Modulo Scheduling with Cache Reuse Information *

Chen Ding† Steve Carr‡ Phil Sweany‡

Abstract

Instruction scheduling in general, and software pipelining in particular face the difficult task of scheduling operations in the presence of uncertain latencies. The largest contributor to these uncertain latencies is the use of cache memories required to provide adequate memory access speed in modern processors. Scheduling for instruction-level parallel architectures with non-blocking caches usually assigns memory access latency by assuming either that all accesses are cache hits or that all are cache misses. We contend that allowing memory latencies to be set by cache reuse analysis leads to better software pipelining than using either the all-hit or all-miss assumption.

Using a simple cache reuse model in our modulo scheduling software pipelining optimization, we achieved a benefit of 10% improved execution performance over assuming all-cache-hits and we used 18% fewer registers than were required by an all-cache-miss assumption. In addition, we outline refinements to our simple reuse model that should allow modulo scheduling with reuse to achieve improved execution performance over the all-cache-miss assumption as well. Therefore, we conclude that software pipelining algorithms for target architectures with non-blocking cache, but without rotating register files, should use a memory-reuse latency model.

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. Computers that can issue multiple operations in a single cycle are typically called *instruction-level parallel* (ILP) architectures. In order to fully utilize ILP hardware, either the compiler or the architecture (or preferably both) must order the operations to be executed to allow for maximum parallelism. This "ordering" of operations is typically called *instruction scheduling*.

In modern processors, main-memory access time is at least an order of magnitude slower than processor speed. A small, fast cache memory is used to alleviate this problem. However, the cache cannot eliminate all accesses to main memory and programs incur a significant penalty in performance when a miss in the cache occurs. To help tolerate the cache miss latency, *non-blocking* caches have been designed to allow cache access to continue when misses occur. This is important to ILP because it allows the instruction scheduler to overlap more operations with memory accesses, possibly hiding main-memory latency. Thus, a significant increase in ILP can be achieved [6].

The existence of a cache, however, produces a problem for the instruction scheduler since the latency of a memory operation is not static. To deal with this situation, instruction schedulers typically either assume that all memory accesses are cache hits or assume that they are all cache misses. Assuming all hits reduces the lifetimes of registers and keeps register pressure to a minimum. However, significant penalties are incurred when a cache miss occurs. Assuming all cache misses tolerates the latency of a cache miss better, but may increase register pressure significantly. Additionally, non-loop scheduling methods may not be able to find sufficient parallelism to hide latency in the all-cache-miss assumption.

To take full advantage of the parallelism available in ILP computers, advanced instruction scheduling techniques such as software pipelining have been developed [3, 10, 14, 18]. Software pipelining allows iterations of a loop to be overlapped with one another in order to take advantage of the parallelism in a loop body. While software pipelining can yield significant performance gains by overlapping loop iterations, it can also require significant register resources. One solution to providing adequate registers for software pipelining involves including special hardware called rotating register files that allow multiple hardware copies of registers, one for each of several possible loop iterations. While it is true that rotating register files dramatically ease the register burden of software pipelining it is also true that few current ILP architectures include them. For those ILP comput-

*This work was partially supported by the National Science Foundation under grants CCR-9409341 and CCR-9308348, as well as a grant from Digital Equipment Corporation.

†Department of Computer Science, Rice University, Houston TX 77251-1892

‡Department of Computer Science, Michigan Technological University, Houghton MI 49931-1295

ers without rotating register files we need a software solution to the problem of software pipelining's register proliferation. While there are several varieties of software pipelining we shall restrict ourselves in this paper to discussion of modulo scheduling, perhaps the most popular software pipelining technique currently available. However, since difficulties with uncertain latencies and exploding register requirements exist no matter how software pipelining is implemented, our discussion and results should apply to other forms of software pipelining as well.

This paper concerns practical issues of implementing modulo scheduling for an architecture with a non-blocking cache, but without rotating register files. For such an architecture we wish, as always, to minimize loop execution time but we also must consider the negative effect that modulo scheduling has on the register pressure of a loop. For indeed, if modulo scheduling increases register demands to the point that considerable register spilling is required, the execution efficiency obtainable by modulo scheduling will be lost.

Modulo scheduling for "traditional" machines without rotating register files relies on Modulo Variable Expansion (MVE) [11], to generate correct code. MVE will assign multiple registers to a single loop value to account for the fact that values' lifetimes typically exist across several loop iterations. This is required by software pipelining's overlapping of values from different loop iterations.

