interval to be achieved. After unrolling, optimizations can be applied to the loop that reduce both the resource requirements and the height of the critical paths. Together, unrolling and unrolling-based optimizations can enable the completion of multiple iterations per cycle in some cases. This paper describes the benefits of unrolling and a set of optimizations for unrolled loops which have been implemented in the IMPACT compiler. The performance benefits of unrolling for five of the SPEC CFP92 programs are reported.

Keywords: *modulo scheduling, software pipelining, optimization, loop unrolling, instruction-level parallelism*

1 Introduction

The scheduling of instructions in loops is of great interest because most programs spend the majority of their execution time in loops. It is often necessary for the scheduler to overlap successive iterations of a loop in order to find sufficient instruction-level parallelism (ILP) to effectively utilize the hardware.

Two classes of loop scheduling schemes have been developed that allow motion of instructions from one iteration to another. The first approach is to unroll the loop body some number of times and then apply a global acyclic scheduling algorithm to the unrolled loop body [1, 2, 3]. This allows the scheduler to overlap the iterations in the unrolled loop body. However, all overlap is lost when the loop-back branch is taken, leaving a long start-up overhead for each iteration of the unrolled body.

The second approach, software pipelining [4, 5, 6], generates code that maintains the overlap of the original loop iterations throughout the execution of the loop. A description of the various approaches to software pipelining is given in [7]. This paper will focus on a class of software pipelining methods called *modulo scheduling* [8]. Modulo scheduling simplifies the generation of overlapped schedules by initiating iterations at a constant rate and by requiring all iterations of the loop to have a common schedule.

The constant interval between the start of successive iterations is called the *initiation interval* (II). A smaller II means higher execution throughput.[1] The *achieved* II depends on:

1. The execution resources, instruction latencies, and registers available in the processor.
2. The instruction types, the dependence pattern, and the register requirements of the computation in the loop body.
3. The quality of the modulo scheduling and register allocation algorithms.

This paper assumes that the processor design is fixed and that high quality modulo scheduling and register allocation algorithms are used. The three characteristics of the computation in the loop body (item 2 above) *are not fixed.* All three can be modified by program transformations. This paper focuses on optimization of the loop

[1]As in the design of hardware pipelines [9], modulo scheduling may increase the latency (the number of cycles needed to complete the first iteration) in order to minimize II. Thus, it is assumed that on average, the loop will execute a sufficient number of iterations to amortize this increase in latency.

body prior to modulo scheduling to reduce the achieved II. Specifically, it describes the advantages of unrolling the loop before modulo scheduling and of performing optimizations which reduce the resource requirements and the height of critical paths in loops.

There are two sources of motivation for this work. First, the II for the loop is restricted to be an integer. If the lower bound on the II computed before scheduling is not an integer, the performance degradation caused by rounding it up to an integer may be reduced by unrolling the loop [8]. A related restriction is that the minimum possible value for II is one. This limits the performance of a modulo scheduled loop to one iteration per cycle. By unrolling the loop and applying optimizations, it is possible to complete multiple iterations per cycle given sufficient execution resources. These restrictions have been known for some time, but the benefits of unrolling prior to modulo scheduling have never been quantified.

The second source of motivation comes from comparisons of modulo scheduling with global acyclic scheduling of an unrolled loop body. Without unrolling prior to modulo scheduling, it is possible for the modulo scheduled loop to perform worse than the acyclicly scheduled loop. The acyclic scheduler is not required to initiate the iterations within the unrolled body at a constant rate nor to generate the same schedule for each of those iterations. Thus, it can achieve an effective II which is not an integer. Additionally, unrolling exposes new opportunities for optimizations which reduce the resource requirements and dependence height. The benefits of unrolling and these unrolling-based optimizations have been quantified for acyclicly scheduled loops [10], but have not been measured for modulo scheduled loops.

This paper describes the benefits of unrolling, and a set of optimizations for unrolled loops which have been implemented in the IMPACT compiler. Unrolling and unrolling-based optimization is applied to the loops in five of the SPEC CFP92 programs and the achieved speedup is measured. The performance benefit of unrolling is also reported individually for each of the 58 loops in those five programs. Statistics which are relevant to modulo scheduling and to unrolling are reported for the benchmarks and their loops.

The paper is organized as follows: Section 2 reviews the background information about the lower bounds on the II and related work. Section 3 describes the benefits of unrolling and the unrolling-based optimizations. The performance results are reported in section 4. A summary and directions for future work are given in section 5.

2 Background

2.1 The Minimum Initiation Interval

It is necessary to choose an initial candidate II before scheduling the instructions. Two lower bounds on II have been developed in the modulo scheduling theory [8], one derived from resource constraints and the other from the constraints imposed by dependence cycles. The maximum of the two is the minimum initiation interval (MII). Scheduling at an II below the MII can be attempted, but will almost surely fail[2] due to lack of resources or failure to meet dependence constraints. In this case, the II is increased and scheduling is attempted again. Choosing the initial II equal to the MII saves scheduling effort. Another important use for the lower bounds on the II is to guide the optimization process [11]. For example, there is no point in doing optimizations to reduce dependence constraints if that bound is already lower than the resource-constrained lower bound.

From the point-of-view of resources, the throughput of a software pipeline is maximized when one of the processor resources is fully utilized. Thus, the resource which is the most heavily used by the loop body determines the resource-constrained lower bound on the II (ResMII). The ResMII is equal to the number of cycles that this resource is used. One way to compute the ResMII is to total up the cycles in which a particular type of resource is used by the instructions in the loop body and divide by the number of that type of resource available in the machine. If the result is not an integer, it is rounded up to an integer. The maximum of this result over all resource types is the ResMII. An alternative algorithm for computing the ResMII is given in [8].

Figure 1 shows the source and assembly code to compute a vector-matrix product. The inner loop is used as an example throughout this paper. The assembly code shown assumes that classic loop optimizations such as induction variable elimination and global variable migration have been performed. Registers r4-r7 and f1-f6 are integer and floating-point registers respectively. For this example, we assume an 8-issue processor with no restrictions on the combination of instructions that may be issued. The issue slots are thus the most heavily used resources resulting in a ResMII of one.

