Design and Evaluation of a Compiler Algorithm for Prefetching

Todd C. Mowry, Monica S. Lam and Anoop Gupta

Computer Systems Laboratory
Stanford University, CA 94305

Abstract

Software-controlled data prefetching is a promising technique for improving the performance of the memory subsystem to match today's high-performance processors. While prefetching is useful in hiding the latency, issuing prefetches incurs an instruction overhead and can increase the load on the memory subsystem. As a result, care must be taken to ensure that such overheads do not exceed the benefits.

This paper proposes a compiler algorithm to insert prefetch instructions into code that operates on dense matrices. Our algorithm identifies those references that are likely to be cache misses, and issues prefetches only for them. We have implemented our algorithm in the SUIF (Stanford University Intermediate Form) optimizing compiler. By generating fully functional code, we have been able to measure not only the improvements in cache miss rates, but also the overall performance of a simulated system. We show that our algorithm significantly improves the execution speed of our benchmark programs—some of the programs improve by as much as a factor of two. When compared to an algorithm that indiscriminately prefetches all array accesses, our algorithm can eliminate many of the unnecessary prefetches without any significant decrease in the coverage of the cache misses.

1 Introduction

With ever-increasing clock rates and the use of instruction-level parallelism, the speed of microprocessors has and will continue to increase dramatically. With numerical processing capabilities that rival the processing power of older generation supercomputers, these microprocessors are particularly attractive as scientific engines due to their cost-effectiveness. In addition, these processors can be used to build large-scale multiprocessors capable of an aggregate peak rate surpassing that of current vector machines.

Unfortunately, a high computation bandwidth is meaningless unless it is matched by a similarly powerful memory subsystem. These microprocessors tend to rely on caches to reduce their effective memory access time. While the effectiveness of caches has been well established for general-purpose code, their effectiveness for scientific applications has not. One manifestation of this is that several of the scalar machines designed for scientific computation do not use caches[6, 7].

1.1 Cache Performance on Scientific Code

To illustrate the need for improving the cache performance of microprocessor-based systems, we present results below for a set of scientific programs. For the sake of concreteness, we pattern our memory subsystem after the MIPS R4000. The architecture consists of a single-issue processor running at a 100 MHz internal clock. The processor has an on-chip primary data cache of 8 Kbytes, and a secondary cache of 256 Kbytes. Both caches are direct-mapped and use 32 byte lines. The penalty of a primary cache miss that hits in the secondary cache is 12 cycles, and the total penalty of a miss that goes all the way to memory is 75 cycles. To limit the complexity of the simulation, we assume that all instructions execute in a single cycle and that all instructions hit in the primary instruction cache.

We present results for a collection of scientific programs drawn from several benchmark suites. This collection includes NASA7 and TOMCATV from the SPEC benchmarks[27], OCEAN – a uniprocessor version of a SPLASH benchmark[25], and CG (conjugate gradient), EP (embarassingly parallel), IS (integer sort), MG (multigrid) from the NAS Parallel Benchmarks[3]. Since the NASA7 benchmark really consists of 7 independent kernels, we study each kernel separately (MXM, CFFT2D, CHOLSKY, BTRIX, GMTRY, EMIT and VPENTA). The performance of the benchmarks was simulated by instrumenting the MIPS object code using *pixie*[26] and piping the resulting trace into our cache simulator.

Figure 1 breaks down the total program execution time into instruction execution and stalls due to memory accesses. We observe that many of the programs spend a significant amount of time on memory accesses. In fact, 8 out of the 13 programs spend more than half of their time stalled for memory accesses.

1.2 Memory Hierarchy Optimizations

Various hardware and software approaches to improve the memory performance have been proposed recently[15]. A promising technique to mitigate the impact of long cache miss penalties is software-controlled prefetching[5, 13, 16, 22, 23]. Software-controlled prefetching requires support from both hardware and software. The processor must provide a special "prefetch" instruction. The software uses this instruction to inform the hardware of its intent to use a particular data item; if the data is not currently in the cache, the data is fetched in from memory. The cache must be lockup-free[17]; that is, the cache must allow multiple outstanding misses. While the memory services the data miss, the program can continue to execute as long as it does not need the requested data. While prefetching does not reduce the latency of the memory access, it hides the memory latency by overlapping the access with computation and other accesses. Prefetches on a scalar machine are analogous to vector memory accesses on a vector machine. In both cases, memory accesses are overlapped with computation and other accesses. Furthermore, similar to vector registers, prefetching allows caches in scalar machines to be managed by software. A major difference is that while vector machines can only operate on vectors in a pipelined manner, scalar machines can execute arbitrary sets of scalar operations well.

Another useful memory hierarchy optimization is to improve data

ASPLOS V - 10/92/MA,USA
© 1992 ACM 0-89791-535-6/92/0010/0062...$1.50

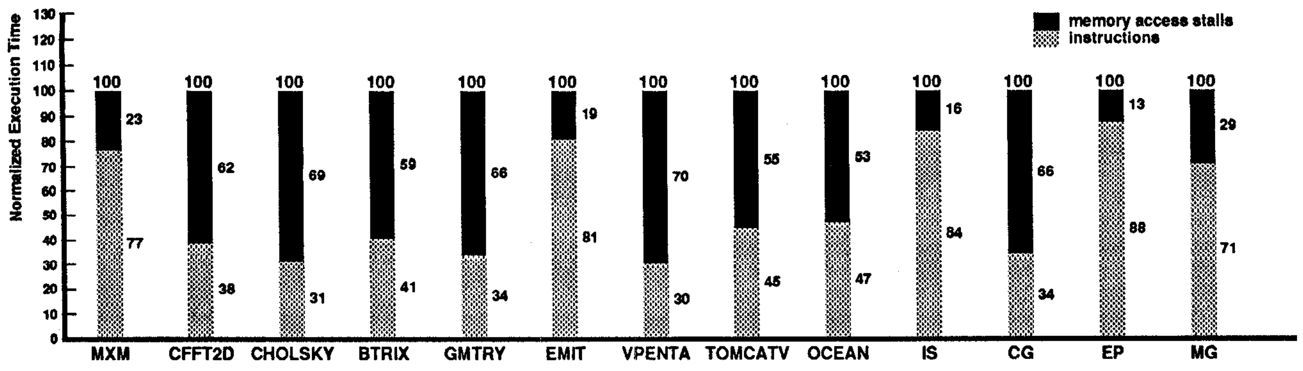


Figure 1: Breakdown of execution of scientific code.

locality by reordering the execution of iterations. One important example of such a transform is blocking[1, 9, 10, 12, 21, 23, 29]. Instead of operating on entire rows or columns of an array, blocked algorithms operate on submatrices or *blocks*, so that data loaded into the faster levels of the memory hierarchy are reused. Other useful transformations include unimodular loop transforms such as interchange, skewing and reversal[29]. Since these optimizations improve the code's data locality, they not only reduce the effective memory access time but also reduce the memory bandwidth requirement. Memory hierarchy optimizations such as prefetching and blocking are crucial to turn high-performance microprocessors into effective scientific engines.

1.3 An Overview

This paper proposes a compiler algorithm to insert prefetch instructions in scientific code. In particular, we focus on those numerical algorithms that operate on dense matrices. Various algorithms have previously been proposed for this problem [13, 16, 23]. In this work, we improve upon previous algorithms and evaluate our algorithm in the context of a full optimizing compiler. We also study the interaction of prefetching with other data locality optimizations such as cache blocking.

There are a few important concepts useful for developing prefetch algorithms. Prefetches are *possible* only if the memory addresses can be determined ahead of time. Prefetches are *unnecessary* if the data are already in the cache (or are being fetched into the cache) at the time of prefetch. Even if a necessary prefetch is issued, it may not be *effective*; it may be too early and the prefetched datum is replaced from the cache before it is used; it may be too late and the datum is used before it has been fetched from memory.

The domain of this work is the set of array accesses whose indices are affine (i.e. linear) functions of the loop indices. A substantial number of data references in scientific code belong to this domain. These references have the property that their addresses can be determined ahead of time, and prefetching these locations is therefore possible.

While these accesses are responsible for many of the cache misses, a significant number of them are also hits, as shown in Table 1. This table suggests that if prefetches were issued for all the affine array accesses, then over 60% of the prefetches would be unnecessary for most of the programs. It is important to minimize unnecessary prefetches[23]. Unnecessary prefetches incur a computation overhead due to the prefetches themselves and instructions needed to calculate the addresses. Prefetching can also increase the demand for memory bandwidth, which can result in delays for normal memory accesses as well as prefetches.

We have developed a compiler algorithm that uses *locality analysis* to selectively prefetch only those references that are likely

Table 1: Hit rates of affine array accesses.