Assuming that memory latencies are all cache misses to avoid paying the cache-miss penalties associated with assuming all cache hits will likely lead modulo scheduling to overlap more lifetimes. This can lead to significantly greater register usage since the register lifetimes get stretched by the assumption that each load is a cache miss. In the presence of rotating registers this additional register pressure can be accommodated (at a significant hardware cost) but when no such hardware is available, MVE may cause an explosion in register pressure.

Given that both all-hit and all-miss assumptions have negative consequences for performance, we would like to recognize those memory accesses that are hits and those that are misses and schedule using the "correct" latency for each memory access. Currently, memory reuse (cache hit) information is available to the compiler [5, 19]. Our contention is that by using such reuse information we can improve software pipelining with respect to using either an all-hit or all-miss latency assumption.

In the remainder of this paper, we first discuss software pipelining (Section 2) with special attention paid to register requirements and pipelining with uncertain memory latencies. Section 3 describes the dependence-based memory reuse analysis that we use. Section 4 details our experimental evaluation of modulo scheduling with reuse information, Section 5 describes refinements to our simple cache model that will allow further improvement over those shown in our experimental results and Section 6 presents our conclusions.

2 Software Pipelining

While local and global instruction scheduling can, together, exploit a large amount of 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 operations.

Allan et al. [3] provide an good summary of current software pipelining methods, dividing software pipelining techniques into two general categories 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 loop 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 [1, 2] and Allan's petri-net pipelining technique [4].

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 [17, 18] *enhanced modulo scheduling* which uses *IF-conversion* to produce a single super-block to represent a loop. Rau [14] provides a detailed discussion of an implementation of modulo scheduling.

2.1 Modulo Scheduling

Our software pipelining implementation is based upon Iterative Modulo Scheduling and follows the method presented by Rau [14]. As such we identify the minimum initiation interval (II) for each innermost loop and attempt to schedule the loop in II instructions. Thus, the overall iterative modulo scheduling technique converts the loop into a prelude, a pipelined loop body, and a postlude.

For example, consider a generalized loop, L1, with of four operations we will call A, B, C, and D. For purposes of illustration let us assume that dependences require a sequential ordering of these operations within a single loop iteration. Thus, even if our target architecture allows 4 operations to be issued at once, a schedule for a single loop iteration, requiring 4 instructions would be

```
do N times
  A
  B
  C
  D
```

due to dependences among the operations. A software pipelined version of this loop might well be able to issue all 4 operations in one instruction by overlapping execution from different loop iterations. This might, under ideal circumstances, lead to a single-instruction loop body of $A^{i+3}B^{i+2}C^{i+1}D^{i}$ where X^{j} denotes operation X from iteration j of the loop[1]. Of course, if the loop body is concurrently executing operations from multiple loop iterations in a pipelined fashion, we need a prelude to set up the software pipeline and a postlude to empty it. The entire idealized loop then becomes:

```
(Prelude)
   A (iteration 1)
   B (iteration 1) A (iteration 2)
   C (iteration 1) B (iteration 2) A (iteration 3)

(Loop Body)
  do N-3 times (with index i)
  A B C D  (A of iteration i+3, B of i+2, C of i+1,
            D of i)

(Postlude)
   D (iteration N-2) C (iteration N-1) B (iteration N)
   D (iteration N-1) C (iteration N)
   D (iteration N)
```

2.2 Increased Register Requirements

As shown in the above example, software pipelining can, by exploiting inter-iteration concurrency, dramatically reduce the execution time required for a loop. Such overlapping of loop iterations also leads to additional register requirements, however. For illustrative purposes let us reconsider our 4-operation loop, L1. Let us assume that operation A computes a value, v, in a register and that operation D uses v. In the initial sequential version of a loop body one register is sufficient to store v's value, since the value computed by the next iteration's A is not available until after D has used the value computed by the current iteration. Notice, however, that, in the software pipelined version, we need to maintain several different copies of v because we have different loop iterations in execution simultaneously. Specifically we need to have as many registers "assigned" to v as we have different iterations of L1 in execution concurrently, namely 4 in our example.