Recurrences also constrain the II. A recurrence is a dependence cycle from an instruction in the loop to the same instruction in a later iteration of the loop. Figure 2 shows the dependence graph for the example loop. The data and control dependences are shown with solid and dashed lines respectively. All anti- and output dependences have been removed assuming that *modulo variable expansion* [12] will be performed after scheduling. Each node is numbered with the id (from Figure 1(b)) of the instruction it represents. Each arc is labeled with two numbers. The first is the minimum delay in cycles required between the start of the two instructions to insure correct execution without interlock stalls. The delays shown are those of the HP PA-RISC PA7100. The second number is the distance which is the number of iterations between the two dependent instructions. Arcs with a distance of 0 are intra-iteration dependences and those with a distance greater than 0 are cross-iteration dependences.

Assume $Delay(c)$ is the sum of the delays of each edge

[2]We say *almost* because in some cases one of the lower bounds is approximate rather than exact [8].

(a) Original Loop

```
  do i = 1,n
    C(i) = 0.0
    do j = 1,m
      C(i) = C(i) + A(j) * B(j,i)
    end do
  end do
```

(b) Assembly Code For Inner Loop

```
    Instr.   Assembly            Register Contents:

L1:   1  f3 = MEM(r8+r4)              f1 = C(i)
      2  f5 = MEM(r2+r4)              f3 = A(j)
      3  f6 = f3 * f5                 f5 = B(j,i)
      4  f1 = f1 + f6                 r8 = &A(1)
      5  r4 = r4 + 4                  r2 = &B(1,i)
      6  ble (r4 r7) L1               r4 = 4*j
                                      r7 = 4*m
                                      r9 = 4*i
```

Figure 1: Example vector-matrix product loop.

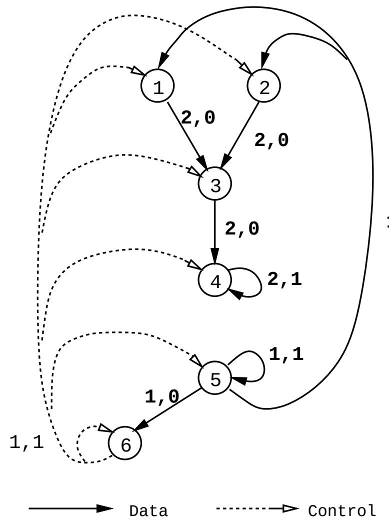


Figure 2: Dependence graph for example loop.

comprising a dependence cycle c and $Distance(c)$ is the number of iterations between the two dependent instances of any instruction in the cycle. Then for all elementary cycles[3] c in the graph, we have the constraint that

$$Delay(c) \leq II * Distance(c)$$

The worst-case constraint among all such cycles determines the recurrence-constrained lower bound on the II (RecMII). Algorithms for computing this bound are given in [8] and [13]. The longest cycle in the example graph is from instruction 4 to itself. The cycle has a delay of 2 and spans from one iteration to the next, resulting in a RecMII of 2. It may appear that there is another recurrence with a delay of two involving instructions 5 and 6, the update of the loop counter and the loop back branch. Such a recurrence would create a RecMII of 2 for every DO loop. In the IMPACT modulo scheduler, the cross-iteration control dependence from instruction 6 to instruction 5 is removed, allowing instruction 5 to be speculatively executed. This is a technique that was proposed for modulo scheduling of WHILE loops [14].

[3]An elementary cycle visits each node in the cycle only once.

2.2 Related Work

There has been extensive prior work on the optimization of loops, some of it targeted directly at software pipelined loops. The work has focused either on reducing the number of instructions (the resource requirement) in the loop body or on changing the dependence pattern to create more ILP. Traditional loop optimizations try to reduce the number and complexity of the instructions in the loop body [15]. These optimizations indirectly reduce the dependence height of a critical path by reducing the number of instructions along the path.

The compiler for the Cydra-5 performed redundant load and store elimination across loop iterations [11]. This optimization reduces the number of memory ports used and reduces the dependence height when a load is on a critical recurrence path. The Cydra-5 compiler also performed *symmetric back-substitution* of data recurrences to reduce dependence height [11]. The compiler for the RS/6000 architecture performs an optimization called *predictive commoning* [16]. This optimization achieves an effect similar to redundant load elimination and common subexpression elimination across loop iterations for loops in which a sequence of values is computed and the value of each member of the sequence, except the first, is computed again in the next iteration. Unrolling is not required for this optimization but can be used to eliminate the copy instructions inserted by the optimization.

Recently, transformations have been proposed which require that the loop be unrolled. *Blocked back-substitution* [17] unrolls the loop b times and reduces the RecMII by a factor of b. Control recurrences within loops can also be accelerated by a factor of b by unrolling the loop b times and applying techniques similar to blocked back-substitution [18]. Loop unrolling is required for these techniques because the code is optimized asymmetrically such that all iterations in the unrolled loop body do not execute the same code.

Optimizations that unroll loops and then reduce the height of dependence chains associated with induction and accumulator variables have been implemented in the IMPACT compiler [10]. These can be viewed as special cases of symmetric back-substitution. These techniques have been evaluated in the context of global acyclic scheduling of the unrolled loop body, but not modulo scheduling.

Unrolling can also enable optimizations which reduce the number of instructions executed per iteration. Compilers which unroll loops before applying a global acyclic scheduling algorithm take advantage of these optimization opportunities [2, 10]. However, the potential benefits of these optimizations for loops that are software pipelined have not been fully explored.

3 Unrolling-Based Optimization

3.1 Loop Unrolling

The requirement that the II be an integer can result in less than full utilization of processor resources, or the allowance of more cycles than necessary for the completion of a recurrence. For example, in Figure 1(b), there are six instructions in the loop body and it was assumed that the processor has eight issue slots. The ResMII is one (rounded up from 0.75) and two issue slots are wasted every cycle. Unrolling allows a smaller non-integral effective ResMII to be achieved. For example, if the loop in Figure 1(b) is unrolled four times, the ResMII for the unrolled loop body is three and the effective ResMII for each iteration within the loop body is 0.75. Unrolling helps to reduce the degradation by creating larger loop bodies which require more resources and a larger ResMII. The larger the ResMII, the smaller the degradation caused by rounding it up to the next integer.

The RecMII will not be an integer if the delay of the limiting cycle is not a multiple of the distance of the cycle. Unrolling helps because it reduces the distance of the recurrence. This makes the RecMII larger, decreasing the degradation caused by rounding it up to the next integer. If the distance of the cycle becomes one, the RecMII becomes an integer. For example, a recurrence with a distance of three becomes a recurrence with a distance of one if the loop is unrolled three or more times.

Figure 3 shows the way the IMPACT compiler unrolls Fortran-style DO-loops. The example loop of Figure 1 has been unrolled 3 times. We use the terminology of [18], and refer to the iterations of the unrolled loop as *major iterations* and the iterations of the original loop as *minor iterations*.

