Improving Software Pipelining by Hiding Memory Latency with Combined Loads and Prefetches

Michael Bedy* Steve Carr† Soner Önder† Philip Sweany‡

Abstract

Modern processors and compilers hide long memory latencies through non-blocking loads or explicit software prefetching instructions. Unfortunately, each mechanism has potential drawbacks. Non-blocking loads can significantly increase register pressure by extending the lifetimes of loads. Software prefetching increases the number of memory instructions in the loop body. For a loop whose execution time is bound by the number of loads/stores that can be issued per cycle, software prefetching exacerbates this problem and increases the number of idle computational cycles in loops.

In this paper, we show how compiler and architecture support for combining a load and a prefetch into one instruction, called a prefetching load*, can give lower register pressure like software prefetching and lower load/store-unit requirements like non-blocking loads. On a set of 106 Fortran loops we show that prefetching loads obtain a speedup of 1.07–1.53 over using just non-blocking loads and a speedup of 1.04–1.08 over using software prefetching. In addition, prefetching loads reduced floating-point register pressure by as much as a factor of 0.4 and integer register pressure by as much as a factor of 0.8 over non-blocking loads. Integer register pressure was also reduced by a factor of 0.97 over software prefetching, while floating-point register pressure was increased by a factor of 1.02 versus software prefetching in the worst case.*

1 Introduction

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 cache miss latency, system designers have developed *non-blocking loads* and *software prefetching* instructions. Non-blocking loads allow cache accesses to continue when misses occur [14], allowing useful work to hide the latency of a cache miss. Software prefetching instructions bring a memory location into the cache in advance of when a load is issued to put the value in a register [10, 21]. Either of these latency hiding techniques can be valuable to the performance of memory systems.

Both of the above latency hiding techniques have disadvantages. Non-blocking loads can increase register pressure in loops significantly by lengthening the lifetimes of loads that are cache misses. Since advanced scheduling techniques such as software pipelining [4, 19, 23] already put a large demand on the register file, the additional pressure due to longer lifetimes can have a detrimental effect on performance. While software prefetching instructions do not increase the register pressure like non-blocking loads, they can cause degradation in loops whose performance is limited by the number of load/store instructions that can be issued per cycle. The additional memory instructions can increase the number of idle computational cycles if there is not a balance between computation and memory instructions.

In this paper, we describe how compilers and architecture can work together to implement *prefetching loads*, a single instruction that performs both a load and prefetch, and detect opportunities for using them effectively. We will show that prefetching loads both enhance the performance and reduce the register pressure of non-blocking load schemes. In addition, we will show that prefetching loads do not require the extra memory instructions required by software prefetching, giving better performance.

This paper begins in Section 2 with background material on memory-reuse analysis and an overview of software pipelining. Then, we give a review previous work on latency hiding in Section 3. Section 4 compares the dif-

*Compaq Computer, 110 Spit Brook Road (ZKO2-3/N30), Nashua NH 03062, *Michael.Bedy@compaq.com.*

†Department of Computer Science, Michigan Technological University, 1400 Townsend Dr., Houghton MI 49931-1295, {*carr,soner*}*@mtu.edu.*

‡Texas Instruments, P.O Box 660199, MS/8649, Dallas, TX 75266-0199, *sweany@ti.com*

ferences between non-blocking loads, software prefetching and prefetching loads. Section 5 presents the compiler and hardware support necessary for prefetching loads. Finally, Section 6 details the experimental evaluation of our proposed technique and Section 7 presents our conclusions.

2 Background

In order to utilize prefetching loads, the compiler must perform data-reuse analysis to determine if a prefetching load is profitable and perform software pipelining to schedule the prefetching load. In this section, we present the data-reuse analysis and software pipelining methods that we use in our compilation system.

2.1 Data-Reuse Analysis

Non-blocking loads, software prefetches and prefetching loads all require the compiler to determine which loads and stores will benefit from latency hiding because they are cache misses. Basically the compiler must determine the data reuse properties of each load in a loop.