Given that software pipelining leads to increased register requirements due to inter-iteration register dependences how can software pipelining overcome this difficulty? Many, including Rau [16] have advocated rotating register files in which each "register" listed in the schedule actually represents a group of registers and hardware is included to rotate among the physical registers associated with each abstract schedule register. This allows each loop iteration in execution simultaneously to have its own "version" of the needed register. Of course, this requires significantly more physical registers than are found in most ILP processors to date, as well as the added hardware complexity to automatically rotate among the available physical registers associated with an "abstract" register listed in the schedule.

When rotating registers are not available on the target architecture, software support is needed to produce correct schedules. The most popular technique is Lam's *modulo variable expansion* (MVE) [10]. MVE overcomes inter-interval dependences by copying the loop body, or kernel M times, where M is the number of different loop iterations included in the longest lifetime for any variable in the loop. 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.

Whether using rotating registers or MVE to ensure semantics-preserving software pipelining, inter-iteration register dependences created by software pipelining continue to be a considerable deterrent to modulo scheduling.

2.3 Scheduling with Uncertain Latencies

Modern processors, whether ILP or not, have been forced to adopt a multi-level memory hierarchy to deal with the fact that processor speeds far exceed memory access speeds and the gap continues to grow. Thus, almost all modern processors include at least one level of cache memory to make use of program locality, thereby significantly reducing average memory access times over what would be possible without a small high-speed memory.

Of course one attribute of memory systems that include cache is that the actual access time for any memory operation is unknown at compile time. Performance will be best if a memory load will be a cache hit most of the time, and thus require on the order of one or two cycles to be resolved. However, the possibility exists that a load will be a cache miss, leading to delays of 20 or more cycles.

This uncertainty in the latency of memory loads creates a problem for instruction scheduling in general and software pipelining in particular. Should a compiler schedule code assuming that all loads are cache hits? Should it assume that all loads are cache misses? Most instruction schedulers assume that all loads will be cache hits and, thus, schedule with a short latency for each load instruction. This is fine when, as is most often the case, the load is in fact a cache hit. When a cache miss occurs, however, a traditional processor stalls. Some modern processors, e.g. the DEC Alpha [7], provide a non-blocking cache that allow cache accesses to continue after a miss. When scheduling for such an architecture it can be especially important to schedule instructions to hide memory latencies as much as possible. Non-blocking caches at least allow for the possibility that assuming cache misses (and thereby inserting as many operations as possible between a load and the first use of the loaded register) might be a viable scheduling alternative.

When considering non-loop code, assuming a cache-hit latency may, in fact, be an excellent scheduling policy. We would expect most loads to be cache hits. In addition, while this optimistic view may lead to unnecessary stalls in non-loop code we may have little viable

[1] This notation is borrowed from Allan et al. [3]

alternative. As for non-loop code, we might not be able to hide the long latency of a cache miss even if we could recognize it. The situation is quite different for software pipelining, however. An arbitrary use of cache-hit latency will lead to processor stalls, just as in non-loop code. Now, of course the stalls are more costly, just due to the fact that most of the execution time of programs is spent in loops. More importantly, when software pipelining a loop, modulo scheduling can, by overlapping more loop iterations, hide almost any memory latency. This is in stark contrast to scheduling of non-loop code and may well make assuming cache-miss latencies a good policy. If we can (by overlapping more loop iterations) guarantee that we hide all latencies, modulo scheduling can guarantee excellent execution time for a loop.

Unfortunately, as we have discussed, overlapping more loop iterations leads to exploding register requirements. If we assume all loads are cache misses we may unnecessarily exhaust registers for the sake of hiding possibly non-existent latencies. Rau [15] and Huff [8] both recommend assuming all loads are cache misses in their modulo scheduling. Note that both assume rotating register files as well, however, which tends to lessen the register problem.

Assuming all loads are cache hits can potentially cripple modulo scheduling's execution efficiency due to stalls, while assuming all loads are cache misses exacerbates the already serious register proliferation problem of modulo scheduling by (perhaps unnecessarily) increasing the number of overlapped loop iterations in an attempt to hide latency. To address the problem of local instruction scheduling with uncertain latencies, Eggers and co-workers [9, 12] have suggested *balanced scheduling* for architectures with non-blocking caches. Balanced scheduling sets memory latencies based, not upon some architecturally predefined value, but rather based upon the number of instructions available to hide the latency of a particular load. While balanced scheduling is a step in the proper direction, in that it uses program information to set latencies, we feel it does not go far enough, at least not for software pipelining. Instead of merely averaging latencies based upon necessary schedule length, we wish to identify those loads that will be cache hits and those that will be cache misses. We propose using well-established memory reuse analysis techniques to identify those loads that will be cache hits and those that will be cache misses and use that information to determine the latency of each load in the loop to be software pipelined.