Benchmark	Affine Access Hit Rate (%)
MXM	91.2
CFFT2D	87.7
CHOLSKY	60.9
BTRIX	68.7
GMTRY	36.0
EMIT	87.1
VPENTA	57.7
TOMCATV	91.5
OCEAN	91.9
IS	89.1
CG	88.9
EP	75.7
MG	95.4

to cause cache misses. To schedule the prefetch instructions early enough, we use the concept of software pipelining, where the computation of one iteration overlaps with the prefetches for a future iteration.

We have implemented our prefetch algorithm in the SUIF (Stanford University Intermediate Form) compiler. The SUIF compiler includes many of the standard optimizations and generates code competitive with the MIPS compiler[28]. Using this compiler system, we have been able to generate fully functional and optimized code with prefetching. (For the sake of simulation, prefetch instructions are encoded as loads to R0.) By simulating the code with a detailed architectural model, we can evaluate the effect of prefetching on overall system performance. It is important to focus on the overall performance, because simple characterizations such as the miss rates alone are often misleading. We have also evaluated the interactions between prefetching and locality optimizations such as blocking[29].

The organization of the rest of the paper is as follows. Section 2 describes our compiler algorithm, Section 3 describes our experimental framework, and Section 4 presents our results. In Section 4, we evaluate the effectiveness of our compiler algorithm on two variations of hardware support, and characterize the robustness of the algorithm with respect to its compile-time parameters. We also evaluate the interaction between prefetching and blocking. Section 5 discusses related work, and includes a comparison between software and hardware prefetching techniques. Section 6 describes future work to improve our algorithm, and finally, we conclude in Section 7.

2 A Prefetch Algorithm

In this section, we will use the code in Figure 2(a) as a running example to illustrate our prefetch algorithm. We assume, for this example, that the cache is 8K bytes, the prefetch latency is 100 cycles and the cache line size is 4 words (two double-word array elements to each cache line). In this case, the set of references that will cause cache misses can be determined by inspection (Figure 2(b)). In Figure 2(d), we show code that issues all the useful prefetches early enough to overlap the memory accesses with computation on other data. (This is a source-level representation of the actual code generated by our compiler for this case). The first three loops correspond to the computation of the `i=0` iteration, and the remaining code executes the remaining iterations. This *loop splitting* step is necessary because the prefetch pattern is different for the different iterations. Furthermore, it takes three loops to implement the innermost loop. The first loop is the *prolog*, which prefetches data for the initial set of iterations; the second loop is the *steady state* where each iteration executes the code for the iteration and prefetches for future iterations; the third loop is the *epilog* that executes the last iterations. This *software pipelining* transformation is necessary to issue the prefetches enough iterations ahead of their use[18, 24].

This example illustrates the three major steps in the prefetch algorithm:

1. For each reference, determine the accesses that are likely to be cache misses and therefore need to be prefetched.
2. Isolate the predicted cache miss instances through loop splitting. This avoids the overhead of adding conditional statements to the loop bodies.
3. Software pipeline prefetches for all cache misses.

In the following, we describe each step of the algorithm and show how the algorithm develops the prefetch code for the above example systematically.

2.1 Locality Analysis

The first step determines those references that are likely to cause a cache miss. This locality analysis is broken down into two substeps. The first is to discover the intrinsic data reuses within a loop nest; the second is to determine the set of reuses that can be exploited by a cache of a particular size.

2.1.1 Reuse Analysis

Reuse analysis attempts to discover those instances of array accesses that refer to the same cache line. There are three kinds of reuse: temporal, spatial and group. In the above example, we say that the reference `A[i][j]` has *spatial reuse* within the innermost loop since the same cache line is used by two consecutive iterations in the innermost loop. The reference `B[j][0]` has *temporal reuse* in the outer loop since iterations of the outer loop refer to the same locations. Lastly, we say that different accesses `B[j][0]` and `B[j+1][0]` have *group reuse* because many of the instances of the former refer to locations accessed by the latter.

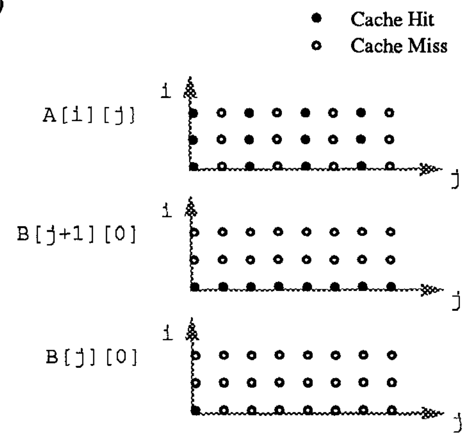
Trying to determine accurately all the iterations that use the same data is too expensive. We can succinctly capture our intuitive characterization that reuse is carried by a specific loop with the following mathematical formulation. We represent an n-dimensional loop nest as a polytope in an n-dimensional iteration space, with the outermost loop represented by the first dimension in the space. We represent the shape of the set of iterations that use the same data by a *reuse vector space*[29].

For example, the access of `b[j][0]` in our example is represented as `B(`$\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix}$`)`, so reuse occurs between iterations

(a)

```
for(i = 0; i < 3; i++)
  for(j = 0; j < 100; j++)
    A[i][j] = B[j][0] + B[j+1][0];
```

(b)



(c)

Reference	Locality	Prefetch Predicate
`A[i][j]`	$\begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} \text{none} \\ \text{spatial} \end{bmatrix}$	$(j \bmod 2) = 0$
`B[j+1][0]`	$\begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} \text{temporal} \\ \text{none} \end{bmatrix}$	$i = 0$

(d)

```
prefetch(&A[0][0]);
for(j = 0; j<6; j += 2) {
  prefetch(&B[j+1][0]);
  prefetch(&B[j+2][0]);
  prefetch(&A[0][j+1]);
}
for(j = 0; j<94; j += 2) {
  prefetch(&B[j+7][0]);
  prefetch(&B[j+8][0]);
  prefetch(&A[0][j+7]);
  A[0][j] = B[j][0]+B[j+1][0];
  A[0][j+1] = B[j+1][0]+B[j+2][0];
}
for(j = 94; j<100; j += 2) {
  A[0][j] = B[j][0]+B[j+1][0];
  A[0][j+1] = B[j+1][0]+B[j+2][0];
}
for(i = 1; i<3; i++) {
  prefetch(&A[i][0]);
  for(j = 0; j<6; j += 2)
    prefetch(&A[i][j+1]);
  for(j = 0; j<94; j += 2) {
    prefetch(&A[i][j+7]);
    A[i][j] = B[j][0]+B[j+1][0];
    A[i][j+1] = B[j+1][0]+B[j+2][0];
  }
  for(j = 94; j<100; j += 2) {
    A[i][j] = B[j][0]+B[j+1][0];
    A[i][j+1] = B[j+1][0]+B[j+2][0];
  }
}
```

Figure 2: Example of selective prefetching algorithm.

(i_1, j_1) and (i_2, j_2) whenever

$$\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i_1 \\ j_1 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i_2 \\ j_2 \end{bmatrix}, \text{ or}$$

$$\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i_1 - i_2 \\ j_1 - j_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

That is, temporal reuse occurs whenever the difference between the two iterations lies in the nullspace of $\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$, that is, span$\{(1,0)\}$. We refer to this vector space as the temporal reuse vector space. This mathematical approach succinctly captures the intuitive concept that the direction of reuse of `B[j][0]` lies along the outer loop. This approach can handle more complicated access patterns such as `C[i+j]` by representing their reuse vector space as span$\{(1,-1)\}$.

Similar analysis can be used to find spatial reuse. For reuse among different array references, Gannon *et al.* observe that data reuse is exploitable only if the references are *uniformly generated*; that is, references whose array index expressions differ in at most the constant term[11]. For example, references `B[j][0]` and `B[j+1][0]` are uniformly generated; references `C[i]` and `C[j]` are not. Pairs of uniformly generated references can be analyzed in a similar fashion[29]. For our example in Figure 2(a), our algorithm will determine that `A[i][j]` has spatial reuse on the inner loop, and both `B[j][0]` and `B[j+1][0]` share group reuse and also have temporal reuse on the outer loop.

2.1.2 Localized Iteration Space

Reuses translate to locality only if the subsequent use of data occurs before the data are displaced from the cache. Factors that determine if reuse translates to locality include the loop iteration count (since that determines how much data are brought in between reuses), the cache size, its set associativity and replacement policy.

We begin by considering the first two factors: the loop iteration count and the cache size. In the example above, reuse of `B[j][0]` lies along the outer dimension. If the iteration count of the innermost loop is large relative to the cache size (e.g., if the upper bound of the `j` loop in Figure 2(a) was 10,000 rather than 100), the data may be flushed from the cache before they are used in the next outer iteration. It is impossible to determine accurately whether data will remain in the cache due to factors such as symbolic loop iteration counts and the other cache characteristics. Instead of trying to represent exactly which reuses would result in a cache hit, we capture only the dimensionality of the iteration space that has data locality [29]. We define the *localized iteration space* to be the set of loops that can exploit reuse. For example, if the localized iteration space consists of only the innermost loop, that means data fetched will be available to iterations within the same innermost loop, but not to iterations from the outer loops.

