TOLERATING LATENCY THROUGH SOFTWARE-CONTROLLED DATA PREFETCHING

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Todd C. Mowry
May 1994

Abstract

The large latency of memory accesses in modern computer systems is a key obstacle to achieving high processor utilization. Furthermore, the technology trends indicate that this gap between processor and memory speeds is likely to increase in the future. While increased latency affects all computer systems, the problem is magnified in large-scale shared-memory multiprocessors, where physical dimensions cause latency to be an inherent problem. To cope with the memory latency problem, the basic solution that nearly all computer systems rely on is their cache hierarchy. While caches are useful, they are not a panacea.

Software-controlled prefetching is a technique for tolerating memory latency by explicitly executing prefetch instructions to move data close to the processor before it is actually needed. This technique is attractive because it can hide both read and write latency within a single thread of execution while requiring relatively little hardware support. Software-controlled prefetching, however, presents two major challenges. First, some sophistication is required on the part of either the programmer, runtime system, or (preferably) the compiler to insert prefetches into the code. Second, care must be taken that the overheads of prefetching, which include additional instructions and increased memory queueing delays, do not outweigh the benefits.

This dissertation proposes and evaluates a new compiler algorithm for inserting prefetches into code. The proposed algorithm attempts to minimize overheads by only issuing prefetches for references that are predicted to suffer cache misses. The algorithm can prefetch both dense-matrix and sparse-matrix codes, thus covering a large fraction of scientific applications. It also works for both uniprocessor and large-scale shared-memory multiprocessor architectures. We have implemented our algorithm in the SUIF (Stanford University Intermediate Form) optimizing compiler. The results of our detailed architectural simulations demonstrate that the speed of some applications can be improved by as much as a factor of two, both on uniprocessor and multiprocessor systems. This dissertation also compares software-controlled prefetching with other latency-hiding techniques (e.g., locality optimizations, relaxed consistency models, and

multithreading), and investigates the architectural support necessary to make prefetching effective.

Key Words and Phrases: Data prefetching, tolerating latency, compiler optimization, computer architecture, shared-memory multiprocessors.

Acknowledgments

Here is my top ten list of people I would like to acknowledge:

10. My fellow officemates in “the trailer”, for devising new methods of opening doors while remaining seated, and ensuring that life as a graduate student was never a dull moment: Andrew Erlichson, J.P. Singh, Jim Laudon, Chris Holt, Mark Heinrich, Steve Woo, Dave Ofelt, Jeff Kuskin, and John Heinlein.

9. My friends and mentors at MIPS, who made my Silicon Valley experience complete, and helped raise my standard of living above the starvation level: Earl Killian, Peter Davies, and Paul Ries.

8. The DASH team, for answering all of my hardware questions, and for including me in their ski trip: Dan Lenoski, Jim Laudon, Kourosh Gharachorloo, Dave Nakahira, Truman Joe, and Wolf-Dietrich Weber.

7. The simulator gurus who cheerfully and tirelessly supported their creations, thus making these quantitative results possible: Steve Goldschmidt for Tango, and Mike Smith for XSIM.

6. The SUIF team, for creating and supporting that wonderful porcine compiler: Steve Tjiang, Michael Wolf, Mike Smith, Jennifer Anderson, Rob French, Dror Maydan, Saman Amarasinghe, Todd Smith, and Bob Wilson.

5. The faculty members who graciously served on my reading and orals committees: Albert Macovksi and Greg Kovacs.

4. The fearless leader of our extended research family, for his wisdom and guidance throughout the years: John Hennessy.

3. My “second” primary advisor, for teaching me what compilers are all about, and for giving me the luxury of having two sources of great advice: Monica Lam.

2. My “primary” primary advisor, for the tremendous time, energy, and wisdom he invested in my graduate education, and for inspiring me to become an academic myself: Anoop Gupta.

1. My spouse, who shared each step of the Stanford graduate experience with me, from our first day of class (when we met) through defending our dissertations (which we did within 24 hours of each other), and who was a very good sport during our “joint” academic job search: Karen Clay.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Microprocessor-based systems are increasingly becoming the workhorse for all scientific and engineering computation. With numerical processing capabilities that already rival older generations of supercomputers, the microprocessors used in these systems will continue to improve dramatically due to every-increasing clock rates and the exploitation of instruction-level parallelism. In contrast to the vector-based machines that have long dominated high-performance computing, these new scalar systems are considerably more cost-effective since they contain commercial microprocessors that are mass-produced for the large general-purpose computing market. In addition, these commodity microprocessors can be used to build large-scale multiprocessors capable of aggregate peak rates surpassing that of current vector machines.

Unfortunately, a high computation bandwidth is meaningless unless it is matched by a similarly powerful memory subsystem. Although microprocessor speeds have been increasing dramatically, the speed of memory has not kept pace. As illustrated in Figure 1.1, the speed of commercial microprocessors has been doubling roughly every three years, while the speed of commodity DRAM has improved by little more than 50% over the past decade. Part of the reason for this is that there is a direct tradeoff between capacity and speed, and the highest priority in improving DRAM has been increasing capacity. The result is that from the perspective of the processor, memory is getting slower at a dramatic rate. This will affect *all* computer systems, making it increasingly difficult to achieve high processor efficiencies. The latency problem is magnified in large-scale multiprocessors, where sheer physical dimensions result in large latencies to remote memory locations.

To deal with memory latency, most computer systems today rely on their cache hierarchy to reduce the effective memory access time. While the effectiveness of caches has been well

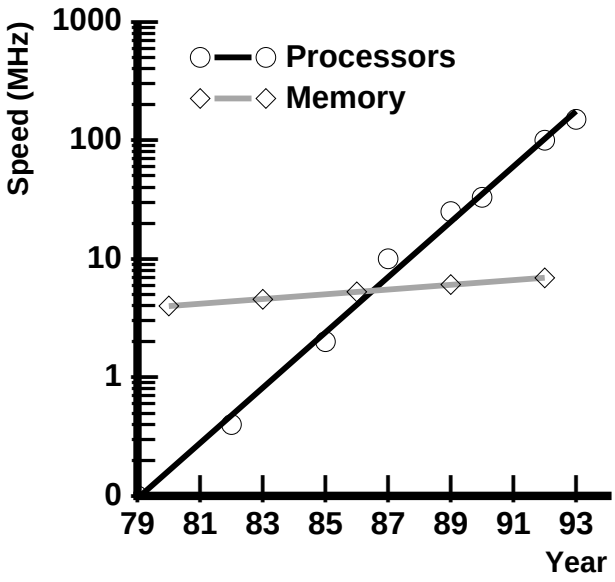


Figure 1.1: Speed of commercial microprocessors and commodity DRAM over the past decade.

established for general-purpose code, their effectiveness for scientific and engineering applications has not. One manifestation of this is that several of the scalar machines designed for scientific computation did not use caches [14, 16].

This thesis investigates a technique called *software-controlled prefetching* which mitigates the impact of long cache miss penalties, thereby helping to unlock the full potential of microprocessor-based systems. The remainder of this chapter provides further motivation for improving cache performance, discusses software-controlled prefetching in light of other techniques for coping with memory latency, presents our research goals, and summarizes related work. We conclude this chapter with a list of the major contributions of this thesis and an overview of the remaining chapters.

1.1 Cache Performance on Scientific and Engineering Codes

To illustrate the need for improving the cache performance of microprocessor-based uniprocessor and multiprocessor systems, we present results in this subsection for a set of scientific and engineering applications. We begin with the uniprocessor architecture. For the sake of concreteness,

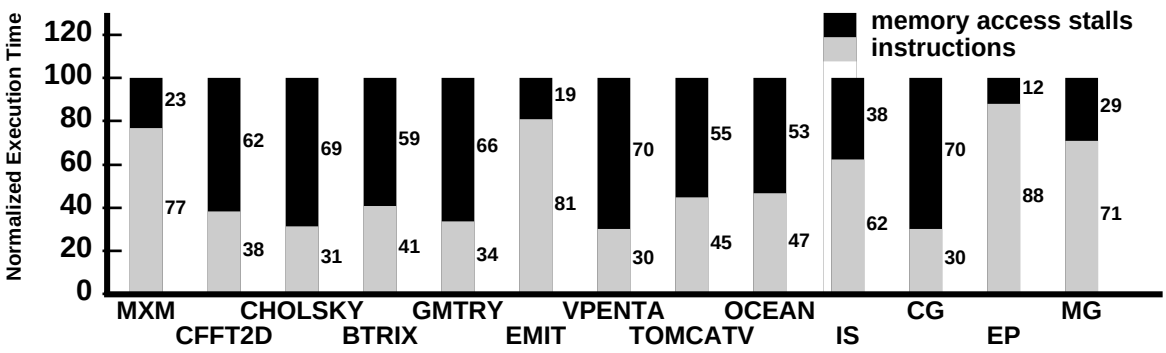


Figure 1.2: Breakdown of execution of scientific and engineering codes on uniprocessor architecture.

we pattern our memory subsystem after a typical MIPS R4000-based workstation. 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 main memory is 75 cycles. In this simple model, we assume that all instructions execute in a single cycle and that all instructions hit in the primary instruction cache. The performance of the benchmarks was simulated by instrumenting the MIPS object code using *pixie* [74] and piping the resulting trace into our detailed cache simulator.

Figure 1.2 breaks down the total program execution time into instruction execution and stalls due to memory accesses for 13 uniprocessor programs taken from the SPEC [77], SPLASH [72], and NAS Parallel [8] benchmark suites. 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.

We conducted a similar experiment to evaluate the impact of memory latency on large-scale shared-memory multiprocessors by simulating the entire SPLASH [72] parallel application suite on an architecture resembling the Stanford DASH multiprocessor [54]. The architecture we model includes 16 R3000 processors running at 33 MHz, two-level cache hierarchies (64 Kbytes/256 Kbytes), and a low-latency interconnection network. Miss latencies for loads range from 15 cycles to the secondary cache to over 130 cycles for "remote dirty" lines. Further details on this architecture and the parallel applications will be presented later in Chapter 4. Figure 1.3 shows the results. Execution time is now broken into three categories: time spent executing instructions,

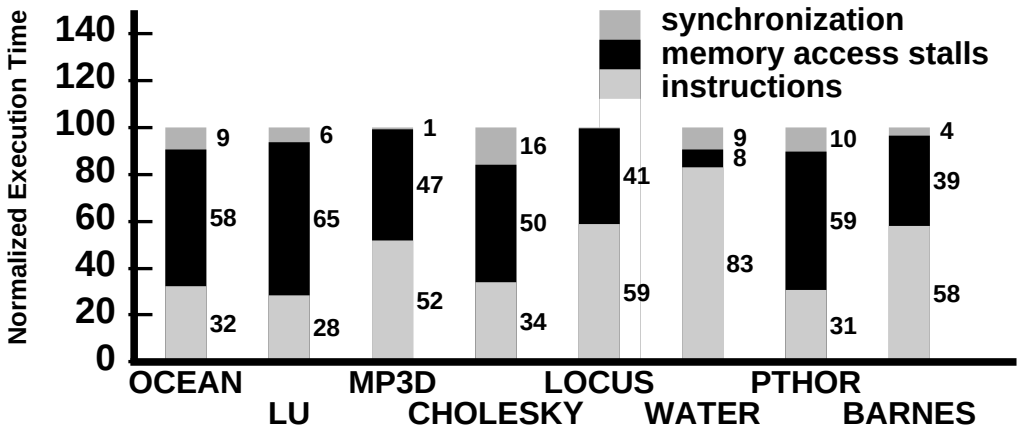


Figure 1.3: Breakdown of execution of scientific and engineering codes on multiprocessor architecture.

time spent stalled for memory, and time spent stalled for synchronization (such as locks and barriers). Once again, memory stalls are significant, with 7 of the 8 applications spending more than 35% of their time stalled waiting for memory accesses to complete.

1.2 Coping with Memory Latency

To reclaim some of the lost performance potential illustrated in Figures 1.2 and 1.3, techniques for coping with memory latency are essential. These techniques fall broadly into two categories: those that *reduce* latency, and those that *tolerate* latency. Techniques for reducing latency include caching data and making the best use of those caches through locality optimizations. Techniques for tolerating latency include buffering and pipelining references, prefetching, and multithreading. We will briefly discuss each of these techniques in this subsection to show how prefetching fits into the overall approach to hiding latency, and to motivate why prefetching itself is worth studying.

1.2.1 Caches

Caches are a critical first step toward coping with memory latency, but are not a panacea, as we saw already in Figures 1.2 and 1.3. Caches reduce latency from a memory access to a cache access whenever data items are found in the cache. The likelihood of finding data in

the cache depends not only on the size and organization of the cache, but also on the inherent *locality of reference* within the application. Locality can occur in both time and space: *temporal locality* is the tendency for a recently-accessed item to be accessed again soon, and *spatial locality* is the tendency for items *near* a recently-accessed item to be accessed soon. Since most applications exhibit a reasonable amount of locality, caches are generally quite useful. As a result, most commercial RISC microprocessors provide support for cache hierarchies, including on-chip primary instruction and data caches. The benefits of caches in multiprocessors have also been recognized, where despite the complication of keeping shared writable data *coherent* [6], a number of multiprocessors with caches have been implemented [3, 41, 47, 55]. Therefore caches are an integral part of the memory latency solution, and the remaining techniques we discuss build upon caching as a foundation.

1.2.2 Locality Optimizations

Locality optimizations attempt to make caches more effective by restructuring computation to enhance data locality. One important example of a locality-improving transformation is *blocking* (also known as *tiling*) [1, 22, 23, 30, 60, 64, 87], which works as follows. Rather than 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 [87]. 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.

For multiprocessors, the concept of data locality can be extended to minimize not only accesses to main memory, but also communication between processors. This involves both the placement of data in processors' local memories (known as *data decomposition*) and the mapping of the computation onto the processors of the parallel machine (known as *computation decomposition*). The most popular approach to this complex optimization problem is for the programmer to explicitly specify the data decomposition through directives in the programming language [11, 38, 43, 67, 78, 82, 88], while the compiler is responsible for decomposing the computation. Another approach is for the compiler to determine both the data and computation decompositions [4, 5, 10, 12, 34, 35, 56].

While locality optimizations are quite useful when they work, their applicability is somewhat limited because not only must there be a better way to structure the code (which is not always the case), but it also must be *legal* to do so. Whenever dependence analysis [58] is inexact,

the compiler usually has to be conservative and assume that dependencies could be violated if the code was restructured. In practice, this means that these types of optimizations are not frequently applicable (as we will observe later in Section 3.4). Therefore to cope with whatever latency cannot be reduced through caching and locality optimizations, we consider techniques for *tolerating* memory latency.

1.2.3 Buffering and Pipelining References

One way to tolerate memory latency is to allow references to be *buffered* and *pipelined*. In current uniprocessor systems, this technique is typically applied only to writes in the form of *write buffers*. Write buffers exploit the fact that a processor does not have to wait for a write to complete as long as it properly observes the effect of the written data in the future. Therefore the processor can perform a write by simply issuing it to the write buffer, provided that future reads check the write buffer for matching addresses. The advantage of a write buffer is not only that the processor does not stall when executing a write, but also that multiple writes can be overlapped to exploit pipelining.

Buffering read accesses is more difficult because unlike writes, the processor typically cannot proceed until the read access completes, since it needs the data that is being read. With *non-blocking loads* and a *lockup-free cache* [45], it is possible to buffer and pipeline reads. A non-blocking load means that rather than stalling at the time the load is performed, the processor postpones stalling until the data is actually used. A lockup-free cache permits multiple outstanding cache misses. By combining the two, it would be possible to buffer multiple reads, and to pipeline their accesses. However, very few commercial microprocessors currently support non-blocking loads due to the complexity involved, and in practice the use of a load value typically occurs shortly after the load is performed. Therefore tolerating read latency through buffering and pipelining is not especially promising.

Buffering and pipelining accesses in a multiprocessor is complicated by the restrictions placed on the causality of accesses in different processors. In the strictest case, known as *sequential* or *strong* consistency [50], all accesses to shared data must appear as though the different processes were interleaved on a sequential machine. While conceptually intuitive and elegant, sequential consistency imposes severe restrictions on the outstanding accesses that a process may have, thus restricting the buffering and pipelining allowed. In contrast, *relaxed consistency models* [2, 18, 26, 27] permit accesses to be buffered and pipelined, provided that explicit synchronization events are identified and ordered properly. Once again, however, the main benefit of these relaxed

Figure 1.4: Illustration of how prefetching improves performance.

consistency models is hiding write latency [26]. To address read latency effectively, we must look beyond buffering and pipelining.

1.2.4 Prefetching

The two main techniques for tolerating read latency as well as write latency are *prefetching* and *multithreading*. The key to tolerating read latency is to split apart the *request* for data and the *use* of that data, while finding enough *parallelism* to keep the processor busy in between. The distinction between prefetching and multithreading is that prefetching finds the parallelism within a *single thread* of execution, while multithreading exploits parallelism across *multiple threads*. To hide the latency within a single thread, the request for the data (i.e. the prefetch request) must be moved back sufficiently far in advance of the use of the data in the execution stream. This effectively requires the ability to *predict* what data is needed ahead of time. In contrast, the multithreading approach splits read transactions by swapping out the currently executing thread when it suffers a cache miss, executing other concurrent threads for the duration of the miss to keep the processor busy, and finally resuming the initial thread once the memory access completes. Prefetching will be discussed in this section, while multithreading will be discussed in more detail in Section 1.2.5.

Figure 1.4 is a simple illustration of how prefetching improves performance. In the case

without prefetching (shown on the left), the processor stalls when it attempts to load two locations (A and B) that are not present in the cache. If prefetches for A and B can be issued far enough in advance in the instruction stream (as shown on the right), then the memory accesses for both locations will have completed before the loads are executed, and hence the processor will not stall. The key observation here is that prefetching not only allows memory accesses to be overlapped with computation, but it also allows memory accesses to be overlapped with other memory accesses (i.e. the accesses can be pipelined).

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.

Prefetching can occur in many different forms. One common type of prefetching occurs whenever cache lines are longer than a single word. In these cases, additional words are brought into the cache on each cache miss. This is most useful when there is abundant spatial locality, such as when iterating across an array in a unit-stride manner. However, increasing the cache line size is not the most effective form of prefetching, since memory bandwidth is wasted whenever useless data is brought into the cache [64]. In addition, long cache lines can aggravate miss rates in shared-memory multiprocessors by causing unnecessary amounts of *false sharing* [19, 81]. As we have seen already in Figures 1.2 and 1.3, a significant amount of latency remains despite the prefetching benefit of multi-word cache lines.

Another form of prefetching could occur with *non-blocking loads* [66]. With a non-blocking load, rather than stalling when the load is executed, any stalls are postponed until the load result is actually needed. So one could imagine that if the loads could be moved far enough in advance of the uses of data, then prefetching could be implemented in this manner. However, there are two important limitations on how far loads can be moved ahead of their uses. First, there is the problem of running out of registers. If the compiler attempts to extend register lifetimes to hundreds of cycles, it will run out of registers very quickly. Second, there is the problem of maintaining program correctness. For example, a load cannot be moved ahead of a store unless it is certain that they are to different locations. Since memory disambiguation is a difficult problem for the compiler to solve (particularly in codes with indirect references), this is likely to be a serious limitation.

Some elaborate prefetching schemes that are strictly hardware-based have also been proposed.

We will discuss those schemes only briefly now, and will examine them in greater detail later in Section 5.3.1. Perhaps the most sophisticated of these techniques is the one proposed by Baer and Chen [7]. With this scheme, the processor maintains a history table to keep track of the types of reference patterns it is seeing. If it detects a pattern of constant-stride access behavior for a particular instruction, it will attempt to prefetch ahead for that reference in the future. This prefetching occurs through a "lookahead PC", which walks ahead of the actual PC using branch prediction. The lookahead PC is used to look up these future instructions in the history table to see whether they should be prefetched. Another scheme proposed by Lee [53] attempted to decode future instructions using a lookahead buffer to detect memory references. One advantage of strictly hardware-based schemes is that they do not incur any instruction overhead, unlike software-controlled prefetching (which we will discuss next). However, their disadvantages include the fact that they are limited to prefetching constant-stride accesses, they are limited by branch prediction (which is less than perfect), and they may entail a significant hardware cost.

Finally, with software-controlled prefetching, explicit prefetch instructions are executed by the processor to move data into the cache. The format of these instructions resembles a normal load instruction, but without a register specifier (since the data is only placed in the cache). Prefetch instructions also differ from normal load instructions in that they are non-blocking and they do not take memory exceptions. The non-blocking aspect allows them to be overlapped with computation, and the fact that they do not take exceptions is useful because it permits more speculative prefetching strategies (e.g., dereferencing pointers before it is certain that they point to legal addresses). The challenges of software-controlled prefetching include the fact that some sophistication is needed to insert the prefetches into the code, and also that the new prefetch instructions will involve some amount of execution overhead. The advantages of software-controlled prefetching are that only a small amount of hardware support is necessary, and a broader class of reference patterns can be covered than simply constant stride accesses (e.g., indirect references, such as in sparse-matrix code).

1.2.5 Multithreading

Figure 1.5 illustrates how multithreading (also known as "multiple-context processing") can also be used to tolerate latency [3, 36, 44, 85]. In this figure, context #1 suffers a cache miss when it attempts to load location A. At this time, context #1 is swapped out and the processor begins to execute context #2. Hopefully by the time context #2 needs to be swapped out (which occurs

Figure 1.5: Illustration of how multithreading improves performance.

when it suffers a cache miss trying to load location B), the memory access for the original context has completed, and therefore context #1 is ready to run again.

Multithreading has two advantages over software-controlled prefetching. First, it can handle arbitrarily complicated access patterns, including situations where it is impossible to predict the addresses ahead of time (and therefore prefetching will not work). Second, since it does not require any software support, it can improve the speed of existing executables without recompilation.

However, multithreading has several disadvantages relative to software-controlled prefetching. First of all, to make a single application execute faster, additional concurrent threads of execution are needed. This concurrency may or may not exist. Particularly in a uniprocessor environment, it is unlikely that a programmer would go through the pain of parallelizing their application for the sake of multithreading. A second limitation is the overhead of switching between contexts. Such overhead occurs because: (i) data cache misses are detected late in the pipeline, and subsequent instructions that have entered the pipe must be flushed; and (ii) saving and restoring context state (e.g., the register file) may take additional time. These switching overheads can potentially offset much of the performance gain of multithreading. Finally, to minimize the context switching overhead, a significant amount of hardware is required (e.g., replicated register files). Therefore multithreading is clearly a more expensive solution than prefetching, both in terms of concurrency demands and hardware support.

Table 1.1: Techniques for coping with memory latency.

Technique	Benefit	Exploits	Hardware Support	Software Support
Caching	Reduces Latency	Locality of Reference	Caches[1]	None
Locality Optimizations	Reduces Latency	Reordering Computation to Enhance Locality	Caches	Locality-Enhancing Transformations
Buffering and Pipelining	Tolerates Write Latency	Writes Can Occur Out-of-Order	Lockup-free Caches[2]	None[3]
Software-Controlled Prefetching	Tolerates Read and Write Latency	Parallelism within a Single Thread	Lockup-free Caches, Prefetch Instruction	Insert Prefetches
Multithreading	Tolerates Read and Write Latency	Parellelism across Multiple Threads	Lockup-free Caches, Thread Control Logic, Replicated State[4]	None

1.2.6 Overall Approach

We now focus on how each of these techniques, particularly prefetching, fits into the "big picture" of coping with memory latency. As a quick summary, Table 1.1 presents the benefits and requirements of each technique. Given these techniques, we would like to apply them in the following order.

First, the latency should be *reduced* as much as possible, through caching and locality optimizations. Reducing latency is preferable over tolerating latency since it actually reduces the demand for main memory bandwidth, which can be crucial. Caches provide the foundation for all of these latency-hiding techniques, and locality optimizations are also attractive since they require no additional hardware support.

After reducing latency, we then want to *tolerate* any remaining latency, starting with the least expensive techniques before resorting to more costly techniques. The first step is buffering and pipelining accesses, which is an effective means of hiding write latency and requires only a lockup-free cache—a requirement common to all latency-tolerating techniques. To address read latency as well, the choices are either prefetching or multithreading. Software-controlled prefetching

[1]Hardware to support coherency mechanism is also needed for multiprocessors.
[2]Lockup-free caches permit both buffering and pipelining. For buffering alone, write buffers are sufficient.
[3]Explicit synchronization must be identified for multiprocessors.
[4]The replicated thread state may include replicated register files.

appears to be the more desirable solution since it requires significantly less hardware support than either hardware-controlled prefetching or multithreading, and perhaps more importantly it can speed up the performance of a *single* thread of execution, rather than requiring multiple concurrent threads as in the case of multithreading. If software-controlled prefetching cannot effectively hide read latency, the final step would be multithreading.

Thus the open question of just how effective software-controlled prefetching can be in practice is a key factor in deciding whether processor architectures should support prefetching, multithreading, or both. Addressing this open question is one of the goals of this dissertation, as we discuss further in the next section.

1.3 Research Goals

This section discusses the research goals of this dissertation. At the highest level, our goal is to evaluate and improve upon the latency-hiding benefits of software-controlled prefetching. We focus specifically on *compiler-inserted* prefetching, since this is obviously preferable to placing the burden of prefetch insertion on the programmer. Since we are interested in practical results rather than theoretical limits, we implement our prefetching algorithms in a state-of-the-art compiler to measure the actual performance of working codes with prefetching. The goals of the compiler algorithm itself are described below.

A key goal for any compiler-based prefetching algorithm is the ability to cover a wide range of programs (e.g., dense-matrix codes, sparse-matrix codes, irregular general-purpose codes, etc.). Ideally, any application suffering from memory latency could benefit from prefetching. However, covering all types of applications and access patterns is an overly-ambitious goal, and therefore we must limit our scope to make substantial headway. Our approach is to start with an important class of applications and handle them well, and then later build upon this core algorithm to cover other important cases. The access pattern we start with is array references where the array indices are *affine* (i.e. linear) functions of surrounding loop indices; such access patterns occur in many types of applications, particularly dense-matrix codes. These applications are interesting both because they tend to suffer from memory latency, due to the large size of the arrays and the often wide separation between reuses, and because the access patterns are regular and predictable, which gives prefetching a good chance of success [61]. Next we extend this algorithm to handle *indirect* array references, which are another important case and are common to sparse-matrix codes. Finally, we address multiprocessing, where memory latencies can be quite

large. By handling all of these cases, our compiler can cover a significant fraction of scientific and engineering codes.

The second key goal of a prefetching compiler algorithm is maximizing the performance improvement for the cases that are covered. Since prefetching involves a *cost* as well as a benefit, care must be taken to minimize these overheads while maximizing the latency-hiding benefits to achieve the best overall performance.

In addition to these compiler-oriented research goals, our goals in the architecture domain are to determine the proper architectural support for prefetching, and to comparatively evaluate prefetching with respect to other latency-hiding techniques, such as locality optimizations, relaxed consistency models, and multithreading.

1.4 Related Work

Several other researchers have also worked on software-controlled prefetching. This section summarizes their work and relates it to ours.

Porterfield [9, 64] was the first to explore software-controlled prefetching for uniprocessors. He proposed a compiler algorithm for inserting prefetches into dense-matrix codes. He implemented his algorithm 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 sophisticated scheme based on dependence vectors and overflow iterations (i.e. the predicted number of iterations where a loop will access more data than fit in the cache). Since the simulation occurred at a fairly abstract level, the prefetching overhead was estimated rather than measured. Overall performance numbers were not presented. Also, the more sophisticated scheme was not automated, since the overflow iterations were calculated by hand, and did not take cache line reuse into account. Despite leaving many important questions unanswered, Porterfield's work demonstrated that software-controlled prefetching was a promising technique that warranted further exploration.

Klaiber and Levy [42] extended Porterfield's work 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 this dissertation, they proposed prefetching into a separate *fetchbuffer* rather than directly into the cache. However, as we will discuss in detail

later in Section 5.1.3, using a separate fetchbuffer has a number of important disadvantages, including the sacrifice of chip area that could otherwise be used for a normal cache, and the fact that it becomes very difficult to makes prefetches *non-binding*, which is a crucial property for multiprocessors.

Gornish, Granston and Veidenbaum [31, 32] presented an algorithm for determining the earliest time when it is safe to prefetch shared data in a multiprocessor with software-controlled cache coherency. Since the prefetches are binding, all control and data dependencies must be carefully considered. Their work is targeted for a block prefetch instruction, rather than the single-line prefetches considered in our study. The proposed schemes are evaluated using a large number of numerical subroutines. Although the speedups predicted from static analysis are quite high, over twofold, the speedups obtained using detailed simulations are limited to 10-20%. The complexity of the compiler algorithm presented in this work illustrates how difficult the compiler's job becomes when *binding* rather than *non-binding* prefetches are used. This is in sharp contrast with the simplicity of our non-binding prefetching algorithm, presented later in Section 4.1.

Chen *et al*. [13] investigated prefetching for non-numerical codes. They attempted to move address generation back as far as possible before loads to hide a small cache miss latency (10 cycles), and found mixed results. Generating addresses early is difficult in non-numerical code because control and data dependencies tend to be tight, and the access patterns can be very irregular. Although these codes are difficult to prefetch, they also tend to make better use of the cache than numerical codes and therefore typically suffer less from memory latency. Because of the difficulty of prefetching highly irregular access patterns and the relatively small expected gains, we chose not to focus on this class of applications during this dissertation.

Tullsen and Eggers [83] post-processed reference traces to evaluate the performance of an "oracle" prefetching scheme on a bus-based multiprocessor architecture with limited bandwidth. They observed that if an application is already bandwidth-limited, prefetching cannot improve performance. However, because of their post-processing trace methodology, they were not able to make proper use of *exclusive-mode* prefetches, which can potentially eliminate up to half of the memory bandwidth consumption. As we will show later in Section 4.3.4, exclusive-mode prefetching eliminates up to 27% of the memory requests in our applications, which would have translated directly into improved performance in their bus-bandwidth-limited architecture.

We now place this related work in the context of our own research. Porterfield's work on software-controlled prefetching for uniprocessors was done concurrently with our investigation of non-binding software-controlled prefetching for multiprocessors as part of the DASH project [54].

Our study, where we inserted prefetches by hand, was the first to consider *non-binding* prefetching for multiprocessors [61]. The compiler algorithm presented in this dissertation partially overlapped the work by Porterfield and by Klaiber and Levy, but has a number of key differences. First, our uniprocessor algorithm is more comprehensive in the types of locality it optimizes for, including temporal, spatial, and group locality. Second, unlike previous studies, we have implemented our algorithm in an optimizing compiler, and can therefore generate overall performance numbers for fully-functional codes. Finally, the scope of our complete algorithm is much broader since it also covers indirect references and multiprocessing.

1.5 Contributions

The primary contributions of this dissertation are the following:

- The proposal of a new compiler algorithm for inserting prefetch instructions in scientific and engineering codes. This algorithm improves upon several previous proposals that focused on dense-matrix uniprocessor codes [31, 42, 64]. In addition, this algorithm handles indirect references, which frequently occur in sparse-matrix codes, and targets large-scale shared-memory multiprocessors as well as uniprocessors.

- A detailed evaluation of the prefetching algorithm based on a full compiler implementation. The prefetching algorithm is implemented in the SUIF (Stanford University Intermediate Form) compiler, which includes many of the standard optimizations and generates code competitive with the MIPS 2.10 compiler[80]. Using this compiler system, we have been able to generate fully functional and optimized code with prefetching. 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. The results of this evaluation show that our algorithm is quite successful at hiding memory latency, improving the performance of some applications by as much as twofold.

- A study of the interaction of prefetching and other techniques for hiding latency, such as data locality optimizations, relaxed consistency models, and multithreading. We find that prefetching is complementary to both locality optimizations and relaxed consistency models, but the benefit of combining prefetching and multithreading is less clear.

- An investigation of the architectural support necessary for software-controlled prefetching, including proposals that may further increase the prefetching performance benefit. In addition to including prefetch instructions in the instruction set, we find that the main support necessary for prefetching is a lockup-free cache. Further enhancements to the architecture may include hardware miss counters to expedite the use of dynamic information, and associativity to reduce the cache conflict problems.

1.6 Organization of Dissertation

Chapter 2 describes our core prefetching algorithm, which handles affine array references and thus dense-matrix code. A key feature of this algorithm is minimizing prefetching overhead by only prefetching references that are predicted to suffer cache misses. This core algorithm is the basis for all of our experiments, and will be extended in later chapters.

Chapter 3 studies the performance benefits of prefetching for uniprocessor applications, beginning with a detailed evaluation of the algorithm described in Chapter 2. Next we evaluate the interaction between prefetching and locality optimizations, which are another important latency-hiding technique for dense-matrix codes. Finally, we extend our core compiler algorithm to handle indirect references (and hence sparse-matrix codes), and measure the resulting performance improvement of relevant applications.

Chapter 4 focuses on prefetching for large-scale shared-memory multiprocessors. These machines are interesting because of their large performance potential, and because they are particularly prone to suffering from memory latency. We begin by discussing how the prefetching compiler algorithm described in Chapters 2 and 3 is modified to address the issues unique to multiprocessing, and then evaluate its effect on the performance of the entire SPLASH [72] application suite. We also compare compiler-inserted prefetching with hand-inserted prefetching to see whether the compiler is living up to its potential, and to discover methods for further improvement.

Chapter 5 explores the architectural issues associated with prefetching, and is divided into three distinct sections. The first section examines the architectural support necessary for the basic prefetching model assumed in Chapters 3 and 4. The second part considers ways to enhance the architecture to further improve prefetching. The third section comparatively studies other latency-hiding techniques that require architectural support, namely hardware-controlled prefetching, relaxed consistency models, and multithreading.

Finally, Chapter 6 contains a summary of the important results in this dissertation, and discusses their implications. It also discusses directions for future work in this area.

Chapter 2

Core Compiler Algorithm for Prefetching

In this chapter we present our compiler algorithm for prefetching dense-matrix codes. These applications are a top priority since they consume large numbers of cycles on supercomputers today, they have poor caching behavior, and yet they have regular enough access patterns that prefetching has a reasonable chance of success. Therefore we start by handling these affine array accesses (i.e. where the index functions are affine expressions of the surrounding loop variables), and later build upon this core algorithm in subsequent chapters to handle other important cases such as indirect references and multiprocessor codes.

This chapter is organized in five sections. The first section discusses some key concepts for compiler-based prefetching, including the need to avoid unnecessary overhead. In light of these goals, Section 2.2 provides an overview of our compiler algorithm, which includes both an analysis phase and a scheduling phase. Details of these two phases are presented in Sections 2.3 and 2.4, respectively. The analysis phase uses *locality analysis* to predict which references should be prefetched, and the scheduling phase first uses *loop splitting* to isolate those dynamic miss instances, and then uses *software-pipelining* to schedule prefetches the proper amount of time in advance. Finally, Section 2.5 makes our algorithm concrete by showing the output for some example code, and also discusses issues that arose when implementing the algorithm in the SUIF compiler, such as whether prefetch insertion should occur before or after scalar optimization. The success of this algorithm will be evaluated later in Chapter 3.

2.1 Key Concepts

In this section we discuss some important concepts that are useful when thinking about prefetching compiler algorithms.

First of all, prefetches are only *possible* if the memory addresses can be determined ahead of time. For example, if the address is dependent on data that is only available immediately before the memory reference, it may not be possible to compute the address far enough in advance. While this is an important concern with irregular computations, such as those containing pointers and linked lists, it is not an issue for the dense-matrix codes considered in this chapter, since the affine array addresses can always be computed ahead of time.

Given that prefetching is possible, a useful metric for determining success is the *coverage factor*, which is the fraction of original cache misses (i.e. without prefetching) that have been prefetched. Ideally, we would like to prefetch all cache misses, thus achieving a coverage factor of 100%.

However, it is not necessarily the case that "more is better" with prefetching, since prefetches issued for data already in the cache result in overhead without any benefit. Such prefetches are referred to as *unnecessary prefetches*, and should be avoided.

Finally, even if a prefetch is issued for data that is not already in the cache, it may not be *effective* at improving performance if the data is not found in the cache during the subsequent memory reference. This can happen for two reasons. Obviously if the prefetch is issued *too late*, there simply isn't enough time to hide the memory latency. But on the other hand, if the prefetch arrives in the cache *too early*, it may be displaced by other references before it has a chance to be referenced. Having discussed these high-level prefetching concepts, we now focus on the compiler algorithm itself.

Any compiler algorithm for inserting prefetches can be viewed as having two distinct phases. The first phase is an *analysis* phase, where the compiler is trying to answer the question: what exactly should be prefetched? The ideal answer to this question is precisely the set of dynamic references that suffer cache misses. By answering this question correctly, the compiler will maximize the coverage factor and minimize unnecessary prefetches. Once the compiler knows what it wants to prefetch, the second phase is to *schedule* those prefetches so that they are effective and so that they add a minimum amount of instruction overhead.

Minimizing overhead is important, because although the benefit of prefetching is hiding latency, prefetches can introduce two forms of overhead. First, there are additional instructions

Table 2.1: 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 issue the prefetches. This includes not only the prefetch instructions themselves, but also any instructions needed to generate the prefetch addresses. Second, any prefetches that do not eliminate cache misses, either because they are unnecessary or ineffective, will increase the load on the memory subsystem,[1], which can lead to increased queueing delays for all references.

For the dense-matrix applications shown earlier in Figure 1.2, we find that while the miss rate is high enough to substantially affect performance, the hit rate of the array references is still very high (over 60% for most programs), as shown in Table 2.1. (Details of this uniprocessor architecture were given earlier in Section 1.1.) Therefore if we were to prefetch all of the array references, then over 60% of the prefetches would be unnecessary for most of the programs. Avoiding these unnecessary prefetches is an important goal of our compiler algorithm, which we describe in the next several sections.

2.2 Overview of Algorithm

We have developed a compiler algorithm that selectively prefetches only those references that are likely to suffer cache misses [62]. Our algorithm consists of the following three major steps:

[1]If the primary cache is checked before prefetches proceed further through the memory hierarchy (we will discuss this in more detail later in Section 5.1.3) then the additional memory subsystem loading caused by unnecessary prefetches will be limited to increased primary cache tag contention.

1. For each static affine array reference, use *locality analysis* to determine which dynamic accesses are likely to be cache misses and therefore should be prefetched.

2. Isolate the predicted dynamic miss instances using *loop splitting* techniques such as peeling and unrolling. This avoids the overhead of adding conditional statements to the loop bodies.

3. Schedule prefetches the proper amount of time in advance using *software pipelining*, where the computation of one iteration is overlapped with prefetches for a future iteration.

The first step in our algorithm comprises the *analysis* phase, thus answering the question of what we should prefetch. The details of this locality analysis algorithm are presented in Section 2.3. The second and third steps constitute the *scheduling* phase of our algorithm, and are described in Section 2.4. We will tie all of these components together in Section 2.5 by showing the code generated for a running example that is introduced in Section 2.3.1 and used throughout the remainder of this chapter.

2.3 Locality Analysis

This section describes the first step in our algorithm, which is to use *locality analysis* to determine which references are likely to cause cache misses.

Before we begin our discussion, we must define and distinguish two fundamental concepts: *reuse* and *locality*. We say that *reuse* occurs whenever the same data item is referenced multiple times. Understanding data reuse is essential to predicting cache behavior since a data item will normally only be found in the cache if its line was referenced sometime in the past. However, reuse might not result in a cache hit if intervening references flush the data item from the cache between uses. If the reused data actually does remain in the cache, we say that the reference that enjoys the cache hit has *locality*. Therefore, it is important to realize that *reuse does not necessarily result in locality*. Instead, the references with data locality are a subset of the references with data reuse.

Given this relationship between reuse and locality, our locality analysis algorithm is broken down into the following three substeps:

1. Discover the intrinsic data reuses within a loop nest through *reuse analysis*. This would be equivalent to solving the locality analysis problem if we had an infinitely large cache.

2. Given that we have a *finite* rather than an infinite cache, determine the set of reuses that actually result in locality. This is accomplished by computing the *localized iteration space* for the given cache size, which is the set of loops within a nesting that can carry locality. By intersecting the intrinsic data reuses with the localized iteration space, we can compute data locality, i.e.

$$\text{Data Reuse} \cap \text{Localized Iteration Space} \Rightarrow \text{Data Locality}$$

3. Express the data locality for each reference in terms of a *prefetch predicate*, which is a logical predicate that is true during each dynamic iteration when the reference is expected to suffer a cache miss. We will later insert prefetches such that a reference is prefetched whenever its prefetch predicate is true.

These first two substeps have been adapted from Wolf and Lam's data locality optimizing algorithm [86, 87], while the third substep is unique to prefetching. Although analyzing data locality is essential for both locality optimizations and prefetching, the differing goals of the two techniques lead to differences in how the analysis is applied. For locality optimizations, the goal of locality analysis is *quantifying* the data locality of various permutations of the loop nest to select the transformation that yields the maximum locality. For prefetching, the goal is *identifying* rather than quantifying references with data locality, and the algorithm does not consider permuting the given loop nest. There are other differences, such as the way the localized iteration space is computed. For Wolf and Lam's algorithm, the assumption is that all loop bounds are large, and therefore the localized iteration space only contains loops other than the innermost loop if transformations such as "cache blocking" are actively applied. In contrast, since prefetching is interested in *discovering* rather than *enhancing* data locality, it is more important to determine the given localized iteration space accurately, so our algorithm attempts to reason about constant and symbolic loop bounds. A final difference is that prefetching attempts to analyze locality in all loop nests, rather than just the well-behaved loop nests where it is considered safe to reorder the computation.

The result of the first two substeps of this algorithm is a mathematical description of the data locality in a *vector space* representation. While this description is precise, a more convenient representation for our purposes is the prefetch predicates which are constructed during the third substep of our locality analysis algorithm. Details of these three substeps will be presented in Sections 2.3.2, 2.3.3, and 2.3.4, respectively. Before going into these details, however, we first introduce the following example.

Figure 2.1: Data locality example.

2.3.1 An Example

The code in Figure 2.1 illustrates the types of locality that our algorithm can discover. We will use this code as a running example throughout the remainder of this chapter. We assume, for this example, that the cache is 8 Kbytes, the prefetch latency is 100 cycles and the cache line size is 4 words (two double-word array elements to each cache line). (Note that for the purpose of illustration, these parameters differ slightly from our previous uniprocessor architecture.) For this example, the set of references that will cause cache misses can be determined by inspection.

For each array reference in our example code, Figure 2.1(b) shows which iterations hit and miss in the cache using *iteration space* plots. In this representation, the horizontal axis corresponds to the `j` loop, the vertical axis corresponds to the `i` loop, and each node represents

an iteration within the loop nest. Therefore as the loop nest is executed, the nodes in the iteration space would be visited in the following order: first the bottom-most row is visited from left-to-right, then the second-from-bottom row is visited from left-to-right, and so on. To simplify the graphs, we only show the first eight out of 100 `j` loop iterations.

The three array references in Figure 2.1 illustrate the three different types of locality: temporal, spatial, and group. *Temporal locality* can occur when a given reference reuses exactly the same data location. *Spatial locality* can occur when a given reference accesses different data locations that fall within the same cache line. *Group Locality* can occur when different references access the same cache line.

The `A[i][j]` reference in Figure 2.1 illustrates spatial locality. In this case, given that each cache line contains two array elements, we would expect misses to occur on every other iteration as cache line boundaries are crossed. These misses are shown graphically in Figure 2.1(b).

The `B[j+1][0]` reference is an example of temporal locality. In contrast with the `A[i][j]` reference, since the `B[j+1][0]` reference traverses the columns rather than the rows of the matrix along the inner loop, there is no cache line reuse and therefore all `B[j+1][0]` references suffer misses the first time through the `j` loop. However, since the same `B[j+1][0]` locations are reused on subsequent `i` loop iterations, and since these 800 bytes of data are likely to remain in the 8 Kbyte cache across `i` loop iterations, we would expect the `B[j+1][0]` references to hit in the cache after the first `i` loop iteration. This effect is also illustrated in Figure 2.1(b).

Finally, the `B[j+1][0]` and `B[j][0]` references are an example of group locality. Group locality is a relationship that can occur between multiple references whenever one reference brings in much of the data used by the other references. In this example, the data for the `B[j][0]` reference is fetched by the `B[j+1][0]` reference during the previous `j` iteration. Consequently the `B[j][0]` reference suffers only a single cache miss during the entire loop nest. Whenever group locality occurs, we only worry about prefetching the *leading reference* within the group, which is the reference that accesses new data first and therefore suffers the bulk of the cache misses. In this example, `B[j+1][0]` is the leading reference, and therefore we would not issue prefetches for `B[j][0]`.

The remainder of this section describes how locality analysis systematically uncovers the types of locality shown in this example.

2.3.2 Reuse Analysis

Since locality can only occur if there is reuse, the first step in locality analysis is determining the intrinsic data reuse through *reuse analysis*. Reuse analysis attempts to discover those instances of array accesses that refer to the same memory line.[2] There are three kinds of reuse, which parallel the three kinds of locality described in Section 2.3.1: temporal, spatial, and group. The difference is that while reuse is an inherent property of code, locality also depends on the ability of the cache to retain data. Therefore if we had an infinitely large cache, which would retain data perfectly, reuse would be equivalent to locality.

Our reuse analysis step is nearly identical to that proposed by Wolf and Lam [86, 87], which we will briefly summarize in this subsection. Our terminology is slightly different, however, since we tailor our reuse categories to more closely match our prefetching algorithm. What they refer to as *self-temporal* reuse is the same as what we refer to as *temporal* reuse, and corresponds to whenever a given array reference accesses the *same* data location multiple times within a loop nest. However, whereas they define *self-spatial* reuse to include accesses anywhere within the same cache line, we define *spatial* reuse only as accesses to *different* locations within the same line. Therefore their definition of self-spatial reuse includes self-temporal reuse as a subset, while our definition of spatial reuse does not. Finally, what they call *group-spatial* reuse is the same as what we call *group* reuse, and occurs whenever different array references access the same cache line. We do not treat their *group-temporal* reuse (different references accessing exactly the same location) as a special case, since it is a subset of our group reuse. As we will demonstrate later in this chapter, our categories of reuse correspond to different cases for scheduling prefetches.

A key simplification of reuse analysis is that rather than trying to precisely compute the sets of iterations (i.e. actual loop index values) that use the same data, which is prohibitively expensive, we instead succinctly capture the intuitive notion 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 (i.e. a finite convex polyhedron bounded by the loop bounds), 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* [87]. The remainder of this subsection describes how this mathematical representation is used to compute temporal, spatial, and group reuse.

[2] "Memory line" refers to cache-line-sized blocks of contiguous memory, which are the unique elements that can be mapped into cache entries.

Temporal Reuse

Since temporal reuse occurs whenever a given reference accesses the same location in multiple iterations, we can isolate these cases by solving for whenever the array indexing functions yield identical results, given different loop indices. To facilitate this task, we represent an array indexing function $\vec{f}(\vec{i})$ which maps n loop indices into d array indices, where n is the depth of the loop nest and d is the dimensionality of the array, as

$$\vec{f}(\vec{i}) = H\vec{i} + \vec{c}$$

where H is a d x n linear transformation matrix, $\vec{i}$ is an n-element iteration space vector, and $\vec{c}$ is a d-element constant vector. For example, the three array references in Figure 2.1 would be written as

$$\texttt{A[i][j]} = \texttt{A}\left(\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}\right),$$

$$\texttt{B[j][0]} = \texttt{B}\left(\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}\right), \text{ and}$$

$$\texttt{B[j+1][0]} = \texttt{B}\left(\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix}\right).$$

Given this representation, temporal reuse occurs between iterations $\vec{i}_1$ and $\vec{i}_2$ whenever $H\vec{i}_1 + \vec{c} = H\vec{i}_2 + \vec{c}$, i.e. when $H(\vec{i}_1 - \vec{i}_2) = \vec{0}$. Rather than worrying about individual values of $\vec{i}_1$ and $\vec{i}_2$, we say that reuse occurs along the direction vector $\vec{r}$ when $H\vec{r} = \vec{0}$. The solution to this equation is the *nullspace* of H, which is a vector space in $\mathcal{R}^n$ (i.e. each vector has n components).

To make this analysis more concrete, consider the `B[j+1][0]` reference from our example in Figure 2.1. This reference accesses the same location in iterations (i_1, j_1) and (i_2, j_2) whenever

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

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

This equation is true whenever $j_1 = j_2$, and regardless of the difference between i_1 and i_2. In our vector space representation, we would say that temporal reuse occurs whenever the difference

Figure 2.2: Example of a more complicated access pattern that can be handled by reuse analysis.

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 also handle more complicated access patterns such as `C[i+j][0]` in Figure 2.2. Intuitively, reuse does not occur along a single loop in this case, but rather along a "diagonal" across loops `i` and `j`, since the same data is accessed whenever the sum of `i` plus `j` is equivalent. Using similar notation as before, `C[i+j][0]` accesses the same data in iterations (i_1, j_1) and (i_2, j_2) whenever

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

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

This equation is true whenever $i_1 - i_2 = j_2 - j_1$. With reuse analysis, we can compute this result directly by solving the nullspace of $\begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}$, which is span $\{(1, -1)\}$. This diagonal reuse vector space agrees with our intuition of when reuse occurs, as illustrated in Figure 2.2(b). Thus we see that this vector space representation allows us to efficiently capture the reuse in a broad range of affine access patterns.