An optimization has already been applied to remove the loop exit branches from the unrolled loop body. A simple check is done before the loop (and for each major iteration) to ensure that there are at least three more minor iterations to be executed. Two extra copies of the original loop body are inserted after the loop at label L2. These copies are executed when the trip count is not a multiple of 3. If the loop is unrolled u times, there are $u-1$ copies of the original loop body inserted after the unrolled loop. If the code expansion is too great, the remainder iterations can be re-rolled into a loop at a cost of lower performance for the those iterations.

This type of unrolling has two benefits. First, the number of branch unit resources required for each major iteration is reduced from 3 to 1. Second, the control dependences associated with the exit branches are removed, allowing the possibility of executing more than one minor

```
     r3 = r7 - 8
     bgt (r4 r3) L2
L1:  f3 = MEM(r8+r4)
     f5 = MEM(r2+r4)
     f6 = f3 * f5
     f1 = f1 + f6
     r4 = r4 + 4
     f3 = MEM(r8+r4)
     f5 = MEM(r2+r4)
     f6 = f3 * f5
     f1 = f1 + f6
     r4 = r4 + 4
     f3 = MEM(r8+r4)
     f5 = MEM(r2+r4)
     f6 = f3 * f5
     f1 = f1 + f6
     r4 = r4 + 4
     ble (r4 r3) L1
     bgt (r4 r7) L3
L2:  f3 = MEM(r8+r4)
     f5 = MEM(r2+r4)
     f6 = f3 * f5
     f1 = f1 + f6
     r4 = r4 + 4
     bgt (r4 r7) L3
     f3 = MEM(r8+r4)
     f5 = MEM(r2+r4)
     f6 = f3 * f5
     f1 = f1 + f6
     r4 = r4 + 4
L3:
```

Figure 3: Example loop after unrolling three times.

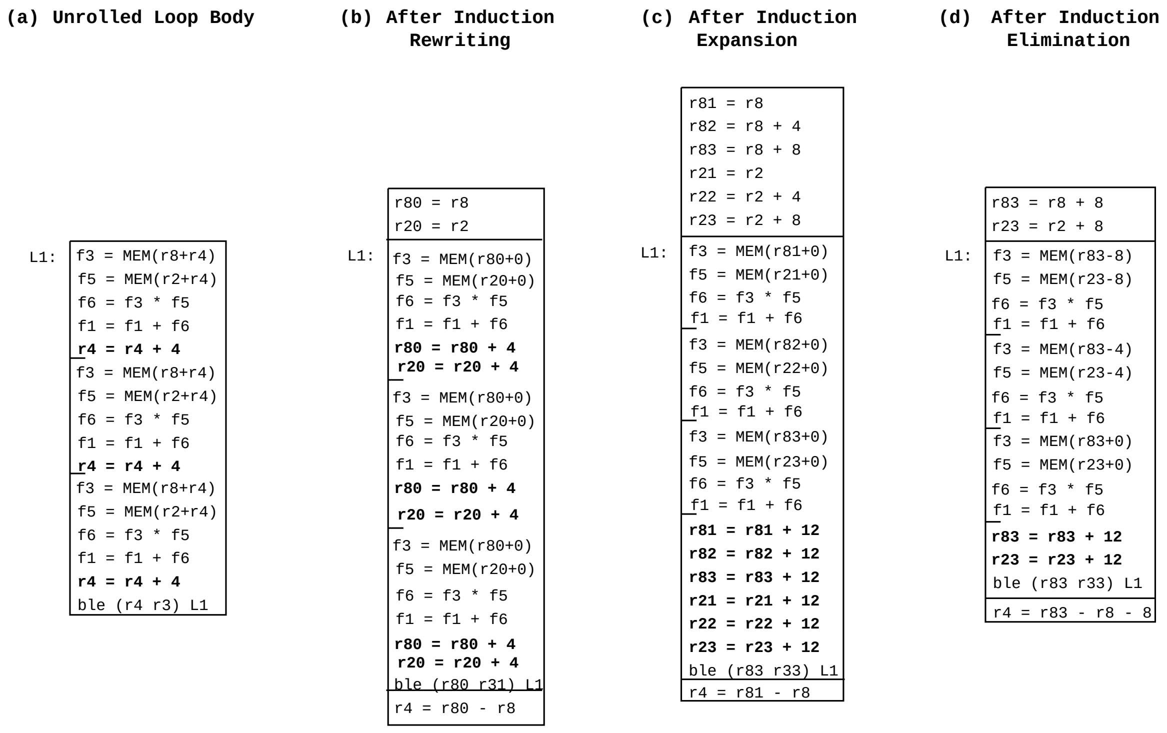
iteration per cycle (resource permitting) without speculation. This can also be viewed as control height reduction [18] where the conditions under which the minor iterations execute have been collapsed into the single check to see if there are at least u minor iterations remaining.

II now refers to the initiation interval for the major iterations. We define II_{eff}, the effective initiation interval for the minor iterations, to be II/u. For the example loop, the $ResMII_{eff}$ falls from 1 to 0.66 as a result of unrolling and exit branch removal.

3.2 IMPACT Unrolling-based Optimizations

This section describes the remaining unrolling-based optimizations done by the IMPACT compiler [10] and their effect on the MII. Figure 4 shows the effect of induction variable optimizations applied to the unrolled loop body. In Figure 4(a), the unrolled loop body without exit branches has been extracted from the code in Figure 3. The induction variable increment instructions have been highlighted.