3 Memory Reuse Analysis

Since modulo scheduling suffers whenever we either assume that 1) all loads are cache hits, or 2) all loads are cache misses, we would like a mechanism to identify those loads that will lead to a cache hit and those that will lead to a cache miss. Given this information, we can schedule each load with its appropriate latency. In this section, we outline such a memory reuse analysis. In our simple model of memory reuse we will assume that each static load in a program will either always be hit or always be a miss. Of course, in general that assumption is not valid. Even given its limitations our simple model allows for an improved modulo scheduler, however. Section 5 describes a more complete model of cache behavior and possible further refinements to modulo scheduling to take advantage of a more sophisticated cache model. Since ours is a dependence-based reuse model we first digress for a brief discussion of program dependence.

```
  DO 10 I = 1,N
    DO 10 J = 1,N
      DO 10 K = 1,N
10      A(I+1,J,K-1) = A(I,J,K) + C(I,J,K)
```

$\langle 1, 0, -1 \rangle$

Figure 1: Example Dependence Graph

3.1 Dependence

A *dependence* exists between two references if there exists a control-flow path from the first reference to the second, and both references access the same memory location. The dependence is

- a *true dependence* if the first reference writes to the location and the second reads from it,
- an *antidependence* if the first reference reads from the location and the second writes to it,
- an *output dependence* if both references write to the location, and
- an *input dependence* if both references read from the location.

If two references, v and w, are contained in n common loops, separate instances of the execution of the references can be described by an *iteration vector*. An iteration vector, denoted $\vec{i}$, is simply the values of the loop control variables of the loops containing v and w. The set of iteration vectors corresponding to all iterations of the of the loop nest is called the *iteration space*. Using iteration vectors, a *distance vector*, $\vec{\delta} = \langle \delta_1, \delta_2, \ldots, \delta_n \rangle$, can be defined for each dependence: if v accesses location Z on iteration $\vec{i_v}$ and w accesses location Z on iteration $\vec{i_w}$, the distance vector for this dependence is $\vec{i_w} - \vec{i_v}$. Under this definition, the k^{th} component of the distance vector is equal to the number of iterations of the k^{th} loop (numbered from outermost to innermost) between accesses to Z. As an example, consider Figure 1. The distance vector for the dependence between the definition and use of array A is $\langle 1, 0, -1 \rangle$.

3.2 Reuse Model

The two sources of data reuse are temporal reuse – multiple accesses to the same memory location – and spatial reuse – accesses to nearby memory locations that share

a cache line or a block of memory at some level of the cache hierarchy. Temporal and spatial reuse may result from *self reuse* from a single array reference or *group reuse* from multiple references. Without loss of generality, in this paper we assume column-major storage for arrays.

The reuse model used in this paper is identical to the one described by Carr, *et al.* [5].To simplify analysis, we concentrate on reuse that occurs between a small number of inner loop iterations. This memory model assumes there will be no conflict or capacity cache misses in one iteration of the innermost loop. To compute cache reuse, we first apply algorithm RefGroup, shown below, to calculate group reuse. Two references are in the same reference group if they exhibit group-temporal or group-spatial reuse (i.e., they access the same cache line on the same or different iterations of an inner loop). In our simple model, any reference having either group or self reuse is considered to always be a cache hit. A reference with no reuse is considered to always be a cache miss.

RefGroup: Two references Ref_1 and Ref_2 belong to the same reference group with respect to loop l if at least one of the two following conditions holds:

1. $\exists$ Ref_1 $\vec{\delta}$ Ref_2 , and
   (a) $\vec{\delta}$ is a loop-independent dependence, or
   (b) δ_l is a small constant d ($|d| \leq 2$) and all other entries are zero,
2. $\exists$ Ref_1 $\vec{\delta}$ Ref_2 , and δ_f is less than the cache-line size and all other entries are zero. δ_f is the distance associated with the induction variable in the first subscript position.

Condition 1 accounts for group-temporal reuse and condition 2 detects some forms of group-spatial reuse.