Spatial Reuse

Computing spatial reuse requires a slight variation on how we compute temporal reuse. Assuming the data are stored in row major order (an assumption we make without loss of generality), accesses to the same cache line will only occur when the same row is accessed.[3] In addition, the row index expressions must be different, but still fall within the size of a cache line. We can test for all of these conditions as follows.

Two different iterations access the same row whenever all but the row index are equivalent. This is in contrast with temporal reuse, where *all* indices, including the row, must be equivalent. To extract this *spatial reuse vector space*, we simply replace the last row in H with zeros to create H_S, and solve for the nullspace of H_S. For example, consider the `A[i][j]` reference in Figure 2.1, where $H = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$, and therefore $H_S = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$. The resulting nullspace of $\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$ is span $\{(0, 1)\}$, which indicates that the same row of `A[i][j]` is accessed along the inner loop.

To check whether *different* elements are being accessed within the same row, we compare whether the temporal and spatial reuse vector spaces are identical. This can occur since reusing the same data item is a degenerate case of reusing the same cache line (i.e. the nullspace of H is a subset of the nullspace of H_S.). If the temporal and spatial reuse vector spaces *are* identical, then there is strictly temporal reuse—if they differ, then there is spatial reuse along the vectors that are unique to the spatial reuse vector space. For example, the `A[i][j]` reference in Figure 2.1 has a temporal reuse vector space of span $\{\}$ (i.e. there is no temporal reuse), and a spatial reuse vector space of span $\{(0, 1)\}$; therefore unique elements within the row are accessed along span $\{(0, 1)\}$ (i.e. the inner loop). For the `B[j][0]` reference, however, the temporal and spatial reuse vector spaces are both span $\{(1, 0)\}$, and therefore there is only temporal reuse.

[3] If the data are stored in *column* major order, then accesses to the same cache line can only occur when the same *column* is accessed.

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    foo(C[i],C[j]);
```

Figure 2.3: Example of non-uniformly generated references.

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    foo(C[2i][j],C[2i+1][j]);
```

Figure 2.4: Example of uniformly generated references that do not have reuse.

Once we identify accesses to different elements within the same row, the final step is to check whether the stride is less than the cache line size. If so, then the reference has spatial reuse.

Group 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 [24]. For example, references `B[j][0]` and `B[j+1][0]` in Figure 2.1 are uniformly generated, while references `C[i]` and `C[j]` in Figure 2.3 are not. In the former case, `B[j][0]` is accessing data brought into the cache by `B[j+1][0]` during the previous `j` iteration, making it very likely that this reuse will result in locality. In the latter case, only iterations near the diagonal (i.e. when `i` = `j`) are likely to have exploitable reuse. Thus we will only consider group reuse among sets of uniformly generated references.

Although uniformly generated references are the likely candidates for group reuse, it is still possible that they never access the same data. For example, the `C[2i][j]` and `C[2i+1][j]` references in Figure 2.4 are uniformly generated, but never overlap since `C[2i][j]` only accesses even rows while `C[2i+1][j]` only accesses odd rows of matrix `C`. To exclude such cases, we check whether a *particular solution* to the common transformation matrix H exists that yields the constant difference between the two array index functions. We express this mathematically by saying that two distinct references `A[`$H\vec{i}+\vec{c}_1$`]` and `A[`$H\vec{i}+\vec{c}_2$`]` access the same data if and

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    foo(C[i][2j],C[i][2j+1]);
```

Figure 2.5: Example of references that access the same cache lines despite never accessing the same data items.

only if

$$\exists \vec{r} : H\vec{r} = \vec{c}_1 - \vec{c}_2. \tag{2.1}$$

For example, the two references in Figure 2.4 would be represented as

$$\texttt{C[2i][j]} = \texttt{C}\left(\begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}\right), \text{ and}$$

$$\texttt{C[2i+1][j]} = \texttt{C}\left(\begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix}\right).$$

These two references access the same data if an integer solution (i_r, j_r) exists to

$$\begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} i_r \\ j_r \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

However, there is no integer solution in this case, since there is no integer i_r such that $2i_r = 1$. In contrast, the `B[j][0]` and `B[j+1][0]` references in Figure 2.1 access the same data since $(i_r, j_r) = (0, 1)$ is one particular solution to

$$\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} i_r \\ j_r \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

While equation (2.1) specifies cases where distinct references access the same *data item*, we are interested in cases where distinct references access the same *cache line*. In Wolf and Lam's terminology, the former case is *group-temporal* reuse, and the latter case is *group-spatial* reuse. We refer to group-spatial reuse simply as *group* reuse, since it is a superset of group-temporal reuse and we have no need to distinguish the two cases.

Figure 2.5 shows an example of two references that access the same cache lines, although never the same data items. Similar to our earlier example in Figure 2.4, `C[i][2j]` and

`C[i][2j+1]` never overlap since they only access even and odd columns of `C`, respectively. However, since they access adjacent elements within the same row on each iteration, they will reuse the same cache lines, provided the lines are long enough to hold at least two array elements.

To detect all of these group reuse cases (including *group-spatial*, we modify equation (2.1) slightly by replacing the last rows in H, $\vec{c}_1$, and $\vec{c}_2$ with zeros to form H_S, $\vec{c}_{S,1}$, and $\vec{c}_{S,2}$, respectively. We therefore say that two distinct references `A[`$H\vec{i}+\vec{c}_1$`]` and `A[`$H\vec{i}+\vec{c}_2$`]` have group reuse if and only if

$$\exists \vec{r}: H_S\vec{r} = \vec{c}_{S,1} - \vec{c}_{S,2}, \tag{2.2}$$

and also provided that the constant difference between the last rows of $\vec{c}_1$ and $\vec{c}_2$ is less than the cache line size divided by the element size.[4] For the `C[i][2j]` and `C[i][2j+1]` references in Figure 2.5, where $H = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}$, $\vec{c}_1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$, and $\vec{c}_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$, and hence $H_S = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$, $\vec{c}_{S,1} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$, and $\vec{c}_{S,2} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$, group reuse only occurs if an integer solution (i_r, j_r) exists to

$$\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i_r \\ j_r \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

Since $(i_r, j_r) = (0,0)$ is a particular solution to this equation, and given a cache line size of at least two array elements, group reuse does occur for this case.

Now that we have demonstrated how the various types of reuse can be computed, the next step in our algorithm is determining when reuse actually results in a locality benefit, given that we have a *finite* rather than an *infinite* cache. To make this distinction, we use the concept of a *localized iteration space*, which is described in the following subsection.

2.3.3 Localized Iteration Space

As we mentioned at the outset of Section 2.3, reuse only translates to locality if the subsequent use of data occurs before the data are displaced from the cache. The likelihood of displacement depends on (i) the amount of data brought into the cache between reuses, which is influenced by

[4] In reality, the alignment of the two references with respect to the cache line boundaries may also be a concern. In the worst case, adjacent references may always straddle cache line boundaries, potentially never reusing the same cache lines. In our implementation, we account for this probabilistically by assuming that two references within half a cache line width of each other probably fall within the same cache line.

Figure 2.6: Example of how loop iteration counts and cache size affect locality.

factors such as loop iteration counts, and (ii) the ability of the cache to retain data, which depends on factors such as the cache size. In this subsection, we describe how our algorithm takes such factors into account to compute locality through the notion of a *localized iteration space*.

We begin with the example in Figure 2.6 which illustrates how loop iteration counts and cache size affect locality. The code in Figures 2.6(a) and 2.6(b) is similar to our original example in Figure 2.1(a), except that the upper bound of the `j` loop is small in Figure 2.6(a) (8 iterations) and large in Figure 2.6(b) (10,000 iterations). The `B[j+1][0]` reference in both cases has temporal reuse along the outer loop (as discussed earlier in Section 2.3.2). In Figure 2.6(a), this reuse is likely to result in temporal locality since the eight `j` iterations bring only 192 bytes into the 8

Kbyte cache between reuses of the same `B[j+1][0]` locations. In Figure 2.6(b), however, the temporal reuse does not result in locality since the 10,000 `j` iterations will sweep 240 Kbytes of data through the 8 Kbyte cache, flushing a given `B[j+1][0]` location before it can be reused.

Predicting accurately whether data will remain in the cache is infeasible for the compiler, due to factors such as symbolic loop iteration counts, cache mapping conflicts, etc. Rather than trying to represent exactly which reuses would result in cache hits, we adopt Wolf and Lam's approach of capturing only the *dimensionality* of the iteration space that has data locality [87]. We define the *localized iteration space* to be the set of loops that can exploit reuse. For example, the localized iteration space includes both loops in Figure 2.6(a), and only the innermost loop in Figure 2.6(b). In the latter case, data fetched in the loop body will be available to subsequent iterations within the same innermost loop, but not to subsequent iterations of the outer loop.

Computing the Localized Iteration Space

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. Figure 2.7 shows our algorithm for computing the localized iteration space, which consists of two major substeps. First, we use our reuse vector information to estimate the amount of data accessed by each loop (i.e. `Tally_Loop_Traffic` in Figure 2.7). This involves visiting each array reference and projecting the amount of data it references onto the surrounding loops. We associate a running total of data accessed with each array reference, and this running total is potentially multiplied by the number of iterations of each surrounding loop, depending on the type of reuse the reference has with respect to that loop. This first substep is a simplified version of algorithms proposed previously [21, 24, 64]. Once the amount of data accessed by each loop has been estimated, the second substep is comparing these amounts with the expected cache capacity to decide which loops are localized (i.e. `Is_Localized` in Figure 2.7). This second substep is a separate top-down and then bottom-up pass over the loops, which allows us to decide that (i) a loop is definitely localized if any surrounding loops are definitely localized (perhaps because they have been actively "blocked" or "tiled" by the locality optimizer), and (ii) a loop is definitely *not* localized if any internal loops are definitely not localized (perhaps because they reference too much data). Such information is particularly helpful when the amount of data referenced by a loop is a symbolic range, and therefore cannot be compared directly against a constant cache size.

Two complications that arise in this algorithm are (i) estimating the amount of data accessed in the presence of symbolic loop bounds, and (ii) modeling cache capacity given the potential for

```
algorithm Compute_Localized_Iteration_Space
        (A: set of affine array references,
         L: set of loops)
    return ();

  (singleT, totalT) := Tally_Loop_Traffic(A);
  foreach l ∈ L, where l is an outermost loop do
    Is_Localized(l, singleT, totalT, False);
  end foreach;
  /* Now all loops have been marked, indicating whether they are in the localized iteration space. */
end algorithm;

algorithm Tally_Loop_Traffic
        (A: set of affine array references)
    return
        (singleT: array of ranges,      /* data traffic for single iteration of each loop */
         totalT:  array of ranges);     /* data traffic for total iterations of each loop */
  /* In this algorithm:                                                                    */
  /*   Reuse(a,l) returns the reuse of reference a with respect to loop l. Possible values */
  /*                 are "Temporal", "Spatial", and "None".                                */
  /*   Iterations(l) returns the total iterations of loop l expressed as a symbolic range. */

  singleT, totalT: array of ranges = 0;

  /* For each array reference, project the data traffic onto the surrounding loops. */
  foreach a ∈ A do
    /* Ignore references with group reuse, unless they are the leading references. */
    if (a is the leading reference in its group) then
      /* Keep a running total of data traffic as it grows from inner to outer loops. */
      runningTotal: range := cacheLineSize;
      for l := innermost to outermost loop surrounding a do
        singleT[l] := singleT[l] + runningTotal;
        if (Reuse(a, l) = None) then
          runningTotal := runningTotal*Iterations(l);
        else if (Reuse(a, l) = Spatial) then
          spatialSavings: int := max(1, cacheLineSize/(sizeof(a)*Stride(a)));
          runningTotal := runningTotal*Iterations(l)/spatialSavings;
        end if;
        if (runningTotal contains the index of loop l) then
          substitute for l in runningTotal;      /* (e.g., l = (n+1)/2, where n = Iterations(l)) */
        end if
        totalT[l] := totalT[l] + runningTotal;
      end for;
    end if;
  end foreach;
  return (singleT, totalT);
end algorithm;
```

Figure 2.7: Algorithm for computing the localized iteration space. (Continued on next page.)

```
algorithm Is_Localized
        (L: loop,                              /* loop to be checked */
        singleT, totalT: array of ranges,      /* data traffic for single and total iterations of loops */
        outerLocalized: boolean)               /* true if surrounding loop is localized */
    return
        (localized: boolean);  /* true if this loop is within the localized iteration space. */

 definitelyLocalized: boolean := ((Fits_In_Cache(singleT[L]) = Yes) or
       (Fits_In_Cache(totalT[L]) = Yes) or withinTile(L) or outerLocalized);
 innerNotLocalized: boolean := False;
 localized: boolean;

 /* First descend over loops inside L to determine whether they are localized. */
 /* Note that this does not assume perfectly-nested loops. */
 foreach l ∈ (loops at next nesting level inside L) do
  if (not Is_Localized(l, singleT, totalT, definitelyLocalized)) then
   innerNotLocalized := True;
  end if;
 end foreach;
 if (innerNotLocalized or (Fits_In_Cache(singleT[L]) = No)) then
  localized := False;
 else if definitelyLocalized then
  localized := True;
 else if (default policy on unknown loop bounds is to assume large loops) then
  localized := False;
 else  /* default policy is to assume small loops */
  localized := True;
 end if;
 Record whether L is localized or not;
 return (localized);
end algorithm;

/* Compute the certainty that the given range of data traffic fits in the cache. */
/* The "certainty" data type has three values: "Yes", "No", and "Maybe". */
algorithm Fits_In_Cache
        (T: range)     /* Range of data traffic, which may include symbolic expressions. */
    return (fits: certainty);

 fits: certainty := Maybe;    /* we are only absolutely certain in the two cases below */
 if (isConstant(T.minimum) and (T.minimum > cacheCapacity)) then
  fits := No;
 else if (isConstant(T.maximum) and (T.maximum < cacheCapacity)) then
  fits := Yes;
 end if;
 return (fits);
end algorithm;
```

Figure 2.7: Algorithm for computing the localized iteration space. (Continued from previous page.)

(a) Example Code

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

(b) Volume of Data Accessed

	`j` loop (100 iterations)			`i` loop (3 iterations)		
		Data Accessed			Data Accessed	
Reference	Type of Reuse	Single Iteration (bytes)	Total Iterations (bytes)	Type of Reuse	Single Iteration (bytes)	Total Iterations (bytes)
`A[i][j]`	Spatial	16	800	None	800	2400
`B[j+1][0]`	None	16	1600	Temporal	1600	1600
`B[j][0]`	(Group with `B[j+1][0]`)	0	0	(Group with `B[j+1][0]`)	0	0
Total	N/A	32	2400	N/A	2400	4000

Figure 2.8: Example of algorithm for estimating volume of data accessed by each loop.

mapping conflicts in direct-mapped or set-associative caches. To address the symbolic loop bound problem, we do three things. First, we use interprocedural constant propagation to eliminate as many symbolic loop bounds as possible. Second, we represent the loop iterations and subsequent data traffic as symbolic ranges, which can include simple linear expressions of symbolic variables. In some cases we can reason about these symbolic values, such as when they are indices of surrounding loops (an example of this will be shown later in Figure 2.9). Third, when all else fails, we use a default policy of assuming loop iteration counts to be either large or small to resolve such cases. (A fourth option which we explore later in Section 5.2.1 is using *control-flow feedback* in such cases.) To address cache mapping conflicts, we 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 these approximations later in Section 3.3, and will suggest further improvements in Section 5.2.1.

Figure 2.8 illustrates how our algorithm computes the localized iteration space for the example code introduced earlier in Section 2.3.1. We begin with the `A[i][j]` reference, which has spatial reuse along the inner loop and no reuse along the outer loop. During a single `j` iteration,

`A[i][j]` will bring (at most) a single line into the cache (16 bytes). To compute the total data accessed by a single pass through the `j` loop, we multiply the current "running total" (16 bytes) by the total number of `j` iterations (100), and divide by the savings factor due to spatial reuse (2), yielding a total of 800 bytes. This is also the amount of data accessed by the `A[i][j]` reference during a single iteration of the `i` loop. Since `A[i][j]` has *no* reuse along the `i` loop, we simply multiply the running total (800 bytes) by the number of `i` iterations (3) to get a total of 2400 bytes accessed by `A[i][j]` in the entire loop nest. Next, we consider the `B[j+1][0]` reference, which has no reuse along the `j` loop and temporal reuse along the `i` loop. The data accessed during a single `j` iteration is again a single cache line (16 bytes), and because there is no reuse along `j`, we multiply by the total `j` iterations (100) to get 1600 bytes accessed during a single `i` iteration. Since `B[j+1][0]` has temporal reuse along `i`, the running total is not increased any further, and the total data accessed by `B[j+1][0]` during the loop nest is therefore 1600 bytes. We can ignore the `B[j][0]` reference, since it has group reuse with `B[j+1][0]`, and `B[j+1][0]` is the leading reference. After tallying each reference's contribution to total data traffic, we see that a single iteration of `j` accesses only 32 bytes (and therefore is definitely localized), and a single iteration of `i` accesses 2400 bytes, which is also likely to fit in an 8 Kbyte cache. Therefore our algorithm would conclude that both loops are within the localized iteration space.

Figure 2.9 presents a more complex example, where the upper bound of the inner loop is a symbolic variable (`i`). In this case, the amount of data accessed during a single `i` iteration is a symbolic expression (`8i` bytes), which cannot readily be compared against a fixed cache size. However, since the compiler understands how the `i` variable behaves, it recognizes that a triangular region within the `A` matrix is being accessed, and that `i` can be substituted with

$$\sum_{i=0}^{14} i = \frac{14(14+1)}{2} = 105$$

to compute a total of 840 bytes being accessed by the `A[i][j]` reference in this loop nest. Since 840 bytes is likely to fit in the 8 Kbyte cache, the compiler would again conclude that both loops are within the localized iteration space.

We represent the localized iteration space as a vector space, so that it can be directly compared with our vector space representation of data reuse. For example, if both loops in a two-level loop nest are within the localized iteration space (as in Figures 2.6(a), 2.8, and 2.9), the corresponding *localized vector space* would be represented as span $\{(1,0),(0,1)\}$. If only the innermost loop is localized (as in Figure 2.6(b)), then the localize vector space would be span $\{(0,1)\}$. We will

(a) Example Code

```
for (i = 0; i < 15; i++)
  for (j = 0; j < i; j++)
    sum = sum + A[i][j];
```

(b) Volume of Data Accessed

	`j` loop (`i` iterations)			`i` loop (15 iterations)		
		Data Accessed			Data Accessed	
Reference	Type of Reuse	Single Iteration (bytes)	Total Iterations (bytes)	Type of Reuse	Single Iteration (bytes)	Total Iterations (bytes)
`A[i][j]`	Spatial	16	`8i`	None	`8i`	840
Total	N/A	16	`8i`	N/A	`8i`	840

Figure 2.9: Example of how symbolic values can be useful when computing volume of data accessed by each loop.

now describe how the intrinsic data reuse and the localized iteration space can be combined to compute data locality.

Computing Locality

Intuitively, a data access has locality (i.e. hits in the cache) if it is reusing a cache line that was referenced sometime in the past *and* this previous reference occurred recently enough that the data is still in the cache. We can isolate these instances by intersecting the *reuse vector space* with the *localized vector space*, i.e.

$$\text{Reuse Vector Space} \cap \text{Localized Vector Space} \Rightarrow \text{Locality Vector Space.}$$

For example, the `B[j+1][0]` reference in Figure 2.1(a) has a temporal reuse vector space of span$\{(1,0)\}$ (i.e. there is temporal reuse along the outer loop), as described earlier in Section 2.3.2. If both loops in the nest are localized (as in Figure 2.6(a)), then the temporal locality vector space is

$$\begin{array}{ccccc} \text{Temporal Reuse} & \cap & \text{Localized Iteration Space} & \Rightarrow & \text{Temporal Locality} \\ \text{span}\{(1,0)\} & \cap & \text{span}\{(1,0),(0,1)\} & \Rightarrow & \text{span}\{(1,0)\}, \end{array}$$

```
for (i = 0; i < 100; i++)
  for (j = 0; j < 10000; j++)
    foo(A[i+1][j],A[i][j]);
```

Figure 2.10: Example of references with group reuse but not group locality.

i.e. the temporal reuse *does* result in temporal locality.

However, if only the innermost loop is localized (as in Figure 2.6(b)), then the temporal locality vector space is

$$\begin{array}{ccccc} \text{Temporal Reuse} & \cap & \text{Localized Iteration Space} & \Rightarrow & \text{Temporal Locality} \\ \text{span}\{(1,0)\} & \cap & \text{span}\{(0,1)\} & \Rightarrow & \text{span}\{\}, \end{array}$$

i.e. there is no temporal locality.

Similar mathematical treatment determines whether spatial reuse translates into spatial locality (i.e. Spatial Reuse $\cap$Localized Iteration Space $\Rightarrow$Spatial Locality).

For group reuse, the localized iteration space is used to partition the references that share group reuse into *equivalence classes*. An equivalence class is a set of references for which group locality exists, and therefore can be treated as a single reference. Equivalence classes are computed by finding a *particular solution* to the difference between the array index equations of two references. If a particular solution exists, and if it is within the localized iteration space, then the references belong in the same equivalence class. In addition, the particular solution result is used to determine the *leading reference* within an equivalence class, which is the reference within the set that will actually suffer the cache miss. The compiler only has to schedule prefetches for leading references.

For example, $\vec{r}_p = (0,1)$ is a particular solution to the difference between the `B[j+1][0]` and `B[j][0]` references in Figure 2.1. Even if the localized iteration space includes only the innermost loop, these two references will be within the same equivalence class since

$$\begin{array}{ccc} \vec{r}_p & \in & \text{Localized Iteration Space} \\ (0,1) & \in & \text{span}\{(0,1)\}. \end{array}$$

Also, since the first non-zero element in $\vec{r}_p$ is positive, this indicates that `B[j+1][0]` is the leading reference, since it accesses the data during an earlier iteration than `B[j][0]`.

As a counterexample, references `A[i+1][j]` and `A[i][j]` in Figure 2.10 share *group reuse* but not *group locality*. In this case, the particular solution is $\vec{r}_p = (1,0)$, and the localized

Table 2.2: Prefetch predicates for the different types of locality.

Type of Locality	Miss Instances	Prefetch Predicate
None	Every Iteration	*True*
Temporal	First Iteration	`i` =0
Spatial	Every l Iterations (l = cache line size)	(`i` mod l) = 0
Group (and not leading reference)	Essentially Never	*False*

iteration space contains only the innermost loop. Therefore, since

$$(1,0) \notin \text{span}\{(0,1)\},$$

`A[i+1][j]` and `A[i][j]` do not share group locality, and will be partitioned into separate equivalence classes.

After computing the localized iteration space and intersecting it with the reuse vector space, we now have a precise description of data locality expressed as a *locality vector space*. The final step in our locality analysis algorithm is converting this vector space representation of data locality into *prefetch predicates*, as described in the next subsection.

2.3.4 The Prefetch Predicate

Although vector space representations are attractive when we compute data locality, a more convenient representation for the purpose of scheduling prefetches is to associate a logical predicate with each reference such that the predicate is true during each dynamic instance when the reference is expected to suffer a cache miss. These miss instances are precisely the cases we want to prefetch, and therefore we refer to these predicates as *prefetch predicates*. In this subsection we describe how the prefetch predicates are constructed based on the locality vector spaces.

The expected dynamic miss instances vary according to the different types of locality. If an access has no locality, it will miss on every iteration. 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. If an access has group locality and is not the leading reference, it will hardly ever suffer cache misses.

Table 2.2 shows the prefetch predicates corresponding to each type of locality. To simplify this exposition (and without loss of generality), we assume here that the iteration count starts at

(a)

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

(b)

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

Figure 2.11: Example of how prefetch predicates are constructed.

0, and that the data arrays are aligned to start on a cache line boundary.[5] As we see in Table 2.2, the prefetch predicate for a reference with no locality is simply "*True*", since misses are expected to occur on every iteration. For an access with temporal locality, the prefetch predicate is only true during the first loop iteration (i.e. when "`i` = 0"). With spatial locality, the prefetch predicate contains a modulo function such that it is true each time a cache line boundary is crossed (i.e. when "`i` mod l = 0", where l is the number of accesses per cache line). Finally, for references with group locality that are not leading references, the prefetch predicate is "*False*", since they are not expected to suffer a significant number of misses and hence should not be prefetched.

The prefetch predicate of a reference within a multi-level loop nest is simply the conjunction of all the predicates imposed by each form of locality within the loop nest. For example, the `A[i][j]` reference in Figure 2.11 has no locality with respect to the `i` loop, which corresponds to a predicate of "*True*", and spatial locality along the `j` loop, which corresponds to a predicate of "(`j` mod 2) = 0"; therefore the overall prefetch predicate for `A[i][j]` is

$$\textit{True} \quad \wedge \quad (\texttt{j} \bmod 2) = 0) \quad \Rightarrow ((\texttt{j} \bmod 2) = 0).$$

Similarly, the `B[j+1][0]` reference in this example has temporal locality along the `i` loop, which corresponds to a predicate of "`i` = 0", and no locality along the `j` loop, which corresponds

[5] In the general case, 0 would be replaced by the lower bound of the loop, and a constant offset would be added to the modulo prefetch predicate for spatial locality in Table 2.2.

to a predicate of "*True*"; hence the overall prefetch predicate for `B[j+1][0]` is

$$(\texttt{i} = 0) \quad \wedge \quad \mathit{True} \quad \Rightarrow (\texttt{i} = 0).$$

The `B[j][0]` reference in Figure 2.11 has a prefetch predicate of "*False*", since it has group locality with `B[j+1][0]`.

Once these prefetch predicates have been constructed, our algorithm has answered the question of "what to prefetch"; the answer is each reference should be prefetched whenever its prefetch predicate is true. Now that the analysis phase of our algorithm is complete, the next step is to schedule the prefetches, as we describe in the next section.

2.4 Scheduling Prefetches

Our algorithm for scheduling prefetches has two goals. First, we would like to minimize the amount of overhead that results from inserting prefetches. Second, we would like the prefetches to be as effective as possible at eliminating cache misses. To address this first goal, we use *loop splitting* techniques such as *peeling* and *unrolling* to isolate only those dynamic instances when the prefetch predicates are true. To address the second goal, we use *software pipelining* to schedule the prefetches so that they arrive in the cache just before they are needed. We discuss both of these steps in our scheduling algorithm in this section.

2.4.1 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 (probably at least as expensive as issuing prefetches all the time), and thus defeats 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*.

Table 2.3 shows the loop splitting transformations for the various prefetch predicates. If the prefetch predicate is either "*True*" or "*False*", then no transformation is necessary since we either prefetch all the time or not at all. An "`i` = 0" predicate resulting from temporal locality requires the first iteration of the loop to be isolated through *peeling*. The generic schema for peeling a loop is illustrated in Figure 2.12. Finally, the "(`i` mod l) = 0" predicate resulting from

Table 2.3: Loop splitting transformations for the various types of locality.

Type of Locality	Prefetch Predicate	Loop Splitting Transformation
None	*True*	None
Temporal	`i` =0	Peel first iteration from loop `i`
Spatial	(`i` mod l) = 0	Unroll loop `i` by l (or if l is too large, strip-mine loop `i` by l)
Group (and not leading reference)	*False*	None

(a) Code Before Peeling

```
for (i = 0; i < n; i++) {
  if (i == 0)
    f(i);
  g(i);
}
```

(b) Code After Peeling

```
f(0);
g(0);
for (i = 1; i < n; i++) {
  g(i);
}
```

Figure 2.12: Generic schema for peeling a loop.

spatial locality requires the loop to be either *unrolled* or *strip-mined* [64] to isolate one in every l iterations. Figures 2.13 and 2.14 show the generic schemas for unrolling and strip-mining a loop, respectively. In general, the unrolling transformation is preferable for small or moderate values of l. However, as l becomes large, perhaps due to large cache line sizes, the overhead of replicating the loop body will eventually become more expensive than the extra loop control overhead involved in strip-mining. For the implementation of the compiler algorithm used in this dissertation, however, only the unrolling transformation was considered for spatial locality, since our cache lines are moderately sized (16 and 32 bytes).

For nested loops, these loop splitting transformations can be applied recursively to handle

(a) Code Before Unrolling

```
for (i = 0; i < n; i++) {
  if ((i mod 4) == 0)
    f(i);
  g(i);
}
```

(b) Code After Unrolling

```
for (i = 0; i < n; i+=4) {
  f(i);
  g(i);
  g(i+1);
  g(i+2);
  g(i+3);
}
```

Figure 2.13: Generic schema for unrolling a loop.

(a) Code Before Strip-Mining

```
for (i = 0; i < n; i++) {
  if ((i mod 64) == 0)
    f(i);
  g(i);
}
```

(b) Code After Strip-Mining

```
for (j = 0; j < n; j+=64) {
  f(j);
  for (i = j; i < j+64; i++)
    g(i);
}
```

Figure 2.14: Generic schema for strip-mining a loop.

each prefetch predicate. 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 a loop is growing. We suppress peeling or unrolling when the loop becomes too large. 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. This is a reasonable choice because when the loop body becomes this large, the cost of a conditional statement is relatively small.

Now that our compiler knows what to prefetch and has isolated those instances in the code with minimal overhead, the final step is to schedule the prefetches the proper amount of time in advance to hide the memory latency through *software pipelining*.

2.4.2 Software Pipelining

Software pipelining [48, 65] is a technique that allows us to hide memory latency by overlapping the prefetches for a future iteration with the computation of the current iteration. While this transformation is straightforward, the key parameter is how far in advance the prefetches should be scheduled. On the one hand, prefetches must be issued early enough to hide memory latency. But on the other hand, if prefetches are issued *too early*, the data may be flushed from the cache before they can be 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

$$\left\lceil \frac{l}{s} \right\rceil \tag{2.3}$$

where l is the expected memory latency and s is the length of the shortest path through the loop body. In our algorithm, l is a parameter given to the compiler, which we set to be the largest expected memory latency, including contention. The s parameter is computed by the compiler for each loop nest, using the `Shortest_Path` algorithm shown in Figure 2.15. The interesting cases in this algorithm are: (i) conditional statements, where we choose the shorter of the "*then*" and "*else*" paths; (ii) loops, where we use the iteration count if it is known—otherwise, we assume at least a single iteration is executed; and (iii) procedure calls, where we use the length of the procedure body, unless there is recursion. We take the ceiling of the ratio to ensure that all of the latency is hidden.

Figure 2.16 shows a simple example of how software pipelining would be used to schedule prefetches. Assuming equation (2.3) yields that 5 iterations are sufficient to hide the memory latency for the loop in Figure 2.16(a), we begin in Figure 2.16(b) with a *prolog* that issues the first several prefetches. Next, a *steady state* loop is executed where both prefetches and computation occur. Finally, an *epilog* loop completes the last several iterations of computation.

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

```
/* This algorithm is called initially with the loop body as the argument. */
algorithm Shortest_Path
        (I: list of instructions)  /* Instructions at a given control-graph level. */
    return (shortestLength: integer);

 shortestLength: integer := 0;

 foreach i ∈ I do
  if (i is a conditional statement) then
   shortestLength := shortestLength +
           min(Shortest_Path(i.then_part),Shortest_Path(i.else_part));
  else if (i is a loop) then
   if (isConstant(i.num_iterations)) then
    shortestLength := shortestLength +
            i.num_iterations*Shortest_Path(i.loop_body);
   else     /* Assume at least one iteration when iteration count is unknown. */
    shortestLength := shortestLength + Shortest_Path(i.loop_body);
   end if;
  else if (i is a non-recursive procedure call) then
   shortestLength := shortestLength + Shortest_Path(i.procedure.body);
  end if;
  shortestLength := shortestLength + 1;
 end foreach;
 return (shortestLength);
end algorithm;
```

Figure 2.15: Algorithm for computing the shortest path through a loop body.

the prefetched data may be replaced (i.e. the loop is outside the localized iteration space), we suppress issuing the prefetch.

2.5 Putting It All Together

Having described all of the components of our core compiler algorithm, we now return to our original example to tie all the pieces together, and also discuss some issues that arose while implementing this algorithm.

2.5.1 Example Revisited

Figure 2.17 summarizes the major steps of our prefetching algorithm. The example code and resulting data locality are shown in Figures 2.17(a) and 2.17(b), respectively. Figure 2.17(c)

(a) Original Loop

```
for (i = 0; i < 100; i++)
  A[i][0] = 0;
```

(b) Software Pipelined Loop

```
for (i = 0; i < 5; i++)            /* Prolog */
  prefetch(&A[i][0]);

for (i = 0; i < 95; i++) {         /* Steady State */
  prefetch(&A[i+5][0]);
  A[i][0] = 0;
}

for (i = 95; i < 100; i++)         /* Epilog */
  A[i][0] = 0;
```

Figure 2.16: Example of how software pipelining is used to schedule prefetches the proper amount of time in advance. For this example, assume that 5 iterations are enough to hide memory latency.

shows the result of the analysis phase of our algorithm, where we have constructed predicates indicating when all three array references should be prefetched. Finally, Figure 2.17(d) shows the resulting code with prefetching, which is generated by the scheduling phase of our algorithm as follows.

First, our algorithm performs loop splitting based on the prefetch predicates. The "`i` = 0" predicate resulting from the temporal locality of the `B[j+1][0]` reference causes the compiler to peel the `i` loop. As we see in Figure 2.17(d), the prefetches for the `B` matrix occur only in the peel of `i`. Also, to isolate the spatial locality of the `A[i][j]` reference, the "(`j` mod 2) = 0" predicate causes the `j` loop to be unrolled by a factor of 2—both in the peel and in the main iterations of the `i` loop. As a result, Figure 2.17(d) shows that there is only one prefetch of the `A` matrix for every two copies of the loop body.

Once the miss instances have been isolated through loop splitting, the compiler then software pipelines the prefetches as follows. Using the algorithm in Figure 2.15, our compiler determines that the shortest path through the loop body is 36 instructions long. Given that memory latency is 100 cycles for this example, equation (2.3) yields that $\left\lceil \frac{100}{36} \right\rceil = 3$ iterations are sufficient to hide the latency. Once this iteration count is determined, the code transformation is mechanical. In Figure 2.17(d) we see the resulting *prolog*, *steady state*, and *epilog* loops in both the peel and the main iterations of the `i` loop. Thus the code in Figure 2.17(d) issues all of the useful

(c) Result of Locality Analysis

Reference	Locality	Prefetch Predicate
`A[i][j]`	$\begin{bmatrix} \texttt{i} \\ \texttt{j} \end{bmatrix} = \begin{bmatrix} \text{none} \\ \text{spatial} \end{bmatrix}$	(`j` mod 2) = 0
`B[j+1][0]`	$\begin{bmatrix} \texttt{i} \\ \texttt{j} \end{bmatrix} = \begin{bmatrix} \text{temporal} \\ \text{none} \end{bmatrix}$	`i` = 0
`B[j][0]`	$\begin{pmatrix} \text{Group with} \\ \texttt{B[j+1][0]} \end{pmatrix}$	*False*

Figure 2.17: Example of selective prefetching algorithm. (Continued on next page.)

(d) Code with Prefetching