In the original loop body, the vector and matrix are ad-


(a) Unrolled Loop Body
(b) After Induction Rewriting
(c) After Induction Expansion
(d) After Induction Elimination

```
(a) Unrolled Loop Body

L1: f3 = MEM(r8+r4)
    f5 = MEM(r2+r4)
    f6 = f3 * f5
    f1 = f1 + f6
    r4 = r4 + 4
    f3 = MEM(r8+r4)
    f5 = MEM(r2+r4)
    f6 = f3 * f5
    f1 = f1 + f6
    r4 = r4 + 4
    f3 = MEM(r8+r4)
    f5 = MEM(r2+r4)
    f6 = f3 * f5
    f1 = f1 + f6
    r4 = r4 + 4
    ble (r4 r3) L1
```

```
(b) After Induction Rewriting

    r80 = r8
    r20 = r2
L1: f3 = MEM(r80+0)
    f5 = MEM(r20+0)
    f6 = f3 * f5
    f1 = f1 + f6
    r80 = r80 + 4
    r20 = r20 + 4
    f3 = MEM(r80+0)
    f5 = MEM(r20+0)
    f6 = f3 * f5
    f1 = f1 + f6
    r80 = r80 + 4
    r20 = r20 + 4
    f3 = MEM(r80+0)
    f5 = MEM(r20+0)
    f6 = f3 * f5
    f1 = f1 + f6
    r80 = r80 + 4
    r20 = r20 + 4
    ble (r80 r31) L1
    r4 = r80 - r8
```

```
(c) After Induction Expansion

    r81 = r8
    r82 = r8 + 4
    r83 = r8 + 8
    r21 = r2
    r22 = r2 + 4
    r23 = r2 + 8
L1: f3 = MEM(r81+0)
    f5 = MEM(r21+0)
    f6 = f3 * f5
    f1 = f1 + f6
    f3 = MEM(r82+0)
    f5 = MEM(r22+0)
    f6 = f3 * f5
    f1 = f1 + f6
    f3 = MEM(r83+0)
    f5 = MEM(r23+0)
    f6 = f3 * f5
    f1 = f1 + f6
    r81 = r81 + 12
    r82 = r82 + 12
    r83 = r83 + 12
    r21 = r21 + 12
    r22 = r22 + 12
    r23 = r23 + 12
    ble (r83 r33) L1
    r4 = r81 - r8
```

```
(d) After Induction Elimination

    r83 = r8 + 8
    r23 = r2 + 8
L1: f3 = MEM(r83-8)
    f5 = MEM(r23-8)
    f6 = f3 * f5
    f1 = f1 + f6
    f3 = MEM(r83-4)
    f5 = MEM(r23-4)
    f6 = f3 * f5
    f1 = f1 + f6
    f3 = MEM(r83+0)
    f5 = MEM(r23+0)
    f6 = f3 * f5
    f1 = f1 + f6
    r83 = r83 + 12
    r23 = r23 + 12
    ble (r83 r33) L1
    r4 = r83 - r8 - 8
```

Figure 4: Example loop after induction variable optimization.

dressed using two different bases (r8 and r2 respectively) and a common offset (r4). This reduces the number of induction instructions for good performance in the original loop. In Figure 4(b), the address calculations have been rewritten in preparation for later induction variable elimination (described below). The objective of the rewriting is to specify each memory address using only one operand. This frees up the other address source operand in each load for use by induction variable elimination. In the unrolled loop, after the rewriting, the vector and the matrix are each addressed using a separate induction variable (r80 and r20 respectively). This rewriting increases the number of induction variable increment instructions in the loop, so it is only done if there will be a net gain after later induction variable elimination.

In Figure 4(c), induction variable expansion [10] has been applied to the loop. For the each induction variable (r80 and r20), 3 temporary induction variables have been created, one for each definition of the original induction variables. The new induction variables are now incremented by 3 times the original increment. They are initialized to the initial series of values in the preheader.

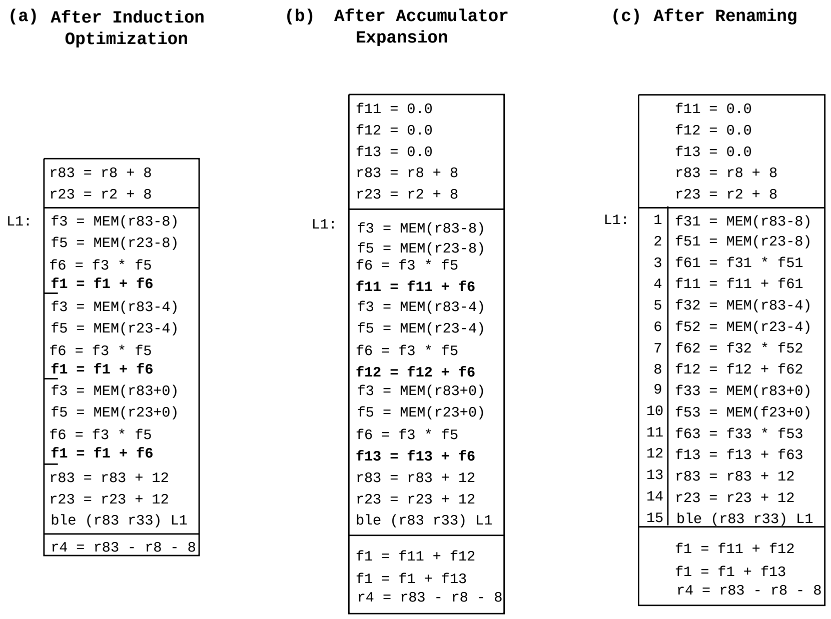
In the example, induction variable expansion reduces the delay for the cycle involving the 3 increments of r80 from 3 to 1. Induction variable expansion is a special case of symmetric back-substitution [17] where the reduction is a simple addition of a loop invariant.

Figure 4(d) shows the loop after the application of induction variable elimination. Elimination of induction variable r22 is done as follows. First, r22 is rewritten in terms of r23: r22 = r23 - 4. Then this definition of r22 is combined with the load which uses r22 in the next iteration [19]. After combining, there are no further uses of r22 and its defining instruction can be removed.

Induction variable elimination significantly reduces the number of integer ALU and issue slot resources required by making use of the separate effective address addition available in most load/store units and the two address operands of the load/store instructions. For the example loop, the number of induction instructions has been reduced from 3 (in Figure 4a) to 2 (in Figure 4d). If the loop is unrolled 8 times, the number of induction instructions is reduced from 8 to 2. Loop unrolling is required for this type of induction variable elimination because the minor loop iterations are no longer identical.

Figure 5 shows the effect of two more optimizations. In Figure 5(b), accumulator variable expansion has been applied [10]. In the original loop, f1 is an accumulator variable. Three temporary accumulators (f11, f12, f13) have been created, one for each definition of the original accumulator. The temporary accumulators are summed after the loop.