To compute self-reuse properties, we consider a representative reference from each RefGroup separately. If the reference is invariant with respect to the innermost loop, it has self-temporal reuse. If the inner-loop induction variable appears only in the first subscript position of the reference, then the reference has self-spatial reuse.

Consider the following example.

```
      DO 10 J = 1,N
        DO 10 I = 1,N
10        A(I,J) = A(I-1,J) + C(J,I)
                 + C(J-1,I) + B(J)
```

`A(I-1,J)` has group-temporal reuse, `C(J-1,I)` has group-spatial reuse, `A(I,J)` and `A(I-1,J)` have self-spatial reuse, `B(J)` has self-temporal reuse and `C(J,I)` has no reuse. In our model, `C(J,I)` is always a cache miss and all other references are always cache hits.

4 Experiment

To evaluate our contention that taking advantage of memory reuse information can improve software pipelining's efficiency, we compiled and simulated 75 Fortran loops in which our software pipelining used one of three different memory latency policies, namely 1) all loads are cache hits, 2) all loads are cache misses, 3) each load is either always a cache hit or always a cache miss, as determined by memory reuse analysis. Our hypothesis is that software pipelining in which the load latency is determined by reuse should yield better execution performance than pipelining with an all-hit latency policy, and while it should lead to slightly poorer execution performance than pipelining with an all-miss latency policy (assuming an infinite number of registers) the reuse policy should lead to significantly fewer registers required for the loop.

4.1 Machine Model

The hypothetical superscalar architecture that we chose for our tests is an instruction-level parallel machine with two integer and two floating point functional units, each of which may issue an instruction in each cycle should data dependences allow. The latency for integer instructions is two cycles, while the latency for floating point instructions is four cycles. Only one load or store can be issued per cycle. All loads and stores use an integer unit and the cache hit latency is two cycles.

Since one of the parameters we wished to investigate was register usage, there were two possible ways to go. We could have chosen a fixed, relatively small number of registers similar to current ILP machines and "measured" register pressure as part of execution time, since spilling would necessarily degrade loop performance. However, in an attempt to separate register concerns and loop performance concerns we chose to include 256 integer and 256 floating point registers in our machine model. In this manner, we ensured that we would not spill and therefore can evaluate the effect on register pressure by a direct measurement of how many registers were required to generate software pipelined code for the loop.

The cache model we have chosen is an 8K direct-mapped cache with 32-byte lines. The cache is non-blocking and allows up to 6 outstanding misses to occur in parallel. The penalty for a miss to cache is an additional 25 cycles. When a miss occurs, two consecutive 32-byte lines are brought into the cache.

We simulated loop behavior only by resetting the simulator for each outermost loop construct. Thus, while we only pipelined innermost loops we counted all nested loops in our simulation results. However, we did not simulate non-loop code.

4.2 Test Programs

We software pipelined 107 Fortran innermost loops from three SPEC programs and an additional 13 loops from Fortran kernels, yielding a total of 120 innermost loops. For 45 of those loops there was no difference in any of the pipelined schedules depending upon whether we used a cache-hit assumption, a cache-miss assumption or a reuse model to determine load latency. Table 1 lists the sources of the 75 loops tested.

We used iterative modulo scheduling to pipeline the loops, and restricted our attention to loops with no control flow or function calls. Thus, we pipelined only single-block loops for this study.

Program	No. of Loops Tested
Spec	
hydro2d	36
su2cor	16
swm256	10
Others	
kernels	13
Total	75

Table 1: Test Loops

4.3 Results

Table 2 gives summary results for the performance, in terms of execution cycles, of the 75 loops tested. The first column shows the "normalized" execution time for code compiled with an all-hit latency policy. The second column shows the same computation for the all-miss latency policy and the third column gives the results of code compiled with reuse information. We normalized the cycles of each of the 75 loops so that whichever of the three compiled codes (hit, miss, reuse) required the fewest cycles was set to 100, and the other two were normalized with that value. The values listed in Table 2 represent the unweighted average of these normalized execution values for all 75 loops. As expected we see that loops pipelined with latencies set by reuse required fewer cycles, on average than those compiled with latencies set by a cache-hit assumption. In fact, the difference in performance between reuse and hit latencies is roughly 10%. Based only on execution cycles, we also expected reuse to perform slightly worse than cache-miss, due in part to our modulo scheduler's oversimplified model of cache behavior that each static load either always be a hit or always be a miss. We anticipated that this would lead to a small performance penalty, but, in fact, virtually all of the roughly 8% degradation we saw in performance between cache miss and reuse policies can be attributed to this simplification, as we will explain shortly (in Section 4.4.) Finally, the summary data shows that miss was not always the best performance policy. If it were always best its value would be 100 instead of 104. In fact several loops showed better performance with reuse than cache miss. This was unexpected and we attribute the fact to the somewhat larger overhead associated with software pipelining when using a cache-miss policy than when using reuse, or cache hit. We discuss this "overhead" in Section 4.4.