The localized iteration space is simply the set of innermost loops whose volume of data accessed in a single iteration does not exceed the cache size. We estimate the amount of data used for each level of loop nesting, using the reuse vector information. Our algorithm is a simplified version of those proposed previously[8, 11, 23]. We assume loop iteration counts that cannot be determined at compile time to be small—this tends to minimize the number of prefetches. (Later, in Section 4.2, we present results where unknown loop iteration counts are assumed to be large). A reuse can be exploited only if it lies within the localized iteration space. By representing the localized iteration space also as a vector space, locality exists only if the reuse vector space is a subspace of the localized vector space.

Consider our example in Figure 2(a). In this case, the loop bound is known so our algorithm can easily determine that the volume of data used in each loop fits in the cache. Both loops are within the localized iteration space, and the localized vector space is represented as span$\{(1,0),(0,1)\}$. Since the reuse vector space is necessarily a subspace of the localized vector space, the reuses will correspond to cache hits, and it is not necessary to prefetch the reuses.

Similar mathematical treatment determines whether spatial reuse translates into spatial locality. For group reuses, our algorithm determines the sets among the group that can exploit locality using a similar technique. Furthermore, it determines for each set its *leading reference*, the reference that accesses new data first and is thus likely to incur cache misses. For example, of `B[j][0]` and `B[j+1][0]`, `B[j+1][0]` is the first reference that accesses new data. The algorithm need only issue prefetches for `B[j+1][0]` and not `B[j][0]`.

In the discussion so far, we have ignored the effects of cache conflicts. For scientific programs, one important source of cache conflicts is due to accessing data in the same matrix with a constant stride. Such conflicts can be predicted, and can even be avoided by embedding the matrix in a larger matrix with dimensions that are less problematic[19]. We have not implemented this optimization in our compiler. Since such interference can greatly disturb our simulation results, we manually changed the size of some of the matrices in the benchmarks (details are given in Section 3.) Conflicts due to interference between two different matrices are more difficult to analyze. We currently approximate this effect simply by setting the "effective" cache size to be a fraction of the actual cache size. We will discuss the robustness of this model in Section 4.2 and suggest some further optimizations in Section 6.

2.1.3 The Prefetch Predicate

The benefit of locality differs according to the type of reuse. If an access has temporal locality within a loop nest, only the first access will possibly incur a cache miss. If an access has spatial locality, only the first access to the same cache line will incur a miss.

To simplify this exposition, we assume here that the iteration count starts at 0, and that the data arrays are aligned to start on a cache line boundary. Without any locality, the default is to prefetch all the time. However, the presence of temporal locality in a loop with index i means that prefetching is necessary only when $i = 0$. The presence of spatial locality in a loop with index i means that prefetching is necessary only when $(i \bmod l) = 0$, where l is the number of array elements in each cache line. Each of these predicates reduces the instances of iterations when data need to be prefetched. We define the *prefetch predicate* for a reference to be the predicate that determines if a particular iteration needs to be prefetched. The prefetch predicate of a loop nest with multiple levels of locality is simply the conjunction of all the predicates imposed by each form of locality within the loop nest.

Figure 2(c) summarizes the outcome of the first step of our prefetch algorithm when applied to our example. Because of the small loop iteration count, all the reuse in this case results in locality. The spatial and temporal locality each translate to different prefetch predicates. Finally, since `B[j][0]` and `B[j+1][0]` share group reuse, prefetches need to be generated only for the leading reference `B[j+1][0]`.

2.2 Loop Splitting

Ideally, only iterations satisfying the prefetch predicate should issue prefetch instructions. A naive way to implement this is to enclose the prefetch instructions inside an `IF` statement with the prefetch predicate as the condition. However, such a statement in the innermost loop can be costly, and thus defeat the purpose of reducing the prefetch overhead. We can eliminate this overhead by decomposing the loops into different sections so that the predicates for all instances for the same section evaluate to the same value. This process is known as *loop splitting*. In general, the predicate $i = 0$ requires the first iteration of the loop to be *peeled*. The predicate $(i \bmod l) = 0$ requires the loop to be *unrolled* by a factor of l.

Peeling and unrolling can be applied recursively to handle predicates in nested loops.

Going back to our example in Figure 2(a), the `i = 0` predicate causes the compiler to peel the `i` loop. The (`j` mod 2) = 0 predicate then causes the `j` loop to be unrolled by a factor of 2—both in the peel and the main iterations of the `i` loop.

However, peeling and unrolling multiple levels of loops can potentially expand the code by a significant amount. This may reduce the effectiveness of the instruction cache; also, existing optimizing compilers are often ineffective for large procedure bodies. Our algorithm keeps track of how large the loops are growing. We suppress peeling or unrolling when the loop becomes too large. This is made possible because prefetch instructions are only hints, and we need not issue those and only those satisfying the prefetch predicate. For temporal locality, if the loop is too large to peel, we simply drop the prefetches. For spatial locality, when the loop becomes too large to unroll, we introduce a conditional statement. When the loop body has become this large, the cost of a conditional statement is relatively small.

2.3 Scheduling Prefetches

Prefetches must be issued early enough to hide memory latency. They must not be issued too early lest the data fetched be flushed out of the cache before they are used. We choose the number of iterations to be the unit of time scheduling in our algorithm. The number of iterations to prefetch ahead is

$$\left\lceil \frac{l}{s} \right\rceil$$

where l is the prefetch latency and s is the length of the shortest path through the loop body.

In our example in Figure 2(a), the latency is 100 cycles, the shortest path through the loop body is 36 instructions long, therefore, the `j` loops are software-pipelined three iterations ahead. Once the iteration count is determined, the code transformation is mechanical.

Since our scheduling quantum is an iteration, this scheme prefetches a data item at least one iteration before it is used. If a single iteration of the loop can fetch so much data that the prefetched data may be replaced, we suppress issuing the prefetch.

3 Experimental Framework

To experiment with prefetching, we extend our base R4000 architecture (described previously in Section 1.1) as follows. We augment the instruction set to include a prefetch instruction that uses a base-plus-offset addressing format and is defined to not take any memory exceptions. Both levels of the cache are lockup-free[17] in the sense that multiple prefetches can be outstanding. The primary cache is checked in the cycle the prefetch instruction is executed. If the line is already in the cache, the prefetch is discarded. Otherwise, the prefetch is sent to a *prefetch issue buffer*, which is a structure that maintains the state of outstanding prefetches. For our study, we assume a rather aggressive design of a prefetch issue buffer that contains 16 entries. If the prefetch issue buffer is already full, the processor is stalled until there is an available entry. (Later, in Section 4.3, we compare this with an architecture where prefetches are simply dropped if the buffer is full.) The secondary cache is also checked before the prefetch goes to memory. We model contention for the memory bus by assuming a maximum pipelining rate of one access every 20 cycles. Once the prefetched line returns, it is placed in both levels of the cache hierarchy. Filling the primary cache requires 4 cycles of exclusive access to the cache tags—during this time, the processor cannot execute any loads or stores.

Since regular cache misses stall the processor, they are given priority over prefetch accesses both for the memory bus and the cache tags. We assume, however, that an ongoing prefetch access cannot be interrupted. As a result, a secondary cache miss may be delayed by as many as 20 cycles (memory pipeline occupancy time) when it tries to access memory. Similarly the processor may be stalled for up to 4 cycles (cache-tag busy time) when it executes a load or store. If a cache miss occurs for a line for which there is an outstanding prefetch waiting in the issue buffer, the miss is given immediate priority and the prefetch request is removed from the buffer. If the prefetch has already been issued to the memory system, any partial latency hiding that might have occurred is taken into account.

As we mentioned before in Section 1, the benchmarks evaluated in this paper are all scientific applications taken from the SPEC, SPLASH and NAS Parallel benchmark suites. For four of the benchmarks (MXM, CFFT2D, VPENTA and TOMCATV), we manually changed the alignment of some of the matrices to reduce the number of cache conflicts.

The prefetching algorithm has a few compile-time parameters, which we consistently set as follows: *cache line size* = 32 bytes, *effective cache size* = 500 bytes, and *prefetch latency* = 300 cycles. As discussed in Section 2.1.2, we choose an effective cache size to be a fraction of the actual size (8 Kbytes) as a first approximation to the effects of cache conflicts (we consider the effects of varying this parameter in Section 4.2). The *prefetch latency* indicates to the compiler how many cycles in advance it should try to prefetch a reference. The prefetch latency is larger than 75 cycles, the minimum miss-to-memory penalty, to account for bandwidth-related delays.