```
prefetch(&A[0][0]);                          /* Peel of i loop (i = 0) ... */
for (j = 0; j<6; j += 2) {                        /* prolog */
  prefetch(&B[j+1][0]);
  prefetch(&B[j+2][0]);
  prefetch(&A[0][j+1]);
}
for (j = 0; j<94; j += 2) {                       /* steady state */
  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) {                     /* epilog */
  A[0][j] = B[j][0]+B[j+1][0];
  A[0][j+1] = B[j+1][0]+B[j+2][0];
}                                            /* ... end of peel of i loop */
for (i = 1; i<3; i++) {                       /* Main i loop iterations (i > 0) ... */
  prefetch(&A[i][0]);
  for (j = 0; j<6; j += 2)                        /* prolog */
    prefetch(&A[i][j+1]);
  for (j = 0; j<94; j += 2) {                     /* steady state */
    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) {                   /* epilog */
    A[i][j] = B[j][0]+B[j+1][0];
    A[i][j+1] = B[j+1][0]+B[j+2][0];
  }
}                                            /* ... end of main i loop iterations */
```

Figure 2.17: Example of selective prefetching algorithm. (Continued from previous page.)

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).

2.5.2 Implementation Experience

We implemented our prefetching algorithm in the SUIF (Stanford University Intermediate Form) compiler, which includes many of the standard optimizations and generates code competitive with the MIPS 2.10 compiler [80]. The experience of actually implementing our algorithm raised some interesting issues.

The first issue is when prefetching should occur relative to the other optimizations performed by the compiler. In particular, should prefetch insertion occur before or after the bulk of scalar optimization? Inserting prefetches *before* scalar optimization has the disadvantage that the code seen by the prefetching pass may change significantly once optimization takes place; these changes may make it difficult to schedule prefetches the proper amount of time in advance. For example, if the optimizer eliminates a significant fraction of instructions in a loop body, the software pipelining algorithm may underestimate the number of loop iterations necessary to provide enough computation to hide memory latency (i.e. the s parameter in equation (2.3) may be overestimated). On the other hand, inserting prefetches *after* scalar optimization has the following two disadvantages: (i) much of the high-level representation of arrays and loops may be unrecognizable, making it difficult to perform locality analysis; and (ii) the compiler must be extremely careful only to insert highly efficient prefetching code, since scalar optimization will not be run again.[6]

In our experience, the advantages of inserting prefetches *before* scalar optimization far outweigh the disadvantages. First, the ability to recognize high-level structures such as arrays and "for" loops is essential for our analysis, and becomes prohibitively difficult once scalar optimization has radically altered the code. Second, the fact that scalar optimization is performed on the prefetching code itself has two strong advantages: (i) it greatly simplifies the task of inserting prefetches, since code transformations such as loop unrolling and software pipelining can be done in a straightforward manner without undo concern for minimizing overhead; and (ii) scalar optimization appears to do a very good job of minimizing prefetching instruction overhead. The overhead is often less than 2 or 3 instructions per prefetch (as we will see later in Chapter 3), and we believe these results are much better than if the entire burden of minimizing overhead

[6]Since scalar optimization is the most time-consuming part of compilation, we assume that running full scalar optimization twice is not a viable option.

Table 2.4: Order in which the optimization passes occur in the SUIF compiler, including prefetching.

SUIF Pass	Description
cpp	C preprocessor
snout	C front end
ipa	interprocedural analysis
prom	register promotion
expander -F	forward propagate register computation into the lower and upper bounds of "for" loops
oynk -Pconst	constant propagation
oynk -Pdstore	dead store elimination
icp	interprocedural constant propagation
oynk -Pconst	constant propagation
oynk -Pivard	detect induction variables and rewrite in terms of the loop induction variable (i.e. "strength increasing")
xp -C	dismantle "if" statements that are always taken or untaken; dismantle "for" loops that are never executed
pf	**perform prefetching analysis and insert prefetches**
expander afosa	expand "high level" SUIF constructs (e.g., "for" loops) into "low level" SUIF (e.g., branch instructions)
oynk -Pconst	constant propagation
oynk	general scalar optimization
oynk -Psr	strength reduction
oynk -Pconst	constant propagation
oynk -Pdstore	dead store elimination
mexp	massage SUIF constructs to MIPS-palatable form
oynk -Preg	register allocation
mgen	generate MIPS assembly code

was placed on the prefetch code-generating pass itself. Regarding the disadvantage of inserting prefetches before scalar optimization, we found that in practice the impact of reduced loop body sizes could be taken into account by simply increasing the target memory latency parameter (i.e. l in equation (2.3)) by a small factor (less than two). Although this may result in prefetches being issued unnecessarily early (e.g., if scalar optimization does not eliminate any instructions from the loop body), it does not appear to have a noticeable negative impact on performance.

There are some optimizations, however, that should be performed *before* prefetching, since they greatly improve the analysis. One such example is interprocedural constant propagation, which helps eliminate symbolic loop bounds and therefore makes it easier to compute the localized iteration spaces. Table 2.4 shows the order in which our compiler performs its optimizations.

The optimizations performed before prefetching in Table 2.4 tend to improve the information content of the loop and array constructs. After prefetches are inserted, these high-level SUIF constructs (such as hierarchical abstract representations of “for” loops) are deconstructed into simpler low-level constructs (such as branch instructions) which more closely match traditional RISC instruction sets. The bulk of scalar optimization (e.g., common subexpression elimination, etc.) is then performed on this low-level representation. In general, we were quite happy with this approach.

We did experience one surprising problem with the scalar optimizer, however, during our early experiments with the compiler. Our initial code with prefetching had very large instruction overheads—often more than 6 instructions per prefetch. We discovered the problem was that our scalar optimizer did not take full advantage of the “base-plus-offset” addressing mode in the MIPS instruction set by reusing the same base register and simply having different constant offsets when two addresses differed by a constant amount. Instead, the optimizer would use a separate base register to compute each address. While this optimization may not have been a high priority for normal code, it was crucial for our prefetching code, since each prefetch tends to differ by a constant amount from a load or store, and loop unrolling creates many copies of array references that differ by constant amounts. Without this optimization, the compiler quickly ran out of registers, and suffered the large overheads of spilling values to memory. Once this optimization was implemented, we saw dramatic reductions in prefetch instruction overhead.

Chapter 3

Prefetching for Uniprocessors

In this chapter, we evaluate the performance benefits of prefetching for array-based uniprocessor applications. Section 3.1 describes the experimental framework used throughout this chapter, including our architectural assumptions, benchmarks, compile-time parameters, and simulation environment. The results of these experiments are presented in four major subsections. First, Section 3.2 contains a detailed evaluation of the algorithm described in the previous chapter for prefetching affine array references. We observe that each component of this core compiler algorithm is effective at achieving its goal, thereby improving overall execution time by as much as twofold. Second, Section 3.3 evaluates the robustness of this algorithm by varying the compile-time parameters that are determined heuristically rather than precisely for a specific architecture (i.e. the effective cache size, the target memory latency, and the policy on unknown loop bounds). The results show that these parameter variations affect only a small subset of the applications, and the performance impact in those cases is generally small; therefore the algorithm appears to be robust. Third, having already examined prefetching in isolation, Section 3.4 evaluates the interaction between prefetching and another powerful latency-hiding technique for dense-matrix codes: locality optimizations. The results illustrate that prefetching and locality optimizations are complementary, and therefore should be combined. Finally, having focused thus far only on affine array references, we extend our core algorithm in Section 3.5 to handle indirect references, which allows us to prefetch sparse-matrix codes. The results demonstrate that this relatively straightforward extension improves performance by as much as an additional 20%. Finally, we conclude the chapter in Section 3.6 with a summary of the important results.

3.1 Experimental Framework

This section presents the architectural assumptions we make, the benchmark applications, the compile-time parameters to our prefetching algorithm, and the simulation environment used to obtain performance results.

3.1.1 Architectural Assumptions

Our uniprocessor architecture consists of a base workstation architecture that has been extended to support prefetching.

Base Architecture

For the sake of concreteness, we pattern the memory subsystem of our base architecture after a typical MIPS R4000-based workstation. 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, "write-back write-allocate", 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.

Extensions for Prefetching

To experiment with prefetching, we extend our base architecture as follows. We augment the instruction set to include a prefetch instruction that uses a base-plus-offset addressing format and is defined not to take any memory exceptions. The advantages of these properties will be discussed in more detail later in Section 5.1.1, but the basic idea is that base-plus-offset addressing minimizes register usage to avoid spilling, and the non-excepting property allows considerable flexibility in scheduling prefetches (e.g., it is acceptable to prefetch off the end of an array if a proper epilog cannot be constructed). Both levels of the cache are lockup-free [45] in the sense that multiple prefetches can be outstanding along with either a single load or store miss.[1] 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*,

[1] Multiple store misses cannot be outstanding because the processor stalls on a store miss.

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 5.1.2, 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; if it attempts to do so, it is stalled.

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.

3.1.2 Applications

The benchmarks evaluated in this study are all scientific and engineering applications drawn from several benchmark suites. This collection includes NASA7 and TOMCATV from the SPEC benchmarks [77], OCEAN–a uniprocessor version of a SPLASH benchmark [72], and CG (conjugate gradient), EP (“embarassingly parallel”—a Monte Carlo simulation), IS (integer sort), MG (multigrid) from the NAS Parallel Benchmarks [8]. Since the NASA7 benchmark really consists of 7 independent kernels, we study each kernel separately (MXM, CFFT2D, CHOLSKY, BTRIX, GMTRY, EMIT and VPENTA). In addition, for our study in Section 3.5 on prefetching indirect references, we also evaluate MP3D (another uniprocessor version of a SPLASH benchmark) and SPARSPAK [25] (a sparse matrix application), since these applications contain many indirect references. Table 3.1 provides a brief summary of the applications, including their input data sets, and Table 3.2 shows some general characteristics of the applications.

For four of the applications (MXM, CFFT2D, VPENTA and TOMCATV), the mapping conflicts in the direct-mapped cache occurred so frequently that we manually changed the alignment

Table 3.1: Description of uniprocessor applications.

Application	Description	Input Data Set
MXM	matrix multiply	256x128 matrix multiplied by a 128x64 matrix
CFFT2D	complex radix-2 FFT on a 2D array	128x256 matrix
CHOLSKY	Cholesky decomposition in parallel on a set of of input matrices	250 4x40 matrices
BTRIX	block tridiagonal matrix solution along one dimension of a four dimensional array	5x5x30x30 matrix
GMTRY	sets up arrays for a vortex method solution and performs Gausian elimination on the resulting arrays	100x5 matrix
EMIT	creates new vortices according to certain boundary conditions	100x5 matrix
VPENTA	inverts 3 matrix pentadiagonals in a highly parallel fashion	128x128 matrices
TOMCATV	vectorized mesh generation with Thompson's solver	257x257 matrices
OCEAN	simulates eddy currents in an ocean basin	98x98 grid
IS	integer bucket sort algorithm	2 million integers
CG	solves unstructured sparse linear system using the conjugate gradient method	5000x5000 sparse matrix with 451,002 nonzeros
EP	monte-carlo simulation	128K random numbers
MG	multigrid solver	34x34x34 matrix
MP3D	simulates rarified hypersonic flow	500 K mols., cylinder.geom (6144 cells), 5 steps
SPARSPAK	sparse Cholesky factorization	graph7 (7 sparse matrices, each with $>$ 5000 nonzeros)

of some of the matrices to help reduce these conflicts. These problematic matrices tend to have dimensions that are powers of two, which causes the cache size (also a power of two) to evenly divide into the size of a row or possibly the entire matrix. Therefore adjacent elements in the same column—and sometimes elements with similar access functions in adjacent matrices—often mapped into the same cache entry, thus resulting in large numbers of conflicts within inner loops. We manually fixed this problem by adding 13 (an arbitrary prime number) to the size of each dimension for these problematic matrices, while being careful that these changes affected only the data layout and not the actual computation. Later, in Section 5.2.2, we will examine these mapping conflicts in more detail, and will evaluate possible architectural enhancements to minimize their performance impact.

Table 3.2: General statistics for the uniprocessor applications. Primary data cache miss counts are for an 8 Kbyte direct-mapped cache.

Application	Instructions *(millions)*	Reads *(millions)*	Writes *(millions)*	Primary Data Cache Misses *(millions)*
MXM	220	116	11	4.3
CFFT2D	278	77	76	8.4
CHOLSKY	202	64	26	17.7
BTRIX	189	85	20	12.9
GMTRY	912	337	93	82.8
EMIT	114	52	11	1.4
VPENTA	155	64	17	13.4
TOMCATV	2638	989	287	86.9
OCEAN	78	28	8	1.7
IS	3236	669	355	68.6
CG	3411	952	407	30.3
EP	9761	1628	515	22.2
MG	113	55	12	1.4
MP3D	1851	268	104	10.5
SPARSPAK	204	28	23	3.3

3.1.3 Compiler Parameters

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, *prefetch latency* = 300 cycles, and *policy on unknown loop bounds* = assume a small number of iterations. The cache line size precisely matches the architecture, while the other parameters are more heuristic in nature. As discussed in Section 2.3.3, 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. The *prefetch latency* indicates to the compiler how many cycles in advance it should try to prefetch a reference (i.e. parameter l in equation(2.3)). The prefetch latency is larger than 75 cycles, the minimum miss-to-memory penalty, to account for bandwidth-related delays. For cases where loop bounds cannot be resolved at compile-time, we assume the number of iterations to be small, which tends to overestimate what remains in the cache. Later, in Section 3.3, we will consider the effects of varying these parameters.

3.1.4 Simulation Environment

To simulate the performance of our applications, we first use the SUIF compiler to generate fully-functional MIPS object code with prefetching. Since the MIPS-I instruction set does not contain a prefetch instruction, our compiler encodes prefetches as loads to `R0`.[2] This encoding is attractive for the purpose of simulation since it has the same addressing mode (base-plus-offset) and register usage (a single source register and no real target register) as an actual prefetch instruction, and therefore produces accurate instruction counts.

The performance of the resulting object code is simulated by using the MIPS *pixie* utility [74] to generate an instrumented version of the code, and then piping the resulting trace into our detailed cache simulator. Our simulator makes the simplifying assumption that all instructions execute in a single cycle and that all instructions hit in the primary instruction cache. Otherwise, the cache tag state and all forms of contention described in Section 3.1.1 (e.g., primary cache tags, memory bus) are modeled in detail.

3.2 Evaluation of Core Compiler Algorithm

In this section we evaluate the effectiveness of our core prefetching algorithm, which was described earlier in Chapter 2. We start with a brief, high-level evaluation of the overall performance of the compiler algorithm. We then focus in greater detail on each of the three key aspects of the compiler algorithm—locality analysis, loop splitting and software pipelining—in Sections 3.2.1, 3.2.2, and 3.2.3, respectively.

The results of our first set of experiments are shown in Figure 3.1 and Table 3.3. Figure 3.1 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.

[2] `R0` is a reserved register that returns the value 0 when used as a source operand. A load to `R0` is essentially a `NOP`, and therefore this code sequence is not normally generated by the compiler.

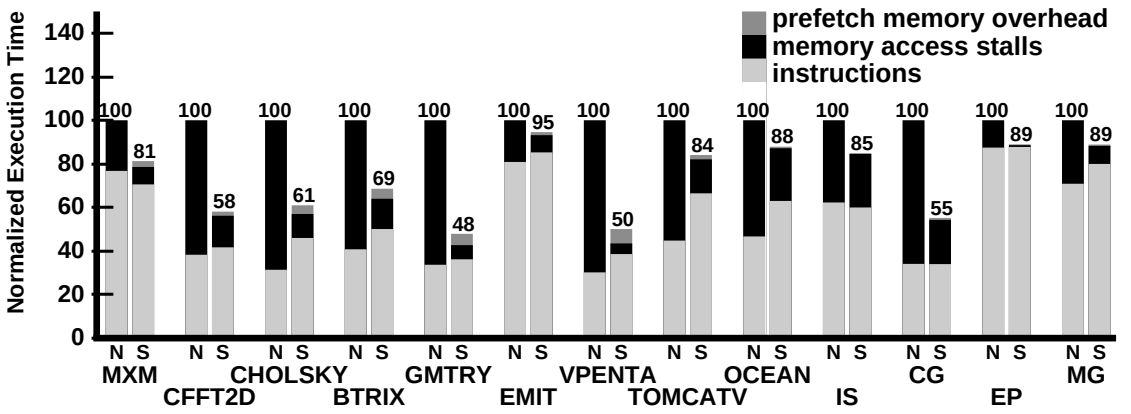


Figure 3.1: Overall performance of the selective prefetching algorithm (**N** = no prefetching, and **S** = selective prefetching).

Table 3.3: 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	6.70	28.5	4.34	28.9	34.4
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

As shown in Figure 3.1, the overall speedup 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 3.3 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, 33% to 90% of the original memory stall cycles are eliminated.

Having established the benefits of prefetching, we now focus on the costs. Figure 3.1 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.

These high-level results suggest that our prefetching algorithm is successful at improving performance. To evaluate the algorithm in greater depth, the following three subsections focus specifically on each key aspect of the selective prefetching algorithm. We begin with the first step in our algorithm, which is using *locality analysis* to determine which references should be prefetched.

3.2.1 Locality Analysis

The goal of using locality analysis is to reduce overhead by prefetching only those references that cause cache misses. To evaluate whether locality analysis is successful, we performed the following experiment. We modified our compiler to prefetch *all* instances of affine array references, which corresponds to setting the *prefetch predicate* for each reference to "*True*" (see Section 2.3.4). We refer to this algorithm as *indiscriminate* (as opposed to *selective*) prefetching. To isolate the impact of locality analysis, we kept all other components of the two algorithms the same. For example, 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.

Ideally the selective prefetch algorithm will achieve the same level of memory stall reduction as indiscriminate prefetching, while decreasing the overheads associated with issuing unnecessary prefetches. The results of this experiment are shown in Figure 3.2 and Table 3.4.

Figure 3.2 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%. As we see in both Figure 3.2 and Table 3.4, most of the benchmarks sacrifice very little in terms

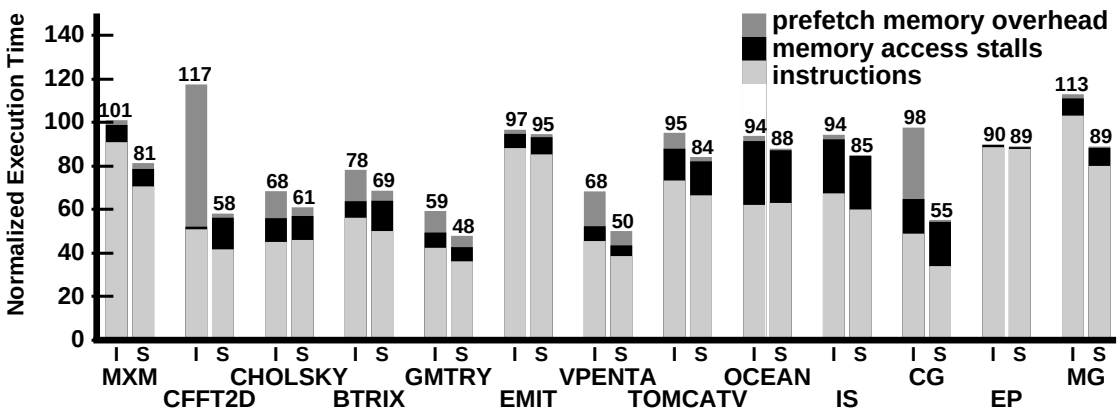


Figure 3.2: Overall performance comparison between the indiscriminate and selective prefetching algorithms (**I** = indiscriminate prefetching, and **S** = selective prefetching).

of memory stall reduction by prefetching selectively. On the other hand, Figure 3.2 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.

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.

Figure 3.3 shows the fraction of unnecessary prefetches and the coverage factor for the two prefetching algorithms. In general, we see the encouraging result that selective prefetching reduces the number of unnecessary prefetches without sacrificing much in terms of coverage factor. Let us consider these numbers in more detail.

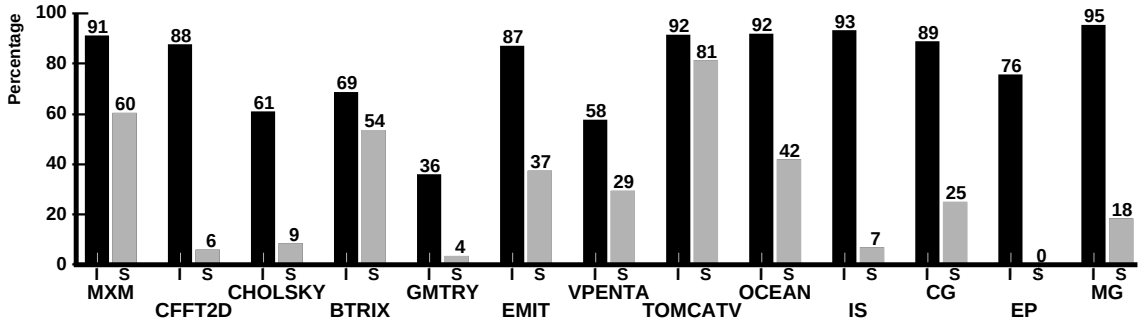
The coverage factor of the indiscriminate case is interesting, since it represents the fraction of cache misses that are within the domain of our analysis in most cases. Looking at Figure 3.3(b), we notice that in 5 of the 13 cases (CFFT2D, CHOLSKY, BTRIX, GMTRY, and VPENTA) the coverage factor is well over 90%, but in 5 of the other cases (TOMCATV, OCEAN, IS, CG, and MG) it is 50% or lower. In the case of CG, the coverage is only 38% because it is a sparse matrix algorithm and we are only prefetching the dense references. The same is true for IS (integer sort), which contains many indirect array references, despite not being a sparse matrix algorithm. (We will improve the coverage for both of these cases later in Section 3.5 when we

Table 3.4: Memory performance improvement for the indiscriminate and selective prefetching algorithms.

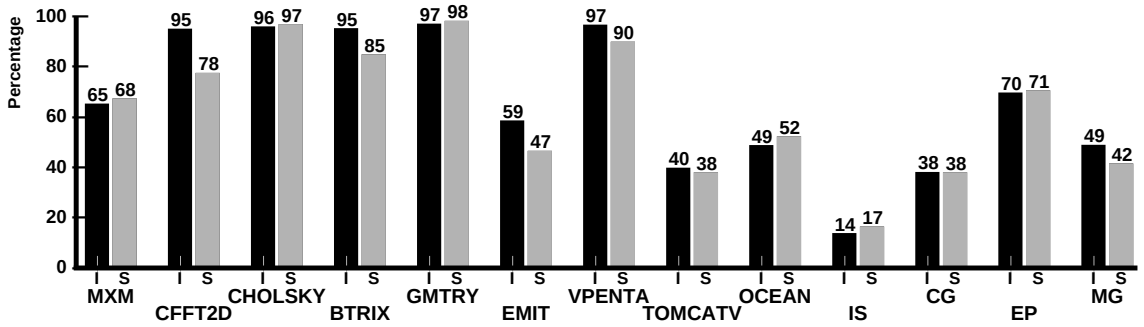
	Indiscriminate Prefetch			Selective Prefetch		
Benchmark	Miss Rate (%)	Average Miss Penalty (cycles)	Memory Stall Reduction (%)	Miss Rate (%)	Average Miss Penalty (cycles)	Memory Stall Reduction (%)
MXM	1.37	13.0	66.0	1.26	14.0	66.2
CFFT2D	0.34	16.6	98.1	1.29	53.4	76.5
CHOLSKY	6.27	12.2	84.1	6.09	12.3	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	4.21	29.3	34.1	4.34	28.9	34.4
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

also prefetch indirect references.) 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 (a likely scenario without locality analysis) 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 (which are not compiled by our prefetching compiler) rather than the programs themselves. Furthermore, library routines present a difficult challenge because the surrounding contexts of the call sites (e.g., surrounding loop nests) are not known in advance. Later, in Section 5.2.1, we will suggest how profiling feedback or dynamically-adaptive code may help in such cases.

Figure 3.3(a) 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



(a) Unnecessary Prefetches



(b) Coverage Factor

Figure 3.3: Statistics for evaluating locality analysis for the uniprocessor applications (**I** = indiscriminate prefetching, and **S** = selective prefetching). Note that the unnecessary prefetch percentages are computed with respect to the number of prefetches issued, which changes between the two cases.

memory subsystem (CFFT2D and CG).

A selective algorithm is successful if it can maintain a similar coverage while lowering the number of unnecessary prefetches. Figure 3.3(a) shows that in 11 of the 13 cases, the coverage is reduced by less than 10%. Table 3.4 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 Figure 3.3(a), we see that the improvement from selective prefetching is dramatic in many cases (CFFT2D, CHOLSKY, IS, EP, MG).

The advantage of selective prefetching is summarized by the ratio of indiscriminate to selective

Table 3.5: Ratio of prefetches issued under the indiscriminate and selective algorithms.

Benchmark	Indiscriminate to Selective PF Ratio
MXM	4.7
CFFT2D	9.4
CHOLSKY	2.3
BTRIX	1.7
GMTRY	1.5
EMIT	6.2
VPENTA	1.8
TOMCATV	2.3
OCEAN	6.6
IS	15.3
CG	6.7
EP	4.1
MG	20.9

prefetches in Table 3.5. 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.

Once we have used locality analysis to predict which dynamic data references should be prefetched, the next step in our algorithm is to isolate those cases through *loop splitting*.

3.2.2 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 3.4 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 [9].

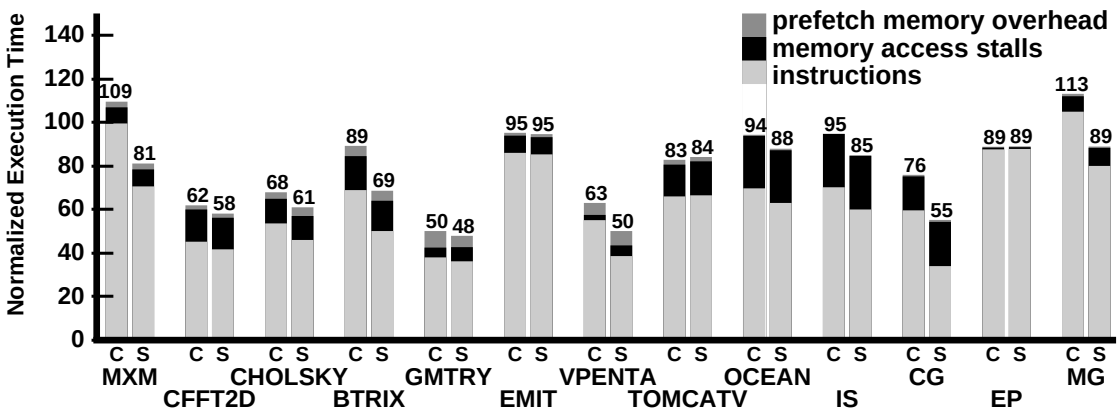


Figure 3.4: Loop splitting effectiveness (**N** = no prefetching, **C** = selective prefetching with conditional statements, and **S** = selective prefetching with loop splitting).

Table 3.6: Average instruction overhead per prefetch.

Benchmark	Indiscriminate Algorithm	Selective Algorithm	
		wrt Original Code	wrt Code After Loop Splitting
MXM	1.3	-2.7	3.9
CFFT2D	1.4	3.5	3.5
CHOLSKY	2.0	4.9	5.8
BTRIX	1.8	1.8	2.1
GMTRY	1.9	0.8	0.8
EMIT	1.6	5.8	5.8
VPENTA	2.5	2.5	2.9
TOMCATV	4.0	7.0	7.0
OCEAN	2.5	17.5	18.4
IS	1.9	-7.9	2.1
CG	2.5	-0.1	7.2
EP	1.8	2.4	2.4
MG	3.4	20.2	20.2

Table 3.6 shows the number of instructions required to issue each 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, we show overhead both with respect to the original code and with respect to code where the loop splitting transformations have been performed but no prefetch instructions are inserted. Loop splitting

generally increases the overhead per prefetch. In a few cases, the overhead with respect to the original code is actually negative, due to the savings through loop unrolling (MXM, IS and CG). In two cases (OCEAN and MG), the overhead has become quite large—more than 17 instructions per prefetch. 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—5 or fewer instructions with respect to the original code for 9 of the 13 benchmarks.

Finally, after predicting and isolating the dynamic miss instances, the last step in our algorithm is to schedule prefetches ahead of the references using *software pipelining*.

3.2.3 Software Pipelining

The goal of software pipelining is to issue prefetches the proper amount of time in advance, such that the data will be found in the cache when it is actually needed. The number of iterations to prefetch ahead must be carefully chosen (see equation (2.3)), since too few iterations will not provide enough time to hide the latency, but too many iterations may cause the data item to be replaced from the cache before it can be referenced.

To evaluate the effectiveness of our software pipelining algorithm, Figure 3.5 shows a breakdown of the impact of prefetching on the original primary cache misses. This breakdown contains three categories: (i) those that are prefetched and subsequently hit in the primary cache (*pf-hit*), (ii) those that are prefetched but remain primary misses (*pf-miss*), and (iii) those that are not prefetched (*nopf-miss*). The effectiveness of the software pipelining algorithm is reflected by the size of the *pf-miss* category. A large value 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 are issued too early, in which case the line has already been replaced in the cache before it is referenced.

The results in Figure 3.5 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 they can be referenced. To adjust for this, one might consider

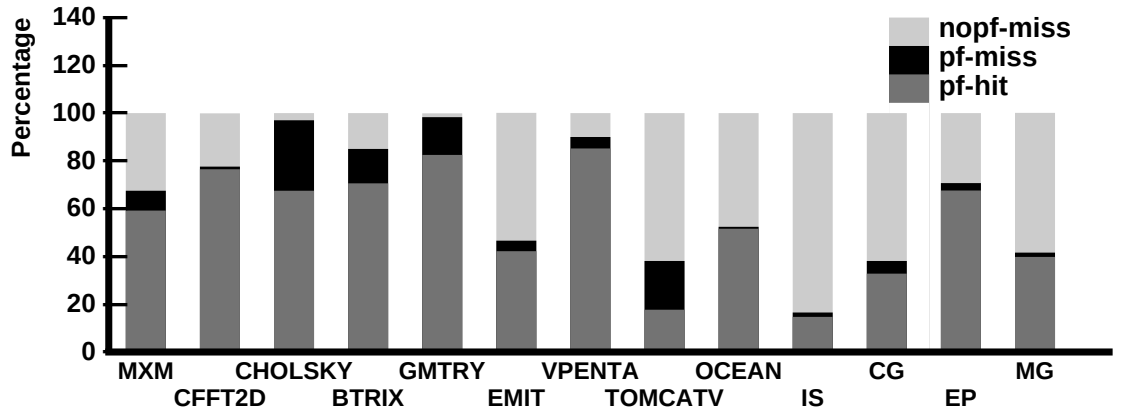


Figure 3.5: Breakdown of the impact of prefetching on the original primary cache misses for the uniprocessor applications.

decreasing the *prefetch latency* compile-time parameter (i.e. parameter l in equation (2.3)), which was set to 300 cycles for these experiments. We will evaluate this possibility later in Section 3.3. However, we observe that when cache conflicts are the problem, they often occur frequently enough that they cannot be avoided by simply adjusting the software pipelining parameters. Later, in Section 5.2.2, we examine these cases in more detail and evaluate whether increasing the cache associativity can help.

Even in cases where prefetched data is replaced from the primary cache before it can be referenced, there is still a performance advantage since the data tends to remain in the secondary cache. Therefore although the miss latency is not eliminated, it is often reduced from a main memory access to a secondary cache access. This was shown earlier in Table 3.3, where selective prefetching reduces the average miss penalty from 24.8 to 12.3 cycles for CHOLSKY, and from 36.6 to 12.5 cycles for TOMCATV.

3.2.4 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.

Now that we demonstrated the success of the algorithm, the next question is whether these speedups can only be achieved by carefully tuning the compile-time parameters that describe the

memory hierarchy (several of them being rather heuristic in nature), or whether the algorithm is fairly robust at achieving good performance. We address this question in the following section.

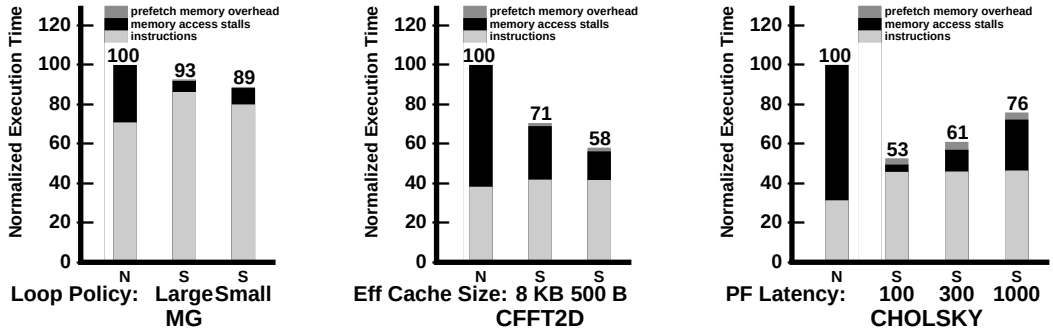
3.3 Sensitivity to Compile-Time Parameters

The selective prefetching algorithm uses several compile-time parameters to model the behavior of the memory subsystem. Specifically these parameters include: (i) cache line size, (ii) whether unknown loop bounds are assumed to be large or small, (iii) effective cache size, and (iv) prefetch latency. The most concrete of these parameters is the cache line size, which is fixed given a specific architecture, and can be set precisely. The other parameters, however, are more heuristic in nature. For example, with a direct-mapped cache, the effective cache size is set to some value less than the actual cache size to model the effects of conflicts, but there is no precise method for determining this value. Similarly, the prefetch latency is set to some value larger than the latency of a memory access to model bandwidth-related delays, but this value is also somewhat arbitrary. To evaluate the robustness of our algorithm, we measured the effects of varying these less obvious parameters.

3.3.1 Policy on Unknown Loop Bounds

The SUIF compiler performs interprocedural constant propagation to statically determine as many loop bounds as possible. The loop bounds are often needed to determine the volume of data accessed by each loop, which in turn is used to decide whether a loop is within the *localized iteration space* (see the algorithm in Figure 2.7). When the loop bounds remain unknown, the compiler may be faced with deciding whether an unknown volume of data fits within the cache. To resolve such cases, our algorithm uses one of two policies: either (i) unknown loop counts are assumed small, and hence the data would tend to fit in the cache; or (ii) unknown loop counts are assumed large, and therefore the data would tend not to fit. For our experiments in the previous section, we consistently used the former policy of assuming unknown loop counts to be small, which tends to overestimate what remains in the cache. We now compare this with the latter policy of assuming unknown loop counts to be large.

When the compiler assumes unknown iteration counts to be large rather than small, 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 3.6(a). In this case, the benefit of the



(a) Loop Policy (b) Effective Cache Size (c) Prefetch Latency

Figure 3.6: Sensitivity of results to compile-time parameters (**N** = no prefetching, **S** = selective prefetching variations).

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 or to generate adaptive code, as we will discuss later in Section 5.2.1. 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.

3.3.2 Effective Cache Size

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). Recall that the reason for this is to approximate the effects of cache conflicts in a direct-mapped cache. 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 3.6(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 3.6(b). (Note that this is in contrast with the effect we see in Figure 3.6(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.

In general, we observe that the volume of data accessed by a single iteration of a loop tends to fall into one of three categories: (i) a small constant amount (often less than 256 bytes), which is particularly common in inner loops; (ii) a very large constant amount (much larger than 8 Kbytes), which is common when the loop contains an inner loop with constant loop bounds; or (iii) an unknown amount. In all three cases, variations within the reasonable range of effective cache sizes have no effect, since in the first case the loop is definitely localized, in the second case the loop is definitely not localized, and in the third case only the policy on unknown loop bounds matters. Overall, the results appear to be robust with respect to effective cache size.

3.3.3 Prefetch Latency

Finally, for our experiments in Section 3.2, 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 value 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 3.6(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 3.6(c). However, in such cases the best approach may be to eliminate the cache conflicts that cause this behavior [49].

In general, we observe that it is better to be conservative with the prefetch latency parameter. Clearly if the value is not large enough to hide latency, it will always hurt performance. If we specify more latency than is actually experienced, it hurts performance *only if* data gets displaced. As caches become larger, this should become less and less of a problem. Besides which, only a relatively small number of new lines can be fetched into the cache in 300-500 cycles. If cache conflicts are a problem within this relatively small window of time, chances are that the conflicts will occur even if the prefetch latency is set to the smallest value that can hide the latency. These chronic cache conflicts must be dealt with in another way, as we will discuss later in Section 5.2.2.

3.3.4 Summary

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.

Now that we have studied our core prefetching algorithm, it is time to "pop up" a level conceptually, and examine its interaction with another important technique for coping with the latency of affine array references: *locality optimizations*.

3.4 Interaction with Locality Optimizations

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 main 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 [87]. In two of the cases (GMTRY and VPENTA), there was a significant improvement in locality, and thus performance. Both of these cases are presented in Figure 3.7. For each case, we show two sets of three performance bars—the three on the left are *without* locality optimizations, and the three on the right are *with* locality optimizations enabled. These latter three bars show locality optimization by itself (**N**) and in combination with the two prefetching schemes (**I** and **S**).

3.4.1 GMTRY: Cache Blocking

In the case of GMTRY, the locality optimizer is able to "block" the critical loop nest. In other words, rather than iterating over large matrices that are too large to fit in the cache, the code is restructured to iterate over smaller "blocks" within the matrices, such that each block does fit in the cache. As the data within in each block is reused many times before proceeding to the next block, these reuses will result in cache hits (i.e. locality) since the reused data can now be

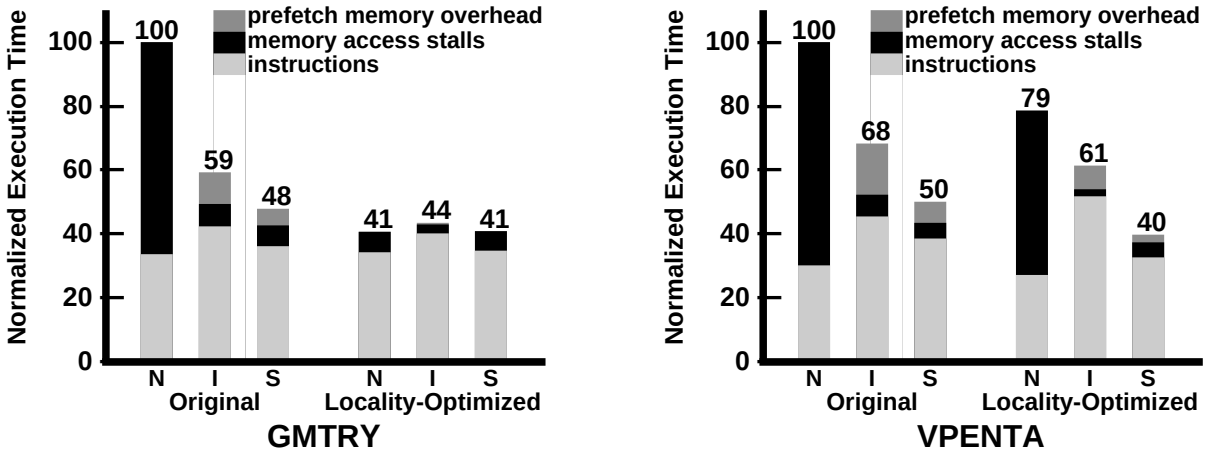


Figure 3.7: Results with locality optimization (**N** = no prefetching, **I** = indiscriminate prefetching, and **S** = selective prefetching).

retained by the cache.

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 of the prefetching 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 overhead by recognizing that blocking has occurred and thereby issuing fewer prefetches.

The best performance overall occurs with the blocking optimization alone. When blocking is combined with *indiscriminate* prefetching, the performance gets worse due to the instruction overhead of issuing large numbers of unnecessary prefetches. However, when blocking is combined with *selective* prefetching, the selective algorithm is clever enough to avoid these unnecessary prefetches and therefore does not hurt performance.

3.4.2 VPENTA: Loop Interchange

For VPENTA, the locality optimizer introduces spatial locality for every reference in the inner loop by interchanging two of the surrounding loops. In other words, rather than iterating along

the *columns* of the matrices, which results in misses on every iteration since the data are stored in row major order, the code has been restructured to iterate along the *rows* of the matrices instead. Therefore references will only miss when they cross cache line boundaries, which happens once every four iterations in this case.

With this locality optimization alone, the performance improves significantly. However, the selective prefetching scheme without this optimization 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 improves by only 11% while the selective prefetching scheme improves by 25%. The selective scheme improves more 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. However, the indiscriminate scheme does not realize that many of its prefetches are now unnecessary, and therefore continues to suffer from large instruction overhead.

The best overall performance, by a substantial margin, comes only through the combination of both locality optimization and selective prefetching.

3.4.3 Summary

It is interesting to compare prefetching with locality optimizations since both techniques rely on essentially the same *locality analysis* to predict cache misses. Despite this commonality, the two techniques differ in many ways. Locality optimizations *reduce* latency, unlike prefetching which *tolerates* latency; from this perspective, locality optimizations appear to be more desirable since they can also reduce bandwidth consumption. However, locality optimizations appear to be more limited in their applicability, as evidenced by the fact that they improved only two of our 13 benchmarks. Intuitively, this is because not only must the locality optimizing algorithm recognize a better way to structure the code (which is not always possible), it must also be *legal* to perform this restructuring. Prefetching, on the other hand, is never restricted by such correctness issues, and therefore appears to have wider applicability.

These 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 further increases performance by hiding any latency that could not be eliminated.

```
for (i = 0; i < n; i++)
  sum += A[index[i]];
```

Figure 3.8: Example of an indirect array reference.

3.5 Prefetching Indirect References

So far in this chapter, we have focused entirely on prefetching *affine* array references—i.e. where the access pattern is a linear function of the surrounding loop variables. These affine references were a good starting point for our compiler algorithm, since they are an important source of memory latency, and because their regular and predictable access patterns make them amenable to prefetching.

In this section, we extend our core algorithm to handle another important array access pattern: *indirect references*. Figure 3.8 shows an example of such a pattern, where the index of the `A` array is itself an array reference (`index[i]`). Indirect references commonly occur in scientific and engineering applications such as sparse-matrix algorithms (to look up the sparse element in a dense storage array), particle-in-windtunnel simulations (to look up the cell containing the particle), etc. We begin in Section 3.5.1 by describing the modifications to our prefetching algorithm that are necessary for handling indirect references. Then in Section 3.5.2, we present experimental results to evaluate the success of our extended algorithm.

3.5.1 Modifications to Compiler Algorithm

Recall from Chapter 2 that there are two phases to a prefetching compiler algorithm: (i) an *analysis* phase, where the compiler determines which references to prefetch, and (ii) a *scheduling* phase, where prefetches are inserted into the code. In this subsection, we describe how both phases are modified to prefetch not only the affine `index[i]` reference in Figure 3.8, but also the indirect `A` reference.

Analysis Phase

Interestingly enough, the locality analysis that we used to predict the caching behavior of affine references (see Section 2.3) cannot work for indirect references, since there is no way to predict at compile-time which data are being accessed. At one extreme, all of the index values may

be identical, and the reference would behave as though it had temporal locality. At the other extreme, each reference may point to a unique cache line, and the reference would behave as though it had no locality. Since we are unable to accurately predict data locality for this case, the two choices are to prefetch all the time or not at all (i.e. there is no such thing as "loop splitting" for this case). For these experiments, we decided to prefetch indirect references all the time. To improve this decision-making process further, profiling feedback or hardware miss counters may prove useful, as we will discuss later in Section 5.2.1.

Scheduling Phase

To schedule the prefetches for indirect references, a minor modification of the software pipelining algorithm is needed. Figure 3.9 shows how software pipelining would work for the example in Figure 3.8. The important thing to focus on is the steady-state code. Assuming that five iterations are needed to hide the memory latency, you can see that in the steady state, the index array is prefetched *ten* iterations ahead. Five iterations after that, the index is dereferenced to compute the indirect array address. This leaves another five iterations to hide the latency of fetching this reference. Therefore, no cache misses will occur. To generate this code, extra prolog and epilog loops are needed. Note that this same technique can be generalized to prefetch an arbitrary number of indirections ahead of time.

One complication with prefetching indirect references is that if the index array is modified within the loop, then the index value might not be valid at the time of prefetch, which can potentially result in a prefetch being issued for an illegal address (e.g., unallocated memory in the virtual address space of the process). With only a single level of indirection, as in Figure 3.9, this will not cause a problem since at worst only the prefetch address may be invalid, and prefetches are defined not to take memory exceptions. However, with two or more levels of indirection, not only may prefetch addresses be invalid, but *load* addresses during dereferences may be invalid as well. Unlike prefetches, load instructions *do* take memory exceptions on invalid addresses, which will prove either costly or fatal for the application.

For example, consider the loop in Figure 3.10(a), where the value of `index1[i]` is only set within the bounds of the `index2` array during the same iteration `A[index2[index1[i]]]` is referenced. Figure 3.10(b) shows the steady state loop for code that prefetches the `index1`, `index2`, and `A` arrays far enough in advance to hide memory latency (a straightforward derivation from the code in Figure 3.9(b)). However, since these prefetches are issued before `index1[i]` is valid, the prefetch of `&index2[index1[i+10]]` may be to an invalid address (which simply

(a) Original Loop

```
for (i = 0; i < 100; i++)
  sum += A[index[i]];
```

(b) Software Pipelined Loop

```
for (i = 0; i<5; i++)          /* Prolog 1 */
  prefetch(&index[i]);

for (i = 0; i<5; i++) {        /* Prolog 2 */
  prefetch(&index[i+5]);
  prefetch(&A[index[i]]);
}

for (i = 0; i<90; i++) {       /* Steady State */
  prefetch(&index[i+10]);       /* prefetch the index early enough... */
  prefetch(&A[index[i+5]]);     /* ... so it is ready for computing the A address */
  sum += A[index[i]];
}

for (i = 90; i<95; i++) {      /* Epilog 1 */
  prefetch(&A[index[i+5]]);
  sum += A[index[i]];
}

for (i = 95; i<100; i++)       /* Epilog 2 */
  sum += A[index[i]];
```

Figure 3.9: Example of how software pipelining is used to prefetch indirect references. For this example, assume that 5 iterations are enough to hide memory latency.

means that the prefetch will be dropped), and the subsequent load of `index2[index1[i+5]]` when computing `&A[index2[index1[i+5]]]` may also be invalid, which would result in a load exception.

To avoid suffering load exceptions with two or more levels of indirection, there are two choices. First, if the compiler can determine that there is no possibility that the index values will be modified within the loop nest, then it is safe to schedule the prefetches as normal. Factors that will make this difficult include memory aliasing and procedure calls. If the compiler is uncertain about whether the index values may be modified, then it must be conservative and prefetch no

(a) Original Loop

```
/* Initial values in the index1 array are outside the bounds of the index2 array. */
for (i = 0; i < 100; i++) {
  index1[i] = foo();   /* index1[i] only becomes valid during iteration i */
  sum += A[index2[index1[i]]];
}
```

(b) Software Pipelined Loop (Steady State Only)

```
for (i = 0; i<85; i++) {
  prefetch(&index1[i+15]);
  prefetch(&index2[index1[i+10]]);      /* prefetch address may be invalid */
  prefetch(&A[index2[index1[i+5]]]);    /* possible load exception */
  index1[i] = foo();
  sum += A[index2[index1[i]]];
}
```

Figure 3.10: Example of how prefetching multiple levels of indirection may result in invalid addresses and possibly a load exception. Assume that 5 iterations are sufficient to hide memory latency.

more than the first level of indirection (e.g., it is safe to prefetch `&index2[index1[i+10]]` but not `&A[index2[index1[i+5]]]` in Figure 3.10(b)). Second, rather than using normal load instructions when computing the indirect prefetch addresses, special *non-excepting* loads could be used instead. For example, rather than suffering a costly memory exception on an invalid address, a non-excepting load might simply return a value of zero. This way, although the prefetch addresses would still be incorrect, at least the only overhead would be wasted instructions, rather than costly (or potentially fatal) exceptions.

In our experiments, we avoid these memory exception problems by only scheduling prefetches for up to a single level of indirection. For array-based codes, a single level of indirection appears to be the most common case (and is the only case we observe in our benchmarks). With linked lists, however, one could imagine arbitrarily large numbers of indirections.

3.5.2 Experimental Results

In this subsection, we evaluate the performance of our algorithm on several applications that contain indirect array references. Two of these cases (CG and SPARSPAK) are sparse-matrix

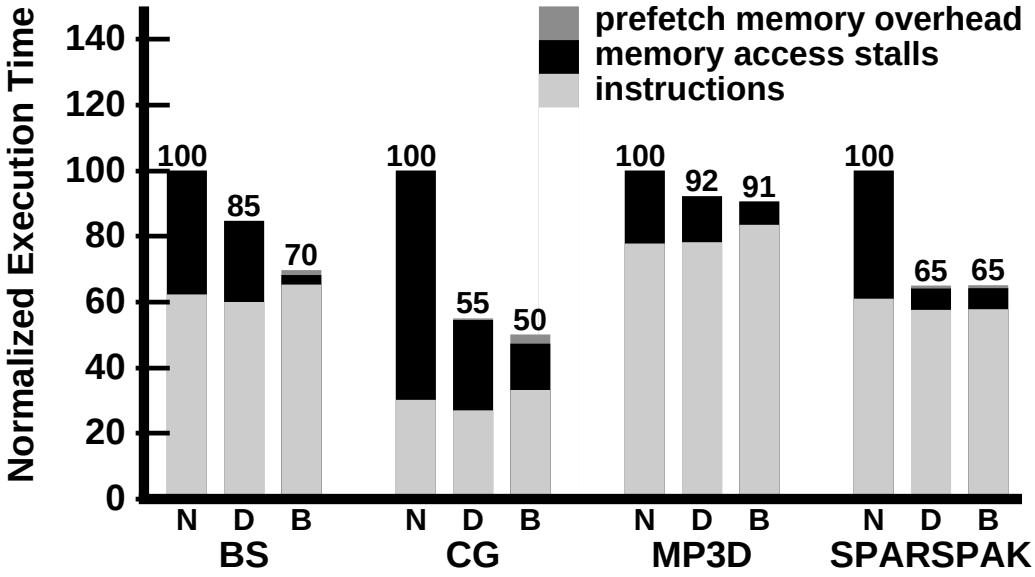


Figure 3.11: Prefetching indirect references in the uniprocessor applications (**N** = no prefetching, **D** = dense-only prefetching, and **B** = both dense and indirect prefetching).

codes, where the indirect references occur while mapping between the sparse matrix and the dense storage array. IS is an integer sort program that uses the "bucket sort" algorithm; in this case, the indirect references occur while computing the the number of occurrences of each integer in a dense histogram-style array. Finally, MP3D is a rarified-air particle simulator, where the indirect references occur when the *(x,y,z)* coordinates of a particle are used to look up the matching "space cell" in a 3-dimensional matrix.

Figure 3.11 shows the results of our experiment. For each application, we show three performance bars: (i) the original case without prefetching (**N**), (ii) the case where we only attempt to prefetch dense (i.e. affine) references (**D**), and (iii) the case where we prefetch both dense and indirect references (**B**).

We see in Figure 3.11 that prefetching significantly improves the performance of each application, with over two thirds of memory stall time eliminated, and speedups ranging from 10% to 100%. The interesting question, however, is how much of this gain came from prefetching indirect references. The answer to this varies across the applications. In the case of IS, roughly half of the benefit was from prefetching the indirect references. At the other extreme, SPARSPAK

Table 3.7: Average instruction overhead per prefetch of indirect reference.

Benchmark	Instruction Overhead
IS	6.3
CG	4.2
MP3D	25.0
SPARSPAK	4.0

saw no additional benefit from prefetching indirect references. This may seem a bit surprising, given that SPARSPAK is a sparse-matrix application. However, in SPARSPAK the elements of the sparse matrix are actually stored in a *dense matrix*, and the indirect references are to pointers into the appropriate rows or columns of the matrix. Therefore the data set size of the indirect references is relatively small, and tends to fit into the primary cache.

Another interesting issue with prefetching indirect references is instruction overhead. In Table 3.7, we isolate the average instruction overhead for prefetching only the indirect references. Notice that these numbers are generally larger than the numbers we saw earlier in Table 3.6. The reason for this is that prefetching an affine reference can take as little as a single instruction (the prefetch instruction itself), since the prefetch address can often be generated by simply changing the constant offset with respect to another load or store. However, computing the prefetch address for an indirect reference requires a minimum of four instructions: (1) a *load*, to load the index value; (2) a *shift*, to convert the index to the number of bytes in an array element; (3) an *add*, to add the byte offset to the starting address of the array; and (4) the *prefetch* instruction itself. In the case of SPARSPAK, our compiler achieved this bare minimum. For IS and CG, the numbers are also respectable. The case that stands out is MP3D, where generating an indirection prefetch address requires an average of 25 instructions. The reason for this is that MP3D must dereference *three* index arrays (x, y, and z) to compute each indirect reference, and in addition these index values must be converted from floating-point to integer values.

In theory, it may be possible to reduce the prefetching instruction overheads further by *saving* the indirection address in a register between the time it is computed for the prefetch and the time it is used for the load or store. However, there are two complications with this optimization. First, the compiler must be absolutely certain that the index value will not change during this interval. Otherwise, not only will the wrong location be prefetched (which only hurts performance), but more importantly the wrong location will be used by the load or store, and therefore the program

will produce incorrect results. Second, storing the address values between the prefetch and the load or store will consume additional registers, which may result in the large overheads of register spilling. To avoid both of these problems, we chose to simply re-compute the indirect addresses each time they are needed in our compiler implementation.

Overall, the gains in reduced memory stalls were large enough to outweigh the increased instruction count in all cases. Although the performance gains for MP3D are small in the uniprocessor case, we will see later in Section 4.3.3 that the gains are large in the multiprocessor case. The bottom line is that through a simple extension of the core algorithm, our compiler is now able to prefetch both affine and indirect array references, which covers a large fraction of array-based scientific and engineering applications.

3.6 Chapter Summary

Our study of compiler-based prefetching for array-based uniprocessor applications has produced the following results:

1. The selective prefetching algorithm presented in Chapter 2 is successful at hiding memory latency while minimizing prefetching overhead, thus improving overall performance by as much as twofold.

2. Our prefetching algorithm is robust with respect to the compile-time parameters that describe the memory hierarchy.

3. Prefetching and locality optimizations are complementary and therefore should be combined. Locality optimizations reduce the number of accesses to main memory, and prefetching tolerates the latency of the remaining misses.

4. Through a minor extension of our software pipelining algorithm, our compiler can automatically prefetch indirect array references.

Chapter 4

Prefetching for Multiprocessors

In this chapter, we turn our attention from uniprocessors to multiprocessors. Multiprocessor architectures are promising because they have the potential of achieving very high computation rates. However, these machines tend to suffer from memory latency even more than uniprocessors, since they have the additional complication of communicating shared data among processors. Also, the physical dimensions of large-scale multiprocessors result in large latencies to remote memory locations. Therefore latency-hiding techniques such as prefetching are essential to exploit the full potential of multiprocessors.

For this study, we focus on a class of machine with the following properties: (i) physically-distributed main memory, (ii) a single address space, and (iii) hardware-coherent caches. These properties are important for achieving high performance while providing ease of programmability for the following reasons. First, main memory is physically distributed across the machine so that each processor continues to have high bandwidth and low latency to its local memory, even as the size of the machine is scaled up. Second, a single address space simplifies the naming of shared objects, and provides an intuitive programming model for the user (the "shared memory" model). Finally, hardware-coherent caches help reduce memory latency by allowing shared writable data to be cached and replicated. Examples of such machines include Kendall Square Research's KSR1 [41], Stanford's DASH [54], and MIT's Alewife [3].

One of the keys to achieving high utilization of a multiprocessor is effectively overlapping communication and computation. Message-passing machines allow the programmer to do this by sending explicit non-blocking messages. However, under the shared-memory abstraction, communication occurs when a processor reads a memory location that has been written by another processor. Since processors typically stall on reads, this means that communication is

not overlapped with computation. Prefetching, however, provides a mechanism by which shared-memory applications can overlap communication (i.e. memory accesses) with computation.

This chapter is organized as follows. We begin in Section 4.1 by discussing the prefetching issues that are unique to multiprocessing, and how they affect our compiler algorithm. Next, we describe our framework for performing multiprocessor experiments in Section 4.2. Then in Section 4.3 we present the results of these experiments, including a detailed evaluation of each major component of the compiler algorithm. In Section 4.4, we vary the cache size to build additional confidence in our results, and to evaluate the effectiveness of the compiler algorithm in handling different mixtures of replacement misses and coherence misses. Section 4.5 compares compiler-inserted prefetching with programmer-inserted prefetching, both for cases where the compiler succeeded in inserting prefetches and for cases where it failed to do so. Finally, in Section 4.6 we summarize the major results of these multiprocessor experiments.

4.1 Multiprocessor Issues and Modifications to Compiler Algorithm

There are three new issues that arise when prefetching for multiprocessors. They are (i) whether *binding* or *non-binding* prefetches are used, (ii) prefetching the additional cache misses due to coherence activity, and (iii) using *exclusive-mode* prefetches to gain ownership of lines that are to be modified. In this section, we discuss each of these issues in detail, and describe how they affect our compiler algorithm.

4.1.1 Binding vs. Non-Binding Prefetches

The first issue we examine is the distinction between *binding* and *non-binding* prefetches. With a *binding* prefetch, the data value is "bound" at the time the prefetch is performed by placing it in either a buffer or a register such that that is the actual value that will be seen by a subsequent access. The problem with a binding prefetch is that if another processor modifies that location during the interval between when the prefetch is issued and the data value is used, the value delivered will be stale. This places significant restrictions on when binding prefetches can be issued.

With a *non-binding* prefetch, however, the data value is not bound until it is referenced by a subsequent load operation. This is implemented by placing the prefetched data in the hardware-coherent cache, rather than a register or separate buffer. Therefore the prefetched data will remain visible to the coherence mechanism, which ensures that later accesses will always see the correct

```
prefetch(&x);          /* prefetch shared variable outside critical section */
  ...
LOCK(&L);              /* enter critical section */
x = x + 1;             /* modify shared variable */
UNLOCK(&L);            /* exit critical section */
```

Figure 4.1: Example of when a binding prefetch would be illegal.

value (albeit at the potential expense of not finding the data in the cache if an invalidation occurs).

To illustrate the difference between binding and non-binding prefetches, consider the code fragment in Figure 4.1. Here a shared variable (x) is modified within a critical section. Since there obviously is not enough time within the critical section to hide the latency of prefetching x, we would like to move the prefetch of x *outside* the critical section to schedule it far enough in advance. However, with a *binding* prefetch, this would be illegal, since another processor may succeed in entering the critical section first, thus modifying x. If this occurred, then the value bound at the time of prefetch would be too small, since it would not reflect the more recent increment, and therefore the program would behave incorrectly. With a *non-binding* prefetch, however, this code sequence would be perfectly legal, since the correct value of x would always be observed inside the critical section.

Thus the key advantage of non-binding prefetching is that it frees the compiler from the burden of *preserving correctness*, and instead allows it to focus on the real issue, which is *improving performance*. Preserving correctness under binding prefetching is a difficult challenge for the compiler, since it requires a full understanding of the communication behavior and any explicit or implicit synchronization that occurs. As a result, compiler algorithms that insert binding prefetches spend most of their effort worrying about whether prefetching is *legal* [31], and are often forced to be conservative due to complications such as imperfect memory disambiguation and explicitly-parallelized code.

To make significant headway in prefetching for multiprocessors, we must move beyond the distractions of correctness and focus instead on the deeper performance issues. For example, even for the relatively simple code in Figure 4.1, it is not clear whether prefetching should be used. On the one hand, if there is significant contention for the lock (e.g., if x is the current bound in a branch-and-bound algorithm), then it is unlikely that the prefetched line will remain in the cache, since another processor is likely to enter the critical section first, thus invalidating the line when it is written. On the other hand, if there is little contention for the lock (e.g., if x

is a migratory object such as a circuit element in a logic simulator), then prefetching outside the critical section may result in large performance gains.

Since latencies in multiprocessors can easily be quite large (e.g., over 100 cycles in DASH), the compiler will need all of the flexibility it can get to move prefetches far enough in advance of the references. Therefore the advantages of non-binding prefetches are essential when compiling for multiprocessors.

How does non-binding prefetching affect our compiler algorithm? The good news is that it does not affect it at all. If we wanted to, we could use exactly the same prefetching algorithm we used for the uniprocessor code, ignoring the fact that we are compiling for a multiprocessor, and the code would still get the correct answer. Therefore our uniprocessor algorithm will serve as the starting point for our multiprocessor prefetching algorithm. In the following two subsections, we discuss the two changes that we do make to the algorithm to optimize it for multiprocessor performance.

4.1.2 Coherence Misses

Although non-binding prefetching allows the compiler to ignore the fact that it is compiling for a multiprocessor from a *correctness* standpoint, there are some *performance* reasons why it should take multiprocessing into account. In this subsection, we discuss the first of these reasons, which is that communication between processors can potentially increase the miss rate by causing more *coherence* misses (e.g., misses due to invalidations when using an invalidation-based cache coherence protocol).

As an example of how communication affects the miss rate, consider the example in Figure 4.2. In this example, two processors are both accessing location A, and both processors initially have copies of A in their caches in a "shared" state. *Processor 1* loads A twice. Assume that during the interval between these loads, no other locations are accessed by *Processor 1* that would interfere with A in the cache. If this access pattern occurred on a uniprocessor, it would be reasonable to expect the second load of A to hit in the cache, since A has not been replaced by other accesses since it was first loaded. However, in the multiprocessor scenario in Figure 4.2, *Processor 2* stores to location A during this interval, thus invalidating A from *Processor 1's* cache, and resulting in a cache miss the second time *Processor 1* loads A. Such coherence misses should be taken into account by the compiler during its analysis phase when it is predicting which references to prefetch.

Figure 4.2: Example of how coherence activity can cause cache misses.

Predicting Coherence Misses

Ideally, we would like to incorporate the notion of coherence misses into our framework of *locality analysis*, so that a single model could predict all forms of misses. How can this be accomplished? As you may recall from Section 2.3, data locality occurs whenever a cache line is reused, provided that the line has not been ejected from the cache since it was last used. In a uniprocessor architecture, a cache line will only be ejected from the cache if it is replaced by another reference that maps into the same cache entry. Our algorithm predicts that such replacements will occur whenever the volume of data accessed between reuses exceeds the effective size of the cache. In a multiprocessor architecture, a cache line can also be ejected from the cache if the line is *modified by another processor* during the interval between reuses, hence invalidating it from the given processor's cache (we assume an invalidation-based coherence protocol). Under locality analysis, the ejection of data from the cache is modeled through the *localized iteration space*, which is the set of loops that can carry data locality. Therefore, to predict coherence misses, we extend the concept of the localized iteration space to include not only the volume of data accessed, but also whether lines are being modified by other processors.

Accurately predicting when another processor modifies a given data line is a very difficult problem for the compiler, and requires a complete understanding of the communication and synchronization patterns of an application, as well the data addresses being accessed. In cases

where the application has been parallelized by the compiler, this may be feasible since presumably the compiler must understand the communication patterns very well to perform the parallelization. However, even in such cases, factors such as dynamic scheduling may make it difficult to predict exactly *when* the modifications occur relative to other processors, and *which* lines are being modified. In addition, while it may be tractable to understand when particular data items are shared among processors (i.e. *true sharing*), it is more difficult to predict the coherence misses that only occur because *separate* items fall within the same cache line (i.e. *false sharing*) [19, 81]. Finally, when the compiler is dealing with explicitly-parallelized programs, as is the case in our experiments[1], it is difficult (if not impossible) for the compiler to extract the communication patterns, since much of this semantic information is contained only in the programmer's head.

Although the compiler cannot precisely understand the communication patterns in the explicitly-parallelized codes used in our experiments, one thing that serves as a useful hint is the explicit *synchronization* statements inserted by the programmer to ensure that shared data are accessed safely. Assuming that a program is "properly labeled" [27] or "data-race-free" [2], synchronization statements should exist between the time when one processor modifies data and a second processor reads that data. Therefore, the compiler interprets explicit synchronization as a hint that data communication may be taking place. Ideally, it would be nice to know the data being protected by any given synchronization variable, since such information would allow the compiler to reason more precisely about which particular variables may have been modified. However, since such semantic information is kept only in the programmer's head, our approach is to conservatively assume the worst, which is that all shared objects may have been modified, and therefore no locality is carried across explicit synchronization. We incorporate this notion into our *localized iteration space* model by saying that a loop is not localized if it either (i) accesses too much data (to model replacement misses, as described earlier in Section 2.3.3), or (ii) contains explicit synchronization (to model coherence misses).

An Example

To make our approach more concrete, consider the example code in Figure 4.3. Matrix `A` is partitioned such that each processor modifies its own row based on a value (`myVal`) which is computed over the entire `A` matrix. This computation is repeated over several timesteps. Barriers

[1] We chose explicitly-parallelized applications because we wanted to use highly-optimized, "real" applications for our study, and therefore did not want to be constrained by the limitations of today's automatic parallelizing compiler technology. The explicitly-parallelized cases are also interesting because the compiler has the least amount of information to work with, which makes getting good performance more challenging for the prefetching algorithm.