While Table 2 provides some indication of the overall performance of the three memory latency choices tested, it hides a great deal of detail. The longer version of the table, with all 75 loops listed individually can be found in Tables 4 and 5. Table 2 shows that the reuse version of a loop improved on the hit version on only 17 of the 75 loops tested, while hit never did better than reuse. That means that all of the roughly 10% average performance improvement was found in less than one fourth of the loops. In fact, for 13 of the loops, the reuse-compiled code was more than 20% faster than the code compiled with hit latencies. The largest difference was a factor of 2.61. The other 56 loops all produced the same results when compiled with hit latencies or reuse latencies. In contrast, while miss resulted in better schedules 34 times out of 75, reuse outperformed miss 19 times, by as much as 41% in one instance. To obtain the roughly 8% average improvement of cache miss to reuse then, cache miss had to be significantly better than reuse for some loops and in fact this is what we found. While reuse outperformed miss by at least 20% only twice, miss was more than 20% faster than reuse for 19 of the 75 loops. The maximum penalty of reuse for any loop was 69%.

Table 3 shows register requirements of the pipelined loops. Notice that while compiling with reuse required about one register more on average than compiling with hit latencies, it needed more than 6 fewer registers than those required by assuming miss latency. This represents a 17.9% registers savings over that needed for schedules that assume miss latency. For architectures with moderate numbers of registers this can be a considerable factor in deciding between using miss latencies and reuse information. When we restrict ourselves to those loops in which miss provided at least 20% better execution performance the difference is even greater. For those loops, reuse required an average of 31.6 registers while miss required 40.9, a savings of 22.8%.

4.4 Discussion

Our basic premise was that compiling with reuse information would allow for more efficient pipelined loops than would compiling with hit latency and for fewer registers required than would compiling with miss latency. Our experimental evidence certainly suggests that this is true. Compared with using hit latency, reuse produced loops requiring 10% fewer cycles on average while requiring less than one additional register on average. When compared with using miss latency, reuse required 6 fewer registers on average, but it did suffer substantial performance degradation on many loops. This led to an overall average degradation of 8% in execution performance.

To understand the reason for this degradation we need to return to the definition of reuse types, namely temporal vs. self-spatial reuse. In temporal reuse, we reuse an individual data item that was previously accessed. Thus, for any reference with temporal reuse, only a small number (d from our reuse model) misses will occur for the entire loop execution. Self-spatial reuse, in contrast, occurs because more than a single data item is brought into the cache at once. In a sense self-spatial reuse is indirect reuse. The "hit" is not due to that particular data item having been previously ac-

Cache Hit	Cache Miss	Reuse
123	104	112

Table 2: Summary Performance Numbers — Normalized

Cache Hit	Cache Miss	Reuse
33.7	40.8	34.6

Table 3: Summary of Registers Required

cessed, but rather from "neighbor" data having been accessed previously. If we assume stride-1 access of data (accessing adjacent data items on successive loop iterations) then rather than d misses for the entire loop, as with temporal reuse, spatial reuse leads to one miss every N loop iterations, where N is the number of adjacent data elements brought into the cache at once. Notice that this is quite different from our compiler's assumption that *every* access is a hit when we have spatial reuse.

Investigation of the 34 loops for which miss led to more effective pipelined schedules showed that they all exhibited spatial reuse. Many of the loops included several spatial reuse loads. This means that in our machine model, each spatial reuse load will incur a 25-cycle penalty each 8 loop iterations (since we bring 8 data items into the cache for each miss). This in itself is responsible for the degraded performance of reuse with respect to assuming miss latency. In Section 5 we suggest some refinements to our cache model that require a combination of software and a small amount of hardware. The refinements should eliminate the penalty that our reuse policy showed with respect to using miss latency.