4 Experimental Results

We now present results from our simulation studies. We start by evaluating the effectiveness of our compiler algorithm, including the key aspects of locality analysis, loop splitting and software pipelining. We evaluate the sensitivity of our performance results to variations in the architectural parameters used by the compiler. We then compare two different architectural policies for handling situations when the memory subsystem is saturated with prefetch requests and cannot accept any more. Finally, we explore the interaction between prefetching and locality optimizations such as cache blocking.

4.1 Performance of Prefetching Algorithm

The results of our first set of experiments are shown in Figure 3 and Table 2. Figure 3 shows the overall performance improvement achieved through our selective prefetching algorithm. For each benchmark, the two bars correspond to the cases with no prefetching (N) and with selective prefetching (S). In each bar, the bottom section is the amount of time spent executing instructions (including instruction overhead of prefetching), and the section above that is the memory stall time. For the prefetching cases, there is also a third component—stall time due to memory overheads caused by prefetching. Specifically, the stall time corresponds to two situations: (1) when the processor attempts to issue a prefetch but the prefetch issue buffer is already full, and (2) when the processor attempts to execute a load or store when the cache tags are already busy with a prefetch fill.

As shown in Figure 3, the speedup in overall performance ranges from 5% to 100%, with 6 of the 13 benchmarks improving by over 45%. The memory stall time is significantly reduced in all the cases. Table 2 indicates that this is accomplished by reducing both the primary miss-rate and the average primary-miss penalty. The miss penalty is reduced because even if a prefetched line is replaced from the primary cache before it can be referenced, it is still likely to be present in the secondary cache. Also, the miss latency may be partially hidden if the miss occurs while the prefetch access is still in progress. Overall, 50% to 90% of the original memory stall cycles are eliminated.

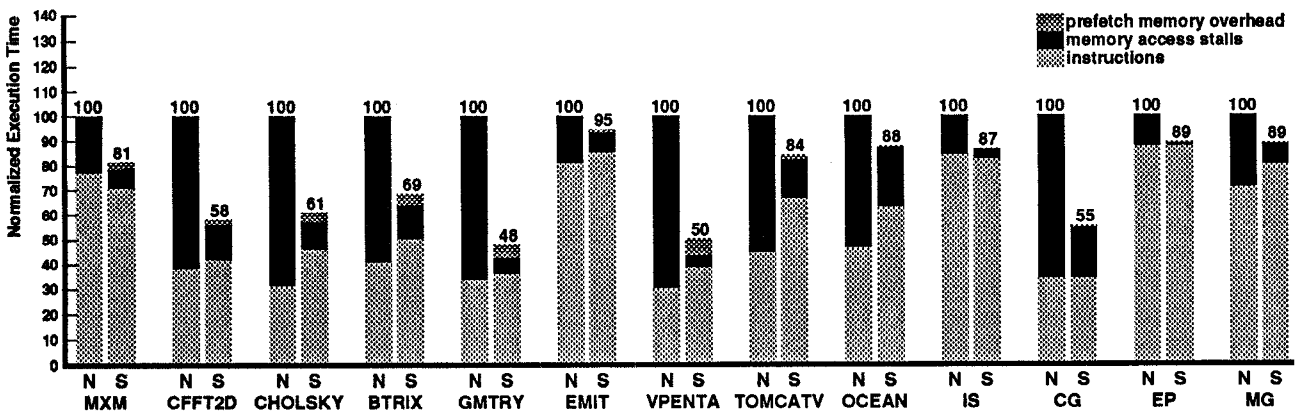


Figure 3: Overall performance of the selective prefetching algorithm (N = no prefetching, and S = selective prefetching).

Table 2: Memory performance improvement for the selective prefetching algorithm.

	No Prefetch			Selective Prefetch		
Benchmark	Refs per Inst	Miss Rate (%)	Average Miss Penalty (cycles)	Miss Rate (%)	Average Miss Penalty (cycles)	Memory Stall Reduction (%)
MXM	0.58	3.35	15.6	1.26	14.0	66.2
CFFT2D	0.55	5.53	53.2	1.29	53.4	76.5
CHOLSKY	0.45	19.69	24.8	6.09	12.3	83.9
BTRIX	0.55	12.29	21.3	3.20	17.0	76.5
GMTRY	0.47	19.29	21.7	3.65	12.3	90.2
EMIT	0.56	2.26	18.7	1.26	13.7	58.4
VPENTA	0.53	16.52	26.4	1.91	12.5	93.0
TOMCATV	0.48	7.06	36.6	4.21	12.5	71.8
OCEAN	0.43	5.07	52.3	2.20	48.6	54.6
IS	0.32	1.62	36.9	0.93	16.0	75.1
CG	0.40	12.57	38.3	8.33	17.5	69.2
EP	0.23	1.05	59.5	0.31	13.8	93.3
MG	0.60	2.09	33.0	1.54	15.7	71.4

Having established the benefits of prefetching, we now focus on the costs. Figure 3 shows that the instruction overhead of prefetching causes less than a 15% increase in instruction count in over half of the benchmarks. In fact, in two of those cases (MXM and IS) the number of instructions actually decreased due to savings through loop unrolling. In other cases (CHOLSKY, BTRIX, VPENTA, TOMCATV, OCEAN), the number of instructions increased by 25% to 50%. Finally, the stalls due to prefetching memory overhead are typically small—never more than 15% of original execution time. In each case, we observe that the overheads of prefetching are low enough compared to the gains that the net improvement remains large. In the following subsections, we present a detailed evaluation of each aspect of our selective prefetching algorithm.

4.1.1 Locality Analysis

The goal of using locality analysis is to eliminate prefetching overhead by prefetching only those references that cause cache misses. To evaluate the effectiveness of this analysis, we performed the following experiment. We implemented a compiler algorithm which prefetches *all* data references within our domain (i.e. array references whose indices are affine functions of loop indices). We refer to this algorithm as *indiscriminate* (as opposed to *selective*) prefetching. The indiscriminate algorithm uses the same software pipelining technique as selective prefetching to schedule prefetches far enough in advance. However, it has no need for locality analysis or loop splitting.

The results of this experiment are shown in Figure 4 and in Tables 3 and 4. The ideal selective prefetch algorithm would achieve the same level of memory stall reduction while decreasing the overheads associated with issuing unnecessary prefetches.

Figure 4 shows that the speedup offered by prefetching selectively rather than indiscriminately ranges from 1% to 100%. In 6 of the 13 cases, the speedup is greater than 20%. Table 3 shows that most of the benchmarks sacrifice very little in terms of memory stall reduction by prefetching selectively. On the other hand, Figure 4 shows that indiscriminate prefetching suffers more from both increased instruction overhead and stress on the memory subsystem. Overall, selective prefetching is effective. In some cases (CFFT2D and MG), selectiveness even turns prefetching from a performance loss into a performance gain.

Table 3: Memory performance improvement for the indiscriminate and selective prefetching algorithms.

	Indiscriminate Prefetch			Selective Prefetch		
Benchmark	Miss Rate (%)	Average Miss Penalty (cycles)	Memory Stall Reduction (%)	Miss Rate (%)	Average Miss Penalty (cycles)	Memory Stall Reduction (%)
MXM	1.37	13.0	66.0	1.26	14.0	66.2
CFFT2D	0.34	16.6	98.1	1.29	53.4	76.5
CHOLSKY	6.27	12.2	84.1	6.09	12.3	83.9
BTRIX	2.10	13.7	87.1	3.20	17.0	76.5
GMTRY	3.22	12.4	89.4	3.65	12.3	90.2
EMIT	1.01	14.2	65.7	1.26	13.7	58.4
VPENTA	2.48	12.5	90.2	1.91	12.5	93.0
TOMCATV	3.25	12.4	73.5	4.21	12.5	71.8
OCEAN	2.38	49.8	45.0	2.20	48.6	54.6
IS	0.87	16.3	76.4	0.93	16.0	75.1
CG	7.20	14.1	75.9	8.33	17.5	69.2
EP	0.35	13.8	92.3	0.31	13.8	93.3
MG	1.03	15.5	72.7	1.54	15.7	71.4

We evaluate the selective algorithm in more detail by using the following two concepts. The *coverage factor* is the fraction of original misses that are prefetched. A prefetch is *unnecessary* if the line is already in the cache or is currently being fetched. An ideal prefetching scheme would provide a coverage factor of 100% and would generate no unnecessary prefetches.