In the example, accumulator variable expansion reduces the delay for the cycles involving f1 by a factor of 3. Accumulator variable expansion is also called interleaved reduction [2] and riffling of reductions [11]. Since accumulator



```
(a) After Induction
    Optimization

    r83 = r8 + 8
    r23 = r2 + 8
L1: f3 = MEM(r83-8)
    f5 = MEM(r23-8)
    f6 = f3 * f5
    f1 = f1 + f6
    f3 = MEM(r83-4)
    f5 = MEM(r23-4)
    f6 = f3 * f5
    f1 = f1 + f6
    f3 = MEM(r83+0)
    f5 = MEM(r23+0)
    f6 = f3 * f5
    f1 = f1 + f6
    r83 = r83 + 12
    r23 = r23 + 12
    ble (r83 r33) L1
    r4 = r83 - r8 - 8
```

```
(b) After Accumulator
    Expansion

    f11 = 0.0
    f12 = 0.0
    f13 = 0.0
    r83 = r8 + 8
    r23 = r2 + 8
L1: f3 = MEM(r83-8)
    f5 = MEM(r23-8)
    f6 = f3 * f5
    f11 = f11 + f6
    f3 = MEM(r83-4)
    f5 = MEM(r23-4)
    f6 = f3 * f5
    f12 = f12 + f6
    f3 = MEM(r83+0)
    f5 = MEM(r23+0)
    f6 = f3 * f5
    f13 = f13 + f6
    r83 = r83 + 12
    r23 = r23 + 12
    ble (r83 r33) L1
    f1 = f11 + f12
    f1 = f1 + f13
    r4 = r83 - r8 - 8
```

```
(c) After Renaming

         f11 = 0.0
         f12 = 0.0
         f13 = 0.0
         r83 = r8 + 8
         r23 = r2 + 8
L1:  1   f31 = MEM(r83-8)
     2   f51 = MEM(r23-8)
     3   f61 = f31 * f51
     4   f11 = f11 + f61
     5   f32 = MEM(r83-4)
     6   f52 = MEM(r23-4)
     7   f62 = f32 * f52
     8   f12 = f12 + f62
     9   f33 = MEM(r83+0)
    10   f53 = MEM(f23+0)
    11   f63 = f33 * f53
    12   f13 = f13 + f63
    13   r83 = r83 + 12
    14   r23 = r23 + 12
    15   ble (r83 r33) L1
         f1 = f11 + f12
         f1 = f1 + f13
         r4 = r83 - r8 - 8
```

Figure 5: Example loop after accumulator expansion and renaming.

variable expansion reassociates the terms in the accumulation, which may change the results of floating-point accumulations, its use is under user control.

In Figure 5(c), variable renaming has been applied to produce the final optimized version of the loop. The 3 iterations of the loop are now independent. Overall, the $RecMII_{eff}$ for the example loop was reduced by a factor of 3: from 2 to 0.66 for the 8-issue processor. The $ResMII_{eff}$ was reduced from 1 to 0.66 due to the removal of the loop exit branches described earlier. Removal of both data and control dependences and reduction of resources were all necessary to achieve these improvements in the MII_{eff}. For example, without the removal of control dependences and the reduction in resources, the MII_{eff} is limited to 1 for this loop. If this example loop is unrolled 8 times, an MII_{eff} of 0.25 can be achieved given sufficient resources. This is 8 times the performance of the original loop in Figure 1!

Although not shown in the running example, unrolling also allows redundant load and store elimination, common subexpression elimination, and copy propagation across minor iterations. These optimizations, along with accumulator and induction variable expansion, can be done without unrolling if the compiler representation has support for *expanded virtual registers* (EVRs) [20]. If the compiler supports EVRs, but the architecture does not have support for rotating registers [21], the loop must still be unrolled to allow modulo variable expansion. Even in this case, EVRs allow the optimizations to be performed without having to first decide how much to unroll the loop. For compilers which do not support EVRs, unrolling is the only way to do load and store elimination and common subexpression elimination across iterations without introducing copy instructions and to do accumulator and induction variable expansion. However, for any compiler, the exit branch removal and induction variable elimination described in this paper require unrolling, as does blocked back-substitution.

4 Experimental Results

In this section, we report experimental results on the importance of the unrolling-based optimizations for software pipelined loops. The results are obtained using the IMPACT compiler and benchmarks from the SPEC CFP92 suite. The data dependence analysis performed includes both the sophisticated array dependence analysis performed by the Omega Test [22], as well as interprocedural analysis of pointer aliases.

In the back end, the classic optimizations and the unrolling-based optimizations described earlier are performed. Modulo scheduling is performed before prepass acyclic scheduling and global register allocation. Modulo variable expansion is done to rename overlapping lifetimes.

The modulo scheduler is a library that can be called from any of IMPACT's code generators. It uses a machine description system [23] to obtain necessary information on instruction latencies and resource usage. The modulo scheduler has been used to pipeline loops for high issue rate versions of the PA-RISC and SPARC architectures. Currently, only single-basic-block DO and WHILE loops without function calls or complex recurrences are pipelined. Loops with a single-node recurrence, a dependence from an instruction to itself, are pipelined.

Table 1 shows the characteristics of 11 of the SPEC CFP92 benchmarks. The *Modulo Sched.* column shows the percentage of the dynamic instructions which are in software pipelined loops. The rest of the columns show the percentage of dynamic instructions within loops that were not software pipelined for the various reasons described below.

Table 1: Benchmark characteristics.

Program	Modulo Sched.	Not Modulo Scheduled Due To				
		Mult. Block	Unstr. Loop	Func Call	Scalar Dep.	Array Dep.
spice2g6	1.1	13.3	60.5	9.7	—	—
doduc	7.3	65.4	0	47.7	—	61.7
mdljdp2	9.6	82.3	0	12	—	86.1
wave5	50.9	17.9	3.2	4.1	—	16.0
tomcatv	90	9.8	0	0	9.8	0
ora	0	96.5	96.5	96.5	—	—
alvinn	99.5	0	0	0	0	0
ear	74.8	20.1	0	0	0	2.7
swm256	99.9	0	0	0	0	0
su2cor	80.3	10.5	0	9.9	6.3	0
hydro2d	56.9	42.2	0	0	0	0

There can be several reasons why a particular loop is not software pipelined, so the rows may add up to more than 100%. Also, only the percentage of instructions in inner loops is shown in the table. The percentages calculated using the total dynamic instructions for the program. Thus, the rows can add up to less than 100% also. The remaining time is spent in acyclic code and outer loops.

Mult. Block means that the loop body consisted of multiple basic blocks. *Unstr. Loop* means that the loop was not a structured DO loop, but rather an implicit loop generated using gotos. Such loops are detected, but not analyzed for dependences in our front end. *Scalar Dep.* means that there was a cross-iteration dependence for a scalar variable. *Array Dep.* means that there was a cross-iteration dependence between accesses to array elements.

