Improving Software Pipelining With Unroll-and-Jam*

Steve Carr
Chen Ding
Philip Sweany
Department of Computer Science
Michigan Technological University
Houghton MI 49931-1295

Abstract

To take advantage of recent architectural improvements in microprocessors, advanced compiler optimizations such as software pipelining have been developed [1, 2, 3, 4]. Unfortunately, not all loops have enough parallelism in the innermost loop body to take advantage of all of the resources a machine provides. Unroll-and-jam is a transformation that can be used to increase the amount of parallelism in the innermost loop body by making better use of resources and limiting the effects of recurrences [5, 6].

In this paper, we demonstrate how unroll-and-jam can significantly improve the initiation interval in a software-pipelined loop. Improvements in the initiation interval of greater than 40% are common, while dramatic improvements of a factor of 5 are possible.

1 Introduction

Over the past decade, the computer industry has realized dramatic improvements in the power of microprocessors. These gains have been achieved both by cycle-time improvements and by architectural innovations like multiple instruction issue and pipelined functional units. As a result of these improvements, today's microprocessors can perform more operations per machine cycle than their predecessors.

To take advantage of these architectural improvements, advanced compiler optimizations such as software pipelining have been developed [1, 2, 3, 4]. Software pipelining allows iterations of a loop to be overlapped with one another in order to take advantage of the maximum parallelism in a loop body. Unfortunately, not all loops have enough parallelism in the innermost loop body to take advantage of all of the resources a machine provides. Parallelism is normally inhibited by either inner-loop recurrences, or by a mismatch between the resource requirements of a loop and the resources provided by the target architecture.

Unroll-and-jam is a transformation that can be used to increase the parallelism in the innermost loop body [5, 6]. Unroll-and-jam can reduce the number of memory operations that need to be issued per floating-point operation in order to alleviate resource constraint problems. In addition, unroll-and-jam creates copies of inner-loop recurrences that are parallel with all other copies [5]. These two effects increase the parallelism available to a software pipelining algorithm.

This paper measures the effectiveness of unroll-and-jam at improving the initiation interval for software-pipelined loops. In our experiments, unroll-and-jam is performed by a Fortran source-to-source transformer called Memoria [7] that is based upon the Para-Scope programming environment [8]. Software pipelining is performed by a retargetable compiler for ILP (Instruction-Level Parallel) architectures, called Rocket [9]. We present experimental evidence that suggests that Memoria can be quite effective at improving the initiation interval generated by Rocket.

2 Software Pipelining

While local and global instruction scheduling can, together, exploit considerable parallelism for non-loop code, to best exploit instruction-level parallelism within loops requires software pipelining. Software pipelining can generate efficient schedules for loops by overlapping execution of operations from different iterations of the loop. This overlapping of operations is analogous to hardware pipelines where speed-up is achieved by overlapping execution of different op-

*This research is partially supported by NSF Grant CCR-9409341 and NSF Grant CCR-9308348. Email addresses: *carr@cs.mtu.edu*, *chding@cs.mtu.edu*, and *sweany@cs.mtu.edu*.

erations. In this section, we first motivate software pipelining with an example and then look at the software pipelining technique used in Rocket .

2.1 Software Pipelining Example

To illustrate the potential of software pipelining, we'll investigate the problem of computing 4x4 matrix products on a machine that has separate add and multiply units that operate concurrently. Assume that in our simple machine an add requires a single instruction to produce a result while the multiplier produces a result with a two-cycle pipe. Figure 1 shows the C code for our 4x4 product example.

If we simply count the additions and multiplications, we see that the innermost loop of the program in Figure 1 contains one add and one multiply, each of which are executed 4^3, or 64, times. A simple schedule (that abstracts the memory accesses, initialization of variables, etc.) would contain the following three instructions.

```
1.   t = a[i,k]  * b[k,j]
2.   nop
3.   c[i,j]  = t + c[i,j];
```

Note that instruction "3" must follow instruction "1" by two instructions because of the 2-instruction execution time for a multiply, coupled with the fact that "1" computes something ("t") used in "3". The result is $3 \times 4^3 = 192$ instruction cycles to perform these arithmetic operations.