Perhaps more puzzling is the fact that for 19 loops the schedule generated with reuse information required fewer cycles than that produced using miss latency. Our intuition suggested that miss should always yield a better schedule, but it did not. Closer investigation of the loops in question showed that, for all of them, the software pipelining "overhead" of prelude and postlude as well as preconditioning was significantly greater for the miss schedule than for the reuse schedule. This is a reasonable expectation because the longer latencies required by the miss policy led to more loop iterations being included in the pipelined kernel. When more iterations are included in the kernel, all of the prelude, postlude and preconditioning suffer. Recall that the prelude sets up the pipeline and the postlude drains it. Thus if we have more iterations within the kernel, more operations are required to set up and drain the pipe. Preconditioning is required to ensure that the entire loop is executed the proper number of times. If modulo variable expansion requires the loop kernel to be unrolled M times to accommodate register requirements, then the (unrolled) kernel for a loop to be executed N times must be executed N/M times. If N/M is not an integer the remainder is executed as a non-pipelined loop body. Longer register lifetimes, due to assuming miss latency, will lead to more kernel unrolling needed by modulo variable expansion and thus, potentially, more executions of the non-pipelined preconditioning code. Investigation of the loops where reuse required fewer cycles than miss showed that, indeed, the precondition loop was executed several more times for the miss pipeline, thus leading to significant performance degradation.

5 Refinements

The experimental data in Section 4 indicates that scheduling with reuse information can achieve performance equivalent to all-cache-miss with lower register pressure if we could properly handle references having self-spatial reuse. Due to potential alignment problems, just assuming the the first out of l (where l is the cache-line size) consecutive references is a miss and the others are hits is not adequate. Below we suggest two possible alternatives.

1. We could use a software-prefetch instruction on self-spatial references to bring in one or more consecutive cache lines. This would allow us to continue to schedule loads and stores as we currently do while avoiding the cache miss penalty. One could also prefetch references that are determined to be all cache misses [13]. However, this could potentially hurt the schedule due to an increase in the number of instructions issued.

2. We could also modify the hardware to prefetch the next cache line on a hit or miss if that line were not already in the cache. This would require no extra instructions and would allow references with self-spatial reuse to miss only on the first reference.

We suggest a combination of prefetching and scheduling with reuse information to obtain the best overall performance. The use of prefetching with self-spatial references eliminates the penalty of missing once per cache line. The use of scheduling with reuse information on cache misses allows us to hide the latency of a miss with little increase in registers over the all-cache-hit assumption and no additional overhead of prefetch instructions for each reference.

6 Conclusion

In this paper, we have demonstrated experimentally that using reuse information while software pipelining is effective. On our benchmark suite we produce on average 10% better schedules than an all-cache-hit assumption (a factor of 2.61 better on one loop) and on average we use 18% fewer registers than an all-cache-miss assumption. Even though all-cache-miss sometimes out performs reuse, it does so at the cost of 23% more registers. We have proposed a combination scheduling-with-reuse/prefetching scheme that will eliminate the edge in performance held by all-cache-miss at the cost of no extra registers.

Given that the cycle time of cache-miss latencies is increasing, software pipelining methods must eliminate the performance degradation caused by these latencies. The methods presented in this paper are an important step in eliminating the latency problem.

References

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

[2] Aiken, A., and Nicolau, A. 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, March 1988), pp. 221–235.

[3] Allan, V., Jones, R., Lee, R., and Allan, S. Software Pipelining. *ACM Computing Surveys 27*, 3 (September 1995).

[4] Allan, V., Rajagopalan, M., and Lee, R. 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).

[5] Carr, S., McKinley, K., and Tseng, C.-W. 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), pp. 252–262.

[6] Chen, T.-F., and Baer, J.-L. Reducing memory latency via non-blocking and prefetching caches. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, Massachusetts, 1992), pp. 51–61.

[7] Edmondson, J., Rubenfeld, P., and Preston, R. Superscalar instruction execution in the 21164 alpha microprocessor. *IEEE Micro 15*, 2 (Apr. 1995), 33–43.

[8] Huff, R. A. Lifetime-sensitive modulo scheduling. In *Conference Record of SIGPLAN Programming Language and Design Implementation* (June 1993).

[9] Kerns, D. R., and Eggers, S. J. Balanced instruction scheduling when memory latency is uncertain. In *Conference Record of SIGPLAN Programming Language and Design Implementation* (June 1993).