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 memory hierarchy. Temporal and spatial reuse may result from *self-reuse* from a single array reference or *group-reuse* from multiple references.[1] In this paper we use the method developed by Wolf and Lam [29] to determine the reuse properties of loads An overview of their method follows.

A loop nest of depth n corresponds to a finite convex polyhedron Z^n, called an iteration space, bounded by the loop bounds. Each iteration in the loop corresponds to a node in the polyhedron, and is identified by its index vector $\vec{x} = (x_1, x_2, \ldots, x_n)$, where x_i is the loop index of the i^{th} loop in the nest, counting from the outermost to the innermost. The iterations that can exploit reuse are called the localized iteration space, L. The localized iteration space can be characterized as a localized vector space if the loop bounds are abstracted away.

For example, in the following piece of code, if $L = span\{(1,1)\}$, then data reuse for both `A(I)` and `A(J)` is exploited.

```
DO I= 1, N
  DO J = 1, N
    A(I) = A(J) + 2
  ENDDO
ENDDO
```

[1] Without loss of generality, we assume Fortran's column-major storage.

In Wolf and Lam's model, data reuse can only exist between *uniformly generated* references as defined below [18].

Definition 1 *Let n be the depth of a loop nest, and d be the dimensions of an array* `A`*. Two references* $\texttt{A}(f(\vec{x}))$ *and* $\texttt{A}(g(\vec{x}))$*, where f and g are indexing functions $Z^n \rightarrow Z^d$, are* uniformly generated *if*

$$f(\vec{x}) = H\vec{x} + \vec{c}_f \text{ and } g(\vec{x}) = H\vec{x} + \vec{c}_g$$

where H is a linear transformation and $\vec{c}_f$ and $\vec{c}_g$ are constant vectors.

For example, in the following loop,

```
DO I= 1, N
  DO J = 1, N
    A(I,J) = A(I,J-1) + A(I+1,J)
  ENDDO
ENDDO
```

the references to `A(I,J)`, `A(I,J-1)` and `A(I+1,J)` can be written as $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} I \\ J \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$, $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} I \\ J \end{bmatrix} + \begin{bmatrix} 0 \\ -1 \end{bmatrix}$, and $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} I \\ J \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix}$, respectively. References in a loop nest are partitioned into different sets, each of which operates on the same array and has the same H. These sets are called *uniformly generated sets* (UGSs).

A reference is said to have *self-temporal* reuse if $\exists \vec{r} \in L$ such that $H\vec{r} = \vec{0}$. A reference has *self-spatial* reuse if $\exists \vec{r} \in L$ such that $H_S\vec{r} = \vec{0}$, where H_S is H with the first row set to $\vec{0}$. Two distinct references in a UGS, $\texttt{A}(H\vec{x} + \vec{c}_1)$ and $\texttt{A}(H\vec{x} + \vec{c}_2)$ have *group-temporal* reuse if $\exists \vec{r} \in L$ such that $H\vec{r} = \vec{c}_1 - \vec{c}_2$. And finally, two references have *group-spatial* reuse if $\exists \vec{r} \in L$ such that $H_S\vec{r} = \vec{c}_{1,S} - \vec{c}_{2,S}$.

References can be partitioned into groups that have group-temporal reuse called *group-temporal sets* (GTS) and into groups that have group-spatial reuse called *group-spatial sets* (GSS), based upon solving the above equations. Since group-temporal reuse is a special case of group-spatial reuse, a GSS can contain many GTSs. The *leader* of a GSS (GTS) is the first reference to access the cache line (memory location) that is accessed by every array reference in the set. The *leading load* is the first load to access a particular cache line (memory location). Assuming that $L = span\{(0,1)\}$, in the previous example all references belong to the same GSS, `A(I,J)` and `A(I,J-1)` belong to one GTS, and `A(I+1,J)` belongs to another GTS. The leader of the GSS is `A(I+1,J)`.

References that have self-temporal, group-temporal or group-spatial reuse within the localized vector space are said to be cache hits. References that have only self-spatial reuse, are said to be cache misses once every l accesses, where l is the cache-line length, and cache hits otherwise. References that have no reuse are cache misses.