```
/* NumProcs = total number of processors */
/* MyProcNum = this processor's ID number */

/* shared matrix A, where each processor owns a row */
shared double A[NumProcs][100];

for (t = 0; t < t_max; t++) {
  local double myVal = 0.0;
  for (p = 0; p < NumProcs; p++) {
    for (i = 0; i < 100; i++)
      /* compute myVal as a function of each element in A */
      myVal += foo(A[p][i],MyProcNum);
  }
  barrier(B,NumProcs);
  for (i = 0; i < 100; i++)
    A[MyProcNum][i] += myVal;     /* add myVal to this processor's row */
  barrier(B,NumProcs);
}
```

Figure 4.3: Example containing explicit synchronization.

are used to synchronize the processors between computing their copy of `myVal` and applying it to the row they own.

If the compiler did not take communication into account, then if `NumProcs` was a small enough constant that the entire `A` matrix fit in the cache, locality analysis would predict that the `A` matrix references would have temporal locality along the outer `t` loop. However, this is incorrect, because modifying the owned rows in the second inner loop will cause them to be invalidated from the other processor's caches, and therefore the temporal *reuse* of the entire `A` matrix in the first inner loop does not result in temporal *locality*.

Our compiler algorithm takes this communication into account by deciding that the `t` loop is outside the localized iteration space, since it contains a barrier statement, and therefore is not likely to carry locality. As a result, the compiler would decide to prefetch the entire `A` matrix each time it enters the first inner loop nesting (i.e. the `p` loop). Note that this is somewhat conservative since the row owned by the processor *does* remain in the cache since it is not modified by other processors. The compiler could take advantage of this if it had a better understanding of the actual communication patterns. However, given that the latency of coherence misses is likely to be large (since the data may be contained far away in another processor's cache), the instruction

overhead of conservatively issuing some unnecessary prefetches may be acceptable given the potential improvement in coverage of coherence misses. We could also improve upon this further through the use of profiling feedback information, as we will discuss later in Section 5.2.1.

4.1.3 Exclusive-Mode Prefetching

In this subsection, we discuss an issue that arises under invalidation-based coherence schemes (the model we assume for the remainder of this section), which is the use of *exclusive-mode prefetching*. Under the invalidation-based coherence model, a processor wishing to read a location receives a *sharable* copy of the line, which allows the line to be replicated in other caches as long as each processor is only reading the line. To write to a line, however, a processor must first acquire an *exclusive* copy of the line by *invalidating* the line from other processors' caches. This prevents the replicated copies from becoming stale, thus preserving coherence.

Just as normal memory accesses have two variations (*shared* accesses for reads, and *exclusive* accesses for writes), it also makes sense to have two types of prefetches: one that fetches a *shared* copy of a line, and one that fetches an *exclusive* copy. If a processor only intends to read a line, it will use the shared-mode prefetch. However, if the processor intends to modify the line—even if the line will be read first and modified shortly thereafter—an exclusive-mode prefetch should be issued to not only fetch a copy of the line, but also to gain ownership.

Proper use of exclusive-mode prefetching can provide two performance benefits. First, it can reduce the latency of the subsequent write since exclusive ownership of the line has already been obtained. This may or may not have a direct impact on execution time, depending on whether writes can be buffered.

The second benefit occurs in the common case where a value is read before it is written. Intuitively, these cases occur frequently because it is more common to *update* a shared variable (e.g., incrementing a shared counter, updating the position of a particle in a wind tunnel), than to simply overwrite it without reading it first. In such "read-modify-write" cases, what normally occurs is that the processor first requests a *sharable* copy of the line, and then immediately afterward requests an *exclusive* copy of the same line to perform the write. Rather than issuing two separate requests, a better approach is to issue a single *exclusive-mode prefetch*, as illustrated in Figure 4.4. Therefore exclusive-mode prefetches can potentially eliminate up to half of the total memory traffic, which can improve the performance of *all* references (both reads and writes) by reducing the amount of contention in the memory subsystem.

Figure 4.4: Illustration of how exclusive-mode prefetching improves performance.

We modify our compiler algorithm to exploit exclusive-mode prefetching as follows. After performing locality analysis, the references have been partitioned into *equivalence classes* (see Section 2.3.3), which are sets of references that can be treated as a single reference. An equivalence class may contain multiple references if they share group locality. We insert an exclusive-mode prefetch rather than a shared-mode prefetch for a given equivalence class if at least one member of the equivalence class is a write. For example, for the code in Figure 4.4(a), locality analysis would determine that both the read and write of `A[i]` are in the same equivalence class. Therefore, despite the fact that the *leading reference* to `A[i]` (i.e. the reference first accessing the data) is a read, our algorithm would schedule a single *exclusive-mode* prefetch of `A[i]`, thus achieving the desired effect illustrated in Figure 4.4(b).

4.1.4 Summary

We have seen how three issues unique to multiprocessing have affected our prefetching algorithm. First, by using *non-binding* rather than *binding* prefetches, our algorithm is not burdened with concerns over violating correctness, and consequently the original uniprocessor algorithm

serves as a viable starting point. Second, to account for the *coherence misses* that result from multiprocessor communication, we extended our analysis to use explicit synchronization as a hint that communication may be taking place. Finally, we exploit *exclusive-mode prefetching* whenever data are written, both to hide the latency of gaining ownership of a line, and to eliminate unnecessary bandwidth consumption in read-modify-write situations.

4.2 Experimental Framework

This section presents the architectural assumptions we make, the benchmark applications, and the simulation environment used to obtain performance results.

4.2.1 Architectural Assumptions

For this study, we have chosen an architecture that resembles the DASH multiprocessor [54], a large-scale cache-coherent machine that has been built at Stanford. Figure 4.5 shows the high-level organization of the simulated architecture. The architecture consists of several processing nodes connected through a low-latency scalable interconnection network. Physical memory is distributed among the nodes. Cache coherence is maintained using an invalidating, distributed directory-based protocol. For each memory block, the directory keeps track of remote nodes caching it. When a write occurs, point-to-point messages are sent to invalidate remote copies of the block. Acknowledgment messages are used to inform the originating processing node when an invalidation has been completed.

We use the actual parameters from the DASH prototype wherever possible, but have removed some of the limitations that were imposed on the DASH prototype due to design effort constraints. Figure 4.5 also shows the organization of the processor environment we assume for this study. Each node in the system contains a 33MHz MIPS R3000/R3010 processor connected to a 64 Kbyte write-through primary data cache. The write-through cache enables processors to do single-cycle write operations. The first-level data cache interfaces to a 256 Kbyte second-level write-back cache. The interface includes read and write buffers. The write buffer is 16 entries deep. Reads can bypass writes in the write buffer if the memory consistency model allows this. Both the first and second level caches are lockup-free [45], direct-mapped, and use 16 byte lines. The bus bandwidth of the node bus is 133 Mbytes/sec, and the peak network bandwidth is approximately 120 Mbytes/sec into and 120 Mbytes/sec out of each node.

The latency of a memory access in the simulated architecture depends on where in the memory

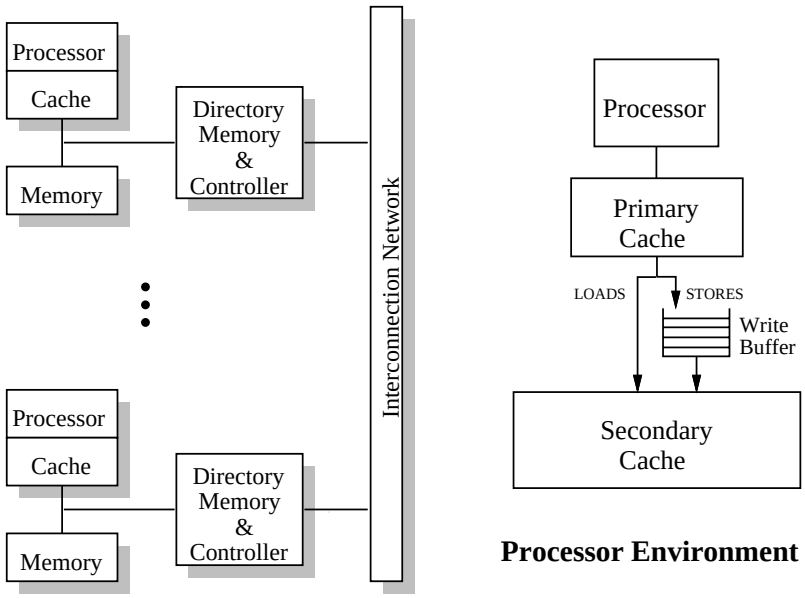


Figure 4.5: Architecture and processor environment.

hierarchy the access is serviced. Table 4.1 shows the latency for servicing an access at different levels of the hierarchy, in the absence of contention (the simulations done in this study do model contention, however). The following naming convention is used for describing the memory hierarchy. The *local node* is the node that contains the processor originating a given request, while the *home node* is the node that contains the main memory and directory for the given physical memory address. A *remote node* is any other node. The latency shown for writes is the time for retiring the request from the write buffer. This latency is the time for acquiring exclusive ownership of the line, which does not necessarily include the time for receiving acknowledgment messages from invalidations, since the release consistency model is used [27].

4.2.2 Applications

In this subsection we describe the computational structure of the parallel applications used in this chapter. This information is useful in later sections for understanding the performance results. The parallel applications we use consist of the entire SPLASH suite [72] plus the LU-decomposition application that we used in earlier studies [33, 61]. These applications are representative of

Table 4.1: Latency for various memory system operations in processor clock cycles (1 pclock = 30 ns).

Read Operations	
Hit in Primary Cache	1 pclock
Fill from Secondary Cache	15 pclock
Fill from Local Node	29 pclock
Fill from Remote Node	101 pclock
Fill from Dirty Remote, Remote Home	132 pclock
Write Operations	
Owned by Secondary Cache	4 pclock
Owned by Local Node	17 pclock
Owned in Remote Node	89 pclock
Owned in Dirty Remote, Remote Home	120 pclock

Table 4.2: Description of multiprocessor applications.

Application	Description	Input Data Set
OCEAN	simulates eddy currents in an ocean basin	98x98 grid
LU	dense LU decomposition with pivoting	200x200 matrix
MP3D	simulates rarified hypersonic flow	100 K mols., cylinder.geom (6144 cells), 5 steps
CHOLESKY	sparse Cholesky factorization	bcsstk15
LOCUS	routes wires for VLSI standard cell designs	Primary2 (25.8 K cells, 3817 wires)
WATER	simulates water molecule interaction	512 mols., 2 steps
PTHOR	simulates a digital circuit at the gate level	RISC (5 K elements), 5 steps
BARNES	performs a hierarchical N-body gravitation simulation	8192 bodies, 3 steps

algorithms used today in scientific and engineering computing environments. One application (OCEAN) is written in Fortran, and the others are written in C. The Argonne National Laboratory macro package [57] is used to provide synchronization and sharing primitives. Table 4.2 provides a brief summary of the applications, along with their input data sets, and Table 4.3 shows some general characteristics of the applications when 16 processors are used (as is the case throughout this chapter).

OCEAN [71] simulates the role of eddy and boundary currents in influencing large-scale ocean movements. It uses Successive Over Relaxation (SOR) to solve two-dimensional grids

Table 4.3: General statistics for the multiprocessor applications.

Application	Instructions Executed *(millions)*	Shared Reads *(millions)*	Shared Writes *(millions)*	Locks	Barriers	Shared Data Size *(KBytes)*
OCEAN	121	12.3	4.3	352	2400	2972
LU	50	5.5	2.7	3,184	29	640
MP3D	209	17.0	5.5	2,176	384	4028
CHOLESKY	1247	181.8	24.0	90,229	16	6856
LOCUS	897	117.3	12.8	46,239	16	5156
WATER	2134	146.4	48.9	302,272	208	484
PTHOR	76	12.9	1.3	98,514	3984	2760
BARNES	337	34.1	10.8	16,530	96	1528

of partial differential equations over a number of time-steps. These time-steps continue until the eddies and mean ocean flow attain a mutual balance. The principal data structures are 25 two-dimensional matrices containing the various values associated with the model's equations. Equal numbers of adjacent columns are statically assigned to each processor. During each time step, the processors iterate over their columns from left to right, performing nearest-neighbor computations with both 5-point and 9-point stencils. Communication occurs when processing the boundary columns since their nearest-neighbors include columns owned by other processors. For our experiments we ran OCEAN with 98x98 grids.

LU performs LU-decomposition for dense matrices. The primary data structure in LU is the matrix being decomposed. Working from left to right, a column is used to modify all columns to its right. Once all columns to the left of a column have modified that column, it can be used to modify the remaining columns. Columns are statically assigned to the processors in an interleaved fashion. Each processor waits until a column has been produced, and then that column is used to modify all columns that the processor owns. Once a processor completes a column, it releases any processors waiting for that column. For our experiments we performed LU-decomposition on a 200x200 matrix.

MP3D [59] is a 3-dimensional particle simulator. It is used to study the pressure and temperature profiles created as an object flies at high speed through the upper atmosphere. The primary data objects in MP3D are the particles (representing the air molecules), and the space cells (representing the physical space, the boundary conditions, and the flying object). The overall computation of MP3D consists of evaluating the positions and velocities of particles over a

sequence of time steps. During each time step, the particles are picked up one at a time and moved according to their velocity vectors. If two particles come close to each other, they may undergo a collision based on a probabilistic model. Collisions with the object and the boundaries are also modeled. The simulator is well suited to parallelization because each particle can be treated independently at each time step. The program is parallelized by statically dividing the particles equally among the processors. The main synchronization consists of barriers between each time step. For our experiments we ran MP3D with 100,000 particles, a 32x6x32 space array, and simulated 5 time steps.

CHOLESKY [69] performs sparse Cholesky factorization using a dynamic version of the supernodal fan-out method. The matrix is divided into supernodes (sets of columns with identical non-zero structures), which are further divided into conveniently-sized chunks called panels. A panel receives updates from other panels to its left, and when all updates have been received, the panel is placed on a task queue. The processors remove panels from this task queue and perform all of their associated modifications, which in turn causes other panels to be placed on the task queue. The principal data structure is the sparse matrix itself, which is stored in a compressed format similar to that of SPARSPAK [25]. The primary operation that is performed repeatedly is adding a multiple of one column to another column. Contention occurs for the task queue and the modified columns, which are protected by locks. For our experiments we ran `bcsstk15` which is a 3948-by-3948 matrix with 56,934 non-zeroes in the matrix and 647,274 non-zeroes in the factor.

LOCUS (our abbreviation for "LocusRoute") [68] is a high-quality global router for VLSI standard cells that has been used to design real integrated circuits. The parallelism in LOCUS comes from routing multiple wires concurrently. Each processor continuously picks up a new wire from the task queue, explores alternative routes, and places the wire along the best route. The principal data structure is a grid of cells called the *cost array*, which is used to record the presence of wires and therefore guides the placement of new wires. The cost array is not protected by locks, although it is accessed and updated concurrently by several processors, since the resulting distortions are considered acceptable. Contention for the task queue is not a problem since each routing task is fairly large-grain. For our experiments we run the largest circuit provided with the application, `Primary2.grin`, which contains 3817 wires and a 1290-by-20 cost array.

WATER is adapted from the Perfect Club Benchmarks [20] and performs N-body molecular dynamics simulation of the forces and potentials in a system of water molecules to predict some physical properties of water in a liquid state. The primary data structure is a large array of

records which contains the state of each molecule. Molecules are statically allocated to the processors. During each time step, the processors calculate the interaction of the atoms within each molecule, and of the molecules with each other. Due to symmetry, each processor only computes the interaction between a molecule it owns and half the other molecules. We run WATER with 512 molecules through 2 time steps.

PTHOR [76] is a parallel logic simulator based on the Chandy-Misra simulation algorithm. Unlike centralized-time algorithms, this algorithm does not rely on a single global time during simulation. The primary data structures associated with the simulator are the logic elements (e.g. AND-gates, flip-flops), the nets (wires linking the elements), and the task queues which contain activated elements. Each processor executes the following loop. It removes an activated element from one of its task queues and determines the changes on that element's outputs. It then looks up the net data structure to determine which elements are affected by the output change and schedules the newly activated elements on to task queues. For our experiments we simulated several clock cycles for a small RISC processor consisting of about 5000 elements.

BARNES (the full name is "Barnes-Hut") is a hierarchical N-body gravitational simulation where each body is modeled as a point mass and exerts forces on all other bodies in the system. To speed up the inter-body force calculations, a set of bodies that is sufficiently far away is abstracted as a simple point mass. To facilitate this clustering, physical space is divided recursively to form an *octree*, until each cell contains at most one body. This tree structure must be traversed on each time step to account for the movement of bodies. The principal data structure is the octree, which is implemented as an array of bodies and an array of cells that are linked together to form a tree. Bodies are statically assigned to processors for the duration of a time-step. During each time-step, each processor calculates the forces of all bodies on its subset of bodies, and the bodies are then moved according to those forces. Finally, the tree is regenerated for the next time step. A set of distributed locks provides exclusive access to the tree when needed. For our experiments, we run BARNES with 8192 bodies through 3 time steps.

4.2.3 Simulation Environment

As in the uniprocessor experiments, the SUIF compiler is used to generate fully-functional object code with prefetching. The performance of this object code is simulated using an event-driven simulator which models the architecture at the behavioral level. For example, the caches and the coherence protocol, contention, and arbitration for buses are all modeled in detail. The simulations are based on a 16 processor configuration. The architecture simulator is tightly coupled to the

Tango-Lite reference generator (the threads-based successor to the process-based Tango reference generator) [29] to assure a correct interleaving of accesses. For example, a process doing a read operation is blocked until that read completes, where the latency of the read is determined by the architecture simulator. Operating system references are not modeled. Unless specific directives are given by an application, main memory is distributed uniformly across all nodes using a round-robin page allocation scheme.

We now arrive at a difficult methodological problem that occurs when simulating large multiprocessors. Given that detailed simulators are enormously slower than the real machines being simulated, one can only afford to simulate much smaller problems/applications than those that would be run on the real machine. However, running small problems on a full-sized machine may result in unrealistic caching behavior, since, for example, the entire working set may fit in the cache. Therefore the question is how to scale the machine parameters so as to get realistic performance estimates.

A thorough examination of this question has been presented by Weber [84]. Weber uses variational analysis (i.e. observing the effects of varying cache size and problem size parameters on performance) and application-specific knowledge to choose appropriate cache sizes given "smaller-than-real" problem sizes for the SPLASH applications. This analysis provides the basis for our own decisions on how to scale the caches. Taking various factors into account regarding the differences between the two studies (e.g., 16 vs. 64 processors, two-level vs. single-level cache hierarchies, and the fact that we generally run larger problem sizes), we scale down the original DASH cache hierarchy from a 64 Kbyte primary and 256 Kbyte secondary cache to an 8 Kbyte primary and 64 Kbyte secondary cache for seven of the eight applications. The exception is PTHOR, where given the small problem size, we use a 2 Kbyte primary cache and 4 Kbyte secondary cache. (However, since prefetches can only be added to PTHOR by hand, performance results are only presented for PTHOR in Section 4.5.) To evaluate the impact of the scaled caches, we present results where the cache size is varied in Section 4.4.

4.3 Experimental Results

In this section, we present experimental results to evaluate the performance of our multiprocessor prefetching algorithm. We begin with a high-level overview of the performance improvements. We then focus in greater detail on several key aspects of the algorithm—locality analysis, scheduling prefetching, prefetching indirect references, and using exclusive-mode prefetches—in

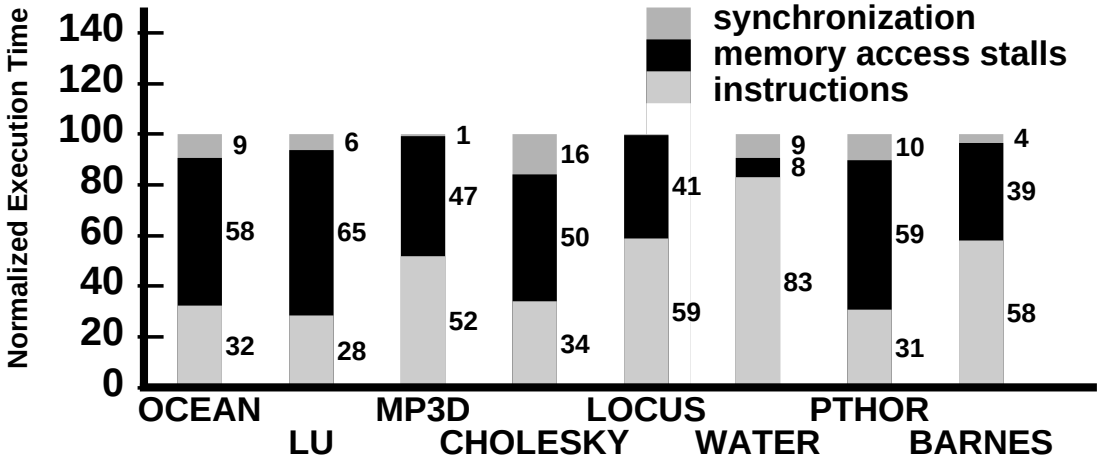


Figure 4.6: Performance of multiprocessor applications without prefetching.

Sections 4.3.1, 4.3.2, 4.3.3, and 4.3.4, respectively. We looked at several of these same aspects of the algorithm earlier in Chapter 3, but only within the context of a uniprocessor environment; many of these issues change or deserve a closer look under a multiprocessor environment. We start now with a brief performance overview.

The performance of the original code (i.e. *without* prefetching) for the eight multiprocessor applications is shown in Figure 4.6. Notice that there is a new component of execution time: time spent stalled for *synchronization*. These synchronization operations include events such as acquiring locks, waiting for other processors to reach barriers (i.e. poor load balancing), and instructions spent spinning on empty task queues. The prefetching compiler algorithm was applicable to five of the eight multiprocessor applications (OCEAN, LU, MP3D, CHOLESKY, and LOCUS). We examine those five applications in this section, and will discuss the other three applications (WATER, PTHOR, and BARNES) later in Section 4.5. The overall performance of the five cases that did improve is shown in Figure 4.7.

The speedups of the applications in Figure 4.7 range from 6% to 113%, with three of the five speeding up by 45% or more. The memory stall times have been reduced by 31% to 88% through a combination of lower miss rates and lower average miss penalties, as shown in Table 4.4. In two cases (OCEAN and LU), more than half of the synchronization latency was eliminated. This was due to improved load balancing, since prefetching helps to reduce the variability between task sizes. However, prefetching did not reduce the synchronization latency of CHOLESKY,

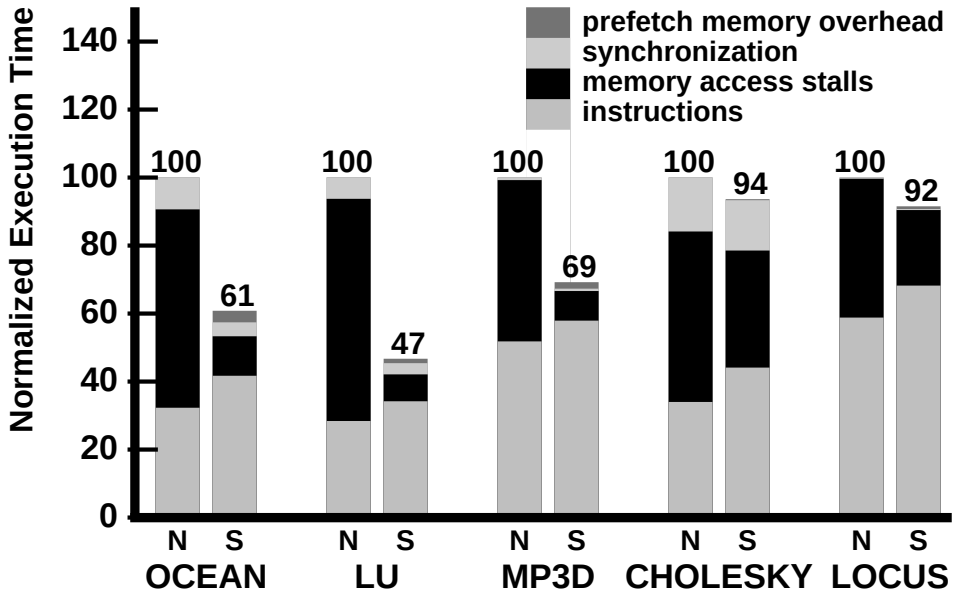


Figure 4.7: Overall performance of the selective prefetching algorithm for the multiprocessor applications (**N** = no prefetching, and **S** = selective prefetching).

where most of the synchronization time is spent in explicit spin loops waiting for more tasks to be produced. Figure 4.7 also shows that the overheads of prefetching (both instruction and memory stall overheads) are relatively small, therefore translating much of the memory stall reduction into actual performance benefit. In the following subsections, we examine each aspect of the multiprocessor prefetching algorithm in greater detail.

4.3.1 Locality Analysis

As in the uniprocessor prefetching algorithm, the goal of locality analysis is to determine which references will hit in the cache so as to reduce prefetching overhead. Locality analysis is slightly different for multiprocessors because it takes explicit synchronization into account when computing the localized iteration space, as we described earlier in Section 4.1.2. To evaluate the effectiveness of locality analysis, we perform an experiment similar to that in Section 3.2.1 by comparing *indiscriminate* prefetching (where array references are prefetched all the time) with *selective* prefetching (which uses locality analysis). The results of this experiment are shown in Figures 4.8 and 4.9.

Table 4.4: Reduction in memory stall times for the multiprocessor applications.

	No Prefetch			Selective Prefetch		
Benchmark	Shared Loads per Inst	Load Miss Rate (%)	Average Load Miss Penalty (cycles)	Load Miss Rate (%)	Average Load Miss Penalty (cycles)	Memory Stall Reduction (%)
OCEAN	0.10	23.11	76.2	9.11	28.2	85.4
LU	0.11	27.07	77.3	13.52	18.8	87.9
MP3D	0.08	12.49	90.1	2.32	83.0	81.7
CHOLESKY	0.15	26.38	29.6	24.66	22.3	31.2
LOCUS	0.13	13.54	39.2	10.97	26.1	45.7

Figure 4.8 shows that prefetching selectively rather than indiscriminately offers speedups ranging from 23% to 39%. As we saw earlier in the uniprocessor results, the reduction in memory stall time is comparable between the two schemes, but the real difference is that selective prefetching has significantly less overhead than indiscriminate prefetching. In two cases (CHOLESKY and LOCUS), prefetching selectively makes the difference between a performance loss and a performance gain.

Figure 4.9 presents two useful metrics for evaluating locality analysis: the percentage of *unnecessary prefetches*, and the *coverage factor*. Comparing both of these metrics between the two schemes, we see that once again selective prefetching eliminates a significant fraction of unnecessary prefetches, while giving up very little in terms of coverage factor. The reduction in total prefetches ranges from a factor of 2.2 to a factor of 7.7.

Looking at the magnitude of unnecessary prefetches remaining after selective prefetching in Figure 4.9(a), we see that three applications are below 25% (OCEAN, MP3D, and CHOLESKY), while the other two are above 45% (LU and LOCUS). The problem in both of these latter cases is that they contain references to important data structures which tend to fit in the cache, but which are occassionally modified by other processors, thus resulting in coherence misses. While it is difficult to predict which dynamic references to these structures will suffer misses, it is even more difficult to isolate those miss instances—i.e. techniques such as loop splitting are not applicable in these cases. Since our algorithm is conservative about coherence misses, it prefetches these references all the time, thus resulting in a relatively large fraction of prefetches being unnecessary. We now describe these two cases in more detail.

In LU, the important data structure is the pivot column. A pivot column is produced by

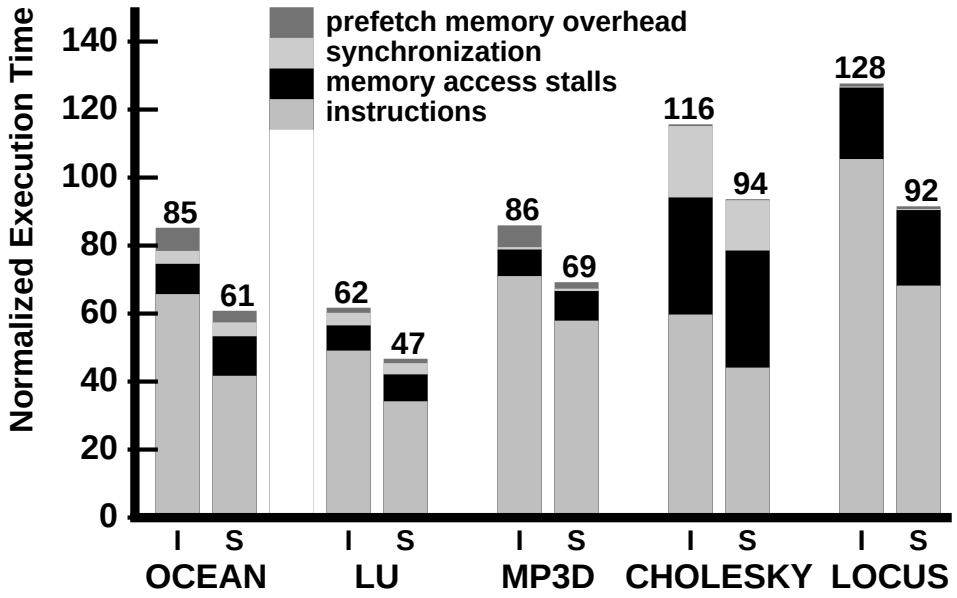
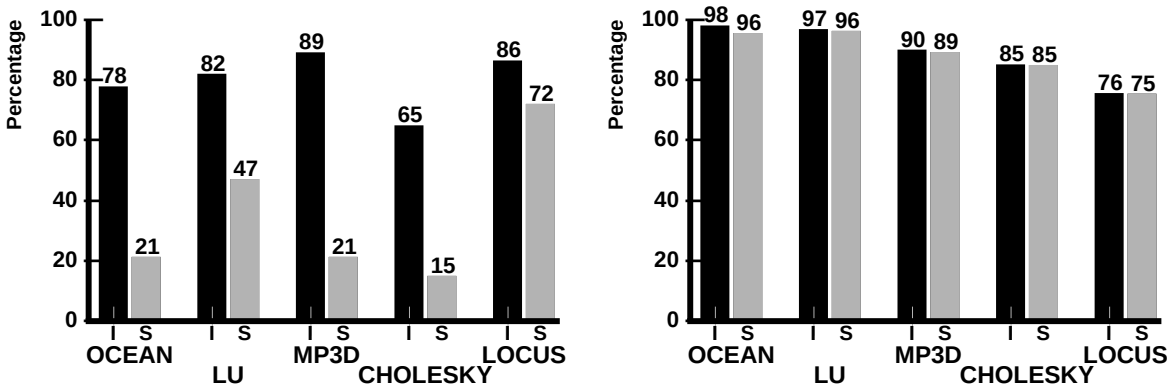


Figure 4.8: Overall performance comparison between the indiscriminate and selective prefetching algorithms for the multiprocessor applications (**I** = indiscriminate prefetching, and **S** = selective prefetching).

a single processor, and is then used repeatedly by the other processors to modify their local columns. This process of applying a pivot column to a local column occurs inside a procedure, which is called once for each local column. The first time a given pivot column is applied to a local column, the pivot column will miss in the cache, since it still resides in the cache of the processor that produced the pivot column. However, on subsequent calls of the same procedure with the same pivot column, the column will hit since it is retained in the cache. Since our prefetching algorithm does not look across procedure boundaries, it cannot recognize the temporal locality of the pivot column, and therefore prefetches it each time the procedure is called. Overcoming this limitation statically would require not only interprocedural analysis, but also the ability to either *inline* or *clone* [15] the procedure body so that prefetches can be scheduled for only the first use of a pivot column. Another possibility is to use *dynamic information* in the form of hardware miss counters, thus allowing the procedure to adapt dynamically to whether the pivot column is in the cache. As we will see later in Section 5.2.1, this latter method is effective at eliminating unnecessary prefetches in LU. However, even without any of these improvements, we observe



(a) Unnecessary Prefetches (b) Coverage Factor

Figure 4.9: Statistics for evaluating locality analysis for multiprocessor applications (**I** = indiscriminate prefetching, and **S** = selective prefetching).

that despite the unnecessary prefetching overhead in LU, it achieves the greatest speedup of all the applications (more than twofold) since the advantage of prefetching the pivot columns when they do miss is quite large.

In the case of LOCUS, the important data structure is the *cost array*, which represents the current placement of wires. Each processor's portion of the cost array tends to fit in its cache, but parts of it are occassionally invalidated as other processors rip up or put down new wires. Since these modifications occur erratically, the compiler cannot predict when they will occur and simply prefetches the cost array each time it is traversed. Therefore a significant fraction of the prefetches are unnecessary, but this is the only way to cover these coherence misses, and it does account for the bulk of the reduction in memory stalls.

LOCUS has the distinction of not only having the highest fraction of unnecessary prefetches, but also of having the lowest coverage factor (75%), as we see in Figure 4.9(b). The reason why the coverage factor is not higher (which is also true for CHOLESKY) is that many of the important array references are inside *while* loops rather than *for* loops. Since a *while* loop has no induction variable (either explicit or implicit), neither locality analysis nor software pipelining is applicable.

Finally, we observe that locality analysis is successful at reducing unnecessary prefetches while maintaining a respectable coverage factor despite our conservative assumption that shared data does not remain in the cache across synchronization statements. In LU and LOCUS, the remaining unnecessary prefetches were *not* caused by adjusting the localized iteration space due

to explicit synchronization (as described in Section 4.1.2). The problem in LU is procedure boundaries, and there is no synchronization in the loops for LOCUS, since the cost array is not protected by locks. In fact, when we recompiled each application and did *not* take explicit synchronization into account, we got precisely the same performance for each application. In other words, explicit synchronization never changed our locality predictions in significant ways. This occurred for two reasons. First, programmers realize that synchronization is costly and therefore avoid putting it in inner loops. Instead, synchronization tends to happen in outer loops and between loop nests, which makes it less likely to affect loop-level data locality. Second, in some cases the programmers decide that the performance advantage of not locking data structures is worth whatever distortion it causes in the result, as is the case in both MP3D and LOCUS. Therefore although communication takes place, there is no explicit synchronization for our algorithm to use as a hint. However, even if the compiler did realize that coherence misses were occurring in these cases, the only choice it has is to prefetch all the time, since the misses occur erratically. Therefore the desired effect was achieved in our experiments despite the lack of sophisticated analysis of communication patterns.

Overall, prefetching selectively through the use of locality analysis appears to be quite successful both for uniprocessors and multiprocessors. Once locality analysis has predicted which references to prefetch, the next step in our algorithm is *scheduling* those prefetches so that they are effective at hiding latency.

4.3.2 Scheduling Algorithm

As in the uniprocessor study, the effectiveness of the scheduling algorithm can be evaluated based on how often prefetched misses result in primary cache hits. Figure 4.10 breaks down what happened to the original primary cache misses, similar to Figure 3.5 in the uniprocessor study. However, since there are more explanations for ineffective prefetches in the multiprocessor architecture, we have broken the misses down into several more specific categories.

The topmost section of the bars in Figure 4.10 (labeled *nopf-miss*) is still the cases that are not prefetched and obviously remain cache misses. These are the misses that are not part of the coverage factor, which was discussed in the previous subsection. The bottom section of the bars (labeled *pf-hit*) includes the original misses that are prefetched and subsequently become primary hits, which corresponds to effective scheduling. The remaining four cases in the middle of the bars (labeled *pf-miss*) are the ones where prefetches are not effective. We will discuss each of these cases in detail.

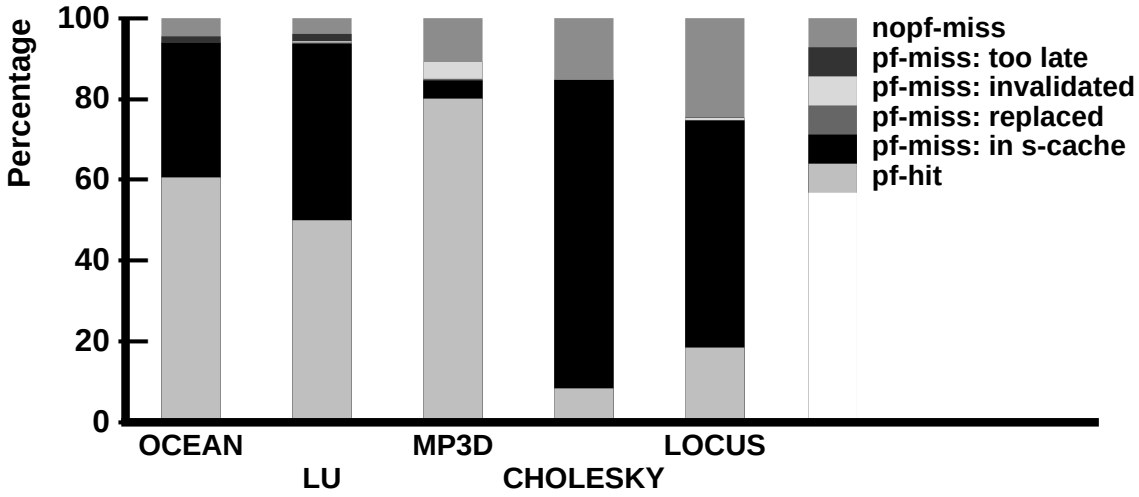


Figure 4.10: Breakdown of the impact of prefetching on the original primary cache misses for the multiprocessor applications.

The "*pf-miss: too late*" category includes cases where the prefetches were not issued early enough to hide the latency, and therefore the data item had not returned to the cache before it was referenced. This case was only noticeable in one application: LU. The problem in LU is that prefetches are delayed due to "hot-spotting" in the network. The hot-spotting occurs because several processors are often waiting for the same column to be produced by a single processor. Once that processor signals that the column is ready, the other processors simultaneously send numerous prefetches to that processor, requesting the data. The prefetches subsequently exceed the bandwidth capabilities of that particular network node, resulting in queueing delays. This problem is *not* remedied by simply scheduling the software pipeline to hide a larger latency, since the prefetches cannot be issued before the synchronization point, and after that point the request rate simply exceeds the bandwidth capacity of the network node. In fact, when LU was recompiled for twice the normal target memory latency (i.e. 600 rather than 300 cycles), the performance was considerably worse. So to some extent these delays are unavoidable.[2]

The "*pf-miss: invalidated*" category includes the cases where the prefetched data item is

[2]One technique that may help in this case is a "producer prefetch", as implemented in the DASH prototype. With a producer prefetch, the processor that generates the column would send it out to the processors waiting for it. This technique has the advantage that it has better hot-spotting performance than "consumer prefetch" (which is what we normally use), since only a single message is sent out from the hotspot, rather than having large numbers of request messages backing up at the hotspot, and then sending out the responses.

brought into the cache but is invalidated before it can be referenced. This situation arises because we use *non-binding* prefetches, and therefore sometimes the data must be invalidated to preserve correctness. This category was most prominent in MP3D, where it accounted for roughly half of the *pf-miss* cases. In MP3D, some of the objects that are prefetched (the "space cells") are very actively shared and modified amongst the processors. Therefore in many cases another processor wants to modify a location between the time it is prefetched and referenced. This effect is magnified to some extent by the fact that we use *exclusive-mode* prefetches, which means that a processor can invalidate other copies of the line when it expects to write the line in the near future, not only when it is actually modifying the line. The misses in the "*pf-miss: invalidated*" category tend to be expensive, since the data is typically found far from the processor (i.e. rather than being in the home memory, it is usually in a remote processor's cache). We see that this is true in Table 4.4, where the average read miss latency for MP3D is only reduced from 90.1 to 83.0 cycles, which is a meager reduction compared to the other applications. Although these invalidations are an interesting effect, the prefetches that are successful still manage to eliminate 82% of the miss stall time in MP3D. The fact that the invalidations allow the prefetches to be non-binding is a benefit that far outweighs the cost of prefetches that are occasionally unsuccessful due to invalidations.

The "*pf-miss: replaced*" category includes cases where the prefetched data is brought into the primary cache but is subsequently replaced from both the primary and secondary caches before it can be referenced. This case did not occur often for any of the applications.

Finally, the "*pf-miss: in-scache*" category includes cases where prefetched data is replaced from the primary cache before it can be referenced, but the data is still found in the secondary cache. Fortunately this is the dominant case for all the applications, as it was in the uniprocessor study. This is the least harmful of the *pf-miss* categories since the latency of a miss to the secondary cache is small relative to other types of misses, and in many cases the latency of an expensive miss has been reduced to a relatively cheap miss (i.e. it represents *partial* latency-hiding). For example, the majority of misses in this category for MP3D were formerly "dirty remote" misses, which have latencies of at least 132 cycles. Therefore the latency is reduced by roughly an order of magnitude when the data are found in the secondary cache.

Although the "*pf-miss: in-scache*" category is the least harmful of the *pf-miss* categories, it is still obviously less desirable than the *pf-hit* category. In the case of CHOLESKY, only 10% of the prefetched primary misses became primary hits—the other 90% are found in the secondary cache. Unfortunately, in this case nearly all of those latter misses were in the secondary cache to

begin with, so there is no partial latency-hiding benefit. On the other hand, most of the misses that became primary hits were originally expensive remote misses. All of these prefetches are for the same structures: the *panels*. Once a panel is "ready", it is repeatedly applied to other panels through a procedure call. When a panel is first used, the prefetches within the procedure succeed in fetching it from a remote location into the primary cache. However, on subsequent calls to this procedure, the prefetches for that same panel fail to bring it from the secondary to the primary cache due to conflicts with other references. These conflicts are with other panel references in the key loop, where eight separate columns within the panel are referenced on each iteration. To some extent, the conflict problem is also aggravated because we schedule for the worst-case latency—although this allows prefetching to work for the expensive misses (which is important), it causes the data to arrive in the primary cache unnecessarily early when found in the secondary cache. Treating these cases differently is difficult since they are simply different calls to the same procedure. It is unfortunate that scheduling is not more effective for CHOLESKY, considering how well locality analysis performs: only 15% of the prefetches are unnecessary and the coverage factor is 85%. The performance of LOCUS is also hindered by a similar conflict problem, but to a lesser extent.

To summarize, we have seen that the scheduling algorithm is often quite successful, which results in substantial speedups for OCEAN, LU, and MP3D. However, there is room for further improvement in CHOLESKY and LOCUS by reducing cache conflicts, which we will address later in Section 5.2.2. Despite these conflicts, which replace data from the primary to the secondary cache, the benefit of hiding remote latencies resulted in the elimination of at least 30% of memory stall cycles in all cases.

4.3.3 Prefetching Indirect References

The prefetching algorithm used so far in this chapter attempts to prefetch both *dense* and *indirect* array references. Indirect references are prefetched as described in Section 3.5.1. Only one of the multiprocessor applications contained a significant number of indirect array references: MP3D. Figure 4.11 breaks down how much of the prefetching benefit came from the dense versus the indirect prefetches.

As we see in Figure 4.11, the overwhelming majority of the benefit was from prefetching the indirect references. This is in contrast with the results we saw earlier in Figure 3.11 for the uniprocessor version of MP3D, where there was very little advantage to prefetching the indirect references. The difference between these two cases is that the indirect references are to objects

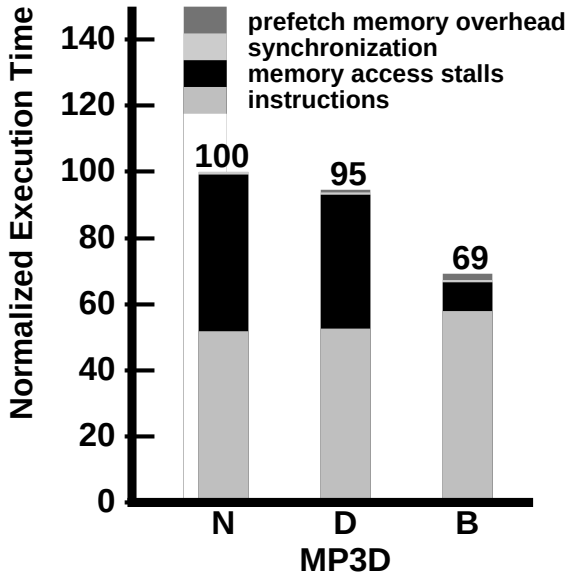


Figure 4.11: Prefetching indirect references in the multiprocessor version of MP3D (**N** = no prefetching, **D** = dense-only prefetching, and **B** = both dense and indirect prefetching).

that are very actively shared and modified amongst the processors (the "space cells"), whereas the dense references are to objects that are rarely shared and reside in a processor's local memory (the "particles"). Therefore the miss latency tends to be substantially larger for the indirect references, since they are often found dirty in a remote processor's cache, in contrast with the dense references, which are found locally.

This application illustrates several aspects of our prefetching compiler algorithm: (i) locality analysis to reduce the overhead of prefetching dense matrix references (as shown in Figure 4.8), (ii) prefetching indirect references (as shown in Figure 4.11), and (iii) non-binding prefetching for multiprocessors (as evidenced by the size of the "*pf-miss: invalidated*" category in Figure 4.10). We now consider the final aspect of our multiprocessor prefetching algorithm, which is using *exclusive-mode* prefetches.

4.3.4 Exclusive-Mode Prefetching

In this section, we evaluate the benefits of exclusive-mode prefetching, which helps to reduce both the miss latencies and the message traffic associated with writes. Unlike read misses,

Table 4.5: Statistics on exclusive-mode prefetching.

Benchmark	Fraction of Prefetches that are Exclusive-mode (%)	Reduction in Message Traffic (%)
OCEAN	38	23
LU	50	9
MP3D	100	27
CHOLESKY	16	15
LOCUS	2	3

which directly stall the processor for their entire duration, write misses affect performance more indirectly, since writes can be buffered. A processor stalls while waiting for writes to complete in two situations: (i) when executing a write instruction if the write buffer is full, and (ii) during a read miss if previous writes must complete before the read miss can proceed. The impact of the former effect can be reduced through larger write buffers. Throughout this study, we use 16-entry write buffers, which we have found to be large enough to make the full-buffer effect negligible. The impact of the latter effect depends on whether reads are permitted to bypass writes (as allowed by the release consistency model), and whether the cache permits multiple outstanding accesses (as allowed by a lockup-free cache).

As we described earlier in Section 4.1.3, our compiler uses an exclusive-mode (rather than a shared-mode) prefetch whenever any member of an *equivalence class* (i.e. a set of references that share group locality) is a write. This catches the important *read-modify-write* cases, and potentially eliminates as much as half of the message traffic. Table 4.5 shows the fraction of prefetches that were exclusive-mode for each of the applications. To evaluate the case where exclusive-mode prefetches are not available, we replace each exclusive-mode prefetch with a normal "shared-mode" prefetch of the same address. Since the multiprocessor architecture we have used so far includes both *release consistency* (which allows writes to be buffered and reads to bypass pending writes) and *lockup-free caches*, write latency has no direct impact on performance. Consequently exclusive-mode prefetching has a negligible performance impact on this architecture. It does, however, reduce the amount of message traffic, as shown in Table 4.5. If the architecture was bandwidth-limited (which in our case it is not), then this reduction in message traffic could have a direct payoff in improved performance.

To evaluate the benefit of exclusive-mode prefetching in an architecture where write latency is not already completely hidden, we performed the same experiment on an architecture that

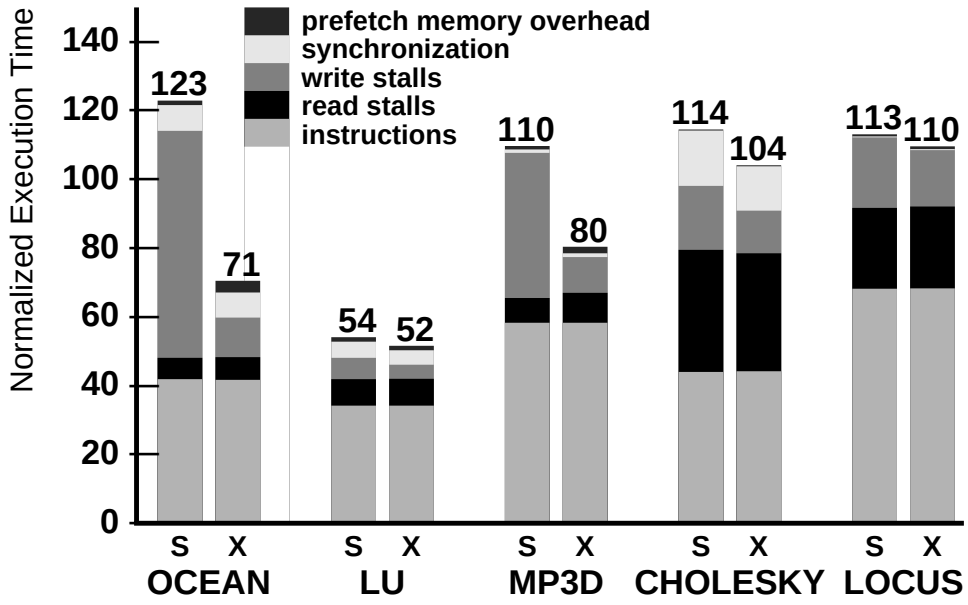


Figure 4.12: Performance with and without exclusive-mode prefetching given *sequential consistency* rather than release consistency (**S** = shared-mode prefetching only, **X** = exclusive-mode prefetching available). Performance is normalized to release consistency without prefetching.

uses *sequential consistency* rather than release consistency. With this stricter consistency model, the processor must stall after every shared access until that access completes. (We will discuss consistency models in greater detail later in Section 5.3.2.) The results of this experiment are shown in Figure 4.12. Notice that the memory stall time in Figure 4.12 has been broken down further into *write stall time* and *read stall time* (under the release consistency model assumed so far in this chapter, nearly all of the memory stall time is read stall time).

Figure 4.12 shows that exclusive-mode prefetching can result in dramatic performance improvements in an architecture using sequential consistency: OCEAN and MP3D achieved speedups of 73% and 37%, respectively. The speedups for CHOLESKY and LOCUS were understandably smaller (10% and 3%) since they make less use of exclusive-mode prefetching, as shown in Table 4.5. In the case of LU, the write latency is small to begin with since the processors only write to their local columns, which tend to fit in the secondary caches.

In summary, exclusive-mode prefetching can provide significant performance benefits in architectures that have not already eliminated write stall times through aggressive implementations

of weaker consistency models with lockup-free caches. Even if write stall times cannot be further reduced, exclusive-mode prefetching can improve performance somewhat by reducing the traffic associated with cache coherency.

4.4 Cache Size Variations

In this section, we study the effects of varying the cache size. Our motivation is twofold. First, we would like to build additional confidence in our results, given our cache scaling methodology. Recall from Section 4.2.3 that in order to achieve realistic problem-size to cache-size ratios, the results presented so far in this chapter are based on an 8K/64K cache hierarchy, which is scaled down from the full 64K/256K caches in DASH. By varying the cache size, we can explore the effects of cache scaling on performance. Second, by running the same problem on varying cache sizes, we effectively change the mixture of misses due to *replacements* (because of the finite cache size) versus misses due to *coherence* (because of the communication of shared data). This allows us to evaluate how well our compiler algorithm prefetches both types of misses.

The results of our experiments are presented in Figure 4.13, where the performance of each application is shown on three different sets of cache sizes. The middle pair of bars are for the scaled cache sizes of 8K/64K that we used throughout this chapter. The rightmost pair of bars are for the full-sized 64K/256K caches in DASH. The leftmost pair of bars are for a smaller cache hierarchy of 2K/8K. We begin by focusing on how the code without prefetching performs across the different cache sizes, and then later compare this to the behavior of the code with prefetching.

If we focus our attention first on the code *without* prefetching, we see a variety of different behaviors in Figure 4.13. At one extreme is MP3D, where varying the cache size has virtually no impact on performance. This is because MP3D sweeps through a very large data set on each time step, and even the 64K/256K caches are not large enough to retain the data (each processor's particles are 200 Kbytes, and the space cell array is 288 Kbytes). This region, where the the data set is much larger than the cache size, is expected to be the case in real-life runs of MP3D, so our results are indicative of the performance on real machines. At the other extreme are applications like LU, CHOLESKY, and LOCUS, where there is a noticeable knee in performance once a key data structure fits in the cache. For LU, this knee occurs between the 8K/64K and 64K/256K cache sizes, and corresponds to when the columns owned by a processor (roughly 20 Kbytes) fit in the primary cache. For both CHOLESKY and LOCUS, the knee occurs between the 2K/8K and 8K/64K cache sizes. In the case of CHOLESKY, this corresponds to an entire panel fitting

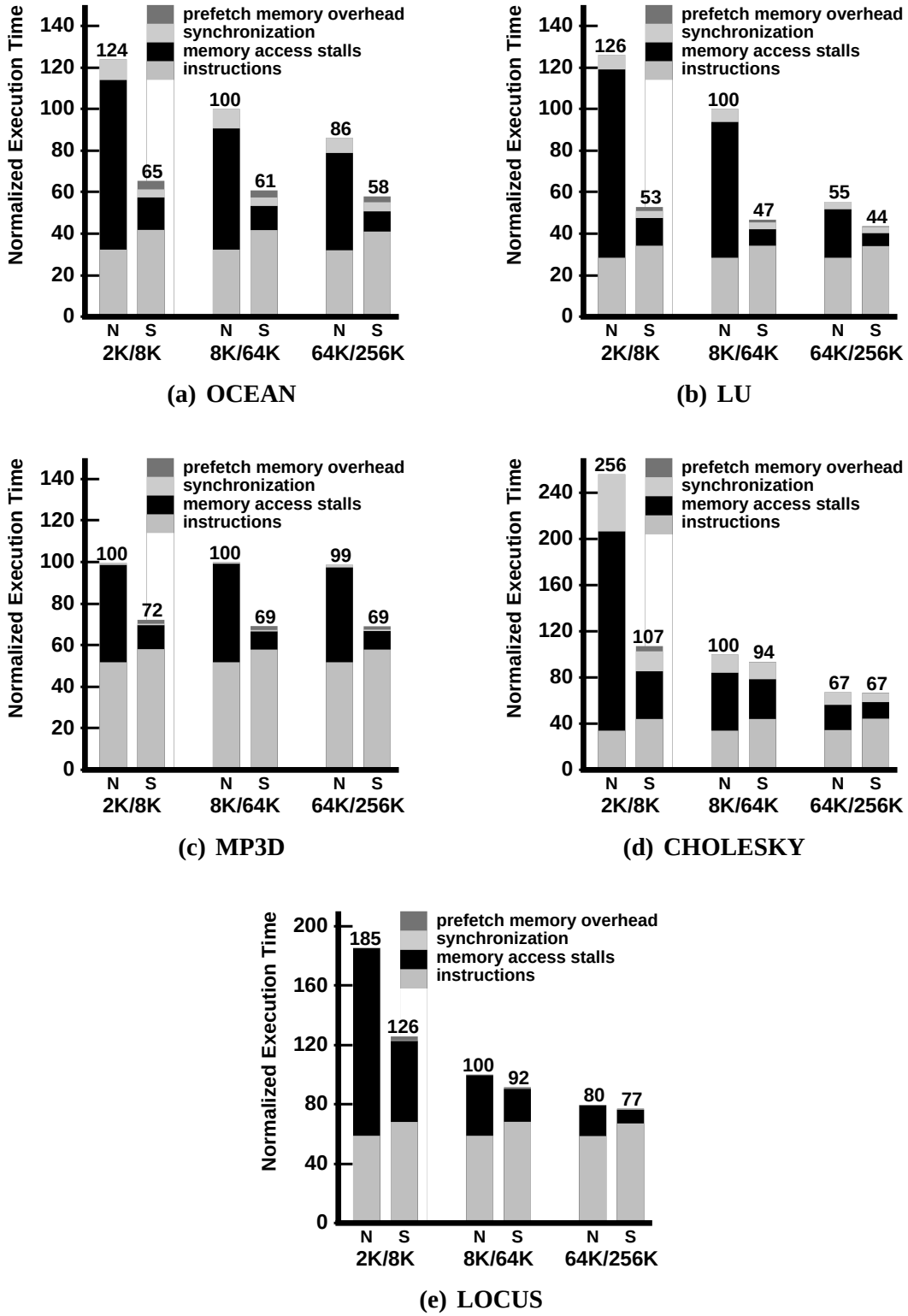


Figure 4.13: Performance of multiprocessor applications with varying cache sizes (**N** = no prefetching, and **S** = selective prefetching).

in the primary cache. For LOCUS, it corresponds to the portion of the cost array where a wire is being routed fitting in the cache. Therefore we see dramatic performance swings (more than 80% improvement) as the knee is crossed in three of the five cases without prefetching.

In contrast, the performance of the code *with* prefetching varies more gradually over the range of cache sizes. For example, while the performance of LU without prefetching improves by 82% when the cache sizes increase from 8K/64K to 64K/256K, the performance with prefetching changes by only 7%. Similarly, the performance improvement as the cache sizes are increased from 2K/8K to 8K/64K is 156% without prefetching vs. 14% with prefetching for CHOLESKY, and 85% without prefetching vs. 37% with prefetching for LOCUS. Why is the prefetching performance less sensitive to cache size? As the cache size is reduced, we see more replacement misses in the code without prefetching. However, since our algorithm is effective at prefetching replacement misses (as demonstrated already in Chapter 3), the resulting latency is therefore hidden, thus making the code with prefetching less sensitive to cache size. In fact, in three of the five cases (OCEAN, LU, and MP3D), the performance of code with prefetching on 2K/8K caches was better than code without prefetching on 64K/256K caches.

We now consider how cache size variations affect the relative performance improvement offered by prefetching. In general, Figure 4.13 shows that prefetching improves performance more with smaller cache sizes. The obvious reason for this is that smaller caches have more latency to hide due to replacement misses, and thus there is more room for improvement. For example, prefetching offers only modest speedups for CHOLESKY and LOCUS on 8K/64K caches, but provides dramatic speedups on 2K/8K caches (139% and 48%, respectively). With these smaller caches, the prefetches for the panels in CHOLESKY and the cost array in LOCUS are almost always useful; with larger caches, these structures only miss during communication, and therefore the prefetches are not as beneficial. As evidence of this reduced benefit, the instruction overhead offsets much of the reduction in memory stalls for both CHOLESKY and LOCUS with 64K/256K caches.

Can these instruction overheads with large caches be reduced by prefetching only the coherence misses? For CHOLESKY, the answer is yes, but only if special techniques for handling procedures are used. The difficulty in CHOLESKY is that each access to a panel occurs as a separate call to the same procedure. To isolate only the coherence misses, we would like to prefetch a panel only the first time it is accessed by a given processor. This cannot occur normally, but would be possible with either inlining or procedure cloning [15]. (Note that a very similar situation occurs in LU.) For LOCUS, however, the answer is no, coherence misses

cannot be isolated. The reason why is that coherence misses in LOCUS only occur when another processor routes a wire in the same portion of the cost array as the given processor. Since these misses are unpredictable and occur sporadically, they cannot be isolated, and therefore we must conservatively prefetch the cost array all the time.

In summary, we have seen that the performance advantages of prefetching often remain significant even as the cache size is varied. This robustness occurs despite the fact that we conservatively assume that data does not remain in the cache across synchronization statements (see Section 4.1.2). Since our algorithm effectively handles replacement misses, the performance with prefetching is relatively insensitive to the cache size, unlike code without prefetching, where performance can suffer greatly once key data structures no longer fit in the cache. Once the caches are large enough that little latency remains, it becomes increasingly important to minimize instruction overhead by prefetching only the coherence misses. However, in some cases it is impossible to isolate only the coherence misses, since they cannot be predicted, and therefore the compiler is forced to be conservative. In such cases, profiling feedback information may potentially be useful, as we will discuss later in Section 5.2.1.

4.5 Programmer-Inserted Prefetching

Software-controlled prefetching relies upon some form of support for inserting prefetches into the code. While having the compiler insert the prefetches automatically is the preferred approach, another possibility is for the programmer to insert the prefetches by hand. Comparing compiler-based prefetching with hand-inserted prefetching is a useful exercise because hand-inserted prefetching is not constrained by the limitations of the compiler technology; it provides a rough "upper bound" on the performance we might expect from the compiler. In this section, we examine applications with hand-inserted prefetching, discussing cases where the compiler was successful (Section 4.5.1), and also cases where the compiler failed (Section 4.5.2).

4.5.1 Cases Where the Compiler Succeeded

In an earlier study, we inserted prefetches by hand into MP3D, LU, and PTHOR [61]. In the case of PTHOR, the access patterns are irregular and difficult to prefetch even by hand—we will discuss that case later in Section 4.5.2. In contrast, the other two cases (MP3D and LU) have regular and predictable access patterns. Therefore inserting prefetches by hand was relatively easy, and we saw large performance gains. Our compiler algorithm is also successful at prefetching

these two cases, which we examine in more detail in this subsection.

MP3D

The MP3D application spends most of its time executing a loop where each processor takes a particle and moves it through one time step. The overwhelming majority of cache misses are caused by references to two structures within this loop: (i) the particle which is being moved (36% of misses), and (ii) the space cell where the particle resides (53% of misses). Particles are statically assigned to processors and are allocated from the shared memory local to each processor, while the memory for the space cells is distributed uniformly among the processors.

We inserted prefetches into MP3D by hand as follows. Since a particle must be referenced to determine the space cell it occupies, we prefetch a particle record two iterations before its turn to be moved. In the iteration following the prefetch, the particle is read, and the associated space cell is determined and prefetched. As a result, when it is time for the particle to be moved, both the particle and space cell records are available in the cache. We also prefetch several other references that occur at time step boundaries. The end result is a coverage factor of 90% for our hand-insertion scheme. Exclusive-mode prefetches are used since the objects are modified during each iteration. Introducing these prefetches required adding 16 lines to the source code.

When our compiler inserted prefetches into MP3D, it recognized that the address of a space cell is computed based on the *x*, *y*, and *z* fields in a particle record (which represent the coordinates of the space cell). Since this is an indirect reference, the compiler used the algorithm described in Section 3.5.1 to prefetch the particles two iterations ahead, and the space cells one iteration ahead. The scheduler determined that only a single iteration is needed to hide the memory latency, since the loop body is rather large. Therefore the compiler duplicated the hand-inserted approach to prefetching particles and space cells, resulting in a coverage factor of 89%. The compiler also prefetched a few other references at time step boundaries, but they turned out to be insignificant.

Figure 4.14(a) shows the performance of both the compiler-based and the hand-inserted prefetching schemes for MP3D. As we see in this figure, they both do quite well. The hand-inserted case performs slightly better simply because the scalar optimizer was able to eliminate a few more instructions in that case, but this difference is basically in the “noise”. Therefore the compiler-based scheme appears to be living up to its potential in this case.

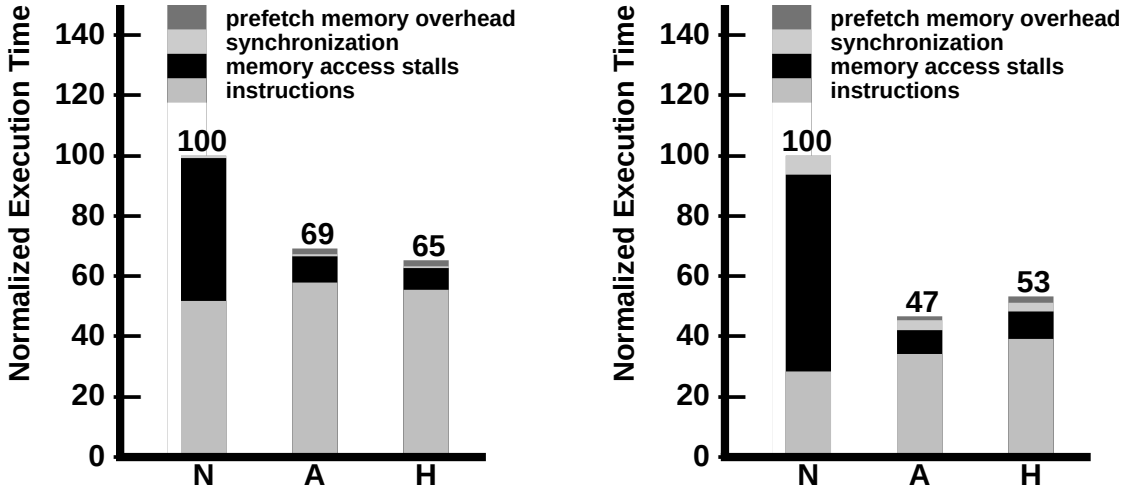


Figure 4.14: Comparison of compiler-based and hand-inserted prefetching for the cases where the compiler succeeded (**N** = no prefetching, **A** = prefetches inserted automatically by compiler, **H** = hand-inserted prefetching).

LU

In LU, the matrix columns are statically assigned to the processors in an interleaved manner. The main computation done by each processor consists of reading a pivot column once it is produced, and applying the pivot column to each column to its right that the processor owns. There are three primary sources of misses in LU: (i) the pivot column when it is read for the first time (9%); (ii) the pivot column when it is replaced by a column it is applied to and needs to be refetched (12%); and (iii) the owned columns that the pivot column is applied to (75%). This last set of misses occurs because the combined size of the owned columns is larger than the size of the cache.

Our strategy for prefetching LU by hand was the following. Each time the pivot column is applied to an owned column, we prefetch the pivot column in shared mode and the owned column in exclusive mode. Although prefetching the pivot column each time causes redundant prefetches, it reduces the misses when the pivot column is replaced from the processor's cache, resulting in a total coverage factor of 96%. We found that it is better to evenly distribute the issue of prefetches throughout the computation rather than prefetching an entire column in a single burst, in order to avoid hot-spotting problems. We also unrolled the loop to minimize

instruction overhead, since there is spatial locality. A total of 8 lines were added to the source code.

Our compiler chose an identical approach for inserting prefetches. It also prefetches the pivot column and owned columns each time they are used. Since the key inner loop is inside a separate procedure, the compiler does not recognize that the pivot column may have temporal locality, and therefore might not need to be prefetched each time. However, it turns out that prefetching the pivot column all the time is the appropriate strategy, and therefore both schemes perform quite well, as we see in Figure 4.14(b). The small difference in instruction count was because the scalar optimizer did a slightly better job of optimizing the compiler-generated code than it did on the hand-inserted code. Both schemes eliminated roughly 90% of the memory stalls, and increased overall performance by more than 85%. Therefore the compiler clearly lives up to the potential of hand-inserted prefetching for both MP3D and LU.

4.5.2 Cases Where the Compiler Failed

As we mentioned at the beginning of Section 4.3, the compiler was only successful at prefetching five of our eight original multiprocessor applications. We will now consider the other three cases (WATER, BARNES, and PTHOR) to see what went wrong.

WATER

As we saw earlier in Figure 4.6, WATER suffers the least from memory latency of all the SPLASH applications, spending only 7% of its time stalled for memory. Although there is little need for prefetching in this case, we discovered nonetheless that our algorithm is unable to cover the misses. The reason why is because the key loop body is not in the same file as its surrounding loop. Since our prefetching algorithm does not perform interprocedural analysis—particularly not across separate files, which becomes very tricky given separate compilation—it fails to recognize the affine access patterns, and therefore does not insert any prefetches at all. With either interprocedural analysis or inlining across separate files, the compiler could easily prefetch the references and hide the memory latency. Since the solution to this problem is well-understood, and since there is little performance gain to be had, we did not bother to insert the prefetches by hand for this case. WATER is an example of a case where strengthening the implementation of the existing algorithm would solve the problem.

BARNES

In contrast with WATER, BARNES spends 38% of its time stalled for memory; therefore hiding latency can potentially result in significant performance improvements. However, since BARNES contains mainly pointers and recursive data structures, our compiler does not insert any prefetches since such references are beyond the scope of our algorithm. To evaluate how much the compiler might be able to improve performance if it could handle such cases, we inserted prefetches into BARNES by hand. We begin with a quick overview of BARNES and a description of how we inserted prefetches, followed by our experimental results.

In BARNES, the main computation consists of traversing the octree structure to compute the gravitational force exerted by the given body on all other bodies in the tree. This is repeated for each body in the system, and the bodies are statically assigned to processors for the duration of each time step. Cache misses occur whenever a processor visits a part of the octree that is not already in its cache, either due to replacements or communication.

We inserted prefetches into BARNES by hand as follows. The octree structure is traversed depth-first, and each internal node can have up to eight children. When we first arrive at a node, we issue prefetches for all immediate children before descending depth-first into the first child. By doing so, we succeeded in covering 78% of the misses, with only 21% of the prefetches being unnecessary. Roughly half of the prefetched primary misses became primary hits—the other half were found in the secondary cache. These prefetches turned out to be moderately expensive in terms of instruction overhead (on average 12 instructions per prefetch) because of the loop overhead and conditional tests at each node. Overall, we achieved a speedup of 9%, as shown in Figure 4.15.

Despite the many pointers in BARNES, it was relatively straightforward to pinpoint the cause of the misses. Through the profiling information provided by our simulator, we discovered that 81% of the miss latency was caused by only six references. In addition, the author of BARNES was readily able to predict these important references without the benefit of profiling information, which indicates that these misses are intuitively obvious to someone familiar with the algorithm. Only five lines of code were added to BARNES to insert the prefetches. The greater challenge in BARNES was scheduling the prefetches early enough to hide the latency, since the control flow consists of recursive procedure calls rather than loops. We were largely successful toward this goal because we understood the structure of the tree and the order in which it was traversed.

Could the compiler insert these same prefetches automatically? With the help of sophisticated pointer analysis to recognize the tree structures and the order in which they are traversed [37],

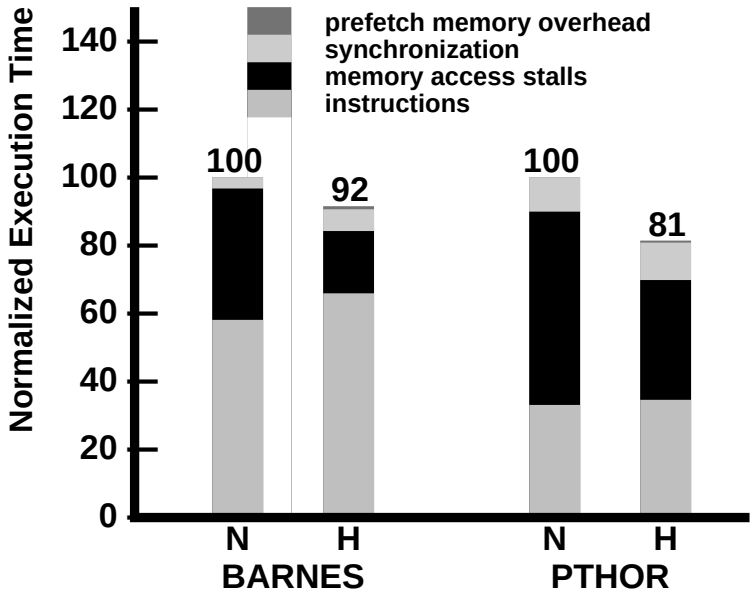


Figure 4.15: Cases where the compiler failed to improve prefetching (**N** = no prefetching, **H** = hand-inserted prefetching).

and perhaps with profiling feedback information to indicate which pointer dereferences suffer misses (see Section 5.2.1), it may be possible. It certainly will not be easy, since pointer analysis is a difficult problem, but it may be possible.

PTHOR

PTHOR is an interesting case because it is second only to LU in the amount of time lost to memory latency (59%), and yet our compiler algorithm fails to insert any prefetches. The problem (as in BARNES) is that PTHOR is an irregular computation containing mainly pointers to linked lists, which are beyond the scope of our algorithm. The approach we take to inserting prefetches by hand was described in detail in an earlier study [61], and we briefly summarize that strategy here.

One of the main data structures in PTHOR is the *element record*, which stores all information about the type and state of a logic element. Several fields in the record are pointers to linked lists, or are pointers to arrays that in turn point to linked lists. During the main computational loop, each processor picks up an activated logic element, computes any changes to the element's outputs, and schedules new input events for elements that are affected by the changes. Prefetching

is complicated by the presence of the linked lists, since to prefetch a list it is necessary to dereference each pointer along the way.

To insert prefetches by hand, we first reorganized the element record and grouped entries based on whether they were likely to be modified, likely to be read but not modified, or likely not to be referenced. Whenever a processor picks an element from a task queue, we prefetch the element record entries accordingly. In addition, we prefetch the first several levels of the more important linked lists. Due to the complex control structure of the application, it is difficult to determine where the misses occur. Despite the aid of profiling information that helped determine which sections of code were generating misses, we were only able to increase the coverage factor to 36%. A total of 29 lines were added to the source code, resulting in a performance improvement of 23%, as shown in Figure 4.15.

Adding prefetching to PTHOR was a qualitatively different experience from adding prefetching to MP3D, LU, and even BARNES. While the process of inserting prefetches took considerably more time, the resulting coverage was much lower. Unlike BARNES, it was very difficult to isolate the misses in PTHOR, even with the help of profiling information, since they are spread across so many different references and often occur only sporadically. Even the author of PTHOR had a difficult time predicting which references should be prefetched. This is partly because PTHOR is a significantly more complicated program, both in terms of its data structures and its control flow. In many cases, although we knew from profiling information that a particular reference was suffering from cache misses, we could find no way to schedule the prefetches that would improve performance. Therefore, of the programs we have studied, PTHOR appears to be the most difficult challenge for the compiler. Given that we did achieve a speedup of 23%, it appears to be a challenge worth pursuing.

4.5.3 Summary

By comparing hand-inserted prefetching with compiler-inserted prefetching, we saw that in cases where the access patterns are regular and predictable (MP3D and LU), the compiler is able to match the performance of hand-inserted prefetching. In the cases where our compiler failed to insert prefetches, the difficulty of fixing the problem ranged from challenging but straightforward to extremely difficult. For WATER, it is a matter of engineering the compiler to handle interprocedural analysis across separate files—a nontrivial task, but one that is feasible. For BARNES, the compiler needs to recognize tree structures and understand how to prefetch them. For PTHOR, however, the data structures, access patterns, control flow, and sources of misses

are complicated enough that it is unclear whether the compiler can be successful. We note that these problems in PTHOR occur even in the uniprocessor version of the code, and are not unique to multiprocessing.

Strengthening the existing algorithm to handle cases such as WATER, and extending it to recognize and prefetch simple recursive data structures such as trees, as in BARNES, would appear to be the next logical steps in enhancing the compiler algorithm.

4.6 Chapter Summary

In this chapter, we extended our uniprocessor prefetching algorithm to handle multiprocessor applications as follows. Through the use of *non-binding* prefetching, our algorithm is freed from concerns over violating multiprocessor correctness, and can focus instead on improving performance. To account for the additional cache misses due to *communication*, we modify our original algorithm to conservatively assume that shared data does not remain in the cache across synchronization statements. Finally, to hide the latency of gaining ownership of a line, and to reduce the bandwidth consumed in read-modify-write sequences, we exploit *exclusive-mode* prefetching.

Our experiments with prefetching for multiprocessors produced the following results:

1. Similar to our uniprocessor results, the compiler algorithm is successful at hiding memory latency while minimizing prefetching overhead, and again improves performance by as much as twofold. These encouraging results occur despite our algorithm's conservative assumptions about the effect of communication on miss rates.

2. Exclusive-mode prefetching eliminated as much as 27% of the total message traffic in our architecture. While this had little direct impact on the performance of our base architecture, we demonstrated that it could improve performance by as much as 73% for an architecture that has not already hidden write latency through relaxed consistency models. In addition, the reduction in message traffic could translate directly into improved performance on a bandwidth-limited architecture (e.g., a bus-based multiprocessor).

3. Our algorithm is generally robust with respect to variations in the cache size. Since our algorithm is effective at prefetching replacement misses, it does not suffer sudden drops in performance as the cache size is decreased, as we sometimes see in code without prefetching. With larger cache sizes, where sharing misses dominate, we still see significant benefits

from prefetching in most cases.

4. When our compiler algorithm succeeds at inserting prefetches, it lives up to the performance potential of hand-inserted prefetching.

5. To further increase the coverage of our algorithm, techniques such as interprocedural analysis and recognizing recursive data structures may be useful.

Overall, these results are quite encouraging, and they demonstrate that automatic compiler-inserted prefetching is an attractive technique for tolerating latency in large-scale multiprocessors.

Chapter 5

Architectural Issues

The success of software-controlled prefetching relies on the combined support of both hardware and software. While part of the attractiveness of this technique is the *simplicity* of the hardware support, that support is nonetheless crucial. This chapter addresses these architectural issues associated with software-controlled prefetching.

We begin in Section 5.1 with a detailed examination of the basic architectural support that was assumed in Chapters 3 and 4. During this examination, we will justify each aspect of the architecture, and will quantify the relevant tradeoffs whenever possible. Next, we look to the future, considering additional ways to improve prefetching performance which involve some amount of architectural support. For example, we will look at using dynamic information to improve the analysis of what to prefetch, and increasing cache associativity to make prefetches more effective. Finally, in Section 5.3, we compare software-controlled prefetching with other latency-hiding techniques that require hardware support, such as relaxed consistency models and multithreading.

5.1 Basic Architectural Support for Prefetching

Chapters 3 and 4 assumed a "base" model for prefetching, including a prefetch instruction, lockup-free caches, etc. In this section, we will examine that model in more detail and discuss some important tradeoffs. We begin in Section 5.1.1 by discussing key properties of prefetch instructions and how to incorporate them into a processor's *instruction set architecture*. Given that prefetches are performance hints, and therefore can be dropped, in Section 5.1.2 we evaluate when it is appropriate to drop prefetches. In Section 5.1.3 we discuss how prefetches that are

not dropped should proceed through the memory hierarchy, and where the resulting data should be placed (e.g., should it go directly into the primary cache). We conclude in Section 5.1.4 by discussing the more significant hardware modifications needed to support prefetching, such as providing *lockup-free caches*.

5.1.1 Instruction Set Architecture

The *instruction set architecture* of a processor serves as the interface between hardware and software. Although prefetches closely resemble *load* instructions, the desired semantics and actions are quite different. This section discusses these semantic and functional issues.

Behavioral Properties

There are three key behavioral distinctions between prefetches and loads; prefetches are (i) *non-binding*, (ii) *non-blocking*, and (iii) *non-excepting*. The *non-binding* property gives prefetches the flexibility to be issued far in advance of the actual references, without worrying about the impact on correctness. The *non-blocking* property allows prefetches to be overlapped with other references and with computation. The *non-excepting* property allows speculative prefetching of addresses which may potentially be invalid. In this subsection, we discuss the importance of each of these properties in more detail.

The *non-binding* aspect of prefetching is implemented by fetching data into the cache rather than a register. As we discussed earlier in Section 4.1.1, non-binding prefetches are essential in multiprocessors since they allow the compiler to prefetch a location without worrying about whether the value may have been modified by another processor in the meantime. Even in a uniprocessor, the non-binding property is important since it avoids the correctness problems that can arise when using registers for temporary storage given imperfect memory disambiguation. For example, if prefetches fetched data into registers, it would be *illegal* to move a prefetch ahead of a store unless it was certain that the store was to a different location (otherwise the prefetched value would be stale). Proving that addresses do not coincide is extremely difficult because of complications such as aliasing, pointers, etc. Therefore, the non-binding property frees the compiler from correctness problems that can occur both *across* threads and *within* a single thread.

An additional advantage of prefetching into the cache rather than the register file is that otherwise the limited size of the register file can be a significant constraint on how far ahead