A dash in the table means that the percentage of the dynamic instructions that are in loops of that type is unknown. Since unstructured loops are not analyzed for dependences, the dependence statistics are unknown for benchmarks which spent a significant amount of time in such loops (such as *spice* and *ora*). The statistics on dependences are currently reported in such a way that if there is a cross-iteration array dependence in the loop, one cannot tell from the report if there was also a cross-iteration scalar dependence. Thus, the number of scalar dependences in *doduc, mdljdp2*, and *wave5* is unknown.

Multiple-basic-block loops are by far the largest category of non-pipelined loops, followed by loops with function calls, unstructured loops, and loops with cross-iteration array dependences. Cross-iteration scalar dependences do not appear to be a large problem so far. Note however, that single-node recurrences associated with accumulator and induction variables are handled by our modulo scheduler and so contribute to the *Software Pipelined* column.

4.1 Benchmark Results

The five benchmarks with the highest percentage of software pipelined loops (*tomcatv, alvinn, ear, swm256*, and *su2cor*) were used to measure the performance benefit of the unrolling-based optimizations. The target processors are multiple issue processors with issue rates between 4 and 16. There are no restrictions on the combination of instructions that may be issued. All speedups are reported over a base single-issue processor. All processors are assumed to have 64 integer registers and 64 floating-point registers. In some cases, the results for 128 floating-point registers are also shown. The latencies used are those of the HP PA-RISC PA7100 processor. A 100% cache hit rate is assumed.

For the multiple issue processors, code is generated two ways, once with and once without the unrolling-based optimizations for the software pipelined loops. In both cases, the loops that are not software pipelined are unrolled and optimized.

The execution time of the programs is calculated using scheduler cycle counts for each basic block and profile information. The benchmarks are profiled after all transformations to insure accuracy. The profiling is done by instrumenting the target (virtual) processor's assembly code and then emulating it on an HP Series 700 workstation. This execution produces benchmark output which is used to verify the correctness of the code transformations.

Figure 6 shows the speedup for each target processor with and without the unrolling-based optimizations for software pipelined loops over a single-issue base processor. For the base processor, loops are not unrolled or software pipelined. The software pipelined loops in *alvinn, ear*, and *swm256* all required less than 64 integer and 64 floating-point registers. However for *tomcatv*, there was a large amount of spilling for the 12 and 16 issue processors, resulting in performance degradation. The IMPACT module scheduler can schedule loops either from the top down or the bottom up. The results shown are for bottom up scheduling. With top down scheduling, the spilling and performance degradation for *tomcatv* is much worse. The loops in these floating-point benchmarks tend to have dependence graphs where the nodes have more incoming dependence edges than outgoing edges (usually only 1 outgoing edge). This creates more opportunities for a greedy top down scheduler to schedule instructions too early (and increase register pressure) than for a bottom up scheduler to schedule instructions too late. The white bars in the figure show the performance of *tomcatv* with 128 floating-point registers. The benchmark *su2cor* also benefits somewhat from additional floating-point registers.

The unrolling-based optimizations produce tremendous performance improvement for *alvinn*. In this benchmark the loops are small and similar to the example loop of Figure 1. Using the PA7100 latencies, single-node recurrences for floating-point accumulator variables impose a RecMII of two, as shown in Figure 2. Without unrolling, the maximum performance achievable for any loop, even

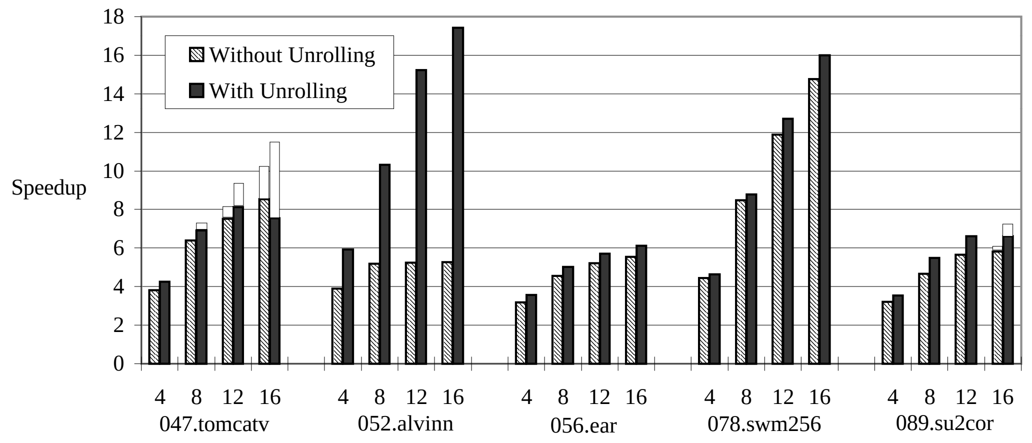


Figure 6: Speedup over single issue processor with and without unrolling.

with infinite resources, is one iteration every two cycles for loops with floating-point accumulator variables, and one iteration per cycle for loops without floating-point accumulator variables. For *alvinn*, the 8-issue processor can support the maximum rate of execution, so without unrolling there is no further speedup for the higher issue rate processors. With unrolling, the loops in *alvinn* execute at a rate of multiple iterations per cycle for the higher issue rate processors.

The unrolling based optimizations improve the performance of *swm256, ear, tomcatv*, and *su2cor* by 8 to 21 percent for the 16 issue processor and 4 to 18 percent for the 8 issue processor. In these programs, the frequently executed loops are larger. In larger loops, induction and branch instructions are a smaller percentage of the total instructions, so the optimizations which reduce them have a smaller effect. Because the modulo scheduler handles only single-node recurrences, the largest RecMII calculated for these benchmarks was two. Larger loops tend to have a ResMII which is larger than this so they do not benefit as much from the height reduction achieved by accumulator variable expansion. Finally, the degradation caused by rounding the MII up to the next integer is not as severe in larger loops. Even so, the unrolling-based optimizations do provide a significant performance improvement for these benchmarks.

The effect of Amdahl's law is visible in Figure 6. *alvinn* and *swm256* have the highest percentage of their execution time in software pipelined loops (practically 100%) and they show the highest speedup over the base processor. *ear* and *su2cor* have the lowest percentage of execution time in software pipelined loops and they have the lowest speedup over the base processor.

4.2 Individual Loop Results

In the five benchmarks, a total of 58 loops were software pipelined. Figure 7 shows the speedup due to the unrolling-based optimizations for each of the 58 loops. The speedups are for the 16-issue machine with unrolling over the 16-issue machine without unrolling. The speedup is calculated by dividing the *achieved* II for the original loop body by the *achieved* II_{eff} for the unrolled loop body.