2.2 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 the execution of operations from different iterations of the loop. This overlapping of operations is analogous to hardware pipelining where speed-up is achieved by overlapping execution of different operations.

Allan et al. [4] provide a good summary of current software pipelining methods, dividing software pipelining techniques into two general categories called *kernel recognition* methods [2, 3, 5] and *modulo scheduling* methods [19, 27, 23]. The software pipelining used in this work is based upon modulo scheduling. 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. The number of cycles between the instantiation of successive loop iterations is called the initial interval (*II*). There are two constraints on the *II* of a loop. The first constraint, called the *ResII*, is the maximum number of instructions in a loop requiring a specific functional-unit resource. The second constraint, called the *RecII*, is found by examining the length of recurrences in the data dependence graph (DDG) for a loop. The minimum initiation interval (*MinII*) is the maximum of the *ResII* and *RecII*. In iterative modulo scheduling [23], first a schedule of *MinII* instructions is attempted. If a schedule is found that does not violate any resource or dependence constraints, modulo scheduling has achieved a minimum schedule. If not, scheduling is attempted with $MinII + 1$ instructions, and then $MinII + 2$, ..., continuing up to the worst case which is the *II* is the number of instructions required for local scheduling. The first value of *II* to produce a "legal" schedule of the DDG becomes the actual initiation interval. After a schedule for the loop itself has been found, code to set up the software pipeline (prelude) and drain the pipeline (postlude) are added. Rau [23] provides a detailed discussion of an implementation of modulo scheduling.

2.3 Increased Register Requirements

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, because the definition and use of a value may span multiple loop iterations. A register may be required for each loop iteration between the definition and use of a value. For example, if a definition and use of a value occurs on the same iteration in a loop one register will suffice for a value. However, if the definition and use of a value are separated by a number of iterations in order to obtain excellent parallelism, a register may be required for every iteration between the definition and use.

3 Previous Work

Callahan, *et al.* [10] describe a simple algorithm for insertion of software prefetches. In one loop iteration, all data needed on the next iteration is prefetched. This simple strategy results in a large percentage of unnecessary prefetches. They eliminate some of overhead by computing the number of loop iterations, called an *overflow iteration*, it takes to fill up the cache and eliminate prefetches for any array references that reuse a value before the overflow iteration occurs.

Mowry, *et al.* [21] describe an algorithm that issues fewer prefetches than Callahan's because it selectively prefetches only those items that are determined to be cache misses by memory reuse analysis. The algorithm also only issues a prefetch for self-spatial references once for each cache line. Additionally, Mowry bases the prefetching distance on the cycle time to memory and the *II* of the loop rather than just prefetching data for the next loop iteration.

Ding *et al.* [17] report on experiments that show non-blocking loads are an effective way to hide memory latency. They present a simple algorithm that assumes that any reference that has any kind of reuse is always a cache hit. Their study shows that using reuse analysis to determine load latencies is superior to assuming that all loads are cache hits or assuming all loads are cache misses.

Sánchez and González [25] describe a method for scheduling non-blocking loads called Cache Sensitive Modulo Scheduling (CSMS). Their method uses cache reuse analysis (including analysis of cache interferences) to determine whether array references will be cache hits or cache misses. If a load is determined to be a cache miss, but register pressure is estimated to be too high or the load increases the *RecII* of the loop, the load is scheduled as a cache hit. CSMS is shown to give better performance then the algorithm implemented by Ding, *et al*.

Sánchez and González also compare CSMS to software prefetching. However, they do not use full selective prefetching. For array references with self-spatial reuse they issue one prefetch per loop iteration or no prefetches, neither of which is as effective as Mowry's technique. Since prefetches for references with self-spatial reuse are only needed once per cache line, not using full selective prefetching can inhibit the performance of software prefetching. More prefetches are issued than necessary if references with self-spatial reuse are prefetches. Or, fewer prefetches than necessary are issued if no prefetches are issued for references with self-spatial reuse.

Carr and Sweany [13] describe a prefetching load that

only prefetches the next cache line, rather than using a prefetch offset. Their method was limited because prefetching one cache line ahead does not allow enough time to hide miss latency. We improve on this by using a prefetch offset to hide the full miss latency of a load.