If the program was compiled with local scheduling only (that is, the instructions were not moved around control block boundaries), we would end up with the code as shown above. An opportunity to improve the efficiency of the program becomes evident when one sees that the next scalar multiply can begin as soon as the addition at the bottom of the loop completes. Software pipelining would "fold" the innermost loop body to allow the adds to overlap with the multiplies. The result is shown below. The "#" delimiter separates operations that are executed in parallel.

```
PRELUDE (INNERMOST LOOP):
  1. t[0]=a[i,0]*b[0,j]
  2.
INNERMOST LOOP BODY
(executed  for  k=1  ...  3):
  1. c[i,j]=t[k-1]+c[i,j] # t[k]=a[i,k]*b[k,j]
  2. nop
POSTLUDE  (INNERMOST LOOP):
  1. c[i,j]=t[3]+c[i,j]
```

The result is now $(2+(2)x3+1) \times 4^2 = 144$ instruction cycles to perform these arithmetic operations, representing a savings of 25% over local scheduling.[1] Note that the loop body now contains operations from what would have been two different iterations of the original loop, as indicated by reading $t[k-1]$ while defining $t[k]$. This overlapping of loop iterations is exactly what allows the more efficient code, but just as hardware pipelines need to be initialized before the first result is available and then drained after the last result has begun execution, software pipelining requires that the steady-state loop body be prefaced with initialization code, typically called the *prelude* and suffixed with a *postlude* to finish off the loop results.

To improve the code efficiency further, we can pipeline the middle loop. The result is shown below. The optimized loop produced using software pipelining now requires only $(1 + ((2) \times 3 + 2) \times 3 + 8) \times 4 = 132$ instruction cycles, a savings of 31.2% over the initial (locally) scheduled loop.[2]

```
PRELUDE   (MIDDLE     LOOP):
 1. t[0]=a[i][0]*b[0][0];

MIDDLE LOOP BODY
(executed  for  J=0  ...  2):
INNERMOST LOOP BODY
(executed for k=1 ... 3):
  1.
  2.  t[k]=a[i,k]*b[k,j] # c[i,j]=t[k-1]+c[i,j]

POSTLUDE  (INNERMOST LOOP):
  1.  nop
  2.  t[0]=a[i,0]*b[0,j+1] # c[i,j]=t[3]+c[i,j]

POSTLUDE  (MIDDLE LOOP):
  1.  nop
  2.  t[1]=a[i,1]*b[1,3] # c[i,3]=t[0]+c[i,3]
  3.  nop
  4.  t[2]=a[i,2]*b[2,3] # c[i,3]=t[1]+c[i,3]
  5.  nop
  6.  t[3]=a[i,3]*b[3,3] # c[i,3]=t[2]+c[i,3]
  7.  nop
  8. c[i,3]=t[3]+c[i,3]
```

2.2 Modulo Scheduling

Allan et al. [1] provide an good summary of current software pipelining methods, dividing software pipelining techniques into two general categories

[1] Actually, software pipelining could improve upon this by overlapping two iterations of the loop within the 2-instruction loop body, yielding a scheduling requiring only 96 cycles. We chose the presented schedule to simplify the discussion.