Table 4 contains several statistics about the effectiveness of the two prefetching algorithms. This includes the percentage of prefetches issued that are unnecessary, and a breakdown of the impact of prefetching on the original cache misses. This breakdown contains three categories: (1) those that are prefetched and subsequently hit in the primary cache (*pf-hit*), (2) those that are prefetched but

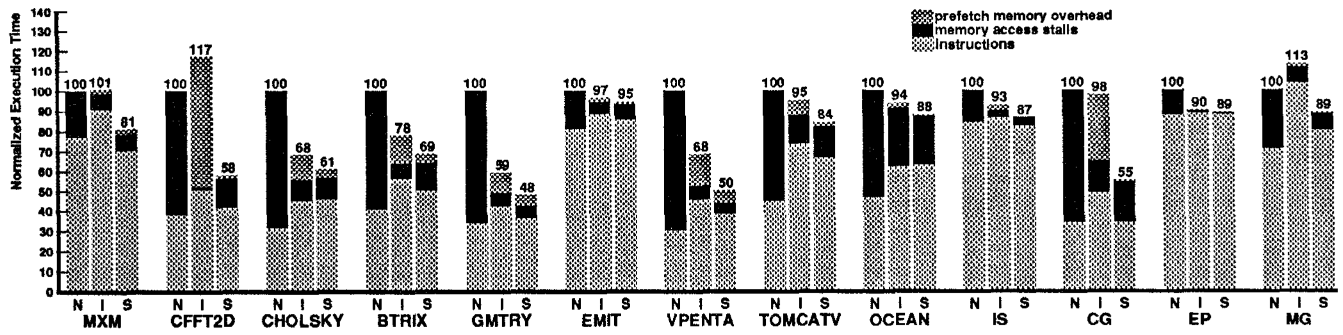


Figure 4: Overall performance comparison between the indiscriminate and selective prefetching algorithms (N = no prefetching, I = indiscriminate prefetching, and S = selective prefetching).

Table 4: Prefetching effectiveness for the indiscriminate and selective prefetching algorithms.

	Indiscriminate Prefetch					Indiscriminate to Selective	Selective Prefetch					
	Unnecessary	Original Miss Breakdown (%)			Inst		Unnecessary	Original Miss Breakdown (%)			Inst per PF	
Benchmark	Prefetch %	pf-hit	pf-miss	nopf-miss	per PF	PF Ratio	Prefetch %	pf-hit	pf-miss	nopf-miss	wrt Orig	wrt Split
MXM	91.2	59.2	6.2	34.6	1.3	4.7	60.4	59.2	8.3	32.5	-2.7	3.9
CFFT2D	87.7	93.8	1.4	4.8	1.4	9.4	6.0	76.5	1.1	22.3	3.5	3.5
CHOLSKY	61.0	67.7	28.4	3.9	2.0	2.3	8.6	67.5	29.5	3.0	4.9	5.8
BTRIX	68.7	79.9	15.5	4.6	1.8	1.7	53.6	70.6	14.4	15.0	1.8	2.1
GMTRY	36.0	81.5	15.7	2.8	1.9	1.5	3.5	82.5	15.8	1.6	0.8	0.8
EMIT	87.1	54.6	4.1	41.2	1.6	6.2	37.4	42.2	4.4	53.5	5.8	5.8
VPENTA	57.7	79.3	17.5	3.3	2.5	1.8	29.4	85.2	4.8	10.0	2.5	2.9
TOMCATV	91.5	22.1	17.8	60.1	4.0	2.3	81.4	17.7	20.4	61.9	7.0	7.0
OCEAN	91.9	42.2	6.7	51.1	2.5	6.6	42.0	51.6	0.8	47.6	17.5	18.4
IS	89.1	46.6	3.3	50.1	1.0	8.4	10.5	42.8	6.1	51.1	-7.9	2.1
CG	88.9	34.6	3.6	61.8	2.5	6.7	25.0	32.8	5.3	61.9	-0.1	7.2
EP	75.7	66.7	3.1	30.3	1.8	4.1	0.0	67.6	3.1	29.3	2.4	2.4
MG	95.4	41.9	7.1	51.1	3.4	20.9	18.4	39.8	1.8	58.5	20.2	20.2

remained misses (*pf-miss*), and (3) those that are not prefetched (*nopf-miss*). The coverage factor is equal to the sum of the *pf-hit* and *pf-miss* categories.

The coverage factor of the indiscriminate case is interesting, since it represents the fraction of cache misses that are within the domain of our analysis in most cases. Looking at Table 4, we notice that in 5 of the 13 cases the coverage factor is well over 90%, but in 5 of the other cases it is less than 50%. In the case of CG, the coverage is only 38% because it is a sparse matrix algorithm and we are only prefetching the dense index arrays. The same is true for IS. MXM (a blocked matrix multiplication kernel) is a surprising case, since all of the important references are obviously affine, yet the coverage factor is only 65%. This is a result of the way we account for prefetches in our experiment; we associate a prefetch only with the very next reference to the same cache line. Suppose the algorithm issues two prefetches for the same line followed by references to two consecutive words in that cache line, we say that the first reference is prefetched but not the second. In the case of MXM, cache conflicts between accesses to different arrays can cause the second access to the same cache line to miss. Similar behavior also occurs in TOMCATV, OCEAN, and MG. Finally, in the cases of EMIT and EP, many of the remaining cache misses occur in library routines rather than the programs themselves.

Table 4 shows that a large fraction of prefetches issued under the indiscriminate scheme are unnecessary. In all but one case, this fraction ranged from 60% to 95%. These unnecessary prefetches can lead to large instruction overheads (MG) or significant delays due to a saturated memory subsystem (CFFT2D and CG).

A selective algorithm is successful if it can maintain a similar coverage while lowering the number of unnecessary prefetches. Table 4 shows that in 11 of the 13 cases, the coverage is reduced by less than 10%. Table 3 also supports this by showing that the miss rates have not increased substantially, and the reduction in memory stall cycles is comparable. In the cases where the coverage did go down, the problem is typically due to the presence of cache conflicts. Comparing the percentages of unnecessary prefetches in Table 4, we see that the improvement from selective prefetching is dramatic in many cases (CFFT2D, CHOLSKY, IS, EP, MG). (Note that these percentages are computed with respect to the number of prefetches issued, which changes between the two cases.)

The advantage of selective prefetching is summarized by the ratio of indiscriminate to selective prefetches in Table 4. Prefetching selectively can reduce the number of prefetches by as much as a factor of 21. At the same time, the coverage factor remains competitive. Overall, this selection process appears to be quite successful.

4.1.2 Loop Splitting

The goal of loop splitting is to isolate the cache miss instances while introducing as little instruction overhead as possible. To quantify the advantage of loop splitting, we implemented the naive alternative for isolating cache miss instances—placing conditional statements inside the loops. Figure 5 shows that for 5 of the 13 benchmarks (MXM,BTRIX,VPENTA,CG and MG), the performance advantage of loop splitting is greater than 25%.

A good measure of the success of loop splitting is the instruction overhead per prefetch issued. Ideally, isolating the cache miss instances will not increase the instruction overhead. One of the advantages of having implemented the prefetching schemes in the compiler is that we can quantify this instruction overhead. Previous studies have only been able to estimate instruction overhead [4].

Table 4 shows the number of instructions required to issue each

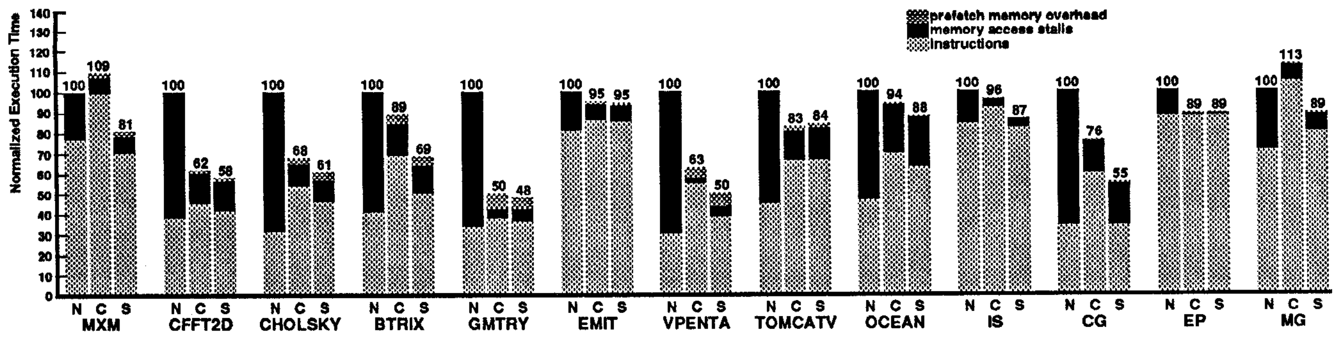


Figure 5: Loop splitting effectiveness (N = no prefetching, C = selective prefetching with conditional statements, and S = selective prefetching with loop splitting).