4 A Motivating Example for Prefetching Loads

As a comparison of non-blocking loads, software prefetching and prefetching loads, consider the following loop.

```
DO J = 1, N
  DO I = 1, N
      ... = A(I,J) + B(J,I)
  ENDDO
ENDDO
```

Reuse analysis would determine that `A(I,J)` is a cache miss once out of every l references, where l is the cache-line size. If we issue a non-blocking load using the miss latency, there is needless increase in register pressure due to longer overlapped lifetimes for every $l-1$ out of l references. Note that it is difficult, in general, to determine which of the l references will incur the miss penalty because the alignment of `A(I,J)` within a cache line may be difficult, or impossible to determine at compile time (*e.g.*, loops in library code). If we assume that `A(I,J)` is always a cache hit, we keep the register pressure lower, but we pay the cache miss penalty once out of every l references. Neither assumption is adequate for loads with self-spatial reuse. In fact our experimentation with this scheme shows that this only give slightly better performance than assuming that all self-spatial load are cache hits. Finally, the reference to `B(J,I)` would be scheduled as a cache miss, potentially increasing register pressure significantly.

Mowry, *et al.*, would insert an explicit software prefetching instruction once every l iterations of the loop for `A(I,J)` and an explicit prefetch instruction for `B(J,I)` every iteration. Assuming that a cache line could hold four elements of `A(I,J)`, Mowry, *et al.*, would unroll the loop by four so that there would be only one prefetch of `A(I,J)` for each cache line. The resulting loop would have 13 memory operations per 4 floating point operations – worse than the two memory operations per floating point operations in the original loop (Note that loop with prefetching still has better performance because of cache performance improvement). Since modern architectures are often able to issue the same number of floating-point instructions and memory instructions in parallel, the software prefetches would exacerbate the already high demand for issuing memory instructions and leave more computational resources idle than if we could prefetch without additional instructions. So, while the latency could be hidden for all references, the loop would require a longer schedule than if no extra memory instructions were issued.

Our enhancement to latency hiding techniques is to introduce a new instruction, called a *prefetching load*, that is intended for references like `A(I,J)` and `B(J,I)` from the example. The instruction is like a normal load except that a prefetching distance is encoded in the offset in register + offset addressing mode or in a special register in register + register addressing mode.[2] The semantics of a prefetching load is to load the data at the address specified in the address register and prefetch the data at the address in the address register plus the offset (or register). If the prefetching distance is large enough, the part of the instruction that actually loads a value into a register will be a cache hit almost every time.

Prefetching loads can be seen as an enhancement to any non-blocking load scheme since using prefetching loads removes the need to extend register lifetimes like non-blocking loads. Additionally, there is no increase in the number of memory instructions issued like software prefetching. Potentially, prefetching loads can get the best of both non-blocking loads and software prefetching. In the example above, a prefetching load can be issued once every l iterations for `A(I,J)`, using loop unrolling, and once every iteration for `B(J,I)`. This would result in keeping the lower register pressure because all loads, and prefetching loads, can be scheduled using the cache hit latency. Also, this would keep the ratio of memory operations to floating-point operations at two to one, as opposed to the thirteen to four in the loop with software prefetching.

5 Prefetching Loads

In this section, we show how to determine which memory references can benefit from prefetching loads. Then, we describe our cache design.

5.1 Compiler Support for Identifying Candidates for Prefetching Loads

For each GSS that has a constant stride between references, we can issue a prefetching load for the leading load in the GSS. There are two types of GSSs that meet this requirement: (1) a GSS that has self-spatial reuse, or (2) a GSS that has no self reuse, but has the inner-loop induction variable appearing in only one subscript. In case (1), the

[2] Note that this does not eliminate register + offset mode for references that use prefetching loads. The prefetching load address can be used as the base address for other references to the same cache-line. The reference that uses the prefetching load will be the first reference to a particular cache line, as discussed in the next section.

prefetching load is once every l loop iterations. In case (2), the prefetching load is issued once per loop iteration. If the leading load of a GSS is removed by *scalar replacement* [9, 11], then no prefetching load is issued for that GSS.