[2] Again, a better schedule is possible, one requiring 84 cycles.

```
int        a[4][4];
int        b[4][4];
int        c[4][4];

main()
{
    int i, j, k;

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

Figure 1: C Program for computing a 4x4 product

called *kernel recognition* methods and *modulo scheduling* methods. In the kernel recognition technique, a loop is unrolled an "appropriate" number of times, yielding a representation for N loops bodies which is then scheduled. After scheduling the N copies of the loop, some pattern recognition technique is used to identify a repeating kernel within the schedule. Examples of kernel recognition methods are Aiken and Nicolau's perfect pipelining method [10, 11] and Allan's petri-net pipelining technique [12].

In contrast to kernel recognition methods, modulo scheduling does not schedule multiple iterations of a loop and then look for a pattern. Instead, modulo scheduling selects a schedule for one iteration of the loop such that, when that schedule is repeated, no resource or dependence constraints are violated. This requires analysis of the data dependence graph (DDG) for a loop to determine the minimum number of instructions required between initiating execution of successive loop iterations. Once that minimum initiation interval is determined, instruction scheduling attempts to match that minimum schedule while respecting resource and dependence constraints. Lam's *hierarchical reduction* is a modulo scheduling method as is Warter's [13, 4] *enhanced modulo scheduling* which uses *IF-conversion* to produce a single superblock to represent a loop. Rau [3] provides a detailed discussion of an implementation of modulo scheduling.

Since Rocket's software pipelining, patterned after Warter's *enhanced modulo scheduling* [13], uses a modulo scheduling algorithm, we shall investigate modulo scheduling in a bit more detail. While Warter's method provides a general framework for our software pipelining, the actual modulo scheduling technique implemented in Rocket closely follows Rau [3]. Modulo scheduling assumes that a single data-dependence graph (DDG) can be built for a loop. To build a single DDG for a loop requires some method of treating the entire loop as a single basic block. Like Warter, we use IF-conversion to transform a loop with arbitrary control flow into a single block and then construct the DDG for that "super-block" using standard dependence analysis. Since we intend to overlap different loop iterations we need to consider loop-carried dependence as well as loop-independent dependence, but well-known algorithms provide this information [14, 15].

Once the DDG is constructed for the loop, modulo scheduling attempts to identify the smallest number of instructions which might separate different loop iterations. This minimum *initiation interval* (II_{min}) represents the shortest time interval between the initiation of consecutive loop iterations. II_{min} depends upon two characteristics of the DDG for the loop (and the parallelism available in the target architecture):

- The *recurrence* constraint, RecII, represents a limit on II_{min} due to dependence arcs within the DDG. The recurrence constraint is due to loop-carried dependences which create cycles in the DDG.
- The *resource* constraint, ResII, represents a limit based upon how much "work" the loop requires

(how many integer operations, floating point operations, memory operations, etc.) and how much parallelism is provided by the architecture (how many floats, ints, etc. can be issued in a instruction.) In this context, a resource can be thought of as a functional unit needed to complete some operation.

Following Rau [3], to find RecII we iterate on potential values for RecII, building a matrix, *MinDist*, for each possible II value. $MinDist[i,j]$ is defined to be the minimum interval between loop operations i and j which maintains data dependence integrity. The MinDist entries of interest are the diagonal elements $MinDist[i,i]$. A positive value for any such diagonal element implies that the operation must follow itself, indicating a failure for that value of RecII. Thus, we try again with a larger value for RecII.

While determining RecII is computationally expensive, calculating ResII is considerably easier. It requires determining limits among the different classes of resources available for the hardware of choice. So, for each resource class (floating point multiplier, floating point adder, read/write pipeline, integer ALU) modulo scheduling needs to know how many such resources can be started in each instruction and it needs to determine how many such resources are needed to execute one iteration of the loop. Using these factors, one can determine the minimum number of instructions required to execute an iteration of the loop based upon the most limited resource, and that minimum number of instructions is ResII. For example, consider a loop which requires 4 floating point multiplies, 8 floating point adds, 10 integer operations and 6 memory operations to complete the execution of a loop iteration. If we are pipelining that loop for a machine which can start 1 floating point add, 1 floating point multiply, and 2 integer operations each instruction and which can start a memory operation every other instruction, then ResII would be 12, because the limiting resource would be the ability to schedule the memory operations. The individual resource limits for this example would be:

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

Since the maximum constraint for any resource is the 12 instructions required for the memory operations, ResII would be 12 in this case. Given both ResII and RecII, the actual minimum initiation interval (II_{min}) is the maximum of ResII and RecII.

Having computed II_{min}, modulo scheduling next attempts to schedule the DDG in II_{min} instructions. Again following Rau, we use a modified conventional acyclic list scheduling method. If ResII is significantly greater than RecII, the nodes are scheduled using traditional list scheduling. If, however, II_{min} is close to RecII, heuristics are used to give priority to nodes which are in the longest dependence cycle. If a schedule of II_{min} instructions can be found which does not violate any resource or dependence constraints, modulo scheduling has achieved a minimum schedule. If not, scheduling is attempted with $II_{min} + 1$ instructions, and then $II_{min} + 2$, ..., continuing up to the worst case which is the number of instructions required for local scheduling. However many instructions are required to "legally" schedule the DDG becomes the actual initiation interval, II.

After finding a schedule for the loop body requiring II instructions, it may be necessary to perform *modulo variable expansion* [2] to circumvent inter-interval dependences which can occur due to register reuse. To overcome such inter-interval dependences, the loop body schedule may need to be copied M times, where M is the number of different loop iterations represented within the loop body schedule. Each register within the (II-length) loop body is then "expanded" to become a group of registers, one per copy of the original loop body, thereby removing conflicts produced by register reuse dependences.

Once a schedule has been found for the loop body and modulo variable expansion has been performed, modulo scheduling needs to add a prelude and postlude for the loop, much as was shown in the example of Section 2.1. Remember that this postlude and prelude are required to initialize and then drain the "software pipe."

After inserting the prelude and postlude code we may need to "clean-up" the loop body to cover the side effects of modulo variable expansion. Since modulo variable expansion will potentially produce multiple copies of the loop body, we need to alter the number of times the loop body actually executes. This is relatively easy if the number of times the single loop body should execute, N, is a multiple of the loop body

expansion factor, F. When F is not a factor of N, we need to set the loop body iteration count to the integer part of N/F and add $mod(N, F)$ additional copies of the single loop body either before the prelude or after the postlude. This process of setting the proper number of loop iterations is called *loop conditioning.*

So, to summarize, the steps required in modulo scheduling once a single block (or super-block) represents the loop include:

- Build a DDG for the block (or super-block), including both loop-independent and loop-carried dependences.
- Since modulo scheduling requires an iterative process to find the best schedule, we need to determine a worst-case schedule which is that schedule required for the loop without software pipelining. Thus, we schedule a single iteration of the loop using normal scheduling techniques. Let N be the length of this schedule.
- Compute the best possible (minimum) II
  - compute ResII
  - compute RecII
  - II_{min} = max(ResII,RecII)
- Attempt to schedule the nodes of the loop in II_{min} instructions, using a resource reservation table of length II_{min}.
- If scheduling in II_{min} instructions is not possible, try $II_{min} + 1$, $II_{min} + 2$, ... N. (We know we can schedule in N instructions without pipelining at all.)
- Having found a schedule for the loop body, perform modulo variable expansion.
- Add prelude and postlude code to initialize and drain, respectively, the software pipeline.
- Perform loop conditioning to counter the side effects of modulo variable expansion.

3 Unroll-and-Jam

Callahan,*et al.*, have shown that when software pipelining is performed, simple inner loop unrolling does not help in the presence of recurrences nor does it help with a mismatch in the resources demanded by a loop and the resources provided by a machine [5]. However, unroll-and-jam, or outer-loop unrolling, can be used to improve the ILP available to a software pipelining algorithm in the above situations [5, 6]. The transformation unrolls an outer loop and then jams the resulting inner loops back together. Using unroll-and-jam we can introduce more parallelism into an innermost loop body. For example, consider our matrix multiply example from Figure 1:

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

Our previous software pipeline of this loop includes one `nop` every loop iteration to account for the recurrence. After unroll-and-jam of the `j`-loop by 1, we have:

```
for(i=0;  i<4; i++)
  for(j=0; j<4; j+=2)
   {
    c[i][j] = 0;
    c[i][j+1] = 0;
    for(k=0; k<4; k++)
     {
      c[i][j]=a[i][k]*b[k][j]+c[i][j];
      c[i][j+1]=a[i][k]*b[k][j+1]+c[i][j+1];
     }
   }
```

Now with two recurrences carried by the `k`-loop, each of which is parallel with the other, the software pipelined loop will not contain a `nop`. In addition to improving parallelism in the presence of recurrences, unroll-and-jam can match the resource demands of a loop with the resources available on a given architecture. This is discussed in some detail below.

3.1 Balance

We assume a pipelined architecture that allows asynchronous execution of memory accesses and floating-point operations (*e.g.* HP PA-RISC). We also assume a typical optimizing compiler – one that performs scalar optimizations only. In particular, we assume that it performs strength reduction, optimizes for machine addressing modes, allocates registers globally (via a coloring scheme) and schedules the pipelines. To measure the performance of program loops given the above assumptions, we use the notion of *balance* defined by Callahan, *et al.* [5].

3.1.1 Machine Balance

A computer is balanced when it can operate in a steady state manner with both memory accesses and floating-point operations being performed at peak speed. To quantify this relationship, we define β_M as the rate at which data can be fetched from memory in words per cycle, M_M, compared to F_M, the rate at which floating-point operations can be performed in flops per cycle:

$$\beta_M = \frac{M_M}{F_M}$$

The values of M_M and F_M represent peak performance where the size of a word is the same as the precision of the floating-point operations.

3.1.2 Loop Balance

Not only do machines have balance ratios, but also loops. More formally, we can define balance for a specific loop, L, as

$$\beta_L = \frac{M_L}{F_L}$$

where M_L is the number of memory operations in loop L and F_L is the number of floating-point operations in L.[3] This model assigns a uniform cost to memory references under the assumption that compiler optimizations can be performed to attain cache locality [16, 17, 18].

Comparing β_M to β_L can give us a measure of the performance of a loop running on a particular architecture. If $\beta_L > \beta_M$, then the loop needs data at a higher rate than the machine can provide and idle computational cycles will exist. Such a loop is said to be memory bound and its performance can be improved by lowering β_L to be as close to β_M as possible.

In our matrix multiply example in Figure 1, the original loop has a balance of 1 because `c[i][j]` can be allocated to a register using scalar replacement [19, 20]. On a machine that can issue two floating-point operations per memory operation, the unroll-and-jammed loop, with a balance of 0.75 (note that `c[i][j]`, `c[i][j+1]`, and the second reference to `a[i][k]` can be allocated to registers), would perform better even in the presence of no pipeline interlock. This is because there is a better match between the resource demands of the loop and the resources provided by the machine.

[3] If the target architecture does not include auto-increment addressing modes, the extra address computations must be considered.

Although there are methods other than unroll-and-jam to deal with recurrences [21], unroll-and-jam can additionally bring memory-bound loops into balance. Inner loop unrolling does not improve memory performance as reuse across the inner loop will already be captured by cache and registers [22]. Unroll-and-jam moves reuse carried by outer loops into the inner loop to reduce the number of memory references and cache misses, thus, decreasing the memory demands of the loop.

3.1.3 Applying Unroll-and-Jam In A Compiler

Previous work has used the following optimization problem to guide unroll-and-jam [6]:

$$\begin{aligned} \textbf{objective function:} \quad & \min |\beta_L - \beta_M|_0 \\ \textbf{constraint:} \quad & R_L \leq R_M \end{aligned}$$

where R_L is the number of registers required by a loop, R_M is the register-set size of the target architecture, and $|\beta_L - \beta_M|_0$ is the *balance norm* having the following definition:

$$|\beta_L - \beta_M|_0 = \begin{cases} \beta_M - \beta_L & \text{if } \beta_L \leq \beta_M \\ \beta_L - \beta_M + \epsilon & \text{otherwise} \end{cases}$$

The decision variables in the problem are the unroll amounts for each of the loops in a loop nest. The register-pressure constraint limits unrolling to the point before floating-point registers are spilled. For the solution to the objective function, ϵ causes the balance norm to favor a slightly compute-bound loop over a slightly memory-bound one.

Essentially, the objective function attempts to match the balance of a loop with the balance of a target machine without creating too much register pressure. The optimization is machine-independent in that it can be retargeted to a new architecture by changing β_M and R_M.