The individual loop speedups are generally better than the full benchmark speedups. There are two reasons for this. First for *tomcatv, su2cor, and ear*, 10-20% of the execution time is spent in loops which are not software pipelined, limiting the applicability of the techniques described in this paper. Second, the smaller loops have the best speedup, but tend to be less frequently executed (*alvinn* is an exception). The small loops, for example, tend to be initialization loops.

Table 2 summarizes some of the characteristics of the software pipelined loops. The column marked *Original* shows the statistics for the original loop body. A dash is used to indicate that a measurement does not apply to the original loop body. The column marked *Unrolled* shows the statistics for the unrolled loop body.

Table 2: Loop statistics.

Measurement	Statistic	Original	Unrolled
Static Instructions	Minimum	5	14
	Maximum	76	137
	Mean	22	65
Unroll Amount	Minimum	—	1
	Maximum	—	8
	Mean	—	6.7
	Median	—	4
MII_{eff}	Minimum	1	0.25
	Maximum	5	5
	Mean	1.9	1.3
Speedup	Minimum	—	1
	Maximum	—	5.33
	Mean	—	1.7

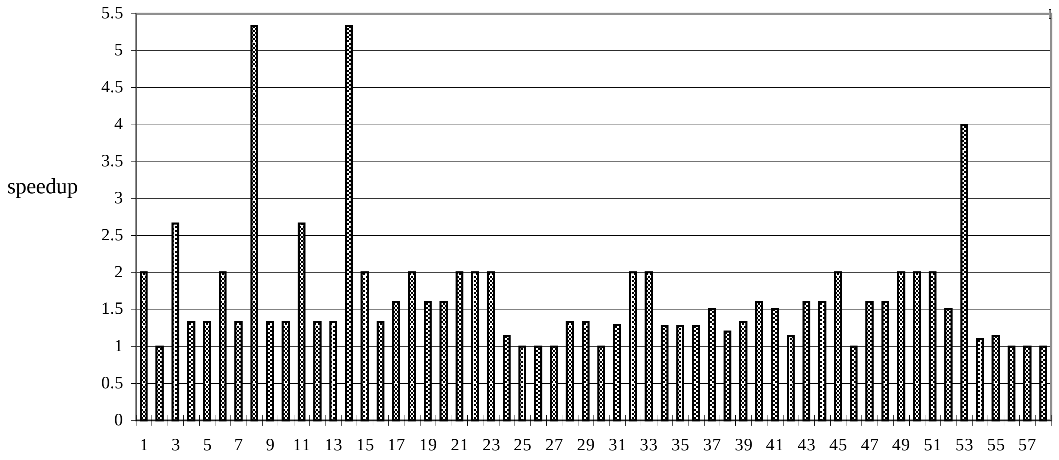


Figure 7: Individual loop speedup. Unrolled over non-unrolled.

For the majority of the loops, the number of instructions in the original loop is fairly small. After unrolling, there are few very large loops. The amount of unrolling is moderate, with a median of 4 times. The optimizer in the IMPACT compiler uses profile information, (the average trip count) and code size to determine the unroll amount. For these floating-point benchmarks, the trip counts are high and the unroll amount is determined by code expansion considerations.

The optimizer generally unrolls the medium to small size loops 8 to 16 times when the loop is scheduled by the acyclic scheduler. For software pipelined loops, only one half that amount of unrolling is done. There are two reasons for this. First, software pipelining creates additional code expansion because of modulo variable expansion and because of the generation of the prologue and epilogue. Therefore, it is desirable to keep the degree of unrolling moderate. Second, the modulo scheduler does not need as much unrolling because it maintains the overlap of iterations across the loop back edge. Investigation of more sophisticated methods for determining the unroll amount is part of the future work.

The minimum and mean MII_{eff} are improved with unrolling as expected. The *Speedup* rows summarize the individual loop speedup results shown earlier. For some loops there is no improvement in the MII_{eff} after unrolling, which can happen when the original loop was too big to be unrolled at all. This case occurs for the largest and most important loop (in terms of execution time) in *tomcatv*.

Performance also may not improve if the resources of the processor are fully or nearly fully utilized by the original loop body. There are several examples in the benchmarks where the original loop has 15 or 16 instructions. For these loops, the reduction in resources due to the optimizations is not enough to reduce the MII_{eff} for the 8 and 16 issue processors. This condition can be detected during optimization when the ResMII is equal to or slightly less than an integer value. If many copies of the most heavily used resource exist, then a large degree of unrolling and optimization would be necessary to reduce the ResMII. Also for these loops, the ResMII is equal to or just slightly less than an integer, so the degradation in rounding up to an integer is not large.

5 Conclusion

5.1 Summary

This paper described a set of unrolling-based optimizations which reduce resource requirements and the height of critical paths in software pipelined loops. Unrolling is the only way to reduce the the effective number of loop-back branches executed per iteration and to allow optimizations which asymmetrically optimize the loop iterations. Unrolling also helps to reduce the degradation caused by rounding the MII up to the nearest integer.

The paper reported the performance improvement achieved by applying unrolling and unrolling-based optimizations to five SPEC CFP92 benchmarks. Speedup of more than 200% was observed for *alvinn*. This benchmark spends all of its time executing small loops. On a high issue rate processor, multiple iterations of the loops can be executed per cycle. This can only be achieved by unrolling the loop. Small loops also benefit the most from the reduction in branch instructions and induction operations. The other benchmarks contained a wider variety of loops and attained speedups between 9 and 22 percent.

5.2 Future Work

Future work includes looking at methods to control the optimizations such that all the constraints on the MII are balanced. For example, there is no point in doing optimizations which reduce the RecMII if it is already less than the ResMII and vice versa. As the ResMII and the RecMII are reduced, the number of physical processor registers can impose a third constraint on the MII. As more parallelism is exploited, more simultaneously live values are generated, requiring more registers [24, 25]. If more simultaneously live values exist than physical registers, spill code must be added and can significantly increase the achieved II of the loop. In this case, it may be possible to achieve a better final II by increasing the candidate II and attempting to schedule the original loop body again [26].