[10] Lam, M. Software pipelining: An effective scheduling technique for VLIW machines. *SIGPLAN Notices 23*, 7 (July 1988), 318–328. *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation.*

[11] Lam, M. Software pipelining: An effective scheduling technique for VLIW machines. In *Conference on Programming Language Design and Implementation* (Atlanta Georgia, June 1988), SIGPLAN '88, pp. 318–328.

[12] Lo, J. L., and Eggers, S. J. Improving balanced scheduling with compiler optimizations that increase instruction-level parallelism. In *Conference Record of SIGPLAN Programming Language and Design Implementation* (June 1995).

[13] Mowry, T. C., Lam, M. S., and Gupta, A. 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, 1992), pp. 62–75.

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

[15] Rau, B. R., Lee, M., Tirumalai, P., and Schlansker, M. S. Register Allocation for Modulo Scheduled Loops: Strategies, Algorithms, and Heuristics. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation"* (San Francisco, CA, June 1992).

[16] Rau, B. R., Yen, D. W. L., Yen, W., and Towle, R. A. The cydra 5 departmental supercomputer: Design philosophies, decisions, and trade-offs. *The IEEE* Computer (January 1989), 12–25.

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

[18] Warter, N. J., Mahlke, S. A., mei W. Hwu, W., and Rau, B. R. Reverse if-conversion. *SIGPLAN Notices 28*, 6 (June 1993), 290–299. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation.*

[19] Wolf, M. E., and Lam, M. S. A data locality optimizing algorithm. *SIGPLAN Notices 26*, 6 (June 1991), 30–44. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.*

Cache Hit		Cache Miss		Reuse	
Execution	Registers	Execution	Registers	Execution	Registers
Hydro2d					
100	16	141	22	100	16
100	16	110	24	110	16
198	19	119	24	100	16
100	18	120	25	100	18
100	18	119	27	100	18
100	47	100	47	101	47
159	36	100	45	107	36
125	18	100	18	105	18
177	45	100	48	114	48
115	35	100	45	112	35
100	16	141	22	100	16
110	23	108	44	100	23
100	25	114	32	100	25
105	25	100	33	105	25
103	41	100	42	104	41
192	22	100	27	100	27
100	15	116	21	100	15
100	15	117	21	100	15
100	15	116	21	100	15
100	15	116	21	100	15
131	41	100	44	131	41
122	38	100	51	122	38
117	20	100	33	117	20
128	21	100	36	128	21
134	21	100	36	134	21
120	20	100	33	118	20
113	19	100	32	113	29
123	20	100	35	123	20
134	19	100	24	134	19
142	20	100	32	142	20
169	21	100	36	169	21
149	46	100	68	149	46
100	17	119	22	100	17
130	14	100	22	130	14
154	46	100	68	154	46
134	14	100	21	134	14

Table 4: Performance Numbers — Normalized Execution and Registers

Cache Hit		Cache Miss		Reuse	
Execution	Registers	Execution	Registers	Execution	Registers
Su2cor					
261	22	100	30	121	30
103	62	100	92	106	62
156	80	100	59	100	59
100	16	100	16	100	16
100	17	114	27	100	17
100	59	110	68	100	59
100	20	117	30	100	20
110	25	100	31	110	25
118	23	100	31	118	23
119	77	100	68	119	77
140	59	100	75	140	59
116	72	100	82	104	75
209	64	100	71	100	71
128	12	100	17	128	12
100	29	100	29	100	29
114	63	100	63	114	63
Swm256					
104	110	100	142	103	110
150	36	100	46	100	46
100	38	105	42	100	38
123	92	100	108	123	92
123	33	100	58	100	58
124	54	100	50	124	54
105	36	100	46	104	36
129	25	100	29	100	29
100	43	110	73	101	43
180	27	100	32	180	27
Kernels					
100	19	100	19	100	19
100	12	100	17	100	12
100	20	100	20	100	20
153	20	104	25	100	25
101	19	100	19	101	19
100	12	100	17	100	12
100	15	100	15	100	15
101	129	102	77	100	130
122	14	100	20	122	14
100	20	100	20	100	20
164	21	100	27	100	27
100	20	100	20	100	20
100	22	108	50	100	22

Table 5: Performance Numbers — Normalized Execution and Registers