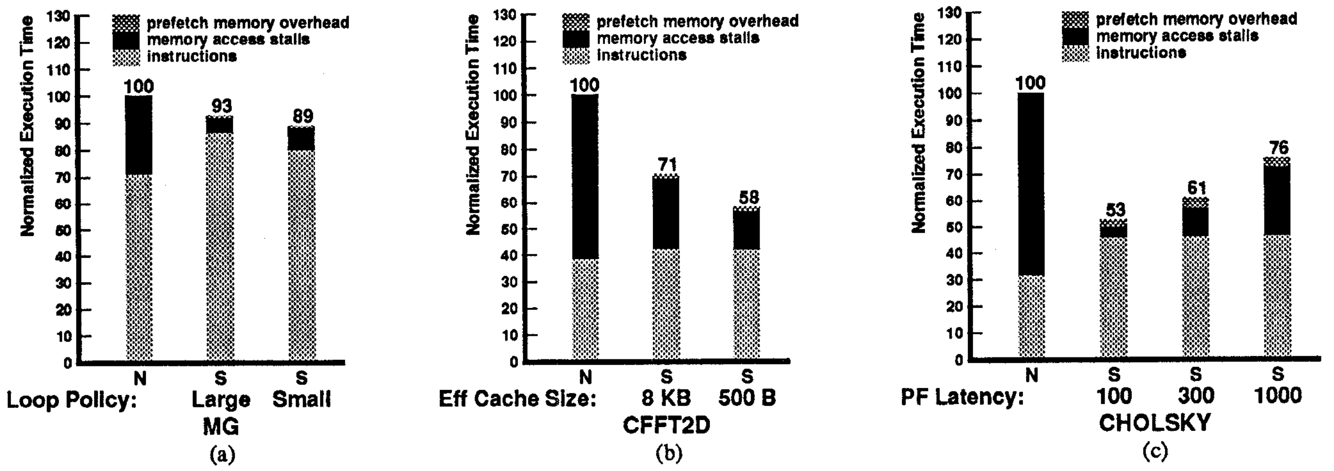


Figure 6: Sensitivity of results to compile-time parameters (N = no prefetching, S = selective prefetching variations).

prefetch. For the indiscriminate prefetching scheme, the overhead per prefetch ranges from 1 to 4 instructions. This is well within the bounds of what one would intuitively expect. One of the instructions is the prefetch itself, and the rest are for address calculation. For the selective prefetching scheme, Table 4 shows the prefetch instruction overhead both with respect to the original code and with respect to code where the loop splitting transformations are performed but no prefetches are inserted. Loop splitting generally increases the overhead per prefetch. In a few cases, the overhead has become quite large (OCEAN and MG). In other cases, the overhead with respect to the original code is actually negative, due to the savings through loop unrolling (MXM, IS and CG). In the case of OCEAN, the loop bodies are quite large, and the combination of loop unrolling and software pipelining makes it necessary for the compiler to spill registers. The penalty for register spills is averaged over just the prefetches, and this penalty can become quite high. In the case of MG, the number of prefetches has been drastically reduced (by a factor of 21). Averaging all the loop and transformation overheads over only a small number of prefetches results in a high instruction-per-prefetch overhead. In most of the cases, however, the overhead per prefetch remains low.

4.1.3 Software Pipelining

The effectiveness of the software pipelining algorithm is reflected by the *pf-miss* figures in Table 4. A large number means that the prefetches are either not issued early enough, in which case the line does not return to the primary cache by the time it is referenced, or not issued late enough, in which case the line has already been replaced in the cache before it is referenced. The results indicate that the scheduling algorithm is generally effective. The exceptions are CHOLSKY and TOMCATV, where over a third of the prefetched references are not found in the cache. The problem in these cases is that cache conflicts remove prefetched data from the primary cache before it can be referenced. However, there is still a performance advantage since the data tends to remain in the secondary cache, and therefore the primary-miss penalty is reduced, as shown earlier in Table 2.

4.1.4 Summary

To summarize, we have seen that in most cases the selective prefetching scheme performs noticeably better than the indiscriminate scheme. The advantage comes primarily from a reduction in prefetching overhead while still maintaining a comparable savings in memory stall time.

4.2 Sensitivity to Compile-Time Parameters

The selective prefetching algorithm uses several compile-time parameters to model the behavior of the memory subsystem. Specifically these parameters include the following: (1) cache line size, (2) whether unknown loop bounds are assumed to be large or small, (3) effective cache size, and (4) prefetch latency. The most concrete of these parameters is the cache line size, which can be set precisely. The other parameters, however, are more heuristic in nature. To evaluate the robustness of our algorithm, we measured the effects of varying these less obvious parameters.

The SUIF compiler performs aggressive interprocedural constant

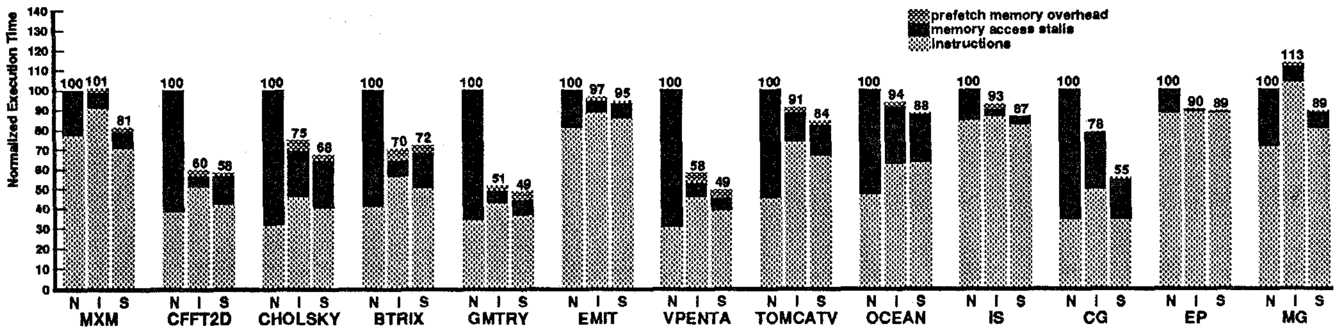


Figure 7: Prefetches are dropped (rather than stalling) when issue buffers are full (N = no prefetching, I = indiscriminate prefetching, and S = selective prefetching).

propagation to statically determine as many loop bounds as possible. When this fails, our current algorithm assumes unknown loop counts to be small, which tends to overestimate what remains in the cache. When the compiler assumes unknown iteration counts to be large, it produces identical code for 11 of the 13 benchmarks—the two benchmarks that change are OCEAN and MG. For OCEAN, the difference in performance is negligible. However, MG performs 4% worse with the large-loop policy, as shown in Figure 6(a). In this case, the benefit of the extra prefetches is more than offset by increased instruction overhead. Although assuming small loop counts happens to be better in this case, the opposite could easily be true in programs where unknown iteration counts are actually large. One solution may be to resolve loop counts through profiling feedback. However, the more interesting result of this experiment is how rarely loop bound uncertainty affects performance. We observe that while nearly half the benchmarks contain loops with unknown bounds, in most cases this has no impact on locality, due to the patterns of data reuse within those loops.

For our experiments so far, the effective cache size has been set to 500 bytes, which is only a small fraction of the actual cache size (8 Kbytes). When the effective cache size is set to the full 8 Kbytes, our compiler generates identical code for 7 of the 13 benchmarks. For 5 of the 6 benchmarks that do change, the difference in performance is negligible. The one case that changes significantly is CFFT2D, as shown in Figure 6(b). In this case, fewer prefetches are issued with the larger effective cache size. However, the prefetches that are eliminated happen to be useful, since they fetch data that is replaced due to cache conflicts. As a result, the performance suffers, as we see Figure 6(b). (Note that this is in contrast with the effect we see in Figure 6(a), where issuing more prefetches hurts performance.) In the case of CFFT2D, many critical loops reference 2 Kbytes of data, and these loops happen to suffer from cache conflicts. An effective cache size of 500 bytes produces the desired result in this case. Overall, however, the results are robust with respect to effective cache size.

Finally, for our experiments in Section 4.1, we set the prefetch latency to 300 cycles. We chose a value greater than 75 cycles to account for bandwidth-related delays. To evaluate whether this was a good choice, we compiled each benchmark again using prefetch latencies of 100 and 1000 cycles. In nearly all the cases, the impact on performance is small. In many cases, the 100-cycle case is slightly worse than the 300-cycle case due to bandwidth-related delays. The most interesting case is CHOLSKY, as shown in Figure 6(c). In this case, prefetched data tends to be replaced from the cache shortly after it arrives, so ideally it should arrive "just in time". Therefore, the lowest prefetch latency (100 cycles) offers the best the performance, as we see in Figure 6(c). However, in such cases the best approach may be to eliminate the cache conflicts cause this behavior[19].

In summary, the performance of our selective algorithm was affected noticeably in only one of the 13 benchmarks for each parameter we varied. Overall, the algorithm appears to be quite robust.