```
PREF const_offset(base_reg)    (e.g., PREF 20(r7))
```

Figure 5.1: Format of prefetch instructions, using "base-plus-offset" addressing mode.

one can prefetch. This is crucial since extending *register lifetimes* to hundreds of cycles (in order to hide large latencies) is almost guaranteed to cause significant register spilling, which can hurt performance considerably. The register lifetime problem is most important in scientific code, where common techniques such as loop unrolling, software-pipelining and register blocking result in very high register pressures even without prefetching. The cache, on the other hand, is substantially larger than the register file, and therefore is not expected to constrain the amount one would reasonably want to prefetch ahead.

The *non-blocking* aspect of prefetching is essential since the very essence of this latency-hiding mechanism is overlapping memory accesses with computation. Normal loads could also potentially be non-blocking, but this would require a mechanism for interlocking and forwarding the data whenever the load result was used before the access completed. (Because of this hardware complexity, few commercial microprocessors have implemented non-blocking loads.) In contrast, it is easy to make prefetches non-blocking since they produce no result value, and therefore no instructions can depend upon their completion.

Finally, the *non-excepting* aspect of prefetching (i.e. prefetches do not take memory exceptions on invalid addresses) is important since it allows data-dependent addresses (e.g., pointers) to be prefetched without being absolutely certain that the address is valid. We have already discussed in Section 3.5.1 how this is important when prefetching indirect references, such as in sparse-matrix code. Even in dense-matrix code, this property is useful by making it safe to prefetch off the end of an array whenever generating a proper epilog would be too expensive (i.e. when it would result in a code size explosion). Therefore the non-excepting property offers considerable flexibility to the compiler since it is much easier to generate valid prefetch addresses *most* of the time rather than *all* of the time.

Format

The format of a prefetch instruction resembles that of a normal load instruction, except that there is no need to specify a destination register. Therefore addressing modes that are appropriate for normal loads are also appropriate for prefetches. The addressing mode used in these experiments was "base-plus-offset", as shown in Figure 5.1. This addressing mode is attractive because