The construction of and solution to the objective function has been shown to be efficient. β_L and R_L can be constructed from the dependence graph of a loop in time proportional to the size of the dependence graph. The solution to the objective function for a particular loop nest can be optimized in $O(\log R_M)$ steps when unroll-and-jam is applied to one loop and $O(R_M)$ steps when unroll-and-jam is applied to two loops [6].

After unroll-and-jam guided by the above objective function, a loop may still contain pipeline interlock, leaving idle computational cycles. To remove these

cycles, we simply ensure that the number of copies of the innermost loop body is at least as large as the length of the longest pipeline in the target architecture. We expect this heuristic to be needed rarely as unrolling for loop balance will likely remove interlock.

In previous work, Carr and Kennedy show that unroll-and-jam guided by the previous optimization formula improves the performance on the IBM RS/6000 [6]. Their experiment presents execution-time improvements to validate their claims. However, the RS/6000 only has limited hardware instruction scheduling and the full benefit of unroll-and-jam could not be measured. The experiment in this paper will look at the effect of unroll-and-jam on software pipelining and show the full improvements possible in ILP.

4 Experimental Evaluation

To evaluate our hypothesis that unroll-and-jam can improve software pipelining's ability to generate efficient code, we performed unroll-and-jam on 28 loops which we subsequently software pipelined. In this experiment, we used the implementation of unroll-and-jam in Memoria and the implementation of software pipelining in Rocket. Our experimental method is graphically described in Figure 2. To obtain our optimized code, we first performed unroll-and-jam on the original Fortran source. Then, we converted the resulting Fortran code to ParaScope's intermediate language (Iloc) and used a translator to convert from Iloc to Rocket's intermediate form. Rocket then built DDGs for the loops to be pipelined and added the loop-carried dependences for the DDG. To measure the effectiveness of the schedules built by software pipelining, we compared the actual iteration interval, II, obtained by software pipelining after unroll-and-jam with the II obtained when software pipelining was used without benefit of unroll-and-jam.

For our target architecture we chose a machine with four integer units and two floating-point units. Only one of the integer units can be used for memory operations. Each integer operation has a latency of two cycles while each floating-point operation has a latency of four cycles. Since we are not currently able to perform register assignment after software pipelining, the machine has essentially an unlimited number of registers. However, our unroll-and-jam configuration assumes a machine with 64 registers and limits its unrolling accordingly. The reason that we chose a disproportionate number of integer functional units is to compensate for the lack of addressing-mode optimization in Rocket.

Our experimental test suite includes the Perfect, SPEC and RiCEPS benchmark suites. Each benchmark was examined for loops with nesting depth two or more. We then applied our transformation system to those loops to determine the improvement in the software pipelining initiation interval by using unroll-and-jam. We investigated 64 nested loops in the benchmark suites to which unroll-and-jam can be applied. Of those loops, unroll-and-jam unrolled 26. The remaining 38 loops were not unrolled because Memoria's heuristics for loop unrolling suggested that no benefit would accrue from unrolling the loop. Memoria uses two simple heuristics to determine when to unroll-and-jam. If Memoria recognizes that the loop balance is greater than the target machine balance, it will unroll in an attempt to lower the loop balance as described in Section 3. Additionally if Memoria detects an inner-loop recurrence, it will unroll the outer loop N times where N represents the target machine's floating point pipe depth. This attempts to remove any pipeline interlock within the inner loop. In the 38 loops Memoria failed to unroll no inner-loop recurrence was found. In addition, one (or both) of two conditions led Memoria to conclude that improved loop balance was not possible through unroll-and-jam. Thirty-one loops contain no outer-loop-carried dependences so no improvement of the balance was possible. Eighteen loops were already determined to be in balance and so no unrolling was deemed necessary to achieve peak efficiency. Of the 26 unrolled loops three contained intrinsic function calls and were therefore not considered in this study since our measurement of RecII is not accurate in the presence of function calls. Of the 23 loops which we both unrolled and software pipelined, all 23 showed schedule improvements due to the unroll-and-jam procedure. In addition, 5 loops extracted from Fortran kernels were software pipelined after unroll-and-jam and each of these 5 loops showed improvement as well. Table 1, lists the 23 benchmark loops as well as the 5 kernel loops along with their improvements in initiation interval due to unroll-and-jam.