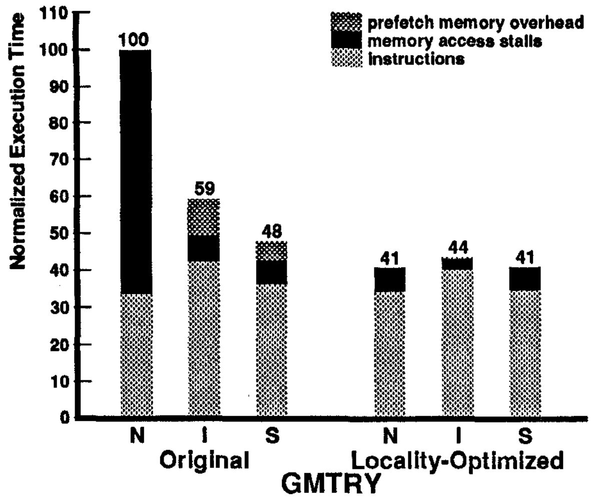
4.3 Dropping Prefetches vs. Stalling

In the architectural model presented so far, the memory subsystem has a finite (16-entry) prefetch issue buffer to hold outstanding prefetch requests. Our model includes a few hardware optimizations to help mitigate the negative effects of the finite buffer size. In particular, a prefetch is only inserted into the buffer if it misses in the primary cache and there is not already an outstanding prefetch for the same cache line. Also, a prefetch is removed from the issue buffer as soon as it completes (i.e. the buffer is not a FIFO queue). However, in spite of these optimizations, the buffer may still fill up if the processor issues prefetches faster than the memory subsystem can service them.

Once the prefetch issue buffer is full, the processor is unable to issue further prefetches. The model we use so far stalls the processor until a buffer entry becomes available. An alternative is to simply drop the prefetch and continue executing. Intuitive arguments might be presented to support either approach. On one hand, if the data is needed in the future and is not presently in the cache (since only prefetches that miss go into the buffer), it may appear to be cheaper to stall now until a single entry is free rather than to suffer an entire cache miss sometime in the future. On the other hand, since a prefetch is only a performance hint, perhaps it is better to continue executing useful instructions.

To understand this issue, we ran each of our benchmarks again using a model where prefetches are dropped rather than stalling the processor when the prefetch issue buffer is full. The results of this experiment are shown in Figure 7. Comparing this with Figure 4, we see that there is a difference in performance for seven of the cases (CFFT2D, CHOLSKY, BTRIX, GMTRY, VPENTA, TOMCATV, and CG). In each of these cases, the performance of the indiscriminate prefetching algorithm is improved by dropping prefetches. The improvement is dramatic in the two cases that had previously stalled the most due to full buffers (CFFT2D and CG). The selective prefetch algorithm, however, did not improve from dropping prefetches since it suffered very little from full prefetch issue buffers in the first place. In fact, in three of the cases (CHOLSKY, BTRIX and GMTRY), the selective algorithm performed slightly worse when prefetches are dropped. Dropping prefetches has the effect of sacrificing some amount of coverage (and therefore memory stall reduction) for the sake of reducing prefetch issue overhead. This effect is most clearly illustrated in the case of CG (compare the I bars in Figures 4 and 7), where memory stall time doubles for the indiscriminate algorithm once prefetches are dropped.

There are two reasons why the performance improves substantially for the indiscriminate prefetching algorithm. The first reason is that dropping prefetches increases the chances that future pre-



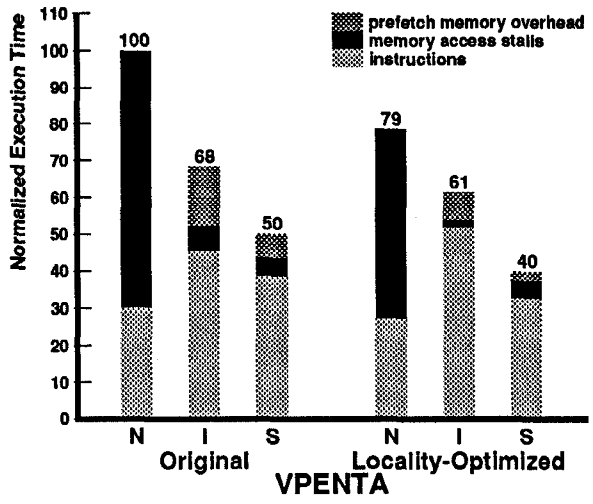


Figure 8: Results with locality-optimizer (N = no prefetching, I = indiscriminate prefetching, and S = selective prefetching).

fetches will find open slots in the prefetch issue buffer. The second is that since the indiscriminate algorithm has a larger number of redundant (i.e. unnecessary) prefetches, dropping a prefetch does not necessarily lead to a cache miss. It is possible that the algorithm will issue a prefetch of the same line before the line is referenced. Since selective prefetching has eliminated much of this redundancy, it is more likely that dropping a prefetch would translate into a subsequent cache miss. However, as we have already seen in Figure 4, the selective algorithm tends to suffer very little from full issue buffers, and therefore performs well in either case.

4.4 Interaction with Locality-Optimizer

Since prefetching *hides* rather than *reduces* latency, it can only improve performance if additional memory bandwidth is available. This is because prefetching does not decrease the number of memory accesses—it simply tries to perform them over a shorter period of time. Therefore, if a program is already memory-bandwidth limited, it is impossible for prefetching to increase performance. Locality optimizations such as cache blocking, however, actually *decrease* the number of accesses to memory, thereby reducing both latency and required bandwidth. Therefore, the best approach for coping with memory latency is to first *reduce* it as much as possible, and then *hide* whatever latency remains. Our compiler can do both things automatically by first applying locality optimizations and then inserting prefetches.

We compiled each of the benchmarks with the locality optimizer enabled [29]. In two of the cases (GMTRY and VPENTA) there was a significant improvement in locality. Both of those cases are presented in Figure 8. In the figure, we show the three original performance bars (seen previously in Figure 4) as well as three new cases which include locality optimization by itself and in combination with the two prefetching schemes.

In the case of GMTRY, the locality optimizer is able to block the critical loop nest. With this locality optimization alone, 90% of the original memory stall time is eliminated. Comparing blocking with prefetching, we see that blocking had better overall performance than prefetching in this case. Although prefetching reduces more of the memory stall cycles, blocking has the advantage of not suffering any of the instruction or memory overhead of prefetching. Comparing the prefetching schemes before and after blocking, we see that blocking has improved the performance of both schemes. One reason is that memory overheads associated with prefetching have been eliminated with blocking since less memory bandwidth is consumed. Also, the selective prefetching scheme reduces its instruction overhead by recognizing that blocking has occurred and thereby issuing fewer prefetches. The best performance overall occurs with blocking, both alone and in combination with selective prefetching.

For VPENTA, the locality optimizer introduces spatial locality for every reference in the inner loop by interchanging two of the surrounding loops. So rather than missing on every iteration, the references only miss when they cross cache line boundaries (every fourth iteration). With this locality optimization alone, the performance improves significantly. However, the selective prefetching scheme without this optimization still performs better, since it manages to eliminate almost all memory stall cycles. Comparing the prefetching schemes before and after the loop interchange, we see that the indiscriminate prefetching scheme changes very little while the selective prefetching scheme improves considerably. The selective scheme improves because it recognizes that after loop interchange it only has to issue one fourth as many prefetches. Consequently it is able to reduce its instruction overhead accordingly. The best overall performance, by a substantial margin, comes only through the combination of both locality optimization and prefetching.

Finally, we would like to note that this is the first time that experimental results have been presented where both locality optimization and prefetch insertion have been handled fully automatically by the compiler. The results have demonstrated the complementary interactions that can occur between locality optimizations and prefetching. Locality optimizations help prefetching by reducing the amount of data that needs to be prefetched, and prefetching helps locality optimizations by hiding any latency that could not be eliminated.

5 Related Work

Several strategies for utilizing prefetching have been presented in the past. Some of these approaches use software support to issue prefetches, while others are strictly hardware-based. In this section, we discuss previous work in both categories.

5.1 Software Prefetching

Porterfield [4, 23] presented a compiler algorithm for inserting prefetches. He implemented it as a preprocessing pass that inserted prefetching into the source code. His initial algorithm prefetched all array references in inner loops one iteration ahead. He recognized that this scheme was issuing too many unnecessary prefetches, and presented a more intelligent scheme based on dependence vectors and overflow iterations. Since the simulation occurred at a fairly abstract level, the prefetching overhead was estimated rather than presented. Overall performance numbers were not presented. Also, the more sophisticated scheme was not automated (the overflow iterations were calculated by hand) and did not take cache line reuse into account.

Klaiber and Levy [16] extended the work by Porterfield by recognizing the need to prefetch more than a single iteration ahead. They included several memory system parameters in their equation for how many iterations ahead to prefetch, and inserted prefetches by hand at the assembly-code level. The results were presented in terms of average memory access latency rather than overall performance. Also, in contrast with our study, they proposed prefetching into a separate *fetchbuffer* rather than directly into the cache. However, our results have demonstrated that prefetching directly into the cache can provide impressive speedups, and without the disadvantage of sacrificing cache size to accommodate a fetchbuffer.

Gornish, Granston and Veidenbaum [13, 14] presented an algorithm for determining the earliest time when it is safe to prefetch shared data in a multiprocessor with software-controlled cache coherency. This work is targeted for a block prefetch instruction, rather than the single-line prefetches considered in this paper.