```
Loop:  LOAD    r2,0(r7)          /* load A[i] */
       PREF    20(r7)            /* prefetch A[i+5] */
       ADD     r3,r3,r2          /* sum += A[i] */
       ADD     r7,r7,4           /* i++ */
       BRANCH  r7 < r8, Loop     /* loop test */
```

Figure 5.2: Example of how prefetches can reuse load/store base registers (in this case `r7`).

prefetch address can typically be generated by using the same base register as the load or store being prefetched and simply adjusting the constant offset to reflect the software pipelining distance. An example of this is shown in Figure 5.2, where the prefetch instruction is fetching data five iterations ahead of the load instruction, and both instructions use the same base register (`r7`). This is important in order to avoid spilling registers once prefetches are inserted.[1] Addressing modes which cannot take advantage of such register reuse should be avoided.

Encoding

One interesting question is whether a prefetch should be encoded with its own unique *opcode*, or whether it should simply be a variation on a normal load instruction. A load instruction could be modified to encode a prefetch in the following two ways. First, some RISC architectures (such as the HP Precision Architecture) allow user-specified *hints* on loads, and one type of hint could be interpreted as a prefetch. Second, many RISC architectures set aside one register which always sources the value zero. Performing a load *to* this reserved register does not change its contents, so such operations could be interpreted as prefetches.

Although a prefetch can be encoded as a form of a load, it is preferable to designate a unique opcode for prefetches (assuming one can be spared) for the following two reasons. First, since prefetches and loads behave quite differently (i.e. prefetches are non-binding, non-blocking, and non-excepting), it may be important to distinguish them early on during decoding, which is presumably easier with distinct opcodes. Second, with distinct opcodes, the bits normally used to specify the destination register of a load can be used instead as "hint bits" to describe various flavors of prefetching. For example, we have already seen in Section 4.3.4 the advantages of

[1] We observed this in practice. At first, our scalar optimizer was not taking advantage of the constant offset difference. The performance impact was devastating in many cases, once register spilling occurred.

Figure 5.3: Possible encoding of prefetch instruction.

having both *shared-mode* and *exclusive-mode* prefetches in multiprocessors under an invalidation-based cache coherence scheme. Other prefetching variations which will be described later in this chapter include: (i) prefetching *multiple* cache lines rather than a single cache line (which may help reduce instruction overhead, as described in Section 5.2.3); (ii) prefetching data that is to remain *uncached* rather than being placed in the cache (which may help reduce cache conflicts, as described in Section 5.2.2); and (iii) whether or not a prefetch should be dropped on a TLB miss (as described in Section 5.1.2). Figure 5.3 shows a possible encoding of the prefetch instruction, based on the MIPS instruction set, including these prefetching hints.

5.1.2 Dropping Prefetches

In this subsection, we examine cases where the overhead of processing a prefetch may potentially be large enough that one might consider dropping the prefetch instead. Since prefetches are performance hints, and have no *side-effects* on user-visible program state, the processor can safely ignore them without violating correctness. In fact, if code with prefetching is run on a processor whose memory system does not support prefetching, the prefetches can simply be treated as NOPs. Presumably the software will be effective at issuing prefetches that improve performance, and we have already seen evidence of this in Chapters 3 and 4. However, there may be cases where the hardware can determine that a prefetch should be *dropped* because the penalty of processing it is too large. For example, what if the prefetch takes a TLB exception, or what if the prefetch issue buffer is already full? We will focus on such issues in this subsection.

TLB Miss

The first step in executing a prefetch is translating the virtual data address to a physical address. Address translation is accelerated in modern RISC processors through a "translation lookaside buffer" (TLB), which is simply a cache of recent virtual-to-physical address mappings. Hence the first question is whether a prefetch should be dropped if its virtual address does not match an entry in the TLB—otherwise a TLB fault handler must be run, which is a relatively expensive operation.

The answer to this question is complicated by two conflicting goals. On the one hand, we would like to hide the latency in situations where we are legitimately suffering frequent TLB misses, and this cannot occur if the prefetch is dropped.[2] An example would be code that iterates across the outer dimensions of large matrices, in which case each reference may be to a unique page. On the other hand, one of the desirable properties of prefetch instructions (as mentioned earlier in Section 5.1.1) is that they are free to reference invalid addresses, in which case we would like to drop the prefetch with minimal performance loss. Since TLBs do not contain invalid address mappings, an invalid address can only be detected by performing full address translation, hence suffering the cost of a TLB miss (which can potentially be hundreds of cycles). This second scenario may occur frequently in code containing pointers and other indirect references, in which case this TLB miss overhead may be prohibitively expensive.

[2] If TLB refills (i.e. address translation) are handled by hardware, then the prefetch can potentially hide the latency of both the TLB refill and the memory access. On the other hand, if TLB refills are handled through software exceptions (as in the MIPS architecture), then only the memory latency can be hidden.

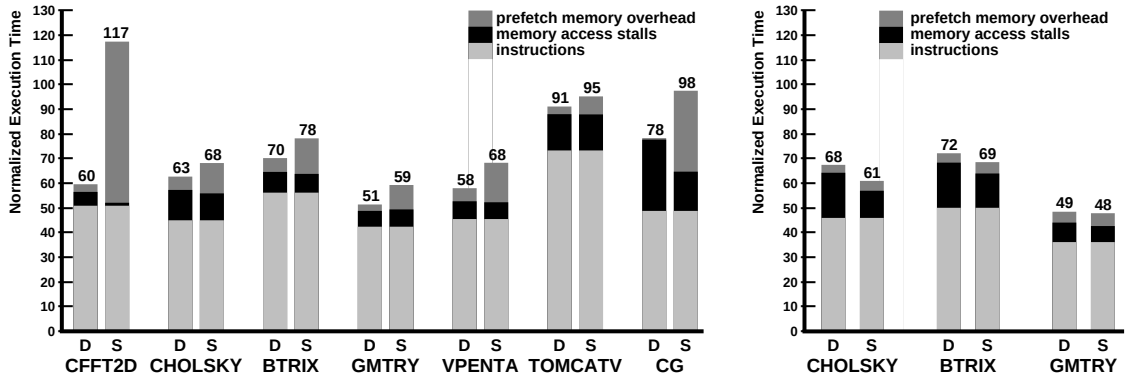
Although choosing between these two goals is difficult, since each is important given its own scenario, we can start by comparing their expected frequencies. The case where legitimate TLB misses are occurring frequently is somewhat unlikely for the following reasons. First, it can only be a sustained problem for applications having both *very* large data sizes and very large (at least a page) strides. Although both of these may occur in some scientific codes, it is far more common to see smaller strides as the code iterates through inner dimensions of matrices. Smaller strides are advantageous since they can exploit *spatial locality* by reusing cache lines, and we would expect locality optimizations such *loop interchange* (as demonstrated in Section 3.4) to continue enhancing this in the future. Second, since legitimate TLB misses would occur even without prefetching, then presumably processor designers have already dealt with this problem by making the TLB sufficiently large. In contrast, invalid prefetch addresses may occur frequently in any code containing indirect references (hopefully not, but it is a possibility). This is independent of both data size and the number of TLB entries. Also, given the inherent difficulty in prefetching code containing recursive data structures (as we encountered with PTHOR and BARNES in Section 4.5.2), the additional burden of TLB miss penalties on invalid addresses is likely to make the task hopelessly frustrating. Therefore if a default policy must be chosen, it is probably better to drop prefetches on TLB misses.

An alternative to choosing a fixed policy is to allow the software to select the more appropriate policy by making use of the prefetch hint bits described earlier in Section 5.1.1. For example, there could be two types of prefetches: "speculative" prefetches, which should be dropped on TLB misses since the address may be invalid, and "non-speculative" prefetches, where it is better to suffer the TLB miss for the sake of hiding the latency. This approach satisfies both goals, and may lead to the best overall performance.

Once a valid physical addresses has been computed for a prefetch, it is ready to be issued to the memory subsystem. The mechanics of how a prefetch normally proceeds through the memory subsystem will be discussed later in Section 5.1.3. However, even after a prefetch has been issued to the memory subsystem, it is still possible to abort it before it completes. The scenario where this might make sense is when the memory subsystem queues are already full, and the prefetch cannot proceed without stalling the processor, as we will discuss next.

Full Prefetch Issue Buffer

For our experiments in Chapter 3, we assume that if the processor attempts to issue a prefetch while the prefetch issue buffer is full, the processor stalls until an entry becomes available. We



(a) Indiscriminate Algorithm (b) Selective Algorithm

Figure 5.4: Dropping vs. stalling on full prefetch issue buffer (**D** = drop, **S** = stall).

now consider what happens if prefetches are instead dropped on a full prefetch issue buffer.

In the architectural model presented so far, the memory subsystem has a finite (16-entry) prefetch issue buffer to hold outstanding prefetch requests. In our model, a prefetch is inserted into the buffer only 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); reordering may occur due to variations in miss latencies. Despite some 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. At that point the choices are either to stall the processor until a buffer entry becomes available, or else 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 uniprocessor benchmarks again using a model where prefetches are dropped rather than stalling the processor when the prefetch issue buffer is full. We ran this model for both the indiscriminate and selective prefetching algorithms. Figure 5.4 shows the cases where this affected performance. We begin by focusing on the indiscriminate algorithm, and then later focus on the selective algorithm.

For all seven cases where the performance of the indiscriminate algorithm changed (shown in Figure 5.4(a)), the performance *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). 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 prefetches 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. 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 (see Figure 3.2(a)), where memory stall time doubles for the indiscriminate algorithm once prefetches are dropped.

The selective prefetch algorithm, in contrast, did *not* improve from dropping prefetches since it suffered very little from full prefetch issue buffers in the first place. In fact, in the three cases shown in Figure 5.4(b), the selective algorithm performed slightly worse when prefetches are dropped. The reason why is that since selective prefetching has eliminated many of the redundant prefetches, it is more likely that dropping a prefetch would translate into a subsequent cache miss. However, as we have already seen in Figure 3.2, the selective algorithm tends to suffer very little from full issue buffers, and therefore performs well in either case.

Summary

To summarize, deciding when to drop prefetches is a complex issue. The one clear-cut case where prefetches should be dropped is when they reference invalid addresses. Because of our desire to minimize the performance overhead for this case (which can arise frequently when speculatively prefetching pointers), we would lean toward dropping prefetches as soon as a TLB miss is detected, even though this means giving up the ability to hide latency during legitimate TLB misses. A better solution may be to distinguish “speculative” and “non-speculative” prefetches, to give software control over this policy. Finally, given that prefetches are issued *selectively*, it makes little difference whether they are dropped once the prefetch issue buffer fills up, since this rarely happens.

Once a prefetch has been issued to the memory subsystem, it is simply a matter of finding the data and then moving it close to the processor. Both halves of this mechanism will be discussed in the next section.

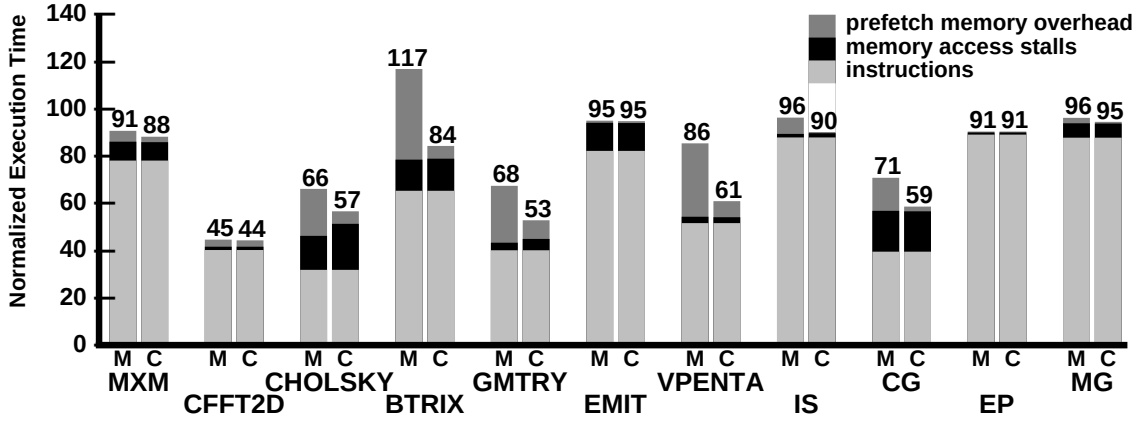


Figure 5.5: Performance when prefetches do not check either caches before going to memory (**M** = go straight to memory, **C** = check caches).

5.1.3 Performing the Prefetch Memory Access

In this section we consider three issues related to performing a prefetch memory access. First, is it important to check the caches while searching for the prefetched data, or can one skip the caches and go straight to main memory? Second, should prefetched data be placed directly into the primary data cache, or somewhere further down in the cache hierarchy? Finally, we will briefly discuss why placing prefetched data in a separate target buffer rather than the normal cache hierarchy is not a good idea in general.

Checking Caches While Searching for the Data

During a normal cache miss, the levels of the memory hierarchy closer to the processor are always checked before proceeding to subsequent levels. For example, the secondary cache is only checked if the data is not found in the primary cache. With prefetching, however, one might argue that since the prefetches are scheduled early enough to hide the worst-case miss latency, it is no longer necessary to check each level of the cache while searching for the data. To evaluate this, we modified the uniprocessor architecture such that prefetches proceed directly to memory without checking either level of the cache. The results of this experiment are shown in Figure 5.5.

As we see in Figure 5.5, it is still important to check levels of the cache close to the processor for the prefetched data.[3] The primary reason for this is to minimize *bandwidth consumption*, not

[3]The two cases that are not shown are TOMCATV and OCEAN. These two applications suffered more from memory latency when the caches are not checked than any of the other applications—so much so that the simulations

latency. The deeper levels of the memory hierarchy are slower, and have less bandwidth to offer. Therefore the prefetches tend to congest the memory system, causing delays both in issuing other prefetches and in servicing normal cache misses. So we see that checking the cache helps alleviate bandwidth-related delays caused by prefetches that can be serviced close to the processor (including *unnecessary* prefetches).

Once the prefetched data has been found, the next step is moving it close to the processor. Just how close is the next question we address.

Prefetching into the Primary Data Cache

On a normal load miss, the data line is brought into all levels of the cache hierarchy, including the primary cache, which makes sense because the processor stalls waiting for the data. On a prefetch miss, however, it is not as obvious that the data should be placed in the primary cache for two reasons. First, the prefetched data may displace other data that will be referenced *before* the prefetched data, thereby causing an *additional* miss rather than *eliminating* a miss. The second problem is that a new source of contention is added since the processor cannot execute loads or stores at the same time that the primary cache is being filled with prefetched data. In our experiments so far, we modeled this effect by assuming that four cycles of exclusive access are needed for each prefetch fill. Therefore a load or store may be delayed by up to four cycles.

Although we have just discussed two potential downsides of prefetching into the primary cache, the upside, of course, is that the latency of a primary-to-secondary miss can be hidden. Note that these downsides are not as applicable to prefetching into the secondary cache, since (i) the secondary cache is quite large (and therefore the chance of displacing important data is small), and (ii) the processor does not actively contend for the secondary cache tags. Therefore there is little argument that prefetched data should be placed in the secondary cache. Consequently, the key question is whether or not to place the data in the primary cache.

To answer this question, we will use both analytical and experimental approaches. Let us begin with an analytical approach of comparing expected costs and benefits. The benefit of prefetching into the primary cache is that for each *successful* prefetch (p_s), the latency of a primary-to-secondary miss (l_s) is eliminated:

$$\text{Benefit} = p_s l_s \qquad (5.1)$$

This includes both cases where the prefetched data is found in the secondary cache (in which case

never ran to completion.

l_s represents the *entire* miss latency), and cases where the prefetched data is found in memory (in which case l_s is only a fraction of the total primary-to-memory miss latency).

The cost of prefetching into the primary cache includes the effects of both increased *conflicts* and *contention*:

$$\text{Cost} = p_c l_s + p_f l_f \quad (5.2)$$

Additional conflicts occur whenever prefetches displace useful data, where p_c is the number of such conflicts, and each conflict results in an extra primary-to-secondary miss (l_s) to later fetch the displaced data. Additional primary cache tag contention can occur during prefetch fills, where p_f is the number of prefetch fills, and l_f is the average number of cycles that a prefetch fill stalls the processor. In practice, l_f should be less than the total fill time, since the processor only stalls if it attempts to execute loads or stores during the fill.

To simplify the algebra in order to directly compare the costs and benefits, we will make the following substitutions into equations (5.1) and (5.2). First, we will express the number of successful prefetches (p_s) in equation (5.1) as the difference between the cases where the prefetched data is brought into the cache early enough (p_t) and the cases where that data is displaced before it can be referenced (p_d), hence $p_s = p_t - p_d$. Next, we will conservatively represent the number of cases where useful data is displaced from the cache (p_c) in equation (5.2) as the number of cases where prefetches are displaced (p_d), since in the worst case, each displaced prefetch will also displace useful data.[4] Finally, we replace the number of prefetch fills (p_f) in equation (5.2) with p_t, the total number of prefetches that arrive early enough in the cache. After making these substitutions, we note that prefetching into the primary cache is worthwhile whenever the following inequality holds:

$$\begin{aligned} \text{Benefits} &> \text{Costs} \\ (p_t - p_d)\, l_s &> p_d l_s + p_t l_f \end{aligned} \quad (5.3)$$

By rewriting equation (5.3), we can express the maximum value of l_f (the average processor stall on a prefetch fill) at which prefetching into the primary cache is advantageous:

$$l_f < \left(1 - \frac{2p_d}{p_t}\right) l_s \quad (5.4)$$

Equation (5.4) shows that in the absence of primary cache *conflicts* due to prefetches (i.e. $p_d = 0$), it is worthwhile to prefetch into the primary cache as long as the average stall time due to primary

[4]Note that this is a conservative approximation because a prefetch may have been displaced by *another* prefetch, in which case the displaced prefetch has no benefit, but also does not *increase* the number of misses by displacing data that would not have otherwise missed.

cache tag contention (l_f) is less than the primary-to-secondary miss latency (l_s). This condition should normally be satisfied since filling the primary cache is almost always a subset of the time it takes to service a primary-to-secondary miss. As the number of primary cache conflicts induced by prefetching (p_d) increases, the maximum acceptable value of l_f will decrease.

Since primary cache conflicts are an important concern, we rewrite equation (5.3) once again, this time solving for p_d:

$$p_d < \frac{1}{2}\left(1 - \frac{l_f}{l_s}\right) p_t \qquad (5.5)$$

Notice from equation (5.5) that in the absence of primary cache tag *contention* (i.e. $l_f = 0$), the break-even point occurs when *half* of the prefetches arriving in the cache early enough (p_t) cause cache conflicts. In this case, half of the prefetches would be successful (thereby eliminating a cache miss), while the other half would displace useful data (thereby adding a new cache miss), and both effects would exactly cancel each other out. As the amount of contention (l_f) increases, it decreases the acceptable number of conflicts (p_d).

To predict whether prefetching into the primary cache is worthwhile for the uniprocessor architecture used in Chapter 3, we can solve equation (5.5) given the parameter values of that architecture: $l_s = 12$ cycles, and $l_f = 4$ cycles. (Note that this is an upper bound on l_f since the processor does not necessarily stall for the entire duration of a primary prefetch fill, which is 4 cycles in this case.) Substituting these values into equation (5.5), we get the following:

$$p_d < \frac{1}{3} p_t \qquad (5.6)$$

Therefore we would expect prefetching into the primary cache to be worthwhile as long as fewer than a third of the prefetches suffer primary cache conflicts. The actual values of l_f and $\frac{p_d}{p_t}$ are shown for each application in Table 5.1. We see that the values of l_f ranges from 1.2 to 2.5 cycles, which is about half of the full primary fill time of four cycles. This makes sense since loads and stores typically occur at most once every other instruction. We also see from Table 5.1 that the only application where cache conflicts potentially occur too often is CHOLSKY, where 37% of prefetches placed in the primary cache are displaced before they can be referenced. However, since CHOLSKY also happens to suffer less than average from primary cache contention ($l_f = 1.50$), the number of displaced prefetches is below the maximum for breaking even (which happens to be 44% for this case). Also, the number of additional misses caused by conflicting prefetches is overstated in this case, since only about half of the displaced prefetches were displaced by data references (the other half were displaced by other prefetches, and therefore did not incur additional misses)—i.e. $p_c \approx \frac{1}{2} p_d$ for CHOLSKY. Hence the analytical

Table 5.1: Average processor stall on a primary prefetch fill (l_f) and the fraction of prefetches that suffer primary cache conflicts ($\frac{p_d}{p_t}$) for each uniprocessor application.

Benchmark	l_f	$\frac{p_d}{p_t}$
MXM	2.00	0.102
CFFT2D	2.36	0.008
CHOLSKY	1.50	0.374
BTRIX	1.95	0.095
GMTRY	2.54	0.097
EMIT	2.36	0.075
VPENTA	2.01	0.013
TOMCATV	2.02	0.261
OCEAN	1.25	0.000
IS	1.17	0.180
CG	1.34	0.190
EP	2.37	0.045
MG	1.82	0.000

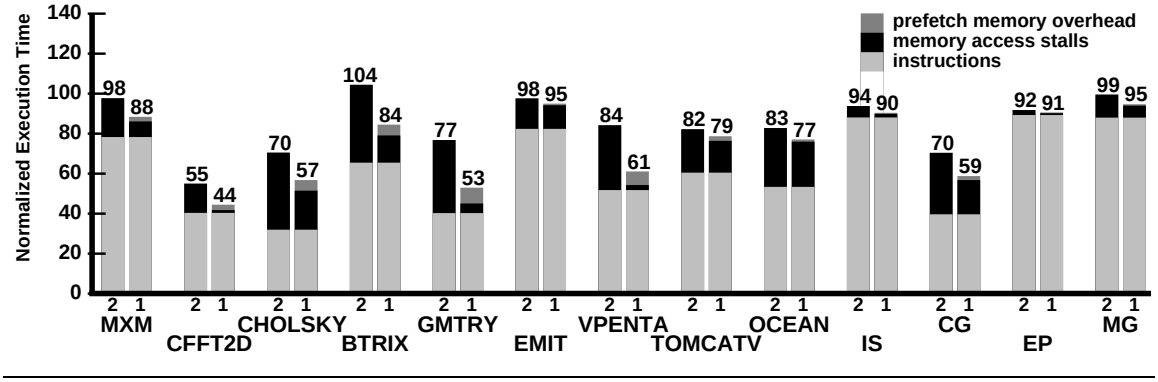


Figure 5.6: Performance when prefetching into the primary cache (**1**) versus prefetching only into the secondary cache (**2**).

model predicts that for the given architecture, prefetching into the primary cache should be a performance win for each of these applications.

To evaluate whether this is true in practice, we simulated the performance of each uniprocessor application, this time only prefetching into the secondary cache. The results of that experiment are shown in Figure 5.6. As we see in this figure, prefetching into the primary cache is in fact a performance win in all cases. In seven of the thirteen cases, there is a speedup of at least 19% due to prefetching into the primary cache rather than just the secondary cache.

Table 5.2: Distribution of where data was found both by prefetch and by subsequent reference. "**X** $\Rightarrow$ **Y**" means prefetch found data at **X**, subsequent reference found data at **Y**, where **X**,**Y** = $\mathbf{C}_1$ (primary cache), $\mathbf{C}_2$ (secondary cache), and **M** (memory).

	Miss Latency is Improved			Miss Latency is Unchanged			Miss Latency is Worse		
Benchmark	$C_2 \Rightarrow C_1$	$M \Rightarrow C_1$	$M \Rightarrow C_2$	$C_1 \Rightarrow C_1$	$C_2 \Rightarrow C_2$	$M \Rightarrow M$	$C_1 \Rightarrow C_2$	$C_1 \Rightarrow M$	$C_2 \Rightarrow M$
MXM	41.0	3.0	0.1	51.1	3.9	0.1	0.9	0.0	0.0
CFFT2D	32.3	66.4	0.4	0.1	0.4	0.4	0.0	0.0	0.0
CHOLSKY	44.4	15.9	0.7	3.4	32.5	0.0	3.2	0.0	0.0
BTRIX	39.7	7.6	0.3	47.1	4.5	0.1	0.8	0.0	0.0
GMTRY	73.5	14.0	1.3	3.1	7.8	0.0	0.3	0.0	0.0
EMIT	56.9	22.4	1.5	14.3	4.5	0.1	0.3	0.0	0.0
VPENTA	48.1	22.6	0.2	27.9	0.8	0.0	0.2	0.0	0.0
TOMCATV	13.1	16.4	11.0	50.7	3.7	0.1	5.0	0.0	0.0
OCEAN	21.9	40.3	0.2	36.9	0.5	0.1	0.0	0.0	0.0
IS	16.6	12.1	1.6	65.0	3.3	0.1	1.2	0.1	0.0
CG	30.9	20.6	3.3	36.2	5.9	0.2	2.8	0.1	0.0
EP	0.0	95.5	4.3	0.0	0.0	0.1	0.0	0.0	0.0
MG	13.3	59.3	1.9	23.5	1.3	0.4	0.3	0.0	0.0

The marginal improvement of prefetching into the primary cache depends on the relative frequency and cost of primary misses that are found in the secondary cache versus misses that go all the way to memory. If the data tends to fit in the secondary cache, then the entire benefit of prefetching will come from prefetching into the primary cache.[5] In the other extreme, if none of the primary misses are found in the secondary cache, then the marginal improvement will be roughly the ratio of the primary-to-secondary and primary-to-memory miss latencies (which would be 12/75 = 16% for this architecture).

To help characterize where the data is being found, Table 5.2 presents a breakdown of exactly where the data is found both by the prefetch and by the subsequent reference. When the value of the "$C_2 \Rightarrow C_1$" category (i.e. cases where the data are successfully prefetched from the secondary to the primary cache) is large relative to the sum of the "$M \Rightarrow C_1$" and "$M \Rightarrow C_2$" categories (i.e. cases where data are prefetched from memory, and are found in either the primary and secondary cache, respectively), we would expect prefetching into the primary cache to account for most of the gains. This agrees with the data in Figure 5.6. For example, GMTRY has the largest value in the "$C_2 \Rightarrow C_1$" column (73.5%), and also shows the largest marginal speedup (45%). In contrast, the prefetched misses in TOMCATV are twice as likely to be found in memory rather than the secondary cache, and therefore the marginal speedup for TOMCATV is considerably

[5] In an architecture that prefetches only into the secondary cache, prefetches would have a cost and no benefit in cases where the data is found in the secondary cache.

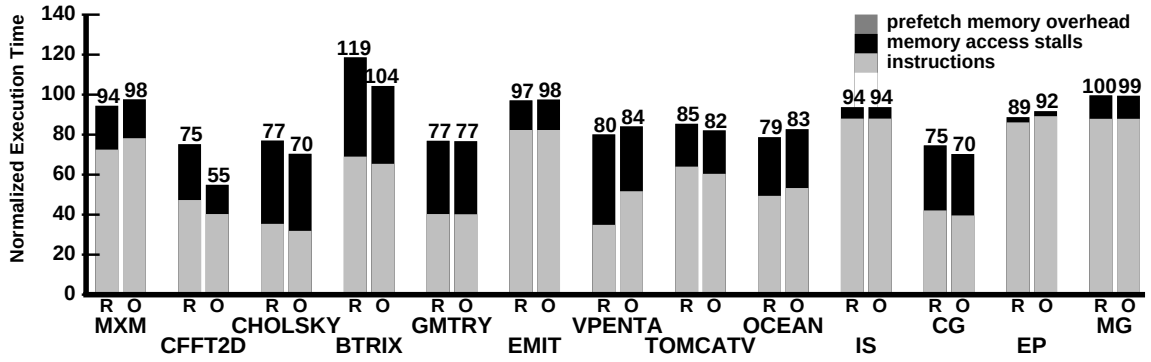


Figure 5.7: Performance when prefetching into the secondary cache, both when compiled for the primary cache size (**O**), and when recompiled for the secondary cache size (**R**).

smaller (4%).

For the experiments in Figure 5.6, we did not recompile the code to take into account the larger size of the secondary cache relative to the primary cache (256K vs. 8K). Compiling for the larger cache size might help reduce instruction overhead by eliminating prefetches of data that reside in the secondary but not the primary cache. However, notice that there are only a few cases in Figure 5.6 where the instruction overhead could potentially offset the marginal difference in reduced memory stalls. To see whether this would make a difference, we recompiled each program, this time increasing the *effective cache size* parameter to 256 Kbytes. The results of this experiment are shown in Figure 5.7.

As we see from Figure 5.7, the results of recompiling the code to take into account the larger cache size were mixed. In five of the cases (MXM, EMIT, VPENTA, OCEAN, and EP) the performance improved, while in six of the other cases (CFFT2D, CHOLSKY, BTRIX, TOMCATV, CG, and MG) the performance was worse. Of the cases that did improve, the benefit was typically small (never more than 5%). In six of the cases the instruction overhead actually increased (due to the overheads of peeling loops to account for temporal locality), despite the fact that fewer prefetches were issued. Of the four cases where there was a significant potential to reduce instruction overhead (BTRIX, VPENTA, TOMCATV and MG), only one case improved: VPENTA. However, even in this case, much of the gain in reduced instruction overhead was offset by larger memory stall times, since a number of the prefetches that were eliminated were in fact useful for reducing latency. In the case of BTRIX, we see that compiling for a larger cache size does not reduce the instruction overhead (it merely eliminates some useful prefetches), and given that these prefetches are being issued, it is clearly preferable that they do something

useful (i.e. prefetch data into the primary cache).

Part of the reason for these mixed results is that it is more difficult to analyze locality in a very large cache, since the approximations inherent in using a fully-associative cache model in the locality analysis become more exaggerated. Also, in many cases the loop bounds are symbolic, and in these cases it does not matter how large the finite cache parameter is—all that matters is the policy for dealing with unknown loop bounds. Since it is unlikely that the compiler will be able to avoid prefetching data already in the secondary cache, we may as well get some performance benefit from these prefetches by moving the data into the primary cache. As secondary caches continue to grow, this should be even more true. Ultimately, however, it is the difference in memory stall reduction, not overheads, that shows that prefetching into the primary cache is the right thing to do.

Although the results presented in this section are specific to this architecture, we would expect the conclusions to remain applicable in the future for the following reasons. First, the average stall due to primary cache tag contention (l_f) should remain smaller than the primary-to-secondary miss latency (l_s). Also, as cache sizes continue to increase, there may be fewer primary cache conflicts (p_d), and more of the data set may be captured by the secondary cache, making it more important to prefetch primary-to-secondary cache misses. Therefore it is likely that prefetching into the primary cache will remain a performance win.

Prefetching into a Separate Prefetch Target Buffer

An alternative approach to moving data close to the processor is to place it outside the normal cache hierarchy in a separate *prefetch target buffer*. We will discuss this approach qualitatively in this subsection. The motivation presented in previous proposals for a prefetch target buffer is that by keeping the prefetched data separate, it cannot interfere with data in the normal cache (which is referred to as the cache "pollution" problem) [13, 42]. While this may be true, this approach has a number of drawbacks.

First, a prefetch target buffer only addresses the problem of conflicts between prefetches and references—it does nothing to help the cases when references conflict with other references, or when prefetches conflict with other prefetches; these latter cases are just as important. As evidence of how important reference-to-reference conflicts are, they were so bad in the original versions of four of the uniprocessor applications that we manually changed the alignment of some matrices to help alleviate these conflicts (as described earlier in Section 3.1). Therefore a more general solution that can address all different types of conflicts would be preferable to a

prefetch target buffer, which only addresses a subset of the conflicts. We will discuss general solutions to cache conflicts later in Section 5.2.2. Second, building a large prefetch target buffer will require sacrificing a significant amount of cache area, thereby reducing the hit rate of normal references. This will be particularly undesirable for applications that do not contain prefetches (either because they have not been recompiled or because they do not benefit from prefetching). Finally, prefetching into a special target buffer makes it difficult for prefetches to be *non-binding* in a multiprocessor environment, since the target buffer must also be kept coherent. In general, treating prefetch accesses as special cases will be more complicated than handling them through the normal cache miss mechanism.

For the types of applications considered in this study, the compiler rarely prefetches data that is not needed in the immediate future, and therefore the problem is not accurately characterized as cache *pollution*. Cache *conflicts*, on the other hand, can be a problem between *any* types of references. Therefore the best approach is to prefetch directly into the primary cache and find other *general* techniques for coping with cache conflicts.

5.1.4 Hardware Modifications to Support Prefetching

We now discuss some specific changes to the hardware that are necessary to support prefetching. First, the processor obviously must be able to decode and process the new prefetch instructions, as described in Section 5.1.1. The main complications of doing so are ensuring that they are non-blocking and are harmlessly dropped whenever the prefetch address is invalid.

It is important to realize, however, that *hardware support for prefetching does not end with just adding prefetch instructions to the instruction set.* It is essential that the bandwidth of the memory hierarchy be increased to support the extra demand imposed by prefetching. An important step toward increasing memory hierarchy bandwidth is allowing multiple outstanding cache misses, which is referred to as having a *lockup-free* cache [45].[6] This added bandwidth makes it possible to hide latency by overlapping memory accesses with *other* memory accesses, not just computation.

This subsection is organized as follows. We begin by discussing issues associated with implementing a lockup-free cache, and relate them to our uniprocessor architecture. We then evaluate the performance tradeoffs for two key parameters in the lockup-free cache design: the number of outstanding misses, and the number of prefetch issue buffer entries. Finally, we

[6] Note that there is even more to providing memory subsystem bandwidth than having a lockup-free cache. In addition, main memory must have sufficient bandwidth to service these multiple outstanding misses.

compare the performance of separate write and prefetch issue buffers with the performance of a combined buffer.

Lockup-Free Cache

A lockup-free cache is a common requirement for most latency-hiding techniques, including prefetching, relaxed consistency models, non-blocking loads, and multithreading. The complexity of implementing a lockup-free cache depends on which of these techniques it is intended to support (as described in detail by Laudon [52]). For example, if the goal is simply to support multiple outstanding prefetches, then it is not strictly necessary for the processor to maintain state on outstanding transactions, as long as the cache is prepared to receive prefetch responses from outside the processor while the processor may be simultaneously issuing new requests. In contrast, supporting multiple outstanding stores (as with relaxed consistency models) or loads (if they are non-blocking) *does* require that special state be maintained for each outstanding access. For stores, the stored data must be merged into the cache line when it returns. For loads, the requested data must be forwarded directly to a register—thus requiring state to associate each outstanding access with the register(s) waiting for the value—and any future uses of that register must interlock if the value has not returned yet.

Kroft [45] presented the original lockup-free cache design, which adds structures called "miss information/status handling registers" (MSHRs) to keep track of outstanding misses. Each MSHR contains enough state to handle one or more accesses of any type to a single memory line. Due to the generality of the MSHR mechanism, the amount of state involved is non-trivial, including the address, pointers to the cache entry and destination register, written data, and various other pieces of state. The majority of subsequent lockup-free cache proposals have been a variation of this original MSHR scheme [63, 70, 79, 46]. An alternative approach is to maintain the state of outstanding misses in the cache tag array itself [17, 52], thereby permitting a larger number of outstanding misses.

For the uniprocessor architecture used in Chapter 3, the lockup-free cache supports a single load or store miss (since loads and stores[7] directly stall the processor) and up to sixteen prefetch misses. The state of outstanding prefetch misses is maintained in a sixteen-entry *prefetch issue buffer*. This structure is simpler than a normal MSHR because it contains only the prefetch addresses and any state necessary to track or control outstanding misses. It does *not* contain any

[7] Stores stall the processor since the R4000 (the processor on which we base our uniprocessor architecture) has a write-back cache and no write buffer.

data, since the prefetched data is placed directly into the cache hierarchy.

To illustrate how the prefetch issue buffer is used, we will walk through the steps of processing a prefetch. When the processor executes a prefetch instruction, the primary data cache is checked during that cycle. If the data is already present in the cache, then the prefetch is considered completed at that point. Otherwise, the prefetch addresses is then compared against the addresses already in the prefetch issue buffer. If there is a match, then the prefetch is dropped. Otherwise, a new entry is allocated in the prefetch issue buffer for the new prefetch.[8] At this point, the prefetch will proceed through the memory hierarchy to first find the data and then place it in the primary cache, as was discussed in Section 5.1.3. An entry in the prefetch issue buffer is deallocated as soon as the prefetch completes (as opposed to waiting for the data to actually be referenced).

Although a prefetch issue buffer is not strictly necessary, it offers the following performance advantages. First, in cases where the prefetch miss cannot be serviced immediately (perhaps because of a limited number of MSHRs), it allows the processor to continue executing by buffering the prefetch request. Second, by keeping track of the prefetches that are already outstanding, it provides a mechanism for *merging* subsequent prefetch and memory references to matching cache lines. Merging a prefetch with a previous prefetch simply means dropping the subsequent prefetch.[9] This helps to avoid unnecessary bandwidth consumption further down the memory hierarchy, similar to the benefit of checking the primary cache when searching for the prefetched data (as discussed earlier in Section 5.1.3). However, with *selective* prefetching, the compiler is generally good at avoiding back-to-back redundant prefetches (at least for these scientific codes). The more important benefit occurs when a load is merged with a previous prefetch that was partially completed, since this allows the load to benefit from whatever partial latency-hiding the prefetch has been able to accomplish. This is especially important for cases where data or control-flow dependencies make it impossible to issue prefetches early enough to hide the entire memory latency, which can occur frequently in pointer-based codes such as PTHOR (e.g., when attempting to prefetch a linked list).

[8] If the prefetch issue buffer is already full, then either the processor must stall until an entry becomes available or else drop the prefetch. The tradeoff between these two choices was described earlier in Section 5.1.2, and for this study we will stall until an entry becomes available.

[9] The one possible exception is when a read-exclusive prefetch follows a read-shared prefetch of the same line. In this case, if the read-shared prefetch has not been issued to the memory system yet, it should be converted to a read-exclusive prefetch. Otherwise, the options are to either drop the read-exclusive prefetch or else issue it separately. We chose the latter for our simulations, but it made little difference because our selective prefetching algorithm is good at avoiding such redundancies.

Figure 5.8: Prefetch issue buffer in the uniprocessor architecture. Note that for prefetches, the fetched data goes directly into the cache, rather than being held in an MSHR. Therefore an MSHR is simply a resource for *controlling* an outstanding miss under this model.

Figure 5.8 shows how the prefetch issue buffer is incorporated into our uniprocessor architecture. In this figure, we have shown distinct MSHRs to allow for the possibility that misses are handled separately from the buffering of prefetch requests. Note that in our model, a prefetch remains in the prefetch issue buffer until it is completed by its MSHR. In our experiments so far, we have assumed a sixteen-entry deep prefetch buffer and seventeen MSHRs (one for each prefetch and one for either a load or store). We chose these large parameters to minimize their effect on performance. We will now examine how many prefetch issue buffer entries and MSHRs are actually needed for our benchmarks.

Before running these experiments with reduced prefetch buffering and MSHR capacities, we can gain insight into what to expect by examining the number of outstanding prefetches in the original "full-capacity" architecture. Figure 5.9 plots the distribution of prefetch misses already outstanding during each prefetch miss for the original architecture. As we see in Figure 5.9,

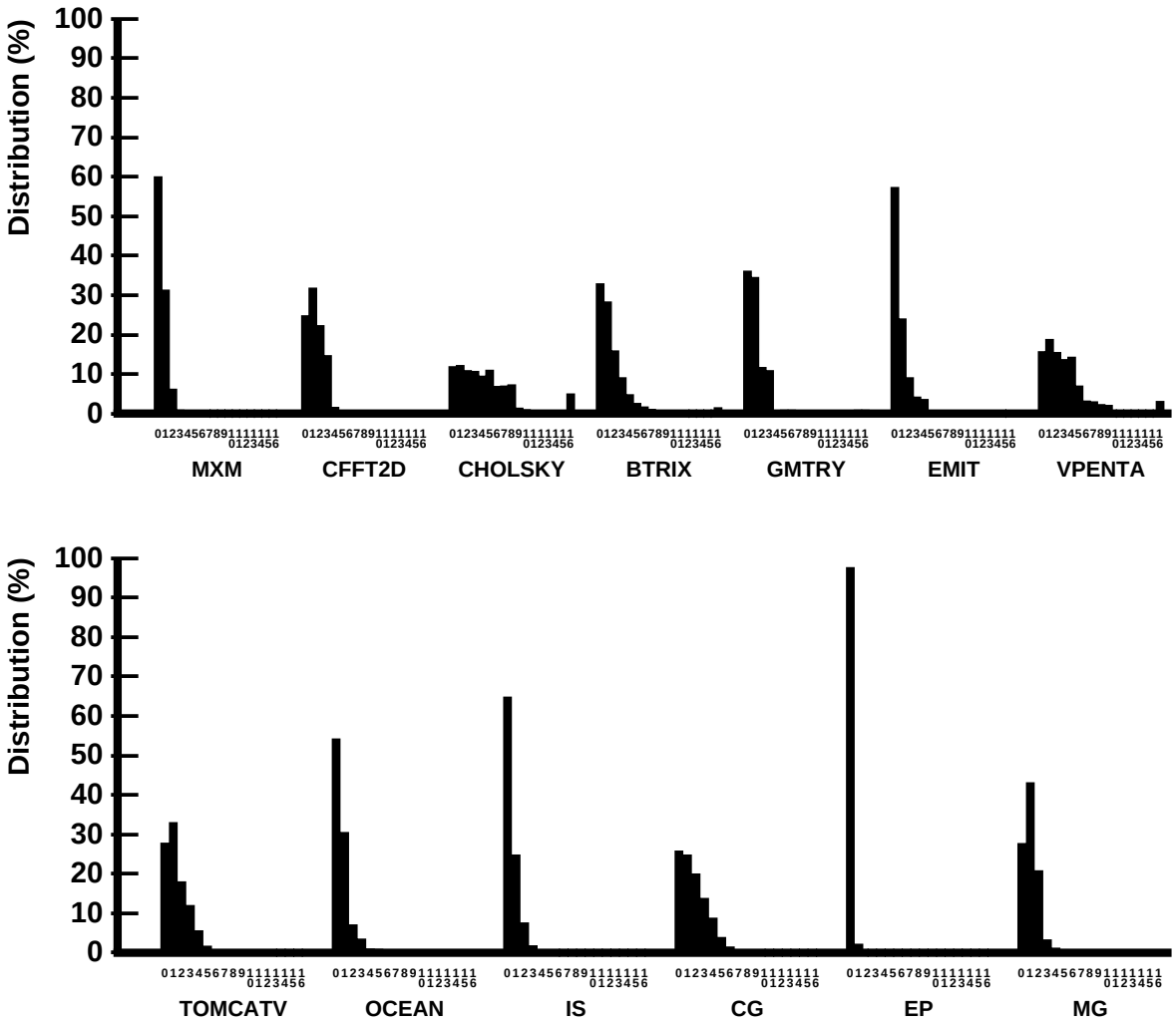


Figure 5.9: Distribution of previously outstanding prefetch misses upon each prefetch miss for the original uniprocessor architecture (which supports up to seventeen outstanding misses).

the distributions vary considerably across applications. On the one extreme is EP, where there is almost never more than one outstanding prefetch. On the other hand there are programs like CHOLSKY, where the distribution is spread out over larger numbers of previously outstanding prefetches (i.e. the distribution has a significant tail). We would expect these latter types of applications to benefit from having multiple MSHRs and deep prefetch buffers.

To measure the actual impact on performance, we first varied the number of MSHRs while holding the depth of the prefetch issue buffer fixed at sixteen entries. Figure 5.10 shows the

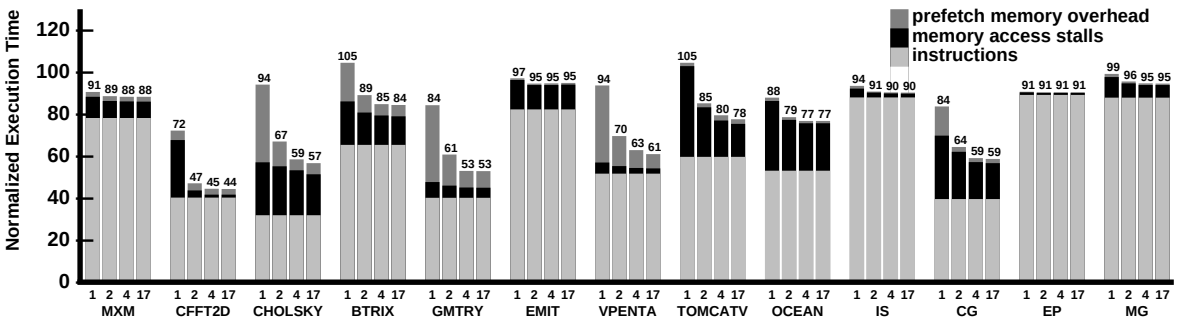


Figure 5.10: Performance when the number of MSHRs is varied.

results of these experiments. A single MSHR corresponds to a "blocking" cache (i.e. one that is not lockup-free), while seventeen MSHRs corresponds to the original architecture. Note that with fewer than seventeen MSHRs, there may potentially be more outstanding miss requests (up to seventeen) than can be serviced at a given time, in which case prefetch misses must compete with load and store misses for the MSHRs. We resolved such cases by giving load or store misses (which directly stall the processor) priority for the next available MSHR—otherwise, the oldest unserviced prefetch will be awarded the next available MSHR.

As we see in Figure 5.10, the benefits of a lockup-free cache are often substantial. Seven of the thirteen applications showed at least an 15% performance improvement by going from one to two MSHRs. In six of those seven cases, there was at least a 5% additional improvement from having four MSHRs. The benefits of moving from four to seventeen MSHRs were typically quite small. These performance improvements correspond to what we would expect from the distributions shown in Figure 5.9. In particular, the largest improvements occur in the benchmarks with the largest tails in their distributions (e.g., CHOLSKY, GMTRY, VPENTA, and CG). Therefore we see that it is important to have a lockup-free cache, and that four outstanding misses are sufficient to capture most of the benefits for these applications.

Given that four MSHRs are sufficient, the next question is whether buffering additional prefetch requests beyond that is advantageous, or whether the prefetch issue buffer should also be reduced to four entries. To evaluate this, we simulated the performance when the number of prefetch issue buffer entries is reduced, while holding the number of MSHRs fixed at four. Figure 5.11 shows the two cases (CHOLSKY and VPENTA) where this caused a noticeable difference in performance (in six of the other cases, there was less than a 2% difference between four and eight buffer entries, and no difference between eight and sixteen buffer entries). CHOLSKY showed

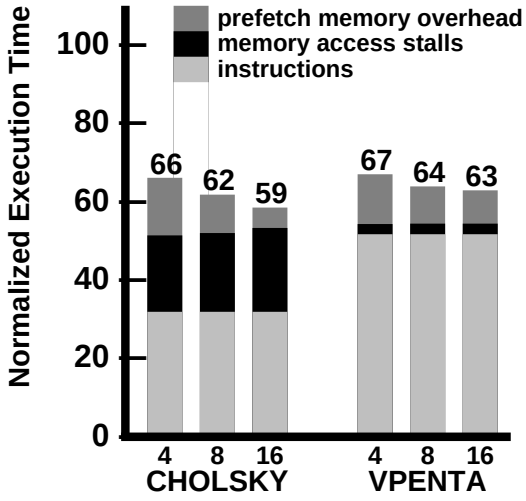


Figure 5.11: Performance when the size of the prefetch issue buffer size is varied between four, eight, and sixteen entries, given four MSHRs.

the largest improvement, with a 6% speedup when going from four to eight buffer entries, and 5% speedup beyond that for having sixteen entries. VPENTA showed a 5% speedup between four and eight entries, and 1% additional speedup with sixteen entries. Therefore we see that additional prefetch buffering is beneficial in a small number of cases. However, given that this benefit is relatively small and infrequent, the hardware cost would also have to be quite small for it to be justified.

Separate Write and Prefetch Issue Buffers

The final hardware-related issue we will discuss is whether it is useful to have a separate prefetch issue buffer in an architecture that already contains a buffer for writes, or whether both writes and prefetches should be placed in the same buffer. One possible performance disadvantage of using a combined buffer is that prefetches may be delayed behind writes. From an implementation perspective, a buffer that only handles prefetch requests would be smaller, since it does not contain written data. However, it may be simpler to build just a single buffer.

The uniprocessor architecture we have been using does not contain a write buffer, but the

multiprocessor architecture does, since it has a write-through primary data cache (versus the copy-back cache of the uniprocessor architecture).[10] In our experiments so far, the multiprocessor architecture has included both a sixteen-entry write buffer and a sixteen-entry prefetch issue buffer. To evaluate the performance impact of having a common buffer, we ran an experiment where both writes and prefetches were placed in a combined sixteen-entry buffer. Our results showed absolutely no difference in performance. This is partly because the lockup-free cache (which allows up to eight outstanding misses for the multiprocessor architecture) handles requests quickly enough that prefetches are rarely delayed behind writes. In an earlier study where we did not use a lockup-free cache [61], the performance advantage of having a separate prefetch issue buffer was also rather small. Therefore the choice of separate or combined buffers should be dictated by whichever is easier to implement, since both schemes offer similar performance.

Summary

Providing a lockup-free cache is essential to obtain the benefits of prefetching. For the architecture and applications we studied, most of the performance benefit was captured by allowing up to four outstanding misses. In some cases there is a performance advantage to providing additional prefetch request buffering beyond these four misses. Finally, combining prefetches and writes in the same buffer showed no significant loss in performance for our multiprocessor architecture.

5.2 Achieving Larger Gains through Prefetching

Having described the basic architectural support for prefetching in the previous section, we now consider techniques which may further improve the performance offered by prefetching. Although many of these techniques involve compiler support as well as architectural support, we have included them in this chapter because of their relevance to the architecture.

To provide a framework for examining all aspects of prefetching performance, let us briefly review the key performance issues. To begin with, since prefetching involves a *cost* as well as a benefit, maximizing overall performance requires both maximizing the latency-hiding benefit and minimizing the cost. Maximizing the latency-hiding benefit involves maximizing the *coverage factor* and maximizing the *effectiveness* of prefetches that are issued. Minimizing the cost involves avoiding *unnecessary prefetches* and minimizing the overheads of prefetches that are issued.

[10] The reason for this difference is that our uniprocessor architecture is based on the MIPS R4000 processor, while the multiprocessor architecture is patterned after DASH, which contains R3000 processors.

The goal of the *analysis* phase of the compiler algorithm is to maximize coverage and minimize unnecessary prefetches by predicting which dynamic references should be prefetched. We will consider ways to improve this decision-making process in Section 5.2.1. The other two goals of maximizing prefetching effectiveness and minimizing prefetching overheads are handled by the *scheduling* phase of the compiler. We will discuss ways to improve prefetching effectiveness in Section 5.2.2, and ways to minimize overheads in Section 5.2.3.

5.2.1 Improving Analysis

The analysis techniques presented so far have relied strictly upon *static information* that is available during compilation. This static analysis could be further improved in the following ways. First, the basic concept of locality analysis, i.e.

$$\text{Intrinsic Data Reuse} \cap \text{Localized Execution Space} \Rightarrow \text{Data Locality}$$

can be generalized to handle other types of reference patterns, such as scalars, pointers, lists, etc. For example, two pointer dereferences would have intrinsic data reuse if the compiler can determine that they point to the same address. The localized execution space for a given reference would be the set of paths in the control-flow graph that can lead to that reference without accessing enough data to flush the cache. If a given reference reuses data from an earlier reference, and that earlier reference falls within the localized execution space of the given reference, then there is data locality. Second, the analysis could be performed on a *global* level rather than a loop-nest level. The largest gains from improving static analysis are likely to come from prefetching large recursive data structures, such as the trees and linked lists we saw in BARNES and PTHOR in Section 4.5.2, and from performing interprocedural analysis, which was important for WATER (as also described in Section 4.5.2). However, analyzing recursive data structures may be quite difficult since it usually requires pointer analysis, which is a difficult problem for the compiler in general [37, 51].

Rather than relying strictly upon static information, another possibility is to make use of *dynamic information*. Dynamic information could be used either at compile-time, through the use of feedback information, or at run-time, by generating adaptive code. We will discuss both possibilities in this subsection.

Incorporating Feedback into Compilation

Dynamic information can be used at compile-time through feedback from earlier runs. The two types of feedback that may be useful for prefetching are *control-flow* feedback and *memory behavior* feedback, which we will briefly discuss.

Control-flow feedback (also known as "branch profiling") records how frequently each control path in the program is executed, thereby measuring branch outcome frequencies. Such information is often used in aggressive instruction-scheduling algorithms where it is important to schedule code beyond branches [75]. Control-flow feedback would be useful in our prefetching compiler algorithm (described in Chapter 2) when computing the *localized iteration space* for loop nests that include symbolic loop bounds. In these cases, the number of loop iterations is not a compile-time constant, and therefore it is difficult to predict whether the volume of data referenced by a loop exceeds the effective cache size. Using control-flow feedback, the compiler can take into account the average number of loop iterations of previous runs when making this decision. Control-flow feedback might also be useful for loops that contain conditional statements. For example, the number of iterations to software-pipeline prefetches ahead depends upon the expected number of instructions in a single loop iteration (see Section 2.4.2). If the loop contains a conditional statement, we currently schedule for the worst case (i.e. the shortest path). However, if the branch outcome almost always favors a much longer path, the compiler algorithm may want to schedule for this longer path instead. Control-flow information is relatively inexpensive to collect, since it simply involves instrumenting the code to count basic blocks. [11]

Feedback on *memory behavior* helps identify which references in the code are suffering the most from memory latency. This information can be collected at various levels of granularity. For example, the Mtool utility [28] collects information at the granularity of a single loop nesting. This is useful for identifying which loop nests suffer the most from latency, and which loop nests are insignificant. At the more fine-grained end of the spectrum, the feedback information might consist of precise miss rates for each reference in the program. One of the challenges of collecting any type of memory behavior information is that the behavior of the memory subsystem is usually hidden from the software. Mtool collects its information through either fine-grain timers or by sampling the program counter. Because of the potential distortion introduced by the large overheads of these techniques, it is only possible to collect information at the level of a loop nest. In order to collect individual miss rates in a program, it is necessary to either *simulate* the

[11] An example of this is the *pixie* utility provided by MIPS Computer Systems, Inc. *Pixified* code generally runs roughly half as fast as the original code.

memory system (which is rather costly), or else make use of user-visible hardware miss counters. However, few commercial microprocessors currently provide support for such miss counters, and therefore collecting individual miss rates remains a non-trivial task.

Memory behavior feedback may be useful in a number of different situations. First, it may be useful when the localized iteration space (LIS) concept would work, but the LIS has been chosen improperly—perhaps because the loop bounds are unknown, or because the effective cache size is incorrect. In this case, the LIS can be adjusted to better match the observed behavior. Second, memory behavior feedback can help detect cases where cache conflicts result in more misses than locality analysis predicts. Similarly, in a multiprocessor environment, it can help detect cases where coherence activity causes additional misses beyond those predicted by locality analysis. Finally, it will be useful for determining whether to prefetch references outside the scope of locality analysis, such as indirect references.

To experiment with feedback, we used our simulator to collect both control-flow and fine-grain memory behavior feedback information. This information was incorporated into the compiler algorithm as follows. The control-flow information was used to compute the average number of iterations for each loop (also known as the "trip count"). As we described earlier in this subsection, this count was used by the compiler to compute the localized iteration space whenever the loop bounds were unknown at compile-time.

The memory feedback information was used to annotate each load and store with both the number of times the reference was executed and the number of times it missed in the primary data cache—hence giving the precise miss rate. After performing locality analysis, we compare the *predicted* miss rate with the *observed* miss rate. If they agree within a certain acceptable margin, we schedule the code as usual. If they disagree, then we first try to find an explanation for the miss rate that is consistent with the intrinsic data reuse. This is important because to schedule the prefetches properly, we need to know *when* the misses occur.

For example, consider the two loop nests in Figure 5.12, and assume that from memory feedback we know the miss rate for loading `A[j][k]` to be 25% in both cases. Since the intrinsic data reuse for `A[j][k]` is quite different between the two cases, the interpretation of the miss rate also differs. For the code in Figure 5.12(a), where `A[j][[k]` has temporal reuse along the `i` loop (`k` is loop-invariant), a 25% miss rate corresponds to each load of `A[j][k]` missing on the first iteration of `i` and hitting on the remaining three iterations. Therefore the compiler would peel the `i` loop to schedule prefetches as normal for temporal locality. In contrast, the `A[j][k]` reference in Figure 5.12(b) has spatial rather than temporal reuse, and therefore we

(a)
```
for (i = 0; i < 4; i++)
  for (j = 0; j < 500; j++)
    A[j][k] = A[j][k] + foo(i);          /* A[j][k] has temporal reuse */
```

(b)
```
for (j = 0; j < 4; j++)
  for (k = 0; k < 500; k++)
    A[j][k] = A[j][k] + foo(i);          /* A[j][k] has spatial reuse */
```

Figure 5.12: Example of why intrinsic data reuse matters when scheduling prefetches even when miss rates are precisely known.

would not expect the same dynamic miss behavior. Assuming there are four `A[j][k]` elements per cache line, the 25% miss rate would correspond to missing on every fourth iteration of the inner loop (`k`) as cache line boundaries are crossed. So rather than peeling the outer loop, in this case the compiler would unroll the inner loop by a factor of four to schedule the prefetches.

This example in Figure 5.12 illustrates that the miss rate alone does not necessarily provide enough information to schedule prefetches properly, since fractional miss rates (i.e. between 0% and 100%) are ambiguous in terms of which dynamic references are hits and which are misses. This ambiguity arises because the information relating individual misses to when they occur is lost in the course of summarizing them as a single miss rate. Even after examining the data reuse within a loop nest, the dynamic misses may still be unclear. For example, the 25% miss rate for loading `A[j][k]` in Figure 5.12(a) may correspond to at least two different miss patterns. First, if the `A[j][k]` references are not already in the cache when the loop is executed, then all of the misses would occur during the first iteration of the `i` loop, as we described earlier. However, if the `A[j][k]` references were already in the cache (perhaps because they were referenced in an earlier loop nest), then the misses may have occurred sporadically across *all* of the `i` loop iterations, perhaps due to occasional cache conflicts with other references. Prefetches scheduled for the former case would not cover the misses of this latter case. Therefore the compiler is faced with choosing the most likely explanation for the observed miss rates.

The approach we take is to first look for explanations consistent with locality analysis by adjusting the LIS and repartitioning equivalence classes as necessary. For example, if the actual miss rate is lower than expected, the compiler checks whether increasing the size of the LIS (thereby increasing the expected locality) would explain the miss rate. Similarly, it considers

decreasing the LIS to explain higher-than-expected miss rates. This is particularly useful when the size of the LIS is unclear from static analysis alone. For example, given an 8 Kbyte primary data cache with 16 byte lines, it would be difficult to predict whether the LIS in Figure 5.12(a) included both surrounding loops since the amount of data referenced in a single iteration of the outer loop is very close to the cache capacity. With only static information, the compiler might normally predict that only the inner loop was within the LIS, and hence the expected miss rate of loading `A[j][k]` would be 100%. However, given the observed miss rate of 25% from feedback, the compiler would increase the LIS to include both loops. Therefore the `A[j][k]` reference would be predicted to have temporal locality, and its predicted miss rate would match the observed rate of 25%.

The compiler adjusts the partitioning of equivalence classes (sets of references predicted to have group locality) based on feedback as follows. First the compiler checks that the *leading reference* (i.e. the reference predicted to actually suffer the misses) of each equivalence class is the only one with a significant miss rate. If multiple references in the same equivalence class have significant miss rates, then the equivalence class is divided accordingly. If none of the references in the equivalence class has a significant miss rate (including the leading reference), then that equivalence class is marked as one that should not be prefetched. To determine which references should not be prefetched, the compiler ranks all references in descending order based on their contribution to total misses. A reference is not prefetched if it has a small miss rate *and* is ranked lower than the references accounting for the first 95% of the misses. This is a better approach than simply using a miss rate threshold, since it takes the *frequency of reference* into account. In other words, if a reference has a low miss rate but is executed so often that it accounts for most of the total misses, we still want to prefetch it. On the other hand, a reference with a higher miss rate that is rarely executed and therefore makes a negligible contribution to total misses can safely be ignored.

To evaluate the benefit of using feedback, we modified our compiler to automatically make use of control-flow and fine-grain memory feedback information. We simulated each application once without prefetching, using our simulator to automatically collect the feedback information. We then compiled each application a second time, this time using the feedback information. Figure 5.13 shows the performance of the one case that improved significantly using feedback: OCEAN.

The difficulty for static analysis in OCEAN is that the critical two-level loop nesting is split across separate files—the outer loop is in one file, and the inner loop is inside a procedure call

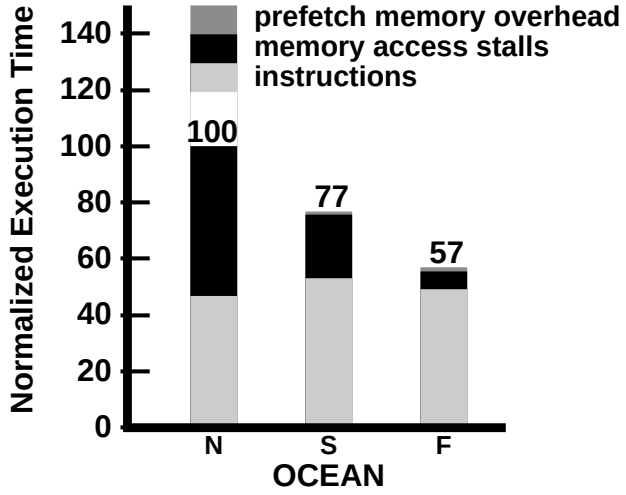


Figure 5.13: Results using feedback (**N** = no prefetching, **S** = prefetching with static analysis only, **F** = prefetching with feedback).

in another file. This loop performs a nearest-neighbor SOR computation, so there is a significant amount of group locality among the references. However, since our compiler does not perform interprocedural analysis across separate files, the prefetching algorithm does not recognize the group locality due to the outer loop, and therefore issues too many prefetches. Once feedback information is available, the compiler immediately recognizes the group locality, thus avoiding the unnecessary prefetches. Interestingly enough, eliminating prefetches actually reduces the memory stall time in this case by eliminating register spilling, since the spilled references were often conflicting with other data references. OCEAN illustrates how feedback may help the compiler overcome the fact that some types of analysis may be too expensive to implement in the compiler.

The main reason why feedback did not improve the other benchmarks is that static analysis alone was already quite successful for these types of codes. Profiling feedback is likely to be more useful in codes that are more difficult to analyze, such as ones containing pointers and recursive data structures. For the scientific codes we studied, however, the benefits of feedback do not appear to be large.

Adapting at Run-time

Although incorporating dynamic feedback information into the compilation process gives the compiler more information to reason with, it has a few shortcomings. First, the feedback process itself is a bit cumbersome, since it requires that the program be compiled twice. Also, depending on the level of detail of the information being collected, the process of executing the code to collect the information may be quite slow. Another difficulty is whether the dynamic profile that was captured is representative of all input data sets. This is a concern because optimizations tailored to one particular input data set may actually hurt performance on other data sets. In some cases it may not be possible to find a single "representative" data set, especially if the cache behavior depends critically upon whether the data set does or does not fit in the cache. This is particularly problematic when the problem size is determined at run-time, in which case it may be impossible to generate a single static piece of code that is appropriate for all input data sets. A similar problem arises in procedures such as library routines where it is impossible to analyze the calling arguments in all situations.

Rather than generating a single static piece of code that contains prefetching, another possibility is to generate code that adapts dynamically at run-time. This involves "specialization" of the prefetching code. The good news is that there are typically only a small number of different cases to specialize for—usually the data either should or should not be prefetched. Therefore when the compiler is uncertain about which case to generate, it could generate both cases, and choose the appropriate one to execute at run-time. The decision of which version to execute might be based on several different things: (1) the problem size (e.g., loop bounds defined at run-time), (2) data alignment in order to detect pathological cache conflict cases, or (3) hardware miss counters.

For example, Figure 5.14 shows a case where checking the problem size is useful. The temporal reuse of the `A[j]` reference in Figure 5.14(a) may or may not result in temporal locality, depending on how large `n` is relative to the cache size. Figure 5.14(b) illustrates how the value of `n` can be checked at run-time to select the appropriate strategy for prefetching `A[j]`.

Another problem that can potentially be detected dynamically is pathological cache interference between array references. When this occurs, the associated prefetches may also be ineffective, so it may be best to avoid the instruction overhead of issuing these useless prefetches. This situation can occur when two array references have similar affine access functions, such as references `A[i]` and `B[i]` in Figure 5.15(a), and when the difference in base addresses is a multiple of the cache size. Figure 5.15(b) illustrates code that adapts to this situation at run-time.

(a) Original Code

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    A[j] = A[j] + foo(i);          /* does A[j] have temporal locality? */
```

(b) Adaptive Code

```
if (n < THRESHOLD_SIZE) {           /* test if problem size is small */
  for (j = 0; j < 8; j += 2)        /* if so, A[j] has temporal locality... */
    prefetch(&A[j]); {               /* ... so only prefetch in the peel of i */
  for (j = 0; j < n-8; j += 2)
    prefetch(&A[j+8]);
    A[j] = A[j] + foo(0);
    A[j+1] = A[j+1] + foo(0);
  }
  for (j = n-8; i < n; i++)
    A[j] = A[j] + foo(0);
  for (i = 1; i < n; i++)       /* no prefetching in remaining iterations of i loop */
    for (j = 0; j < n; j++)
      A[j] = A[j] + foo(i);
}
else {                                /* no temporal locality for large problem size... */
  for (i = 0; i < n; i++) {           /* ... so prefetch A[j] on each iteration of i loop */
    for (j = 0; j < 8; j += 2)
      prefetch(&A[j]);
    for (j = 0; j < n-8; j += 2) {
      prefetch(&A[j+8]);
      A[j] = A[j] + foo(i);
      A[j+1] = A[j+1] + foo(i);
    }
    for (j = n-8; i < n; i++)
      A[j] = A[j] + foo(i);
  }
}
```

Figure 5.14: Example of adapting at runtime by checking problem size.