The most noteworthy result from Table 1 is that, not only did all 28 unrolled loops show schedule improvements by software pipelining when unroll-and-jam was applied first, but that the amount of improvement is quite significant, ranging from a low of 15.3% to a high of 80%. Performing an unweighted average on the % Improvement column shows that, on average, the 28 loops showed an improvement of 43%. Of

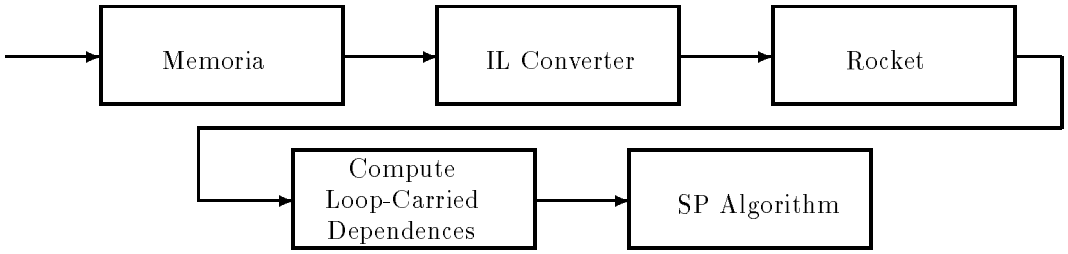

Figure 2: Experimental Method

Program	SubProgram/	Before Unrolling			After Unrolling				%
	Loop Number	RecII	ResII	II	times	RecII	ResII	II	Improvement
RICEPS									
simple	conduct1	4	4	5	17	37	52	52	38.8
	conduct2	4	4	5	51	4	154	154	39.6
SPEC									
dnasa7	btrix1	4	12	12	12	4	48	48	66.7
	btrix2	4	12	12	5	4	44	44	26.7
	btrix3	7	41	41	2	56	45	58	29.3
	btrix4	4	12	12	5	26	40	40	33.3
	cholsky1	4	4	5	25	7	98	98	21.6
	gmtry	4	4	5	49	4	99	99	59.6
	vpenta1	9	23	23	3	18	49	49	29.0
	vpenta2	4	18	18	3	14	30	30	44.4
PERFECT									
adm	radb2.lo	4	16	16	10	4	52	52	67.5
	radbg1	4	3	5	50	206	52	206	17.6
	radbg1*	4	3	5	5	8	8	8	68.0
flo52	collc	4	3	5	4	4	10	10	50.0
	dflux2	4	4	5	25	14	100	100	20.0
	dflux3	4	4	5	25	4	74	74	40.8
	dflux5	4	4	5	25	14	76	76	39.2
	dflux6	4	5	5	4	14	14	14	30.0
	dflux7	4	4	5	25	14	76	76	39.2
	dfluxc1	4	4	5	8	5	21	21	47.5
	dfluxc3	4	5	5	16	5	45	45	43.8
	dfluxc4	4	5	5	17	10	72	72	15.3
	dfluxc5	4	4	5	25	14	76	76	39.2
	dfluxc6	4	4	5	25	5	51	51	59.2
KERNELS									
	dmxpy	4	4	5	49	8	99	99	59.6
	fold	4	4	5	19	8	27	27	71.6
	mmjik	4	4	5	4	8	9	9	55.0
	mmjki	4	4	5	9	4	9	9	80.0
	sor1k	4	6	6	10	44	37	47	21.7
AVERAGE									43.0

Table 1: Software Pipelining Improvement with Unroll-and-Jam

the 28 loops unrolled, 10 showed improvement greater than or equal to 50%, corresponding to a speed-up of 2 or better.

In addition, our results suggest that unroll-and-jam will increase the number of loops in which II_{min} is determined by ResII rather than RecII. While ResII exceeded RecII for only 11 of the 28 loops before unroll-and-jam, ResII was greater than RecII for 23 out of 28 loops after unroll-and-jam was applied. This is a positive effect as well, since it suggests that we are indeed making better use of the target architecture's hardware. Perhaps of less importance, but still interesting, is the fact that unroll-and-jam leads to loops for which software pipelining is more likely to achieve an iteration interval of II_{min}. Software pipelining was able to find II equal to II_{min} for 13 of 28 loops before unroll-and-jam, while 26 of 28 loops achieved II of II_{min} after unroll-and-jam. We attribute this directly to the greater likelyhood that II_{min} will be determined by ResII for the unrolled loops.

One loop, found in *radbg1*, produced results which look a bit odd at first glance. Notice that Table 1 includes radbg1 twice, once with a "*", to reflect that we changed the source code for the nested loop. First consider the results for the unchanged loop. Notice that we unrolled 50 times and that RecII for the unrolled version is 206! This seems odd in its own right. The reason that radbg1 required such a long recurrence iteration interval is that the innermost loop consists of a somewhat lengthy chain of intra-iteration (loop independent) dependences combined with an induction variable which, of course, represents a recurrence. However, this is the only recurrence in the nested loop, as originally written. Meanwhile the outer loop contains a loop-carried dependence which led Memoria to unroll the outer loop. However, this merely led to a longer chain of intra-iteration dependences within the inner loop with still the same recurrence on the induction variable. This, in turn, led to the unusually large RecII. By interchanging the inner- and outermost loops of radbg1 (leading to radbg1*) unroll-and-jam allows for much better software pipelining, as can be seen from the table, where the radbg1 allowed improvement of 17.6% and radbg1* showed improvement of 68%. This suggests that judicious use of loop interchange might well allow unroll-and-jam to produce even more improvement in software pipelining than was found in this study.