Chen *et al.* [5] considered prefetching of non-scientific code. They attempt to move address generation back as far as possible before loads to hide a small cache miss latency (10 cycles). The paper also suggested that prefetch buffers would be helpful for reducing the effects of cache pollution. It is considerably more difficult to generate addresses early in non-scientific code where the access patterns can be very irregular. At the same time, these applications tend to make better use of the cache in a uniprocessor.

5.2 Hardware Prefetching

While software-controlled prefetching schemes require support from both hardware and software, several schemes have been proposed that are strictly hardware-based. Porterfield [23] evaluated several cacheline-based hardware prefetching schemes. In some cases they were quite effective at reducing miss rates, but at the same time they often increased memory traffic substantially. Lee [20] proposed an elaborate lookahead scheme for prefetching in a multiprocessor where all shared data is uncacheable. He found that the effectiveness of the scheme was limited by branch prediction and by synchronization. Baer and Chen [2] proposed a scheme that uses a history buffer to detect strides. In their scheme, a "look ahead PC" speculatively walks through the program ahead of the normal PC using branch prediction. When the look ahead PC finds a matching stride entry in the table, it issues a prefetch. They evaluated the scheme in a memory system with a 30 cycle miss latency and found good results.

Hardware-based prefetching schemes have two main advantages over software-based schemes: (1) they have better dynamic information, and therefore can recognize things such as unexpected cache conflicts that are difficult to predict in the compiler, and (2) they do not add any instruction overhead to issue prefetches.

However, the hardware-based schemes have several important disadvantages. The primary difficulty is detecting the memory access patterns. The only case where it does reasonably well is for constant-stride accesses. However, for the types of applications where constant-stride accesses are dominant, the compiler is quite successful at understanding the access patterns, as we have shown. Additionally, in the future our compiler will be able to prefetch complex access patterns such as indirection which the hardware will not be able to recognize. Hardware-based schemes also suffer from a limited scope. Branch prediction is successful for speculating across a few branches, but when memory latency is on the order of hundreds of cycles, there is little hope of predicting that many branches correctly. Also, hardware-based schemes must be "hard-wired" into the processor. For a commercially available microprocessor that is targeted for many different memory systems, this lack of flexibility can be a serious limitation—not only in terms of tuning for different memory latencies, but also a prefetching scheme that is appropriate for a uniprocessor may be entirely inappropriate for a multiprocessor [22]. Finally, while hardware-based schemes have no software cost, they may have a significant hardware cost, both in terms of chip area and possibly gate delays.

6 Future Work

The scope of this compiler algorithm was limited to affine array accesses within scientific applications. By prefetching only the affine accesses, we covered over 90% of the cache misses for roughly half of the benchmarks, while the coverage was closer to 50% of the misses for the remaining benchmarks. In order to increase our coverage, we need to handle more complex access patterns. A good starting point would be sparse matrices, which would account for many of the remaining misses in CG.

Another problem we observed was the large number of conflicts that can occur in direct-mapped caches. These conflicts can have a severe impact on memory performance, regardless of whether prefetching is used. In the case of our algorithm, random conflicts make it even more difficult to predict the contents of the cache. For our experiments, we alleviated this problem by manually changing the alignment of some matrices in four of the applications. However, the burden of fixing such problems should not rest entirely on the programmer.

The use of profiling feedback might also improve the accuracy of our algorithm. As we mentioned earlier in Section 4.2, control-flow profiling information could be used to resolve unknown iteration counts. Also, techniques for profiling memory performance may give the compiler more insight into the caching behavior of the program. At one extreme, the exact miss rates of each reference could be measured—however, this would require *simulating* the application, which may be prohibitively expense. On the other hand, less expensive profiling techniques may provide the compiler with enough useful information, such as which loops suffer the greatest memory penalty. We are currently evaluating this tradeoff between the cost of collecting profiling information and benefit offered in terms of performance.

7 Conclusions

This paper focuses on translating the concept of prefetching into real performance. Software-controlled prefetching not only incurs an instruction overhead, but can also increase the load on the memory subsystem. It is important to reduce the prefetch overhead by eliminating prefetches for data already in the cache. We have developed an algorithm that identifies those references that are likely to be cache misses, and only issues prefetches for them.

Our experiments show that our algorithm can greatly improve performance—for some programs by as much as a factor of two. We also demonstrate that our algorithm is significantly better than an algorithm that prefetches indiscriminately. Our algorithm reduces the total number of prefetches without decreasing much of the coverage of the cache misses. Finally, our experiments show that software prefetching can complement blocking in achieving better overall performance.

Future microprocessors, with their even faster computation rates, must provide support for memory hierarchy optimizations. We advocate that the architecture provide lockup-free caches and prefetch instructions. More complex hardware prefetching appears unnecessary.

8 Acknowledgments

We thank the entire SUIF compiler team for developing and supporting the SUIF compiler. We particularly thank Michael Wolf for the locality optimizer, and Steve Tjiang for the scalar optimizer. This research is supported by DARPA Contract N00039-91-C-0138. Anoop Gupta is also supported by an NSF Presidential Young Investigator Award.

References

[1] W. Abu-Sufah, D. J. Kuck, and D. H. Lawrie. Automatic program transformations for virtual memory computers. *Proc. of the 1979 National Computer Conference*, pages 969–974, June 1979.

[2] J-L. Baer and T-F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of Supercomputing '91*, 1991.

[3] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS Parallel Benchmarks. Technical Report RNR-91-002, NASA Ames Research Center, August 1991.

[4] 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, April 1991.

[5] W. Y. Chen, S. A. Mahlke, P. P. Chang, and W. W. Hwu. Data access microarchitectures for superscalar processors with compiler-assisted data prefetching. In *Proceedings of Microcomputing 24*, 1991.

[6] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman. A vliw architecture for a trace scheduling compiler. In *Proc. Second Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 180–192, Oct. 1987.

[7] J. C. Dehnert, P. Y.-T. Hsu, and J. P. Bratt. Overlapped loop support in the cydra 5. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, pages 26–38, April 1989.

[8] J. Ferrante, V. Sarkar, and W. Thrash. On estimating and enhancing cache effectiveness. In *Fourth Workshop on Languages and Compilers for Parallel Computing*, Aug 1991.

[9] K. Gallivan, W. Jalby, U. Meier, and A. Sameh. The impact of hierarchical memory systems on linear algebra algorithm design. Technical Report UIUCSRD 625, University of Illinios, 1987.

[10] D. Gannon and W. Jalby. The influence of memory hierarchy on algorithm organization: Programming FFTs on a vector multiprocessor. In *The Characteristics of Parallel Algorithms*. MIT Press, 1987.

[11] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computing*, 5:587–616, 1988.

[12] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, 1989.

[13] E. Gornish, E. Granston, and A. Veidenbaum. Compiler-Directed Data Prefetching in Multiprocessors with Memory Hierarchies. In *International Conference on Supercomputing*, 1990.

[14] E. H. Gornish. Compile time analysis for data prefetching. Master's thesis, University of Illinois at Urbana-Champaign, December 1989.

[15] A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, and W-D. Weber. Comparative evaluation of latency reducing and tolerating techniques. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 254–263, May 1991.

[16] A. C. Klaiber and H. M. Levy. Architecture for software-controlled data prefetching. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 43–63, May 1991.

[17] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 81–85, 1981.

[18] M. S. Lam. Software pipelining: An effective scheduling technique for vliw machines. In *Proc. ACM SIGPLAN 88 Conference on Programming Language Design and Implementation*, pages 318–328, June 1988.

[19] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, April 1991.

[20] R. L. Lee. *The Effectiveness of Caches and Data Prefetch Buffers in Large-Scale Shared Memory Multiprocessors*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, May 1987.

[21] A. C. McKeller and E. G. Coffman. The organization of matrices and matrix operations in a paged multiprogramming environment. *CACM*, 12(3):153–165, 1969.

[22] T. Mowry and A. Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87–106, 1991.

[23] A. K. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications*. PhD thesis, Department of Computer Science, Rice University, May 1989.

[24] B. R. Rau and C. D. Glaeser. Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing. In *Proceedings of the 14th Annual Workshop on Microprogramming*, pages 183–198, October 1981.

[25] J. P. Singh, W-D. Weber, and A. Gupta. Splash: Stanford parallel applications for shared memory. Technical Report CSL-TR-91-469, Stanford University, April 1991.

[26] M. D. Smith. Tracing with pixie. Technical Report CSL-TR-91-497, Stanford University, November 1991.

[27] SPEC. *The SPEC Benchmark Report*. Waterside Associates, Fremont, CA, January 1990.

[28] S. W. K. Tjiang and J. L. Hennessy. Sharlit: A tool for building optimizers. In *SIGPLAN Conference on Programming Language Design and Implementation*, 1992.

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