As an example, consider the following loop.

```
DO J = 1, N
   DO I = 1, N
      A(I,J) = B(I,J) + B(I-2,J) +
               C(J,I) + C(J,I-1)
   ENDDO
ENDDO
```

In this loop, both `B(I,J)` and `C(J,I)` would be loaded using a prefetching load. Since we need only issue a prefetching load once out of every l references for `B(I,J)`, we unroll the loop by a factor of l and issue the prefetching load for just the original reference.

The prefetch distance is determined by calculating the number of loop iterations that are needed to hide the memory latency. For a software pipelined loop, this is[3]

$$\left\lceil \frac{\text{latency}}{\text{II}} \right\rceil \times \textit{data size}$$

Unfortunately, we cannot be assured of the alignment of a memory reference within the cache line. The alignment may vary on different executions of the loop and invocations of the containing function, or the alignment may not be known at compile time due to separate compilation. This makes it quite difficult to determine which of the l successive references will be hits and which will be misses. For references with self-spatial reuse this can have a significant effect on performance. If the address being prefetched is not aligned on a cache-line boundary, the prefetch will be less effective since a cache miss will still be incurred for a prefetch that is still in flight.

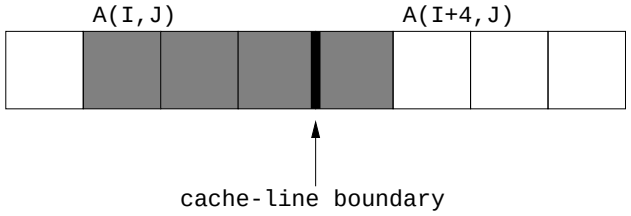

Figure 1. Alignment within a Cache Line

Consider prefetching `A(I,J)` from our previous example as shown in Figure 1. If we assume that a cache line contains four elements of `A`, it is possible for those four elements to be contained in two cache lines as shown in the shaded area. Thus, a miss to a line whose prefetch has not finished will occur once out of every four references.

To handle this case, we will set the prefetch distance to be one additional cache line ahead of what is computed above, *e.g.*, the line containing `A(I+4,J)`. If we prefetch the line with `A(I+4,J)` enough in advance, when `A(I,J)` is referenced both cache lines in the figure will be present in the cache. Note that the line containing `A(I,J)` would have been prefetched by an earlier iteration of the loop.

5.2 Cache Design

A prefetching load instruction specifies two memory operations, consisting of the actual load and the prefetching part. In order to handle both parts concurrently, we use a standard two-way interleaved cache to support up to two simultaneous accesses to the cache – one to each module. The MIPS R10000 processor employs such a cache with two banks [20]. The cost of interleaving the cache is the introduction of a multiplexer between the cache and the CPU (Figure 2).

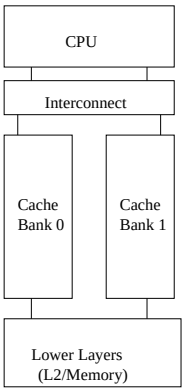

Figure 2. Interleaved Cache

In our design, a prefetching load will only proceed if both modules are available. If either cache module is not available, the instruction delays for one cycle and then tries to access both modules again on the next cycle. This continues until both modules can be accessed by the prefetching load. Note that if the prefetch address and the load address access the same cache module, the compiler will increase the prefetch offset by the l so that each cache lookup is in a different module. As a result, no delays are encountered because of bank conflicts of the two memory access components of the prefetching load instruction.

An alternative design to replicating the cache port is to allow the load portion of a prefetching load instruction to continue as if it is a regular load instruction and let the prefetching portion take over the cache port in a separate

[3] We assume that the stride of self-spatial references is one. This formula can easily be adapted to the case where the stride is greater than one

pipeline stage. In such a design, prefetching loads would keep a given cache port for two cycles as opposed to a single cycle utilization for cache hits. Since few prefetching loads are introduced compared to the total number of loads issued, this should not severely effect the performance of regular load instructions. This alternative approach has not been fully explored yet and deserves further study.