Overall, this experiment certainly supports the hypothesis that unroll-and-jam can improve software pipelining's ability to generate efficient code. Considering that unroll-and-jam, when applied, improved the performance of every loop tested and that the average improvement was 43% over software pipelining without unroll-and-jam, we conclude that unroll-and-jam should be attempted whenever software pipelining of nested loops is performed.

5 Conclusions

To achieve efficient code for loops, compilers for instruction-level parallel (ILP) architectures use software pipelining, which overlaps operations from different loop iterations to obtain a more compact instruction schedule. We show how unroll-and-jam can be used to increase parallelism available to software pipelining, allowing even better schedules to be found. By judiciously applying unroll-and-jam we can improve the "balance" between memory operations and floating point operations within the nested loop, allowing software pipelining to better utilize the target architecture's hardware.

Our results show that performing unroll-and-jam can significantly improve schedules generated by software pipelining. All 28 loops on which unroll-and-jam was applied showed software pipelining improvements over that attainable without unroll-and-jam. While these improvements averaged 43% over all 28 loops, 10 loops exceeded 50% improvement, and one showed 80% improvement, corresponding to a factor of 5 speed-up. Based upon this empirical evidence, we recommend that unroll-and-jam should be attempted whenever software pipelining of nested loops is performed.

References

[1] V. Allan, R. Jones, R. Lee, and S. Allan, "Software Pipelining," *ACM Computing Surveys*, vol. 27, no. 3, September 1995.

[2] M. Lam, "Software pipelining: An effective scheduling technique for VLIW machines," *SIGPLAN Notices*, vol. 23, pp. 318–328, July 1988. *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation.*

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

[4] N. J. Warter, S. A. Mahlke, W. mei W. Hwu, and B. R. Rau, "Reverse if-conversion," *SIGPLAN Notices*, vol. 28, pp. 290–299, June 1993. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation.*

[5] D. Callahan, J. Cocke, and K. Kennedy, "Estimating interlock and improving balance for pipelined machines," *Journal of Parallel and Distributed Computing*, vol. 5, pp. 334–358, 1988.

[6] S. Carr and K. Kennedy, "Improving the ratio of memory operations to floating-point operations in loops," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 6, pp. 1768–1810, 1994.

[7] S. Carr, *Memory-Hierarchy Management.* PhD thesis, Rice University, Department of Computer Science, September 1992.

[8] D. Callahan, K. Cooper, R. Hood, K. Kennedy, and L. Torczon, "ParaScope: A parallel programming environment," in *Proceedings of the First International Conference on Supercomputing*, (Athens, Greece), June 1987.

[9] P. H. Sweany and S. J. Beaty, "Overview of the Rocket retargetable C compiler," Tech. Rep. CS-94-01, Department of Computer Science, Michigan Technological University, Houghton, January 1994.

[10] A. Aiken and A. Nicolau, "Optimal loop parallelization," in *Conference on Programming Language Design and Implementation*, (Atlanta Georgia), pp. 308–317, SIGPLAN '88, June 1988.

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

[12] V. Allan, M. Rajagopalan, and R. Lee, "Software Pipelining: Petri Net Pacemaker," in *Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, (Orlando, FL), January 20-22 1993.

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

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

[15] G. Goff, K. Kennedy, and C.-W. Tseng, "Practical dependence testing," *SIGPLAN Notices*, vol. 26, pp. 15–29, June 1991. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.*

[16] K. Kennedy and K. McKinley, "Optimizing for parallelism and memory hierarchy," in *Proceedings of the 1992 International Conference on Supercomputing*, (Washington, DC), pp. 323–334, July 1992.

[17] T. C. Mowry, M. S. Lam, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching," in Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, (Boston, Massachusetts), pp. 62–75, 1992.

[18] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," *SIGPLAN Notices*, vol. 26, pp. 30–44, June 1991. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.*

[19] D. Callahan, S. Carr, and K. Kennedy, "Improving register allocation for subscripted variables," *SIGPLAN Notices*, vol. 25, pp. 53–65, June 1990. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation.*

[20] S. Carr and K. Kennedy, "Scalar replacement in the presence of conditional control flow," *Software Practice and Experience*, vol. 24, pp. 51–77, Jan. 1994.

[21] P. Kogge, *The Architecture of Pipelined Computers.* New York: McGraw-Hill, 1981.

[22] S. Carr, K. McKinley, and C.-W. Tseng, "Compiler optimizations for improving data locality," in Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, (Santa Clara, California), 1994.