If a lower bound on the loop's final register requirement for a given II were available, it would be useful during both optimization and scheduling. During optimization it could be used to stop optimization before excessive register pressure is generated. During scheduling, the candidate II's for which the lower bound is less than the number of physical registers could be ruled out. A lower bound on the average number of simultaneously live values was reported by Huff [13]:

$$MinAvg = \left\lceil \frac{\sum_v MinLT(v)}{II} \right\rceil$$

where v is a loop variant, a value that is written in the loop. MinLT is a lower bound on the length of the lifetime of v and is a function of the processor latencies and II. A register-constrained lower bound on the II (RegMII) can be computed iteratively by starting with the maximum of ResMII and RecMII and incrementing II until MinAvg plus the number of loop invariants (read-only values) is less than the number of physical processor registers. How close MinAvg is to the actual number of registers required during allocation depends on how well the scheduler optimizes the register lifetimes and on the quality of the register allocator. For Huff's scheduler, and a Cydra-5-like processor model, MinAvg was reasonably close to the maximum number of simultaneously live values [13]. More research needs to be done to determine how tight this bound is for our scheduler and processor models and how effectively it can be used to control optimizations and to save scheduling effort.

A fundamental issue with unrolling is determining the optimal number of times to unroll. More unrolling can result in a greater reduction in resource usage and dependence height. However, the degree of unrolling should be limited if the register pressure becomes too high or if there are diminishing returns. Analytical and empirical methods to determine the amount to unroll deserve further investigation. Another consideration is limiting code expansion. Future work may measure the code expansion due to the unrolling-based optimization and investigate methods to control it.

Acknowledgments

The IMPACT modulo scheduler and the underlying research for this paper have benefited from discussions with Bob Rau, Mike Schlansker, and Scott Mahlke at HP Labs. Scott Mahlke and John Gyllenhaal implemented the unrolling-based optimizations. The authors would like to thank Dave Gallagher and Grant Haab for proofreading the initial and final versions of this paper respectively. The authors would also like to thank all the members of the IMPACT research group and the anonymous referees whose comments and suggestions helped to improve the quality of this paper significantly.

This research has been supported by the National Science Foundation (NSF) under grant MIP-9308013, Joint Services Engineering Programs (JSEP) under Contract N00014-90-J-1270, Intel Corporation, the AMD 29K Advanced Processor Development Division, Hewlett-Packard, SUN Microsystems, NCR and the National Aeronautics and Space Administration (NASA) under Contract NASA NAG 1-613 in cooperation with the Illinois Computer laboratory for Aerospace Systems and Software (ICLASS).

References

[1] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Transactions on Computers*, vol. c-30, pp. 478–490, July 1981.

[2] 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," *The Journal of Supercomputing*, vol. 7, pp. 51–142, January 1993.

[3] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The Superblock: An effective technique for VLIW and superscalar compilation," *The Journal of Supercomputing*, vol. 7, pp. 229–248, January 1993.

[4] 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 20th Annual Workshop on Microprogramming and Microarchitecture*, pp. 183–198, October 1981.

[5] K. Ebcioglu and T. Nakatani, "A new compilation technique for parallelizing loops with unpredictable branches on a VLIW architecture," in *Languages and Compilers for Parallel Computing*, pp. 213–229, 1989.

[6] A. Aiken and A. Nicolau, "Optimal loop parallelization," in *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pp. 308–317, June 1988.

[7] B. R. Rau and J. A. Fisher, "Instruction-level parallel processing: History, overview, and perspective," *Journal of Supercomputing*, vol. 7, pp. 9–50, January 1993.

[8] B. R. Rau, "Iterative modulo scheduling: An algorithm for software pipelining loops," in *Proceedings of the 27th International Symposium on Microarchitecture*, pp. 63–74, December 1994.

[9] J. H. Patel and E. S. Davidson, "Improving the throughput of a pipeline by insertion of delays," in *Proceedings of the 3rd Annual Symposium on Computer Architecture*, pp. 159–164, January 1976.

[10] S. A. Mahlke, W. Y. Chen, J. C. Gyllenhaal, W. W. Hwu, P. P. Chang, and T. Kiyohara, "Compiler code transformations for superscalar-based high-performance systems," in *Proceedings of Supercomputing '92*, pp. 808–817, November 1992.

[11] J. C. Dehnert and R. A. Towle, "Compiling for the Cydra 5," *The Journal of Supercomputing*, vol. 7, pp. 181–227, January 1993.

[12] M. S. Lam, "Software pipelining: An effective scheduling technique for VLIW machines," in *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pp. 318–328, June 1988.

[13] R. A. Huff, "Lifetime-sensitive modulo scheduling," in *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pp. 258–267, June 1993.

[14] P. Tirumalai, M. Lee, and M. Schlansker, "Parallelization of loops with exits on pipelined architectures," in *Proceedings of Supercomputing '90*, November 1990.

[15] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools.* Reading, MA: Addison-Wesley, 1986.

[16] K. O'Brien, B. Hay, J. Minish, H. Schaffer, B. Schloss, A. Shepherd, and M. Zaleski, "Advanced compiler technology for the RISC System/6000 architecture," in *IBM RISC System/6000 Technology*, 1990.

[17] M. Schlansker and V. Kathail, "Acceleration of first and higher order recurrences on processors with instruction level parallelism," in *Proceedings of Languages and Compilers for Parallel Computing, 6th International Workskop*, August 1993.

[18] M. Schlansker, V. Kathail, and S. Anik, "Height reduction of control recurrences for ILP processors," in *Proceedings of the 27th International Symposium on Microarchitecture*, pp. 40–51, December 1994.

[19] T. Nakatani and K. Ebcioglu, "Combining as a compilation technique for VLIW architectures," in *Proceedings of the 22nd International Workshop on Microprogramming and Microarchitecture*, pp. 43–55, September 1989.

[20] B. R. Rau, "Data flow and dependence analysis for instruction-level parallelism," in *Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing*, pp. 236–250, 1992.

[21] J. C. Dehnert, P. Y. 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*, pp. 26–38, April 1989.

[22] W. Pugh, "A practical algorithm for exact array dependence analysis," *Communications of the ACM*, vol. 35, pp. 102–114, August 1992.

[23] J. C. Gyllenhaal, "A machine description language for compilation," Master's thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1994.

[24] W. Mangione-Smith, S. G. Abraham, and E. Davidson, "Register requirements of pipelined processors," in *Proceedings of the International Conference on Supercomputing*, pp. 260–271, 1992.

[25] P. P. Chang, D. M. Lavery, S. A. Mahlke, W. Y. Chen, and W. W. Hwu, "The importance of prepass code scheduling for superscalar and superpipelined processors," *IEEE Transactions on Computers*, vol. 44, pp. 353–370, March 1995.

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