6 Experiment

We have implemented non-blocking loads, software prefetching and prefetching loads in our experimental software systems and performed an experiment on 106 loops from the SPEC95 benchmark suite and common kernels. Table 1 shows the benchmarks from which the loops come. Memoria, a source-to-source Fortran transformer based upon the D System infrastructure [1], performs the memory reuse analysis and determines which loads need latency hiding. This information is passed on to Rocket [26] via optimized intermediate code [6]. Rocket then software pipelines the code using appropriate reuse-based memory latencies.

Benchmark	# of Loops
101.tomcatv	2
102.swim	3
103.su2cor	22
104.hydro2d	38
110.applu	19
125.turb3d	7
141.apsi	3
kernels	11

Table 1. Benchmark Loops

Our target architecture for this experiment is a superscalar machine based upon the Unlimited Resource Machine (URM) [22] and has two integer functional units, each with a memory port, and two floating-point functional units. The architecture requires 3 cycles for integer operations and 3 cycles for floating-point operations. The cache for all experiments is as discussed in Section 5.2. We use 16K, 32K, and 64K direct-mapped and two-way set associative caches, each with a 32-byte line size. We use main memory cycle times of 25 and 75 cycles. There is a miss buffer with 16 entries. The architecture assumes that all branches will be taken. Finally, there are 128 integer and 128 floating-point registers. Since our software pipelining implementation does not allow register spilling, a large register set is needed to support scheduling of non-blocking loads.

We have implemented 4 different latency hiding schemes in our software system. Each method uses memory reuse analysis as described in Section 2.1 to determine which memory references are cache hits and which are cache misses. The first scheme (NBLH) uses only a non-blocking load to hide latency and assumes that all array references having any of self-temporal, self-spatial, group-temporal or group-spatial reuse are always cache hits. All other array references are assumed to be cache misses. The second scheme (NBLM) is the same as the first scheme, except that array references that only have self-spatial reuse are assumed to be always a miss. NBLH gives lower register pressure at the cost of performance. NBLM gives better performance at the cost of register pressure.

We have not implemented CSMS for this experiment. The main advantage of CSMS over the simpler methods is its handling of cache misses on recurrences. Prefetching loads could be used as an extension to CSMS also, where we'd expect to see improvements when self-spatial reuse is dominant.

The third scheme tested (Pf) uses software prefetching to hide latency. In this scheme we use full selective prefetching as done by Mowry [21]. Finally, the fourth scheme (Pfld) used for latency hiding is prefetching loads. If an array reference is a cache miss, but is not amenable to prefetching loads (*i.e.*, there is not a constant stride between memory accesses), we issue a non-blocking load using the cache miss latency.

To generate code, we first apply loop unrolling to loops that have array references with only self-spatial reuse. We unroll the loop by the number of array values that fit in a cache line. This allows us to use selective prefetching (and prefetching loads) on references with self-spatial reuse. We perform the unrolling for each latency-hiding scheme so that they each operate on the same code. After unrolling, we perform scalar replacement [12] and then perform memory reuse analysis on the resulting code. We also use array padding for arrays whose dimension sizes cause self interference [24]. The scalar optimizations that we use are constant propagation [28], global value numbering [8], partial redundancy elimination [7], operator strength reduction [15] and dead code elimination. We also generate code using register-plus-offset addressing mode to reduce the integer register pressure and address arithmetic. It is important to note that using register-plus-offset addressing mode is important to the performance of software prefetching. Previous work [16] has shown that the performance of software prefetching is degraded by approximately 20% if proper address arithmetic is not generated. After the code has been optimized, it is then software pipelined using our implementation of Rau's iterative modulo scheduling [23].

6.1 Initiation Interval

Table 2 shows the geometric mean increase in *II* for NBLH, NBLM and Pf versus prefetching loads. As pre-

dicted, software prefetching has the highest average *II* because of the additional memory operations. Since many loops are bound by memory accesses, adding additional memory operations increases the *ResII* and thus, the *II*. Prefetching loads achieved a slightly lower *II* than both NBLH and NBLM. This is likely due to the change in latencies for some memory operations.