(a) Original Code

```
for (i = 0; i < 1000; i++)
  A[i] = B[i];              /* do A[i] and B[i] conflict in the cache? */
```

(b) Adaptive Code

```
diff = (&A[0] - &B[0]) % CACHESIZE;
/* test if A[i] and B[i] fall within the same cache line */
if ((diff > -LINESIZE) && (diff < LINESIZE)) {
  if ((diff > 0) {                      /* if so, test if A[i] leads B[i] */
    for (i = 0; i < 8; i += 2) {        /* if so, prefetch A[i] only... */
      prefetch(&A[i]);                  /* ... since it will be used first */
    }
    for (i = 0; i < 992; i += 2) {
      prefetch(&A[i+8]);
      A[i] = B[i];
      A[i+1] = B[i+1];
    }
    for (i = 992; i < 1000; i++)
      A[i] = B[i];
  } else {              /* prefetch B[i] only, since it will be used first */
    for (i = 0; i < 8; i += 2) {
      prefetch(&B[i]);
    }
    for (i = 0; i < 992; i += 2) {
      prefetch(&B[i+8]);
      A[i] = B[i];
      A[i+1] = B[i+1];
    }
    for (i = 992; i < 1000; i++)
      A[i] = B[i];
  }
} else {                /* prefetch both A[i] and B[i] if they do not conflict */
  for (i = 0; i < 8; i += 2) {
    prefetch(&A[i]);
    prefetch(&B[i]);
  }
  for (i = 0; i < 992; i += 2) {
    prefetch(&A[i+8]);
    prefetch(&B[i+8]);
    A[i] = B[i];
    A[i+1] = B[i+1];
  }
  for (i = 992; i < 1000; i++)
    A[i] = B[i];
}
```

Figure 5.15: Example of adapting at runtime by checking for data alignment conflicts.

(a) Original Code

```
for (i = 0; i < 1000; i++)
  sum = sum + A[i];          /* is A[i] already in the cache? */
```

(b) Adaptive Code

```
RESET_MISS_COUNTERS();        /* first reset hardware miss counters */
for (i = 0; i < 8; i += 2)    /* issue the first several prefetches */
  prefetch(&A[i]);
/* test whether the prefetches of A[i] have been hitting in the cache so far */
if (PREFETCH_MISS_COUNT() < SMALL_NUMBER) {
  for (i = 0; i < 1000; i++)      /* if so, stop prefetching */
    sum = sum + A[i];
} else {          /* otherwise, continue prefetching throughout remainder of loop */
  for (i = 0; i < 992; i += 2) {
    prefetch(&A[i+8]);
    sum = sum + A[i];
    sum = sum + A[i+1];
  }
  for (i = 992; i < 1000; i++)
    sum = sum + A[i];
}
```

Figure 5.16: Example of adapting at runtime by checking hardware miss counters.

Finally, Figure 5.16 illustrates how hardware miss counters can be used to dynamically detect whether data is already in the cache. These miss counters keep a running total of primary cache misses. It may be useful to have four miss counters, with each one devoted to counting one of the following types of misses: (i) loads, (ii) stores, (iii) prefetches, and (iv) exclusive-mode prefetches. Keeping separate counts is useful because it provides more information to the software. Also, there is a significant difference between seeing loads and stores hit in the cache (which is a good thing), and seeing prefetches hit in the cache (which is a bad thing). Miss counters should not be too expensive to implement since they only consume a small amount of chip area and should not affect cycle time.

As we see in Figure 5.16, the run-time overhead of using miss counters can be minimized as follows. First reset the counter before executing a loop. Then after executing some number

of loop iterations, stop and check the counter to see if the data has been in the cache so far. If so, do not prefetch the data for the remaining loop iterations. Otherwise, continue prefetching. Assuming there are a reasonable number of loop iterations (the software can also test for this), the relative overhead of this check should be quite small. One way to detect misses is to start off the first several iterations without prefetching, and then check to see how many load or store misses occur. The drawback of this approach is that it does not eliminate the miss latency of those first several iterations. Perhaps a better approach, as illustrated in Figure 5.16(b), is to start off issuing prefetches for the first several iterations, and then test to see whether the *prefetches* have been hitting in the cache. If they have, assume the data is already in the cache and do not issue further prefetches.

While Figure 5.16 shows how miss counters can be used within a single-level loop nest, their run-time overhead can be reduced even further by using them across outer loop iterations in multi-level loop nests. For example, Figure 5.17 demonstrates how miss counters can be used to adapt to temporal locality along an outer loop.

Having described how hardware miss counters can be used, we will now show experimental results for two different benchmarks: BCOPY, and LU. We start with BCOPY, which is a "block copy" library routine for copying a block of data from one location to another. Although BCOPY is a very simple routine, it is interesting for two reasons. First, since it is a library routine, the compiler cannot make any assumptions about the input parameters and cannot analyze the call sites for locality. Second, since BCOPY is frequently called by the operating system to move data around, improving BCOPY may have a significant impact on system performance. We rewrote BCOPY by hand to take advantage of hardware miss counters, similar to the code in Figure 5.16. We added hardware miss counters to our simulator, and made them visible to the software through special function calls. We used a simple workload to drive BCOPY, since we were mainly interested in testing the ends of spectrum. The workload consisted of a loop which repeatedly called BCOPY with two distinct arrays and a given block size as arguments. Since identical block copies are performed on subsequent iterations, there can potentially be a temporal locality benefit if the block can remain in the cache. Figure 5.18 shows the performance of BCOPY using various block sizes and loop iteration counts.

As we see in the "500x10" case (where BCOPY is called ten times with 500 byte blocks) in Figure 5.18, the original code without prefetching suffers a significant amount of miss latency. These misses occur the first time the routine is called, since the data remains in the cache for subsequent calls. As we see from the middle bar, the code which statically prefetches the blocks

(a) Original Code

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    A[j] = A[j] + foo(i);          /* does A[j] have temporal locality? */
```

(b) Adaptive Code

```
for (j = 0; j < 8; j += 2)          /* perform the first i loop iteration normally */
  prefetch(&A[j]);
for (j = 0; j < n-8; j += 2) {
  prefetch(&A[j+8]);
  A[j] = A[j] + foo(0);
  A[j+1] = A[j+1] + foo(0);
}
for (j = n-8; i < n; i++)
  A[j] = A[j] + foo(0);
RESET_MISS_COUNTERS();      /* reset the miss counters before the second i loop iteration */
for (j = 0; j < 8; j += 2)          /* perform the second i loop iteration normally */
  prefetch(&A[j]);
for (j = 0; j < n-8; j += 2) {
  prefetch(&A[j+8]);
  A[j] = A[j] + foo(1);
  A[j+1] = A[j+1] + foo(1);
}
for (j = n-8; i < n; i++)
  A[j] = A[j] + foo(1);
/* check whether A[j] remained in the cache during second i loop iteration */
if (PREFETCH_MISS_COUNT() < SMALL_NUMBER) {
  for (i = 2; i < n; i++)           /* if so, stop prefetching */
    for (j = 0; j < n; j++)
      A[j] = A[j] + foo(i);
}
else {          /* otherwise, continue prefetching */
  for (i = 2; i < n; i++) {
    for (j = 0; j < 8; j += 2)
      prefetch(&A[j]);
    for (j = 0; j < n-8; j += 2) {
      prefetch(&A[j+8]);
      A[j] = A[j] + foo(i);
      A[j+1] = A[j+1] + foo(i);
    }
    for (j = n-8; i < n; i++)
      A[j] = A[j] + foo(i);
  }
}
```

Figure 5.17: Example of adapting at runtime to temporal locality along an outer loop by checking hardware miss counters.

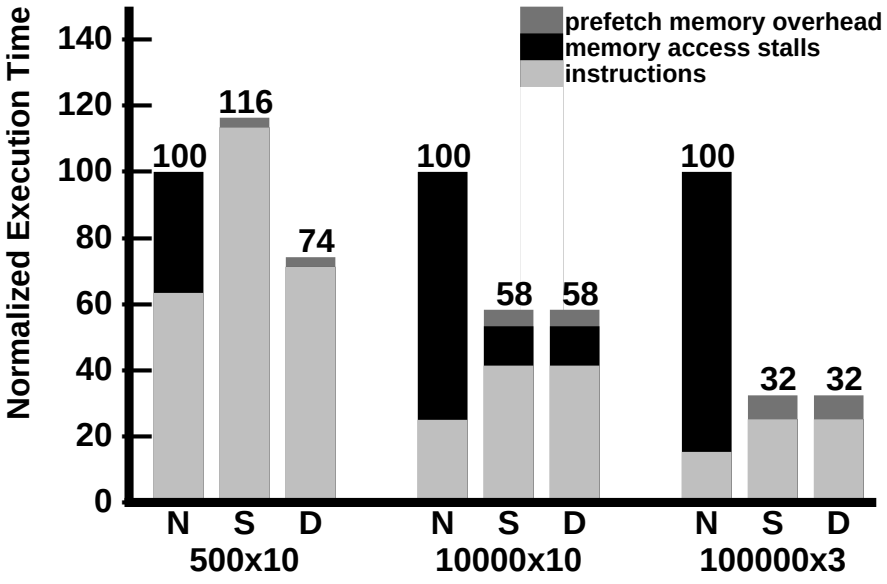


Figure 5.18: Results with adaptive version of BCOPY (**N** = no prefetching, **S** = statically prefetch all the time, **D** = adapt prefetching dynamically). "**BxT**" means the same **B**-byte block is copied to the same destination **T** times. Performance is renormalized for each case.

all the time actually performs worse than the original case, due to the large instruction overhead of the many unnecessary prefetches. The dynamically adaptive code (shown as the righthand bar) performs the best, since it recognizes that the data should be prefetched the first time, but not on subsequent calls. The other two cases in Figure 5.18 show larger block sizes that do not fit in the cache. In these cases it is best to prefetch all the time, so stopping to check the miss counters is pure overhead. However, we see that this overhead is small enough that it has no noticeable impact on performance. Therefore the adaptive code provides the benefit of eliminating unnecessary prefetch overhead without introducing any significant cost.

The other application we modified to exploit hardware miss counters was LU, which performs parallel LU-decomposition on dense matrices. In LU, the same procedure is called repeatedly to apply a pivot column to other columns. If the columns are large relative to the cache size, then it is best to prefetch the pivot column each time it is referenced. On the other hand, if a column can fit in the cache, then the pivot column will only suffer misses the first time it is referenced. However, since this code is in a separate procedure, and since our compiler does not perform procedure *cloning* [15], the only static option is to prefetch the column all the time. Once again,

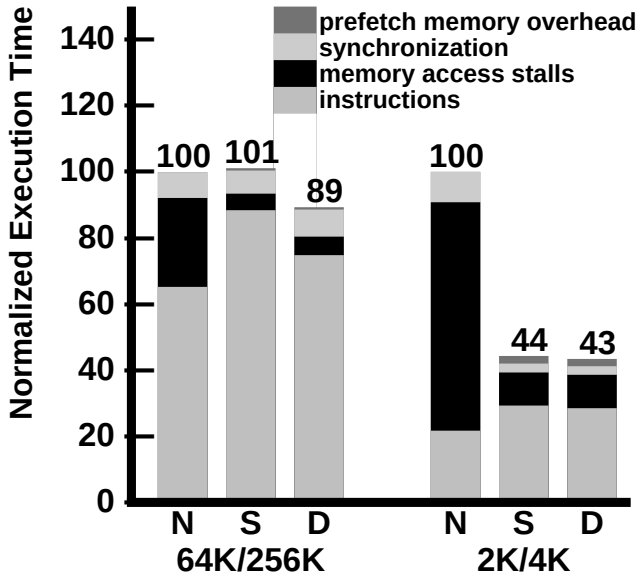


Figure 5.19: Results with adaptive version of LU (**N** = no prefetching, **S** = statically prefetch all the time, **D** = adapt prefetching dynamically). Performance is re-normalized for each cache size.

we modified this procedure by hand, similar to BCOPY, to use hardware miss counters. The results of our experiments with LU are shown in Figure 5.19.

We ran LU with both large and small problem-size to cache-size ratios. The lefthand side of Figure 5.19 shows the small ratio, meaning that columns tend to fit in the cache. In this case, the static approach of prefetching the columns all the time suffers from a significant amount of instruction overhead. In contrast, the adaptive code eliminates much of this unnecessary overhead while still hiding much of the memory latency, thereby resulting in the best overall performance. Looking at the righthand side of Figure 5.19, where the data size is large relative to the cache, we see the interesting result that the adaptive code still offers the best performance. This is because LU performs a triangular matrix solve, and therefore the last several columns always tend to fit in the cache. The savings of not issuing unnecessary prefetches for these last several columns more than offsets the instruction overhead of the dynamic test in this case. Therefore the adaptive code is clearly the best choice for LU.

Summary

We have demonstrated that profiling feedback can potentially help the compiler generate more effective prefetching code by giving it more information about the dynamic behavior of the application. However, using feedback in the compilation process can be somewhat time-consuming and clumsy, and also raises the issue of finding representative input data sets. Perhaps a better alternative for using dynamic information is to generate code that adapts at runtime. We have demonstrated that by exploiting user-visible hardware miss counters, the software can dynamically adapt to the cache behavior to achieve the best overall performance. Based on the success of using such miss counters in BCOPY and LU, it would appear that processor designers should seriously consider adding user-visible miss counters to their processor architectures. In addition to their benefit in adaptive code, these counters will also make it possible to collect detailed memory feedback information with very little overhead. From our own experience, this information is quite valuable for debugging memory performance, and is very useful when inserting prefetches into irregular codes (e.g., BARNES and PTHOR) where static locality analysis is extremely difficult.

5.2.2 Improving Effectiveness

Choosing the right references to prefetch is only one part of maximizing the prefetching benefit—making those prefetches effective at eliminating cache misses is also critical. The measure of success for effective prefetching is that all prefetched references should find their data in the cache. The first step toward this goal is timing the arrival of the prefetched data in the cache to maximize the likelihood of it being found there by the subsequent reference. Ideally, prefetches should arrive in the cache "just in time" to be referenced, since prefetches arriving too late obviously cannot prevent misses, and prefetches arriving too early are susceptible to being replaced by other references. The second step toward effective prefetching is avoiding excessive cache conflict problems that can render ineffective even prefetches that are scheduled the proper amount of time in advance.

The prefetching algorithm described so far addresses only the first of these two goals, and it does so by computing the proper number of iterations to software-pipeline prefetches ahead (see Section 2.4.2). This task is made difficult only by the variability in both memory access times and path lengths through the loop body. To be conservative, the compiler schedules for the largest expected memory latency (300 cycles in our experiments) and the shortest loop body path.

Therefore the prefetched data can potentially arrive quite early in the cache if it is found close to the processor. However, despite being conservative, the compiler is usually quite successful at scheduling prefetches effectively.[12] In the cases where prefetches are ineffective, the problem is caused more by excessive cache conflicts than by the timing of the prefetches. The intuitive explanation for this is that although prefetches may arrive early, the number of other lines that are brought into the cache during this interval is relatively small compared to the cache size. Therefore it is only likely that prefetched data will be displaced if the frequency of cache mapping conflicts is unusually high. To address this problem, we will now focus on techniques for coping with these cache conflicts.

Dealing with Cache Conflicts

Many commercial RISC microprocessors have direct-mapped primary caches. This is because direct-mapped caches have faster access times than set-associative caches, and the difference in speed often outweighs the difference in miss rate for general-purpose codes. General-purpose codes (e.g., the C compiler) have relatively few mapping conflicts on direct-mapped caches because their access patterns tend to be randomly distributed throughout the cache. In contrast, scientific codes that stride through matrices (particularly ones dimensioned to powers of two) can have pathologically bad mapping conflicts. In four of the applications we studied (MXM, CFFT2D, VPENTA, and TOMCATV), the cache conflicts in the original code were so bad that we manually realigned the data to help alleviate these problems.

The cache conflict problem can be addressed either in hardware or in software. One software-based approach is to place the burden of avoiding conflicts on the programmer. Once the programmer is aware that conflicts are a problem, the solution to the problem is often quite obvious. In the applications where we fixed conflicts by hand, we simply added thirteen (an arbitrary prime number) to the size of each matrix dimension that was a power of two. This was enough to prevent elements with similar access functions in adjacent rows or matrices from mapping to the same cache entries. Although this is fairly straightforward, ideally the programmer should not have to worry about this. A more appealing software-based approach would be for the compiler to automatically realign data to fix mapping problems. However, the difficulty here is that once the data are moved, every possible reference to the data must also be adjusted appropriately. This

[12] Scheduling prefetches the right amount of time in advance becomes much more difficult when software pipelining cannot be used (e.g., when traversing a linked list). In such cases, dependencies may make it quite difficult to move prefetches sufficiently far in advance to hide the latency.

may be particularly difficult given explicit address arithmetic, pointers, aliasing, or compilation across separate files. So although eliminating the problem in software may sound like the ideal approach, it is unclear whether this is a practical solution.

Cache conflicts can be addressed in the hardware through associativity of some form. While associativity has the advantage of reducing conflicts by allowing locations to map to multiple cache entries, it has the disadvantage of slowing down cache access rates due to added complexity. Therefore minimizing the degree of associativity is an important concern. One option is to have a set-associative primary cache, where addresses are mapped to N-way associative sets. Another approach is to keep the cache direct-mapped but also add a "victim cache" [40] off to the side. A victim cache is a small buffer containing the last several lines replaced from the cache. While set-associative caches are the most common approach for adding associativity, the victim cache approach is appealing because it is tailored to cases where data are reused shortly after being displaced—this is precisely what happens with prefetching conflicts.

We evaluate both of these techniques in this subsection by incorporating them into our uniprocessor architecture. The applications we consider are the ones that suffered most noticeably from conflicts, including two cases where prefetches were often ineffective due to conflicts (CHOLSKY and TOMCATV, as discussed in Section 3.2.3), and the four original codes before we manually realigned data (MXM, CFFT2D, VPENTA, and TOMCATV). The results of these experiments are shown in Figures 5.20 through 5.28.

Each graph shows curves both with and without prefetching (**PF** and **NOPF**, respectively) for victim caches ranging from one to 256 entries, and for set-associative caches ranging from 2-way to 8-way associativity. Two curves are shown for the set-associative cases: one with random replacement (**Random**), and one with a least-recently-used (**LRU**) replacement policy. The LRU policy is slightly more effective, but is typically more expensive to implement.

The success of victim caching and set-associativity varied across the applications, and we discuss each case individually below.

CHOLSKY: We begin with CHOLSKY (the uniprocessor NASA7 SPEC benchmark), as shown in Figure 5.20. CHOLSKY is an interesting case because cache conflicts only occur once prefetching is added. The conflicts occur in two similar loop nests, one of which is shown in Figure 5.21. The references `B(I,L,K+JJ)` and `B(I,L,K)` are separated by 8032(`JJ`) bytes (`JJ` is a loop index, and therefore its value changes). Since `JJ` is always greater than zero, the `B` references never directly conflict in an 8 Kbyte cache, and hence the original code does not have a cache conflict problem. As we see in Figure 5.20, the **NOPF** cases

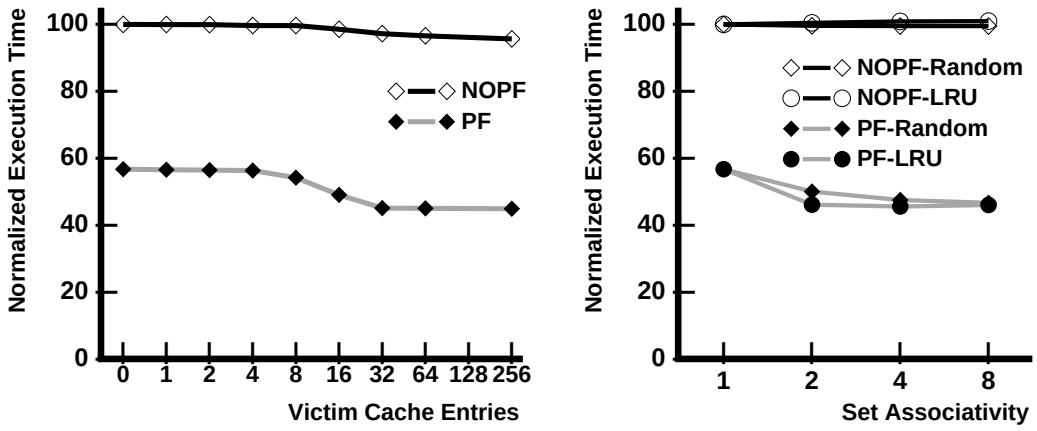


Figure 5.20: Performance of CHOLSKY with victim caches and set-associative primary caches.

```
do I = 0, NRHS
  do K = 0, N
    do JJ = 1, MIN(M, N-K)
      do L = 0, NMAT
        B(I,L,K+JJ) = B(I,L,K+JJ) - A(L,M-JJ,K+JJ) * B(I,L,K)
```

Figure 5.21: Loop that suffers cache conflicts in CHOLSKY.

show little improvement with associativity or victim caching.

To insert prefetches, the compiler unrolls loop `L` in Figure 5.21 four times for the spatial locality of the `A` reference, and prefetches are software-pipelined three iterations ahead. Therefore the `B` references are being prefetched twelve references ahead, which spans 384 bytes (i.e. twelve separate 32-byte cache lines). This region is large enough that the `B` references overlap when they are close enough together. What happens then is that the prefetch of `B(I,L,K+JJ)` displaces the prefetch of `B(I,L,K)`, and the load of `B(I,L,K)` displaces the prefetch of `B(I,L,K+JJ)`, thereby rendering both prefetches ineffective. These conflicts only occur for certain values of JJ, and they happen about 25% of the time.

As we see in Figure 5.20, a 2-way set-associative cache (with LRU replacement) is enough to eliminate this conflict problem. A victim cache only eliminates the problem when it is 32 entries deep. This is because we prefetch each reference twelve iterations ahead, and

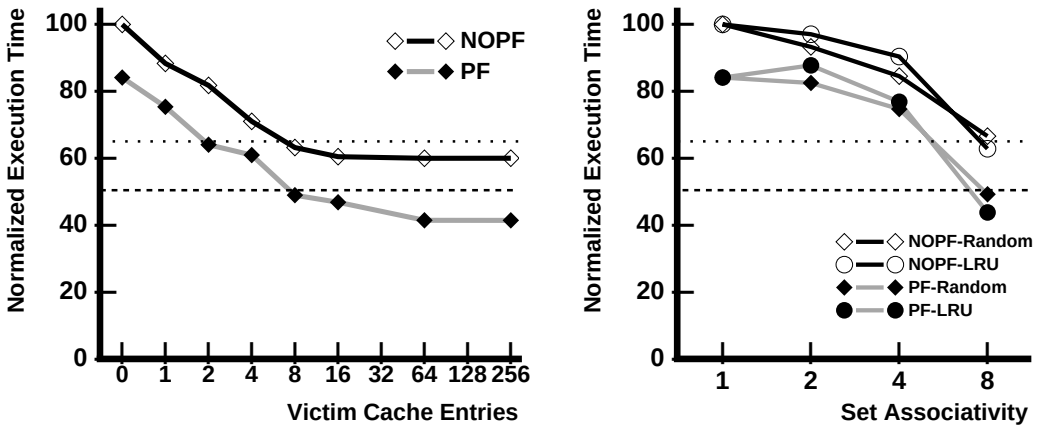


Figure 5.22: Performance of the original TOMCATV code with victim caches and set-associative primary caches. The performance of TOMCATV on the original direct-mapped architecture after arrays are manually realigned is shown by the dotted (no prefetching) and dashed (with prefetching) horizontal lines.

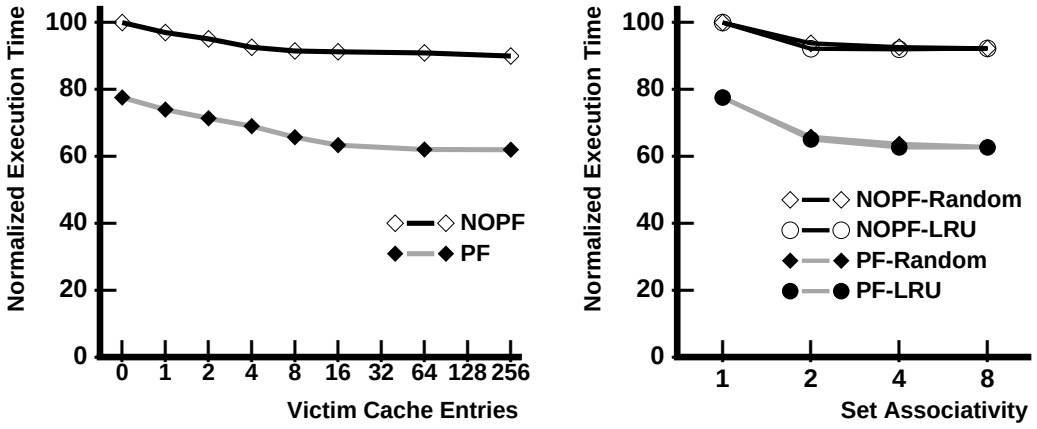


Figure 5.23: Performance of the realigned version of TOMCATV with victim caches and set-associative primary caches. Note that performance is normalized to the speed of this realigned code.

up to 24 conflicting misses must be captured. This type of conflict pattern is better suited to associativity.

TOMCATV: TOMCATV is a much more complicated case. The key loop in TOMCATV contains seven distinct references that conflict with each other at various times. References are

prefetched only three iterations ahead since the loop body is fairly large, and the conflict patterns are rather complex.

We begin by focusing on the original TOMCATV code, as shown in Figure 5.22. For each of these cases where we show original code before the arrays are manually realigned (i.e. TOMCATV, MXM, CFFT2D, and VPENTA), we also indicate the performance of the code *after* realignment using dashed and dotted horizontal lines, which correspond to the cases with and without prefetching, respectively, on the original direct-mapped architecture. We observe that either eight victim cache entries or an 8-way set-associative cache are needed to match the performance of simply realigning the arrays. Once the arrays are realigned, as shown in Figure 5.23, a 2-way set-associative cache is quite helpful, and there is steady improvement from increasing the victim cache size up to sixteen entries. Large degrees of both victim caching and set-associativity fare well in the case of TOMCATV, but clearly realigning the arrays is the most important optimization.

MXM: The original MXM code is a very interesting case, and the resulting performance is shown in Figure 5.24. This is a partially-blocked matrix multiply, and the key loop is shown in Figure 5.25. The conflicts in this code occur between the `A` and `C` matrices. For both matrices, the size of the inner dimension is 2 Kbytes (256 double-precision elements). In a single pass through the `I` loop, each of the four `A` references will sweep through their own quarter of the cache, never conflicting with each other. However, the `C` reference will line up directly with one of the `A` references (it rotates between each of them). Without these conflicts, the `A` and `C` references would suffer misses once every four references, as they cross cache line boundaries. These conflicts will cause the `C` reference and one of the `A` references to miss on each iteration. Increasing the associativity to 2- or 4-ways does not improve performance, since the `C` and the four `A` references will always map into the same set. In fact, the performance is dramatically *worse* with a 4-way LRU cache because the code cycles repeatedly through 5 references, so the least recently used reference is the worst one to throw out. The code finally improves with 8-way set-associativity since all five conflicting references can fit in the same set.

Unlike associativity, victim caching performs very well in this case, as we see in Figure 5.24. A single victim entry allows all five references to remain in the cache, thereby dramatically improving the **NOPF** case. The prefetching case is fetching data two iterations ahead, and does somewhat worse than **NOPF** with a single victim entry because it replaces the crucial

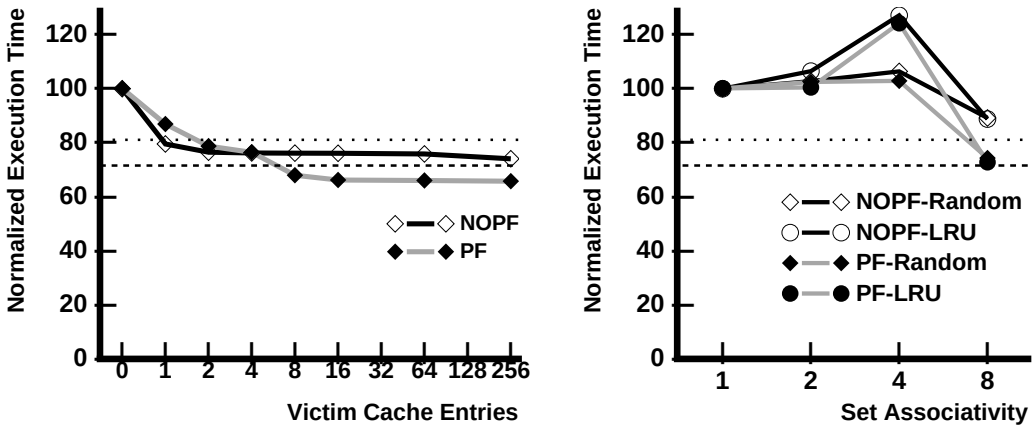


Figure 5.24: Performance of the original MXM code with victim caches and set-associative primary caches. The performance of MXM on the original direct-mapped architecture after arrays are manually realigned is shown by the dotted (no prefetching) and dashed (with prefetching) horizontal lines.

```
do J = 1, M, 4
  do K = 1, N
    do I = 1, L
      C(I,K) = C(I,K) + A(I,J)*B(J,K) + A(I,J+1)*B(J+1,K)
               + A(I,J+2)*B(J+2,K) + A(I,J+3)*B(J+3,K)
```

Figure 5.25: Loop that suffers cache conflicts in MXM.

victim entry with data displaced by prefetching. With eight or more victim cache entries, we finally see the full prefetching benefit, since both the current and the prefetched data sets can be captured. We also see that by restructuring the code by hand, cache conflicts can be avoided altogether, and this matches the **NOPF** performance with a single victim cache entry. The performance of the restructured code with prefetching is only exceeded by eight or more victim entries, where it is possible to avoid occasional conflicts with other references.

CFFT2D: In the original CFFT2D code, the conflicts occur between references of the `X` matrix in two separate loop nests, shown in Figure 5.27. The inner dimension of the `X` matrix contains 128 complex numbers, which is 2 Kbytes of data. In the code without prefetching, conflicts occur only in the second loop nest (Figure 5.27(b)) whenever the difference between `II`

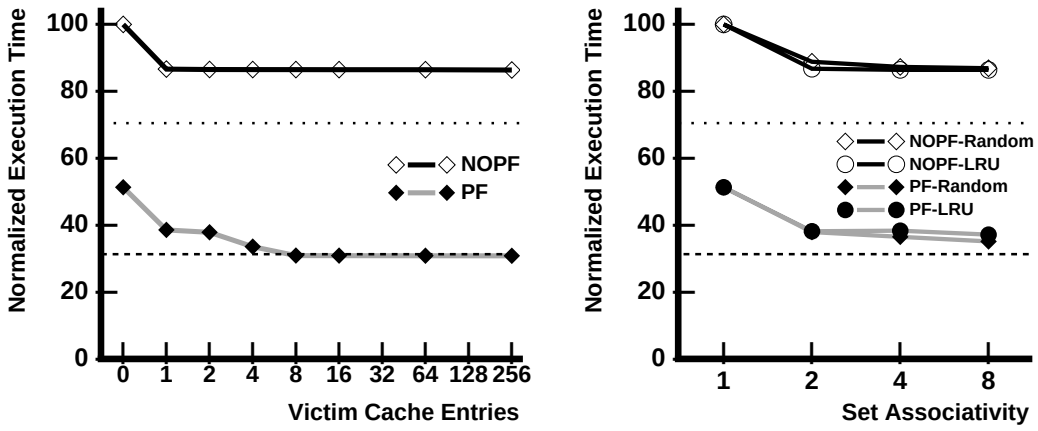


Figure 5.26: Performance of the original CFFT2D code with victim caches and set-associative primary caches. The performance of CFFT2D on the original direct-mapped architecture after arrays are manually realigned is shown by the dotted (no prefetching) and dashed (with prefetching) horizontal lines.

(a)

```
do K = 1, N
  CT = X(II,K) - X(IM,K)
  X(II,K) = X(II,K) + X(IM,K)
  X(IM,K) = CT * CX
```

(b)

```
do K = 1, N
  CT = X(K,II) - X(K,IM)
  X(K,II) = X(K,II) + X(K,IM)
  X(K,IM) = CT * CX
```

Figure 5.27: Loops that suffer cache conflicts in CFFT2D.

and `IM` is a multiple of four (which occurs more than 50% of the time). Either a single victim cache entry or a 2-way set-associative cache fixes this problem, as we see at the top of Figure 5.26. This does not match the performance of the restructured code, because it still suffers from conflicts in the direct-mapped secondary cache (a 2-way set-associative secondary cache should fix this).

In the prefetching case, this second loop is unrolled by a factor of two and we prefetch three iterations ahead. When the references line up, obviously the prefetches will also interfere with each other. Consequently the prefetch of `X(K,IM)` was often replaced by the prefetch and reference of `X(K,II)` before it could be used. However, the more important conflict cases with prefetching are in the first loop nest (Figure 5.27(a)). Here we prefetch six iterations ahead along the outer dimension. Since each of these accesses is separated by 2 Kbytes, they map into only four entries in the 8 Kbyte direct-mapped cache, so each reference ends up displacing its own prefetches. Six or more victim cache entries are enough to capture these conflicts, and we see a noticeable performance improvement in Figure 5.26 with eight entries. An eight entry victim cache does better than 8-way set-associativity in this case because all of the prefetches map into the same set, and the replacement policy is not perfect. The manually-realigned code with a direct-mapped cache performs better than an 8-way set-associative cache and as well as a victim cache with eight or more entries.

VPENTA: Finally, there is the VPENTA code shown in Figure 5.28. The critical loop nest for this application contains eight matrices which line up directly on top of each other in a direct-mapped cache, as shown in Figure 5.29. Each 2-D matrix is 128 Kbytes large, and the 3-D `F` matrix behaves like three large 2-D matrices. So ten unique references line up in the same entry of the cache on each loop iteration. Consequently the performance of this code is limited almost entirely by the degree of associativity, up to an 8-way set-associative cache or an 8-entry victim cache. Prefetching offers little improvement for these configurations because there is nowhere for the prefetched data to go. Finally, with a victim cache of sixteen or more entries, some of the prefetched data is retained, and we see a widening performance gap. The performance levels off at 64 entries because we prefetch two iterations ahead, so the working set is slightly larger than 32 lines, including the various stack references in the loop. Even with a 256 entry victim cache, the original code with prefetching does not match the restructured code without prefetching. This is because of the large number of secondary cache conflicts (the 256 Kbyte secondary cache would need to be at least 5-way set-associative to avoid conflict misses). By simply increasing the values of the matrix dimension parameters from 128 to 141, the original code without prefetching runs nearly three times faster. When this is combined with prefetching, the code runs five times faster on this direct-mapped cache architecture.

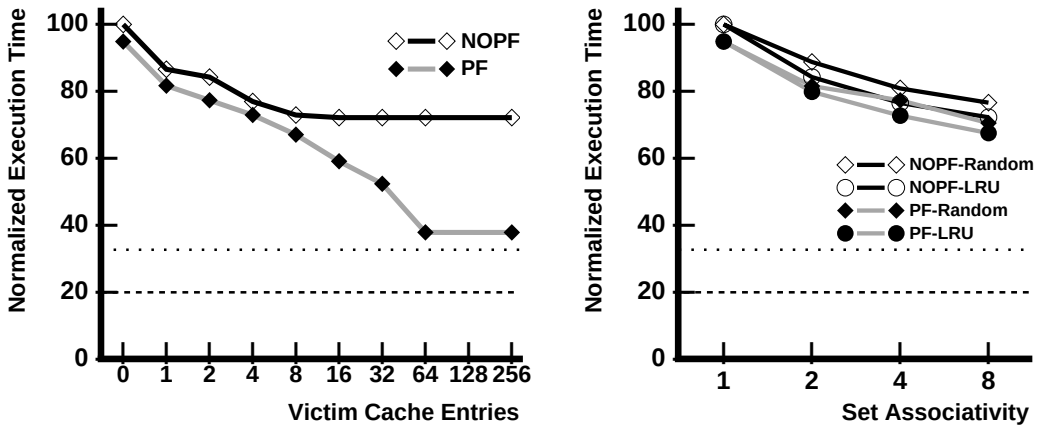


Figure 5.28: Performance of the original VPENTA code with victim caches and set-associative primary caches. The performance of VPENTA on the original direct-mapped architecture after arrays are manually realigned is shown by the dotted (no prefetching) and dashed (with prefetching) horizontal lines.

```
do J = JL+2, JU-2
  do K = KL, KU
    RLD2 = A(J,K)
    RLD1 = B(J,K) - RLD2*X(J-2,K)
    RLD = C(J,K) - (RLD2*Y(J-2,K) + RLD1*X(J-1,K))
    RLDI = 1.D0/RLD
    F(J,K,1) = (F(J,K,1) - RLD2*F(J-2,K,1) - RLD1*F(J-1,K,1))*RLDI
    F(J,K,2) = (F(J,K,2) - RLD2*F(J-2,K,2) - RLD1*F(J-1,K,2))*RLDI
    F(J,K,3) = (F(J,K,3) - RLD2*F(J-2,K,3) - RLD1*F(J-1,K,3))*RLDI
    X(J,K) = (D(J,K) - RLD1*Y(J-1,K))*RLDI
    Y(J,K) = E(J,K)*RLDI
```

Figure 5.29: Loop that suffers cache conflicts in VPENTA.

To summarize these results, we have seen a mixture of effects. In cases like CHOLSKY where a small number of references conflict only because their prefetched data sets overlap, a small amount of set-associativity is the best approach. Victim caches do not perform as well in these cases where the depth of software pipelining is large, since the replaced data must be held across that many iterations (i.e. there is a multiplicative effect of degree of conflict times pipelining distance). In other cases such as MXM where the conflicting data maps into the same sets in a set-associative cache, victim caching is better suited to the problem. In other cases, the

same degree of associativity (either N-way set-associativity or an N-entry victim cache) works equally well.

In a few cases we noticed two dips in the performance curves. The first dip corresponds to holding the "active" working set in either the primary cache or the victim cache (e.g., two victim entries in MXM and CFFT2D), and the second dip occurs when the prefetched active set can also be held at the primary cache level (e.g., eight victim entries in MXM and CFFT2D). When conflicts already occur (e.g., the original VPENTA code), prefetching may be doomed since the prefetched data will also conflict. However, prefetching may cause new conflicts, especially when striding through arrays that are only slightly separated in the cache.

Since prefetch-specific conflicts occur within a certain window of time, the victim cache approach sounds appealing, since it is tailored to capturing recently displaced data. However, with latencies of over a hundred cycles, more than sixteen victim cache entries may be needed, which is probably too large for an associative lookup.

In all cases, the software-based restructuring approach did quite well. Our restructuring algorithm was simply to add a prime number (thirteen) to the size of array dimensions that were a power of two. While care must be taken with this approach, it worked quite well and has the added benefit of reducing conflicts in the secondary cache. In one case (VPENTA), no reasonable amount of hardware could solve the cache conflict problems.

Prefetching into a Separate Target Buffer

Earlier in Section 5.1.3, we discussed the possibility of placing prefetched data in a separate target buffer, rather than the normal cache hierarchy. We suggested that as a general policy, this was not a good idea. However, there is one case where this may make sense, which is when prefetched data is only used once (i.e. it has *no* locality). When data has no locality, it will not be reused after it is brought into the cache. If a large amount of data with no locality is placed in the cache, it may displace other data that would have a locality benefit. Therefore one possibility is to issue *uncached prefetches* whenever the compiler determines that a reference has no locality.

Using uncached prefetches has several complications. First, if the compiler incorrectly predicts that a reference has no locality when it does in fact have locality, the performance will suffer as a result. For example, the `A[j][0]` reference in Figure 5.30 may or may not have temporal locality, depending on the size of `n`. If the compiler assumes `n` is large and therefore `A[j][0]` has no locality, it should realize that using uncached prefetches for `A[j][0]` will hurt performance if `n` turns out to be small. Therefore uncached prefetches should only be used

```
for (i = 0; i < 1000; i++)
  for (j = 0; j < n; j++)
    A[j][0] = A[j][0] + foo(i);
```

Figure 5.30: Example where it is not clear whether to use uncached prefetches.

when the compiler is certain that a reference has no locality.

The second complication, as we mentioned earlier in Section 5.1.3, is the hardware complexity of building the separate target buffer. Given this complexity, it is unclear whether it would be worth building such a structure even to support uncached prefetches.

Prefetching Set Hints

In architectures with set-associative caches, a more attractive technique for preventing data that streams through the cache from displacing other useful data may be a prefetching "set hint" that specifies the set in which prefetched data should be placed. For example, in blocked matrix algorithms, it is desirable for the blocked data to remain in the cache, and not be displaced by the non-blocked data. This could be accomplished by prefetching the blocked data into set 0 of a two-way set-associative cache, and the non-blocked data into set 1. Similarly, if the operating system wished to perform a large block copy operation that would normally flush the entire cache, it could instead prefetch the data only into set 1, thus leaving set 0 intact.

These new prefetching hints might be referred to as "retained" and "streamed" prefetches, which would correspond to placing data in particular subsets of a set-associative cache (e.g., "retained" prefetches go into set 0, and "streamed" prefetches go into set 1). Normal prefetches (i.e. without either of these hints) and loads and stores would use the normal set replacement algorithm to decide where data should be placed.

One advantage of prefetching set hints is that they require no complexity beyond a normal set-associative cache. Therefore the clock rate will not be affected, and the normal coherence mechanism will ensure that prefetches are non-binding in a multiprocessor environment. Another important advantage is that in the default case, the entire cache area can be utilized by any types of references. This is contrast with having a special prefetch target buffer, where normal loads and stores can never utilize the cache area devoted to the target buffer. Thus prefetching set hints provide the flexibility to partition the cache storage area only in cases where the programmer or

compiler has a strong reason to believe that doing so is beneficial.

5.2.3 Reducing Overheads

Since software-controlled prefetching has a cost as well as a benefit, care must be taken when inserting prefetches that the cost does not offset much of the latency-hiding benefit. The first step toward minimizing cost is prefetching *selectively* to avoid the pure overhead of unnecessary prefetches. Our results in Sections 3.2.1 and 4.3.1 demonstrate that selective prefetching can reduce much of the prefetching overhead, and we discussed ways to improve this analysis further in Section 5.2.1. While the remaining overhead after selective prefetching is typically quite small in comparison with the reduction in memory stall time, there are still a few cases where additional speedups of at least 10% could be achieved if it was possible to eliminate the remaining instruction overhead. In this section we will address the second step toward reducing prefetching cost, which is minimizing the instruction overhead of the prefetches that are issued.

Before we begin this discussion, let us consider how future trends are likely to affect the relative importance of prefetching instruction overhead. The first relevant trend is that the gap between processor and memory speeds will continue to grow. As this occurs, the cost of even the current level of instruction overhead will diminish relative to the latency-hiding benefit of each useful prefetch. The second important trend is continued improvements in the ability of processors to exploit instruction-level parallelism through techniques such as *superscalar* processing [75]. Since prefetch instructions can always be executed in parallel with other operations (because no other operations depend upon their completion), they should benefit well from the exploitation of instruction-level parallelism. Therefore the absolute overhead of processing prefetch instructions is likely to decrease. The combined effect of both of these trends is that prefetch instruction overhead should become less significant in the future.

Given these trends, why do we care about prefetch instruction overhead at all? The first reason is that although prefetch instructions can theoretically be executed in parallel with other operations, this will only result in no overhead if there are resources available for executing the prefetches that are normally idle. However, the functional units needed to compute prefetch addresses and issue prefetches will also be busy handling normal loads and stores. Due to competition for these critical resources, it is unlikely that prefetch instruction overhead will be completely hidden. The second reason is that prefetch instruction overhead is an inherent problem in applications where there are few instructions between cache misses. For these applications, the difference of only a single instruction per prefetch can result in a large fractional increase in

total instructions. For example, consider CHOLSKY in Figure 3.1, where selective prefetching increases the instruction count by roughly 50%. In this case the analysis is nearly perfect—only 9% of prefetches are unnecessary, and the miss coverage is 97%. The large instruction overhead is because cache misses occur rather frequently (once every 11 instructions), and issuing each prefetch requires several instructions (5, on average). Eliminating only a single instruction per prefetch would decrease the instruction count by roughly 10% in this case.

Therefore, since the instruction overhead of useful prefetches may be a concern in some cases but is probably not a major hindrance in general, we will discuss techniques for reducing this overhead only briefly in this section.

Avoid Spilling Registers

The total instruction overhead depends not only on how frequently prefetches are issued, but also on how many instructions are necessary to issue each prefetch. Ideally only a single instruction would be needed for each prefetch—the prefetch instruction itself. However, Table 3.6 shows that several additional instructions are often needed to generate prefetch addresses. In some cases, as many as 20 extra instructions were added for each prefetch. From our experience with the SUIF compiler, the primary cause for these large overheads is *register spilling*. Register spilling occurs whenever the register allocator runs out of registers, and therefore must "spill" values by saving and restoring them from memory. Once register spilling occurs, the instruction count within a loop body can increase dramatically.

One of the keys to avoiding register spilling is to reuse addressing registers between prefetches and loads and stores whenever possible. This works because prefetch addresses are often separated by a constant distance from load or store addresses. This difference can be folded into the constant offset field in the "base-plus-offset" addressing mode. Therefore the prefetch can be issued without consuming any additional registers, as we illustrated earlier in Figure 5.2. The importance of this optimization was demonstrated during our early experience with the compiler, when the scalar optimizer had not yet implemented this optimization. The instruction overhead was quite large in this early code because of the large amount of register spilling. Once the scalar optimizer began to exploit these common base registers, the overheads were dramatically reduced in many cases.

The second potential cause of register spilling is the loop unrolling which our algorithm performs to exploit spatial locality. Once a loop is unrolled, the compiler can optimize across the several replicated copies of the loop body. In most cases this allows the compiler to reduce

the overall instruction count for the following reasons: (i) branch instructions can be eliminated between unrolled iterations; (ii) it is more likely that "hazard" slots after multi-cycle operations can be filled with independent instructions; and (iii) register allocation can be optimized across the unrolled iterations, possibly eliminating loads and stores. One such example is MXM, where the savings through loop unrolling resulted in fewer instructions in the code with prefetching than in the original code (see Figure 3.1). On the other hand, the potential downside of loop unrolling is creating too many intermediate values for the register allocator to handle, thus resulting in register spilling. This occurred in the uniprocessor version of OCEAN, where each prefetch ended up costing 18 instructions. This number is so high because register spilling occurs for a large number of values in the loop, and these additional instructions are averaged over only the small number of prefetches. The result of this spilling is that OCEAN's instruction overhead is noticeably large despite the fact that misses occur infrequently (only once every 46 instructions). One way for the compiler to avoid these register-spilling problems is to perform loop unrolling with a greater awareness of its effect on register pressure. In our algorithm, we only suppressed loop unrolling to avoid code explosion, and did not take register pressure into consideration.

Although these techniques for avoiding register spilling are strictly compiler-based, we will now discuss ways to further reduce instruction overhead which also require architectural support.

Block Prefetches

Given that each prefetch brings a single line of data into the cache, the minimum amount of prefetch overhead (assuming perfect miss coverage) is one additional instruction for each cache miss. To reduce the overhead below this, the prefetch instructions must specify larger amounts of data to be fetched. For example, a prefetch could be defined to fetch multiple consecutive cache lines, rather than just a single cache line. We will refer to these multi-line prefetches as "block prefetches".

Block prefetching is advantageous when there is spatial locality. In such cases a single block prefetch supplants the multiple prefetches for the individual cache lines. For example, if a block prefetch fetches four cache lines, it can potentially eliminate up to 75% of the prefetch instruction overhead. Although block prefetches may be helpful when there is spatial locality, they may hurt performance in the absence of spatial locality by displacing useful data and wasting memory bandwidth. This negative effect has been observed in previous studies where additional consecutive cache lines were automatically prefetched by the hardware [64]. Another potential downside is that bringing the additional lines into the cache earlier increases their chance of being

displaced before use.

Therefore, rather than defining all prefetches to be block prefetches, a more attractive approach is for the compiler to intelligently select between using *single-line* prefetches and *block* prefetches based on whether the associated references enjoy spatial locality. Figure 5.3 illustrated how both types of prefetches could be encoded in the instruction set architecture through the prefetching hint field. Incorporating block prefetches into the compiler algorithm in this manner is straightforward—block prefetches are used whenever there is spatial locality, and single-line prefetches are used otherwise. The compiler adjusts the line size parameter to match the block size when it schedules block prefetches, hence modifying the modulo factors in prefetch predicates involving spatial locality accordingly. Note that as block prefetching increases the number of iterations between prefetches, it will eventually become more attractive to use *strip mining* [64] (as described earlier in Section 2.4.1) rather than *unrolling* to do the loop splitting, since the negative effects of loop unrolling dominate once they are unrolled too many times.

Block prefetching is a straightforward extension of normal prefetching. The main hardware complexity of supporting it is that the cache controller must handle single requests which fetch multiple lines. While block prefetches may help to reduce instruction overhead, it will only do so by at most the ratio of the larger prefetch block size to the normal cache line size. Given that large block sizes may hurt performance by causing additional primary cache conflicts, this ratio is likely to be relatively small.

Programmable Streams

To surpass the overhead reduction offered by block prefetches, the final step would be eliminating the "per iteration" instruction overhead altogether. The two previous proposals which accomplish this are (i) issuing large block prefetches for the entire data structure outside the loop (as illustrated in Figure 5.31, or (ii) using strictly hardware-based prefetching. However, both of these approaches have drawbacks, which we will briefly discuss.

The large block prefetch approach was studied by Gornish *et al.* [31, 32] in the context of a software-coherent shared-memory multiprocessor with binding prefetches. The potential problem of large block prefetches is that data may arrive in the cache *too early*, thereby exposing it to possible replacement before it can be used. In addition, the bursts in network traffic caused by large block requests can lead to queueing delays. We observed these negative effects when using large block prefetches in LU during an earlier study [61]. While hardware-based prefetching schemes can stream data into the cache more evenly, they suffer from a number of disadvantages

(a) Original Code