	Memory Latency	
Method	***25 cycles***	***75 cycles***
NBLH	1.01	1.02
NBLM	1.01	1.02
Pf	1.08	1.08

Table 2. Geometric Mean Change in II vs. Prefetching Loads

6.2 Performance

Table 3 reports the geometric mean speedup of prefetching loads over each of the other latency hiding techniques. As expected NBLH performs the worst of the four techniques. Prefetching loads have an geometric mean speedup of 1.24–1.53 over NBLH. Pfld outperforms NBLM by 1.09–1.3.

	16K DM Cache		**16K 2-way Cache**	
	Memory Latency		*Memory Latency*	
Method	*25 cycles*	*75 cycles*	*25 cycles*	*75 cycles*
NBLH	1.24	1.50	1.24	1.51
NBLM	1.07	1.28	1.08	1.29
Pf	1.04	1.07	1.04	1.06
	32K DM Cache		**32K 2-way Cache**	
	Memory Latency		*Memory Latency*	
	25 cycles	*75 cycles*	*25 cycles*	*75 cycles*
NBLH	1.24	1.51	1.25	1.50
NBLM	1.09	1.29	1.09	1.29
Pf	1.04	1.07	1.05	1.06
	64K DM Cache		**64K 2-way Cache**	
	Memory Latency		*Memory Latency*	
	25 cycles	*75 cycles*	*25 cycles*	*75 cycles*
NBLH	1.24	1.51	1.25	1.53
NBLM	1.09	1.29	1.09	1.30
Pf	1.05	1.08	1.05	1.06

Table 3. Geometric Mean Speedup of Prefetching Loads

Pfld also outperforms software prefetching by a geometric mean speedup of 1.04–1.08. The performance improvement for Pfld is due to the larger achieved *II* for Pf. Pf had geometric mean increase in instructions executed of a factor of 1.07 at 25 cycles and 1.10 at 75-cycles. The reason for the difference deals with the unroll factors and the number of iterations for the pre-loops needed for unrolling.

In our test cases, Pfld had the best performance on 46% of the loops and tied for the best performance with Pf on 38% of the loops. Pf had the best performance on 10% of the loops and one of the non-blocking load schemes performed best on 6% of the loops. Most of the cases where prefetching performed best, it was by a factor of less that 1.02. Some of theses cases can be attributed to higher loop overhead due to differing unroll amounts. In the cases where non-blocking loads performed best, either the loop had too few iterations to benefit from prefetching or there was additional cache interference caused by aggressive prefetching.

6.3 Register Pressure

Table 4 shows the geometric mean increase in register pressure versus Pfld. Pfld not only provided better performance than NBLH, NBLM and Pf, but it also required fewer registers. NBLH required a factor of 1.01 more integer registers and a factor of 1.03–1.05 more floating-point registers. NBLM required a factor of 1.10–1.9 more integer registers and a factor of 1.26–2.51 more floating-point registers.

	Integer		**Floating Point**	
	Memory Latency		*Memory Latency*	
Method	*25 cycles*	*75 cycles*	*25 cycles*	*75 cycles*
NBLH	1.01	1.01	1.03	1.05
NBLM	1.10	1.26	1.9	2.51
Pf	1.03	1.03	0.98	1.02

Table 4. Geometric Mean Increase in Register Pressure vs. Prefetching Loads

Finally, the register pressure for Pf and Pfld was very close to the same. Pfld had a small decrease in integer register pressure over Pf and a slight increase in floating-point register pressure. This is likely due to the variance in the code generated due to different *II*s being used.

7 Conclusion

In this paper, we have shown that combining loads and prefetches into one instruction gives better results than non-blocking loads and explicit software prefetches alone. Prefetching loads eliminate the need to extend register lifetimes by scheduling some references as cache misses without increasing the resource requirements of a loop as done with

software prefetching. In this way, prefetching loads can get the best features of both non-blocking loads and software prefetching.

Our experiment with prefetching loads showed geometric mean speedup of 1.07–1.53 over using non-blocking loads and a speedup of 1.04–1.08 over software prefetching. Just as importantly, we observed a geometric mean reduction in floating-point register pressure by as much as a factor of 0.4 and a reduction in integer register pressure by as much as a factor of 0.8 versus non-blocking loads. Prefetching loads used a factor of 0.97 fewer integer registers than software prefetching and a factor of 1.02 more floating-point registers in the worst case.

In the future, we will investigate alternate cache designs and other structures to support prefetching loads. The primary goal will be to define a structure that effectively supports that access patterns of prefetching loads.

Given that memory latencies are increasing and that aggressive scheduling techniques such as software pipelining are necessary to get good performance on modern architectures, we need better methods to reduce the negative effects of long latencies. The use of prefetching loads as proposed in this paper is a promising step in alleviating the effects of long latencies for scientific program loops.

Acknowledgments

This research was partially supported by NSF grant CCR-9870871. The authors wish to thank RaeLyn Crowell, Don Darling and Dave Poplawski for their help in developing the software used in our experiments.

References

[1] V. Adve, J.-C. Wang, J. Mellor-Crummey, D. Reed, M. Anderson, and K. Kennedy. An integrated compilation and performance analysis environment for data parallel programs. In *Proceedings of Supercomputing '95*, San Diego, CA, Dec. 1995.

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

[3] 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*, pages 221–235, Nancy, France, March 1988.

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

[5] 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.

[6] P. Briggs. The massively scalar compiler project. Technical report, Rice Univeristy, July 1994. Preliminary version available via anonymous ftp.

[7] P. Briggs and K. D. Cooper. Effective partial redundancy elimination. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 159–170, Orlando, FL, June 1994.

[8] P. Briggs, K. D. Cooper, and L. T. Simpson. Value numbering. *Software – Practice & Experience*, 27(6):701–724, June 1997.

[9] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 53–65, White Plains, NY, June 1990.

[10] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, Santa Clara, California, 1991.

[11] S. Carr and K. Kennedy. Scalar replacement in the presence of conditional control flow. *Software Practice and Experience*, 24(1):51–77, Jan. 1994.

[12] 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*, pages 252–262, Santa Clara, California, 1994.

[13] S. Carr and P. Sweany. Improving software pipelining with hardware support for self-spatial loads. In *The Third Workshop on Interaction between Compilers and Computer Architecture (INTERACT-3)*, San Jose, CA, October 1998.

[14] T.-F. Chen and J.-L. Baer. 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*, pages 51–61, Boston, Massachusetts, 1992.

[15] K. D. Cooper, L. T. Simpson, and C. A. Vick. Operator strength reduction. Technical Report CRPC-TR95635-S, Center for Research on Parallel Computation, Rice Univeristy, October 1995.

[16] R. Crowell. An experimental evaluation of compiler-based cache management techniques. Master's thesis, Michigan Technological University, Mar. 1998.

[17] C. Ding, S. Carr, and P. Sweany. Modulo scheduling with cache reuse information. In *Proceedings of EuroPar '97*, Passau, Germany, August 1997.

[18] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformations. In *Proceedings of the First International Conference on Supercomputing*. Springer-Verlag, Athens, Greece, 1987.

[19] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 318–328, Atlanta, GA, July 1988.

[20] MIPS Technologies, Incorporated. *R10000 Microprocessor Product Overview*, October 1994.

[21] 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*, pages 62–75, Boston, Massachusetts, 1992.

[22] D. Poplawski. The unlimited resource machine (URM). Technical Report 95-01, Michigan Technological University, Jan. 1995.

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

[24] G. Rivera and C.-W. Tseng. Data transformations for eliminationg conflict misses. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 38–49, Montreal, Canada, June 17-19 1998.

[25] F. Sánchez and A. González. Cache-sensitive modulo scheduling. In *Proceedings of the 30th International Symposium on Microarchitecture (MICRO-30)*, Research Triangle Park, NC, December 1997.

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

[27] N. J. Warter, S. A. Mahlke, W.-M. Hwu, and B. R. Rau. Reverse if-conversion. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 290–299, Albuquerque, NM, June 1993.

[28] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, Apr. 1991.

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