```
for (i = 0; i < 10000; i++)
  sum += A[i];
```

(b) Code with Large Block Prefetch

```
block_prefetch(&A[0],10000);          /* prefetch entire A array */
for (i = 0; i < 10000; i++)
  sum += A[i];
```

Figure 5.31: Example of how instruction overhead can be eliminated by issuing large block prefetches outside the main loop.

which we will describe in detail later in Section 5.3.1.

To improve upon both of these techniques, we introduce the notion of *programmable streams*. With programmable streams, the software decides what to prefetch and when the prefetches should be issued. Similar to the "large block prefetch" approach, this information is provided only once outside the loop, so there is no per-iteration instruction overhead. However, in contrast to the "large block prefetch" approach, where all data is fetched at once, programmable streams provide a mechanism for "flow control" so that data can be streamed into the cache at a rate matching the computation. This flow control is accomplished by associating a "trigger instruction" address with each stream. A trigger instruction can be any normal instruction in the loop, provided that it is executed once per loop iteration. Once a programmable stream has been initialized by the software, the hardware issues the next prefetch in the stream each time the trigger instruction is executed. Therefore the data arrives in the cache in the same manner as with software pipelining, but with essentially no instruction overhead.

To initialize a programmable stream, the software must specify at least the following: (i) a starting prefetch data address, (ii) the difference between consecutive prefetch data addresses, and (iii) a trigger instruction address. The values are stored in a hardware structure representing the stream. The only value which changes is the "current prefetch data address", which is initially the starting prefetch address, but is updated after each new prefetch is issued upon a matching trigger instruction. Once the loop completes, the software may choose to deallocate the stream to free the corresponding hardware entry for later use.

Further refinements of this basic approach may be achieved by having the software provide more information to the hardware. For example, the prefetch block size may be specified to make

the software more portable across different cache configurations. Also, to optimize for cases with spatial locality, the software could specify the number of iterations between prefetches, and the hardware could account for this by maintaining a modulo counter with each stream. Finally, given that the hardware knows the size of memory latency, the hardware could compute the number of iterations to pipeline ahead and automatically issue the prolog prefetches if the software specifies the number of instructions in the loop body.

Summary

In general, instruction overhead is not likely to be a significant problem. To prevent it from becoming a problem, the compiler should be careful to avoid register spilling problems. For codes that benefit from spatial locality, block prefetches may help reduce instruction overhead. Almost all of the instruction overhead of dense-matrix prefetches could be eliminated through programmable streams, although the corresponding hardware support is much more significant.

5.3 Alternative Latency-Hiding Techniques

Software-controlled prefetching is only one of several different approaches for coping with latency. We discussed a number of these techniques in Section 1.2, and in Section 3.4 we demonstrated that *locality optimizations* and prefetching are complementary. While locality optimizations are strictly a compiler-based technique, we will now consider several other techniques that require architectural support to see how they compare and interact with software-controlled prefetching. We begin in Section 5.3.1 by comparing *hardware-controlled* prefetching with *software-controlled* prefetching. Next, we will discuss *relaxed memory consistency models* in Section 5.3.2, which are primarily useful for hiding write latency in multiprocessors. Finally, we evaluate *multithreading* in Section 5.3.3, which is a technique for hiding latency by exploiting parallelism across multiple threads of execution.

5.3.1 Hardware-Controlled Prefetching

While software-controlled prefetching requires support from both hardware and software, several schemes have been proposed that are strictly hardware-based. Porterfield [64] 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 [53] 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 [7] proposed a scheme that uses a history buffer to detect constant-stride access patterns. In their scheme, a "lookahead PC" speculatively walks through the program ahead of the normal PC using branch prediction. When the lookahead 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 encouraging results.

To compare hardware-controlled prefetching with software-controlled prefetching, we will discuss how hardware-controlled prefetching addresses the three goals introduced in Section 5.2—namely performing analysis, maximizing effectiveness and minimizing overheads associated with prefetching.

Analysis

The goal of the *analysis* phase of prefetching is to decide which references should be prefetched. With the software-based approach, this analysis is performed by considering all references in the code and deciding whether or not they need to be prefetched. If so, the prefetches are scheduled by moving them back in time within the code. The hardware, however, does not have the luxury of examining the entire program and then moving prefetches back in time. The main challenge of deciding what to prefetch in hardware is predicting what locations will be referenced in the future. In the various hardware-based prefetching schemes that have been proposed, this prediction is achieved in one of three ways: (i) assume that there is abundant spatial locality (as in the schemes Porterfield studied [64]); (ii) decode ahead in the instruction stream (as in Lee's proposal [53]); or (iii) maintain a history of past access patterns (as in Baer and Chen's proposal [7]). The idea behind the first technique is that whenever a cache line is accessed, neighboring cache lines will be accessed in the near future, so they should also be brought into the cache. The problem, of course, is that this is not always true, and fetching lines unnecessarily can hurt performance both by displacing useful data and by causing memory queueing delays. The second technique attempts to predict access patterns by simply "getting ahead" in the instruction stream. However, the limitations of buffering capacity and branch prediction accuracy make it difficult to decode far enough ahead to hide large latencies. Finally, by maintaining history information, the hardware attempts to recognize constant-stride access patterns and prefetches ahead whenever those same instructions are executed. This last scheme appears to be the best of the three hardware-based schemes since it can prefetch constant-stride access patterns with arbitrary stride lengths (unlike the "long cache line" schemes which only exploit unit-stride patterns) without the expense of

decoding ahead in the instruction stream (as in Lee's scheme). However, the analysis capability of a history-based scheme has the following limitations: (i) it can only recognize constant-strides accesses (unlike our compiler which can prefetch indirect references); (ii) it can only react to historical behavior (unlike the compiler, which can anticipate future misses); and (iii) the detection scheme depends upon the history table being large enough to contain the entire working set of references, which may not be possible (particularly given that this structure is *content-addressable* in order to look up the address of the lookahead PC).

So far we have described how the hardware predicts which locations will be *referenced* in the future. In contrast with the software-based approaches, the hardware typically does not bother to predict whether or not the data is already in the cache.[13] Instead, for each reference the hardware predicts will be referenced in the future, it simply checks the primary cache to see whether the data is already present. The obvious advantage of always probing the cache is that the hardware will never be fooled into predicting that data is in the cache (thereby not issuing a prefetch) when it actually is not. For example, if the operating system swaps out the process and later restarts it, a software-based scheme would have no idea that the entire working set had been flushed from the cache, but a hardware-based scheme would still prefetch the data correctly. Part of the reason why the hardware always probes the cache is that doing so involves no instruction overhead. However, it may cause cache contention overheads, as we will discuss later in this subsection. In general, the primary disadvantage of performing the analysis in the hardware is that it cannot detect complex access patterns such as indirect references.

Effectiveness

Prefetching data the proper amount of time in advance of when it is used is tricky for the hardware, since it has difficulty knowing where the instruction stream will be a hundred or so cycles in the future. Whereas the compiler uses software pipelining, the hardware must rely on branch prediction and possibly instruction lookahead buffering in order to issue the prefetches at the right time. This can be quite difficult, for example, when a loop contains a conditional statement whose outcome varies erratically. For example, if the outcome of function `foo(`i`)` in Figure 5.32 is unpredictable, the lookahead mechanism will be ineffective since it must back up and start over each time the branch is mispredicted. With software-pipelining, however, the data would still be prefetched properly, since the compiler realizes that subsequent loop iterations are executed regardless of the outcome of the conditional statement.

[13] The exception would be if the stride detection hardware only looked at references that suffered cache misses.

```
for (i = 0; i < n; i++) {
  A[i] = x;
  if (foo(i)) x++;
}
```

Figure 5.32: Example where imperfect branch prediction makes it difficult to look far enough ahead in the instruction stream.

The second challenge of making prefetches effective is avoiding cache conflicts, which is a problem common to both hardware-based and software-based techniques. Therefore the techniques discussed earlier in Section 5.2.2 are also applicable to hardware-controlled prefetching.

Overhead

For software-controlled prefetching, the overheads include both instruction and memory contention overheads. Hardware-based techniques obviously do not suffer from instruction overhead, which is one of their advantages. However, they should suffer at least as much (if not more) from memory hierarchy contention, since the hardware probes the cache for *each* predicted reference, rather than only for references predicted to suffer cache misses (as software does). Consequently the primary cache tag contention may be quite high for hardware-based approaches. In addition, hardware-based techniques may suffer from TLB contention, since they must somehow deal with virtual addresses in order to follow the access patterns across physical pages. The TLB may need to be accessed either every reference or once every time a page is crossed, and this may contend with normal instructions that access the TLB. For the software-controlled case, access to the TLB is simplified since this is a normal part of processing an instruction that references memory.

Summary

Hardware-controlled prefetching primarily offers two advantages over software-controlled prefetching. First, old code does not need to be recompiled to take advantage of prefetching. However, this dissertation has demonstrated that the compiler technology for automatically inserting prefetches can be quite successful and is straightforward to implement. Therefore since prefetching compilers should be readily available in the future, this does not appear to be a compelling argument. In particular, scientific programmers usually care enough about performance that they

are willing to recompile their code. The second advantage of hardware-controlled prefetching is that it adds no instruction overhead. However, as we have already seen in Chapters 3 and 4, the instruction overhead of software-controlled prefetching is typically quite small, so this also appears not to be much of an advantage.

Hardware-controlled prefetching has some important disadvantages compared to software-controlled prefetching. First, it is limited only to constant-stride access patterns, and therefore cannot prefetch the indirect references which our compiler can handle (as demonstrated in Section 3.5). Since the compiler is also quite successful at prefetching the constant-stride cases (as we have demonstrated), software-controlled prefetching is likely to offer better *coverage* than hardware-controlled prefetching. We would expect this trend to continue in the future as the compiler becomes more sophisticated. Second, although hardware-based schemes have no software cost, they may have a significant *hardware cost*, consuming chip area and possibly affecting cycle time. Therefore since software-controlled prefetching has been shown to be quite effective, offers a broader coverage of misses, and is *much* simpler to implement in the processor, it appears to be a better solution than hardware-controlled prefetching.

5.3.2 Relaxed Memory Consistency Models

One way to cope with the latency of cache misses suffered by loads and stores is to buffer and pipeline their accesses. However, due to features of large-scale multiprocessors such as caches, distributed memory, and general interconnection networks, it is likely that multiple accesses issued by a processor will be performed out of order. This may lead to incorrect program behavior if the program depends upon accesses completing in a certain order. Therefore it may be necessary to restrict the types of buffering and pipelining that are permitted. These restrictions are dictated by the *memory consistency model* supported by the multiprocessor.

Several memory consistency models have been proposed. The strictest model is that of *sequential consistency* (SC) [50]. It requires the execution of a parallel program to appear as some interleaving of the execution of the parallel processes on a sequential machine. While conceptually intuitive, this model imposes severe restrictions on the buffering and pipelining of memory accesses. One of the least strict models is the *release consistency* model (RC) [27]. It requires that synchronization accesses in the program be identified and classified as either *acquires* (e.g., locks) or *releases* (e.g., unlocks). An acquire is a read operation (which can be part of a read-modify-write) that gains permission to access a set of data, while a release is a write operation that gives away such permission. This information is used to provide flexibility in the

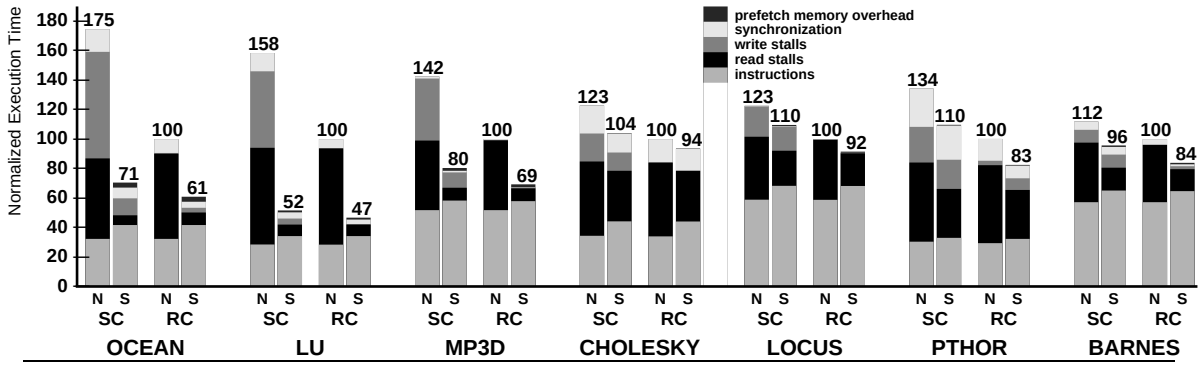


Figure 5.33: Performance with sequential consistency (**SC**) versus release consistency (**RC**), normalized to RC without prefetching (**N** = no prefetching, **S** = selective prefetching).

buffering and pipelining of accesses between synchronization points. The main advantage of the relaxed models is the potential for increased performance. The main disadvantage is increased hardware complexity and a more complex programming model. In this section we will evaluate the benefits of relaxed consistency models and explore their interaction with prefetching.

The multiprocessor architecture we have been using so far uses RC. The writes are buffered and the reads are allowed to bypass pending writes. With the lockup-free cache, a single read miss and an arbitrary number of write misses (limited only by the write buffer size) may be processed simultaneously.[14] Therefore the latency of writes should have no direct impact on performance under our implementation of RC. The SC implementation that we evaluate is satisfied by ensuring that the memory accesses from each process complete in the order that they appear in the program. This is achieved by delaying the issue of an access until the previous access completes. Since the processor already stalls on reads until they complete, the only modification necessary to satisfy SC is to explicitly stall after every write until the write access completes.

Figure 5.33 shows the performance of the multiprocessor applications under the SC and RC models. The memory stall time in these bars has been broken down further into both *read stalls* and *write stalls*. Comparing the cases without prefetching under the two models, the main performance impact of RC is to eliminate the time spent stalled for writes. In several cases (e.g., OCEAN, LU, and MP3D) this resulted in dramatic performance improvements (more than 40%). The pipelining of writes under RC also reduced synchronization stall times somewhat by allowing release operations (e.g., unlocks) to be propagated faster. While relaxing the consistency model

[14]The RC model would even allow multiple read misses to occur simultaneously, but this does not occur in our architecture since it does not support *non-blocking loads*.

effectively hides the latency of write accesses, the latency of read misses still remains.

Overall, the speedup due to prefetching under SC is typically at least as large as it is under RC. The reduction in read stall time is similar under both SC and RC. However, the reduction in write stall time varies depending on how effectively exclusive-mode prefetching is used. In three cases (OCEAN, LU, and MP3D) exclusive-mode prefetching eliminates most of the write latency, and therefore the speedup due to prefetching under SC is larger than under RC.

Comparing the cases with prefetching under the two models, the SC case approaches the absolute performance of the RC case when either (i) exclusive-mode prefetching effectively hides write latency (e.g., LU) or (ii) there is little write latency under SC to begin with (e.g., CHOLESKY). The case with the largest performance gap is PTHOR, where there is a significant amount of write latency, and little of it is hidden by prefetching. However, even when prefetching is relatively successful at reducing write latency under SC, enough remains that the best overall performance always comes through the combination of both prefetching and RC.

Hence, we see that prefetching and relaxed consistency models are complementary. Relaxed consistency models eliminate write latency in shared-memory multiprocessors, and prefetching reduces the remaining read latency.

5.3.3 Multithreading

Although relaxed consistency models are effective at eliminating write latency, they do not address the problem of read latency. While prefetching is one technique for hiding read latency, another technique is for the processor to support multiple hardware contexts [3, 36, 39, 73, 85] (also known as *multithreading*). As we mentioned earlier in Section 1.2.5, multithreading has two advantages over prefetching. First, it can handle arbitrarily complex access patterns—even cases where it is impossible to predict the accesses ahead of time (and therefore prefetching cannot succeed). This is because multithreading simply reacts to misses once they occur, rather than attempting to predict them. Multithreading tolerates latency by attempting to overlap the latency of one context with the computation of other concurrent contexts. The second advantage of multithreading is that it requires no software support (assuming the code is already parallelized), which as we mentioned in the previous section is only an advantage if the user is unwilling or unable to recompile old code. Multithreading has three limitations: (i) it relies on additional concurrency within an application, which may not exist; (ii) some amount of time is lost when switching between contexts; and (iii) to minimize context-switching overheads, a significant amount of hardware support is necessary. In this section, we will evaluate multithreading and

explore its interactions with software-controlled prefetching.

The performance improvement offered by multithreading depends on several factors. First, there is the number of contexts. With more contexts available, the processor is less likely to be out of ready-to-run contexts. However, the number of contexts is constrained by hardware costs and available parallelism in the application. Previous studies have shown that given processor caches, the interval between long-latency operations (i.e. cache misses) becomes fairly large, allowing just a handful of contexts to hide most of the latency [85]. The second factor is the context switch overhead. If the overhead is a sizable fraction of the typical run lengths (time between misses), a significant fraction of time may be wasted switching contexts. Shorter context switch times, however, require a more complex processor. Thirdly, the performance depends on the application behavior. Applications with clustered misses and irregular miss latencies will make it difficult to completely overlap computation of one context with memory accesses of other contexts. Multithreading processors will thus achieve a lower processor utilization on these programs than on applications with more regular miss behavior. Lastly, multiple contexts themselves affect the performance of the memory subsystem. The different contexts share a single processor cache and can interfere with each other, both *constructively* (by effectively prefetching another context's working set) and *destructively* (by displacing another context's working set). Also, as is the case with release consistency and prefetching, the memory system is more heavily loaded by multithreading, and thus latencies may increase.

In this study, we use processors with two and four contexts. We do not consider more contexts per processor because 16 4-context processors require 64 parallel threads and some of our applications do not achieve very good speedup with that many threads. We use two different context switch overheads: 4 and 16 cycles.[15] A four-cycle context switch overhead corresponds to flushing/loading a short RISC pipeline when switching to the new instruction stream. An overhead of sixteen cycles corresponds to a less aggressive implementation. In our study, we include additional buffers to avoid thrashing and deadlock when two contexts try to read distinct memory lines that map to the same cache line. All of these experiments assume an RC model.

Results with Multithreading Alone

We begin our investigation by evaluating multithreading in its own right. Later we will examine the benefits of combining multithreading with prefetching.

[15] We show OCEAN with only two contexts because the problem size we use cannot be run with 64 processes.

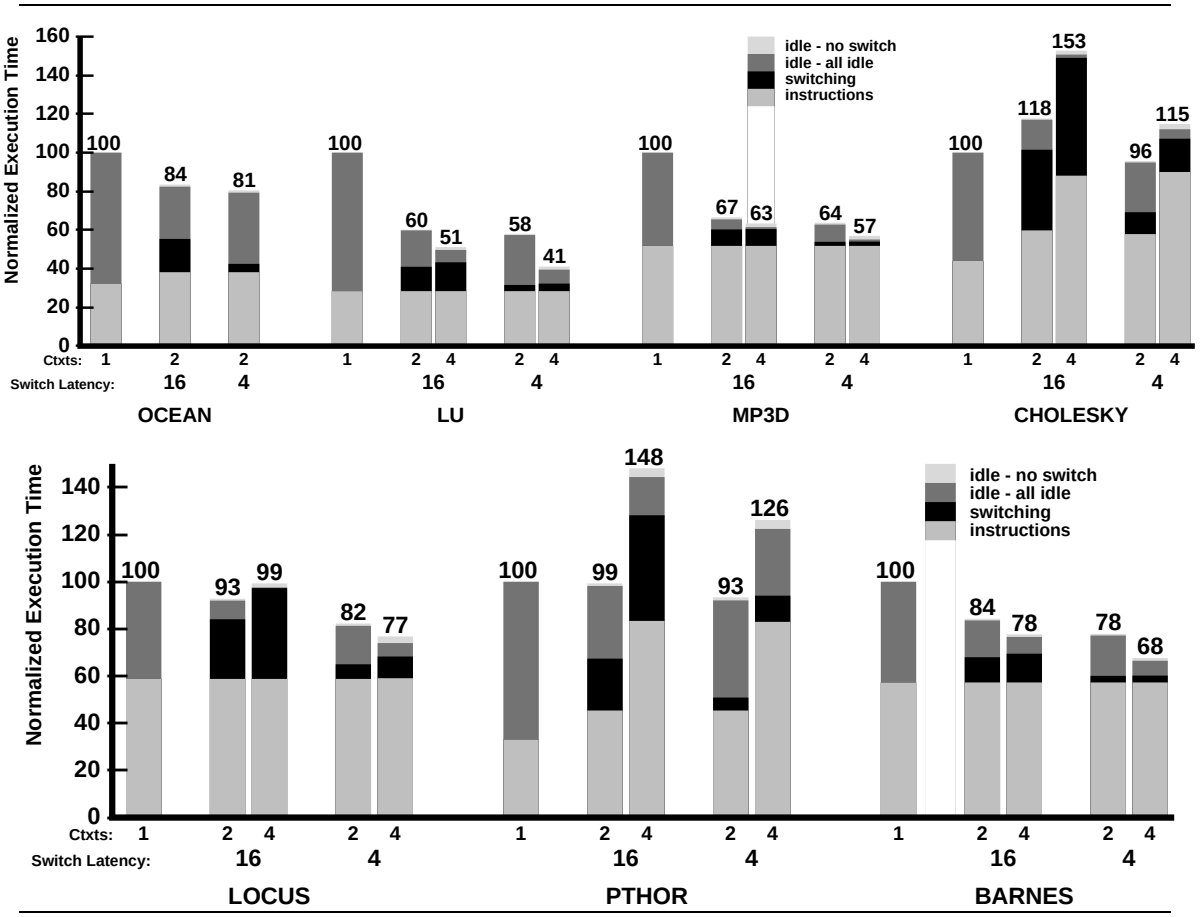


Figure 5.34: Performance of multithreading with 1, 2 and 4 contexts and switch latencies of 4 and 16 cycles.

Figure 5.34 shows performance results for 1, 2, and 4-context processors with context switching penalties of 4 and 16 cycles. Each bar in the graphs is broken down into the following components: time spent executing *instructions*,[16] time spent *switching* between contexts, and time when the processor is *idle*. The idle time is broken down further into *all idle* time which is when all contexts are idle waiting for a reference to complete, and *no switch* time which represents time when the current context is idle but is not switched out. Most of the latter idle time is due to the fact that the processor is locked out of the primary cache while fill operations of other contexts complete.

[16]The *instructions* category includes both "useful" instructions and instructions wasted while spinning on idle work queues. For our previous experiments, these latter instructions were included in the *synchronization* rather than the *instruction* category.

Table 5.3: Statistics on multithreading behavior.

Benchmark	Median Run Length	Average Primary Miss Latency
OCEAN	24	77
LU	36	75
MP3D	55	90
CHOLESKY	23	42
LOCUS	14	41
PTHOR	23	85
BARNES	30	58

Most of the applications benefited from multithreading. The noteworthy exceptions are CHOLESKY and PTHOR, where the performance is worse with four contexts than with a single context. The reason for this is that these two applications do not scale well to 64 processes, and therefore the processes spend too much time spinning waiting for work. This extra spinning time can be seen as the increase in the *instruction* category in Figure 5.34.

To provide some insight into these results, Table 5.3 shows the median *run length* and average primary miss latency for each application. A rough estimate of the number of contexts necessary to hide memory latency is the miss latency divided by the run length. For example, MP3D has one of the more favorable ratios (roughly two-to-one), which helps explain why two contexts eliminate a large fraction of the idle time. In contrast, OCEAN has a ratio of more than three-to-one, which helps explain why two contexts eliminate only part of the idle time.

However, a favorable run-length-to-miss-latency ratio does not ensure good performance. For example, in BARNES this ratio would suggest that two contexts would be sufficient to hide the latency, but in fact only about half of the *all-idle* time is eliminated. The reason for this is the clustering of cache misses. Also the cache miss rates can deteriorate as the different contexts compete for the same cache; we observe this effect in LOCUS, where the primary data miss rate more than doubles from 14% to 30% as we go from one to four contexts.

The importance of minimizing the context switch latency varies depending on whether there is frequently another ready-to-run context during a context switch. On the one hand, when some of the applications are run with only 2 contexts (e.g., OCEAN, LU, and PTHOR), there typically is not a ready-to-run context during a context switch, and therefore reducing the switch penalty from 16 to 4 cycles has little impact on performance. On the other hand, the switch penalty

does affect performance significantly in most cases with 4 contexts, and even in some cases with 2 contexts (e.g., CHOLESKY and LOCUS). Therefore, given that there are enough contexts to hide the latency, it is important to minimize the context switch latency.

To summarize, we see that multithreading can increase performance significantly when the run length to latency ratio is favorable. However, enough parallelism must be available in the application to keep the additional contexts busy. We further observe that destructive interference of the contexts in the processor cache can undo any gains achieved. Interference is more of a problem with multithreading than with prefetching because multiple working sets interfere with each other in the same cache. The smaller the number of cycles required for context switching, the lower the total overhead due to multithreading. A context switch cost of 16 cycles introduces significant overhead, whereas the overhead is much more reasonable with a 4-cycle switch penalty.

Combining Multithreading with Prefetching

Finally, let us consider the combined effect of multithreading and prefetching. The main benefit of combining the two, of course, is that each scheme can compensate for the other scheme's weaknesses. For example, prefetching can increase the hit rate, thus increasing the run lengths and ensuring that a small number of contexts suffice. Similarly, multithreading can ensure that the processor does not remain idle for misses where prefetching was not effective. However, the two schemes can also have negative interactions. First, both prefetching and multithreading add overhead. So if the latency of a reference could be totally hidden by one scheme alone, the second one only contributes overhead. Secondly, the two techniques may interfere with each other. For example, when multiple contexts are used, the time between issue and use of a prefetch may increase substantially, thus increasing the chance of the prefetched data being invalidated or replaced from the cache before being referenced. Depending on the relative magnitudes of the above effects, the performance of an application may increase or decrease when both schemes are used.

Figure 5.35 shows the performance of multithreading, both with and without prefetching (all of these results are for the four cycle switch penalty). As we see in this figure, the results are mixed. In some cases the best overall performance is with four contexts and no prefetching (LU, MP3D, LOCUS, and BARNES), in other cases it is with prefetching and a single context (OCEAN, PTHOR), and in one case it is through the combination of prefetching and two contexts (CHOLESKY). With four contexts, the negative effects of combining prefetching and multithreading appear to dominate. When only two contexts are used, the addition of prefetching nearly

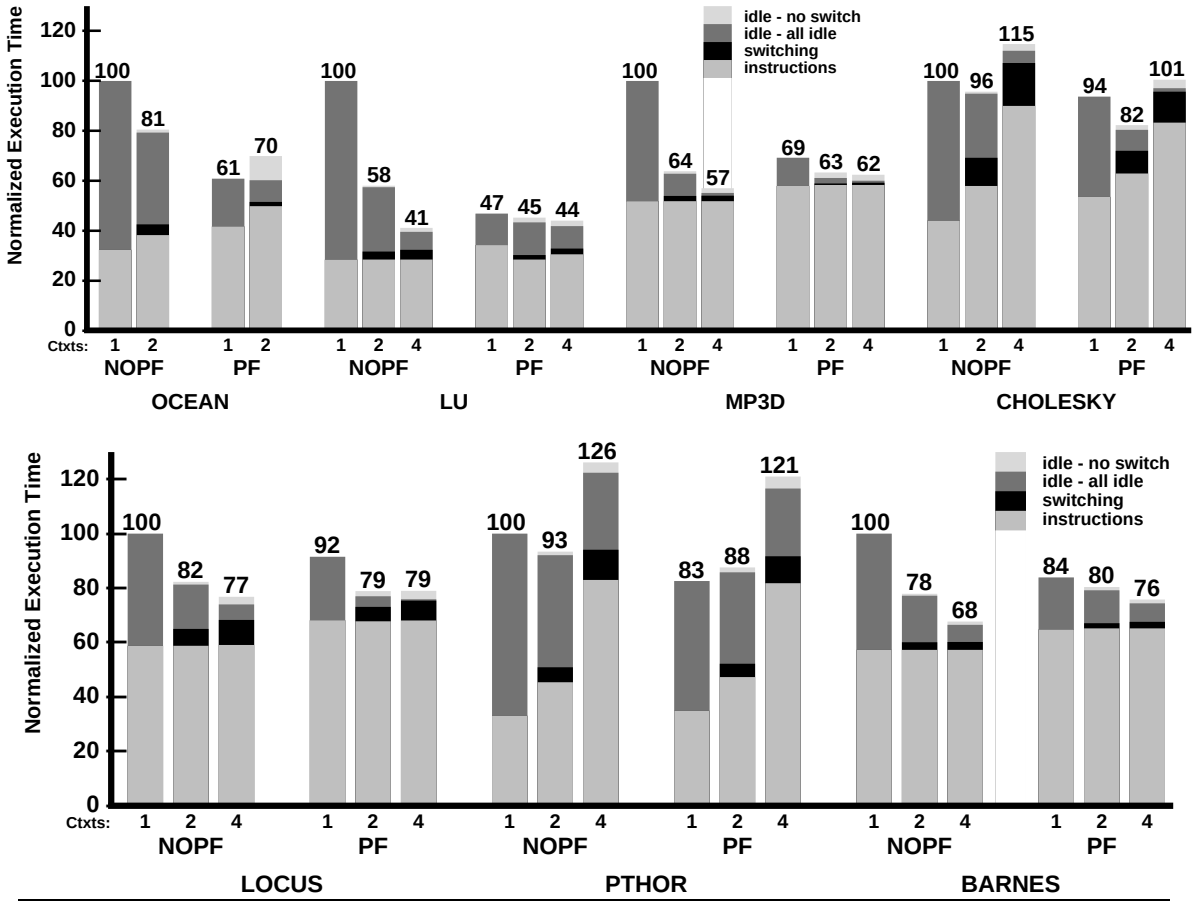


Figure 5.35: Effect of combining multithreading with prefetching (multithreading schemes have a 4-cycle switch latency).

always improves performance. In fact, with only two contexts, the best overall performance in four of the seven cases (LU, MP3D, CHOLESKY, and LOCUS) is achieved by combining prefetching and multithreading.

In our study, when we added prefetching to the applications, we did not have multithreading in mind. In some cases this may have had a negative impact on the results for combining prefetching and multithreading. For example, for a single-context processor, it is reasonable to be quite aggressive and add prefetches in situations where we expect only a small portion of the latency to be hidden. This occurred most frequently in PTHOR and BARNES, where control dependencies make it difficult to move prefetches back far enough. However, for a multiple-context processor, this may be a bad decision. If the multithreading processor would have hidden

the latency anyway, prefetch overhead has been added without any benefit.

Summary

In contrast with locality optimizations and relaxed consistency models, which are clearly complementary to prefetching, the interaction between multithreading and prefetching is more complex. In part this is because both techniques attempt to achieve the same goal, which is hiding read latency. Therefore if either technique is highly successful, there is little benefit of adding the second technique. For example, we observed that with four contexts (which are usually sufficient to hide most of the latency), there is little benefit of adding prefetching, and in fact the performance is often worse. On the other hand, we saw cases where prefetching outperforms multithreading because the applications do not scale well to large numbers of processes. Perhaps the most surprising of these latter cases is PTHOR, where although we had great difficulty inserting prefetches due to its highly irregular access patterns, it performs even worse with multithreading since it contains little additional task-level parallelism.

Prefetching and multithreading appear to be complementary when there are only two contexts—in all but two cases, prefetching improved by adding a second context, and in all but one case, two contexts improved by adding prefetching. In such cases, prefetching boosts the effectiveness of the smaller number of contexts by increasing the hit rate and thus the interval between context switches. At the same time, multithreading improves prefetching performance by hiding the latency of misses that are not prefetched. Therefore, if hardware costs dictate that only a very small number of contexts can be supported (too few to fully hide memory latency), the most attractive solution may be to combine multithreading and prefetching.

5.4 Chapter Summary

In this chapter, we explored three major sets of architectural issues related to prefetching: basic architectural support for prefetching, possible enhancements to achieve even larger gains through prefetching, and alternative latency-hiding techniques that require hardware support. We now briefly summarize each section below.

The key issues in providing basic architectural support for prefetching are the following:

1. Unlike normal loads, prefetches are *non-binding*, *non-blocking*, and *non-excepting*. By giving prefetch instructions unique opcodes, the bits normally used to specify a destination

register can be used instead for prefetching hints (e.g., *exclusive-mode* vs. *shared-mode* prefetches).

2. While dropping prefetches is a complex issue, the best approach appears to be dropping prefetches on TLB misses, and not dropping them on full prefetch issue buffers (given selective prefetching).

3. When performing a prefetch access, the caches should be checked while searching for the data, and the data should be placed directly in the primary cache.

4. The main hardware support necessary for prefetching is a lockup-free cache. Supporting up to four outstanding misses is useful, and buffering more than four outstanding prefetch requests offers only a limited advantage.

The following are the major issues in achieving even larger gains through prefetching:

1. Profiling feedback can potentially improve performance, but has some important drawbacks. A more attractive technique for exploiting dynamic information may be to generate adaptive code. Both of these techniques will benefit from user-visible hardware miss counters.

2. Providing associativity through either set-associative caches or victim caches is an important step toward dealing with cache conflicts. Neither of these solutions is as attractive as eliminating the problem in software.

3. To avoid excessive prefetching overhead, the compiler must be careful to avoid register spilling. To reduce overheads further, *block prefetches* may be useful when spatial locality exists.

Finally, prefetching compares with other latency-hiding techniques as follows:

1. Software-controlled prefetching appears to be superior to hardware-controlled prefetching, since the software approach results in better coverage and requires less hardware support.

2. Prefetching and relaxed consistency models are complementary. Relaxed consistency models eliminate write latency, and prefetching addresses the remaining read latency.

3. The interaction between prefetching and multithreading is complex, and appears to be complementary only with very small numbers of contexts.

Chapter 6

Conclusions

Techniques for coping with memory latency are essential to achieve high processor utilization. Such techniques will become increasingly important in the future as the gap between processor and memory speeds continues to widen. As we discussed in Chapter 1, the best approach to dealing with latency is to first *reduce* it as much as possible, through techniques such as caching and locality optimizations, and then *tolerate* whatever latency remains. The latency of writes can be hidden by buffering and pipelining the write accesses, which is accomplished in shared-memory multiprocessors through relaxed consistency models. However, read latency is only addressed effectively through either prefetching or multithreading. Of these two techniques, software-controlled prefetching appears to be more attractive because it can speed up a single thread of execution, and because it requires much simpler hardware than multithreading. This dissertation has addressed the open question of how effective software-controlled prefetching can be in practice. We have addressed this question by proposing and implementing a new algorithm for inserting prefetches into array-based scientific and engineering codes.

The key results of this dissertation are the following:

1. Software-controlled prefetching can be quite effective at tolerating memory latency in scientific and engineering applications on both uniprocessor and large-scale multiprocessor architectures. In all but a few cases, 50% to 90% of the original memory stall time is eliminated, which translates into improvements in overall performance of over 45% for a majority of the applications we studied. In several cases, overall performance improved by a factor of two.

2. The compiler can do a very good job of inserting prefetches into code automatically, and

it can cover a wide domain of scientific and engineering applications. *Locality analysis* is successful at predicting exactly which references should be prefetched, *loop splitting* techniques help minimize prefetching overhead, and *software pipelining* is effective at scheduling prefetches to hide memory latency. We demonstrated the effectiveness of our algorithm through a full compiler implementation and detailed performance studies. The success of the compiler algorithm is encouraging, since it relieves the programmer from the burden of inserting prefetches manually.

3. Prefetching is complementary to other latency-hiding techniques, including locality optimizations and relaxed consistency models. Locality optimizations complement prefetching by reducing the number of cache misses (thus reducing the resulting prefetching overhead), and prefetching hides much of the remaining latency. Similarly, relaxed consistency models complement prefetching by completely hiding write latency, and prefetching addresses the remaining read latency.

4. Latency-hiding techniques requiring expensive hardware support (e.g., hardware-controlled prefetching, multithreading) do not appear to be necessary for the classes of applications considered in this study.

5. Since prefetching can only improve performance if additional bandwidth is available in the memory subsystem, it is essential that the hardware provide this additional bandwidth through techniques such as lockup-free caches. We observed that supporting up to four outstanding cache misses can improve performance substantially. Providing sufficient memory bandwidth should be the focus of hardware design.

6.1 Future Work

The goals of our compiler research were twofold: to cover a wide range of applications, and to maximize the performance benefit for the cases that are covered. In this section, we briefly discuss how our research can be extended along both of these dimensions.

The scope of our algorithm was limited to array-based scientific and engineering applications. While such applications represented an important first step for prefetching, clearly there are other types of applications and reference patterns that also deserve attention. Perhaps the most obvious next step is to address applications containing large recursive data structures, such as the trees and linked-lists that accounted for so much of the memory latency in BARNES and PTHOR.

To handle such cases, the compiler will need powerful pointer analysis techniques to recognize these recursive structures and to understand the manner in which they are being traversed.

To achieve larger benefits from prefetching in the cases that are covered, a number of the techniques discussed in Section 5.2 deserve further exploration. In particular, the use of dynamic information through either profiling feedback or adaptive code is likely to become more important as the compiler increases its scope to include access patterns such as pointers, where it is difficult if not impossible to predict locality based on static information alone. In addition, we have seen how chronic cache conflicts can potentially render prefetching ineffective, and that often the most desirable solution is fix the problem in software rather than hardware. Techniques that allow the software to automatically detect and prevent such conflicts would be very desirable, and would improve the performance of code even without prefetching.

Finally, this research has focused only the latency of accessing main memory. The general concept of prefetching can potentially be extended to handle other important forms of latency, such as accessing file systems and communicating across networks.

Bibliography

[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] S. Adve and M. Hill. Weak ordering - A new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, May 1990.

[3] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz. April: A processor architecture for multiprocessing. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 104–114, May 1990.

[4] S. P. Amarasinghe and M. S. Lam. Communication optimization and code generation for distributed memory machines. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 126–138, June 1993.

[5] J. M. Anderson and M. S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 112–125, June 1993.

[6] J. Archibald and J.-L. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, 1986.

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

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

[9] 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.

[10] A. Carle, K. Kennedy, U. Kremer, and J. Mellor-Crummey. Automatic data layout for distributed-memory machines in the D programming environment. In *Proceedings of AP'93 International Workshop on Automatic Distributed Memory Parallelization, Automatic Data Distribution and Automatic Parallel Performance Prediction*, Saarbrücken, Germany, March 1993.

[11] B. Chapman, P. Hehrota, and H. Zima. Programming in vienna fortran. In *Third Workshop on Compilers for Parallel Computers*, pages 121–160, July 1992.

[12] S. Chatterjee, J. Gilbert, R. Schreiber, and S. Teng. Automatic array alignment in data-parallel programs. In *Proceedings of the Twentieth Annual ACM Symposium on the Principles of Programming Languages*, January 1993.

[13] 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.

[14] 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.

[15] K.D. Cooper, M.W. Hall, and K. Kennedy. A methodology for procedure cloning. *Computer Languages,* 19(2), April 1993.

[16] 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.

[17] M. Dubois, L. Barroso, Y.-S. Chen, and K. Oner. Scalability problems in multiprocessors with private caches. In *Proceedings of Parallel Architecture and Languages Europe '92*, pages 211–230, June 1992.

[18] M. Dubois, C. Scheurich, and F. A. Briggs. Synchronization, coherence, and event ordering in multiprocessors. *Computer*, 21(2):9–21, February 1988.

[19] S. J. Eggers and T. E. Jeremiassen. Eliminating false sharing. In *Proceedings of the 1991 International Conference on Parallel Processing*, volume I, pages 377–381, August 1991.

[20] M. Berry et al. The perfect club benchmarks: Effective performance evaluation of supercomputers. Technical Report CSRD 827, Center for Supercomputing Research and Development, Illinois, May 1989.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] A. George, J. Liu, and E. Ng. User's guide for SPARSPAK: Waterloo sparse linear equations package. Technical Report CS-78-30, Department of Computer Science, University of Waterloo, 1980.

[26] K. Gharachorloo, A. Gupta, and J. Hennessy. Performance evaluation of memory consistency models for shared-memory multiprocessors. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 245–257, April 1991.

[27] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[28] A. J. Goldberg. *Multiprocessor Performance Debugging and Memory Bottlenecks*. PhD thesis, Stanford University, August 1992.

[29] S. R. Goldschmidt and H. Davis. Tango introduction and tutorial. Technical Report CSL-TR-90-410, Stanford University, 1990.

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

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

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

[33] 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.

[34] M. Gupta. *Automatic Data Partitioning on Distributed Memory Multicomputers*. PhD thesis, College of Engineering, University of Illinois at Urbana-Champaign, September 1992.

[35] M. Gupta and P. Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):179–193, March 1992.

[36] R. H. Halstead, Jr. and T. Fujita. MASA: A multithreaded processor architecture for parallel symbolic computing. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 443–451, June 1988.

[37] L. J. Hendren. *Parallelizing Programs with Recursive Data Structures*. PhD thesis, Cornell University, January 1990.

[38] S. Hiranandani, K. Kennedy, and C. Tseng. Compiling fortran d for mimd distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[39] R. A. Iannucci. Toward a dataflow/von Neumann hybrid architecture. In *Proc. Int. Symp. Comput. Arch.*, pages 131–140, June 1988.

[40] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, May 1990.

[41] Kendall Square Research. *Kendall Square Research 1 (KSR1) Technical Summary*, 1992.

[42] 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.

[43] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Supporting shared data structures on distributed memory machines. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, March 1990.

[44] J. S. Kowalik, editor. *Parallel MIMD Computation : The HEP Supercomputer and Its Applications*. MIT Press, 1985.

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

[46] J. Kubiatowicz, D. Chaiken, and A. Agarwal. Closing the window of vulnerability in multiphase memory transactions. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 274–284, October 1992.

[47] D. J. Kuck, E. S. Davidson, D. H. Lawrie, and A. H. Sameh. *Experimental Parallel Computing Architectures: Volume 1 – Special Topics in Supercomputing*, chapter Parallel Supercomputing Today and the Cedar Approach, pages 1–23. North-Holland, New York, 1987.

[48] 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.

[49] 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.

[50] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):241–248, September 1979.

[51] W. Landi, B. G. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 56–67, June 1993.

[52] J. P. Laudon. *Architectural and Implementation Tradeoffs for Multiple-Context Processors*. PhD thesis, Stanford University, Stanford, California, 1994. In preparation.

[53] 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.

[54] D. Lenoski, K. Gharachorloo, J. Laudon, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford DASH multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[55] D. Lenoski, K. Gharachorloo, J. Laudon, A. Gupta, J. Hennessy, Mark Horowitz, and Monica Lam. Design of Scalable Shared-Memory Multiprocessors: The DASH Approach. In *Proceedings of COMPCON'90*, pages 62–67, 1990.

[56] J. Li and M. Chen. The data alignment phase in compiling programs for distributed-memory machines. *Journal of Parallel and Distributed Computing*, 13(2):213–221, October 1991.

[57] E. Lusk, R. Overbeek, et al. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston, Inc., 1987.

[58] D. E. Maydan. *Accurate Analysis of Array References*. PhD thesis, Stanford University, September 1992.

[59] J. D. McDonald and D. Baganoff. Vectorization of a particle simulation method for hypersonic rarified flow. In *AIAA Thermodynamics, Plasmadynamics and Lasers Conference*, June 1988.

[60] 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.

[61] 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.

[62] 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*, volume 27, pages 62–73, October 1992.

[63] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss. The IBM research parallel processor prototype (RP3): Introduction and architecture. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 764–771, 1985.

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

[65] 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.

[66] A. Rogers and K. Li. Software support for speculative loads. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, volume 27, pages 38–50, October 1992.

[67] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Proceedings of the SIGPLAN '89 Conference on Program Language Design and Implementation*, June 1989.

[68] J. Rose. Locusroute: A parallel global router for standard cells. In *Design Automation Conference*, pages 189–195, June 1988.

[69] E. Rothberg and A. Gupta. Techniques for improving the performance of sparse factorization on multiprocessor workstations. In *Proceedings of Supercomputing '90*, November 1990.

[70] C. Scheurich and M. Dubois. Lockup-free caches in high-performance multiprocessors. *Journal of Parallel and Distributed Computing*, 11(1):25–36, January 1991.

[71] J. P. Singh and J. L. Hennessy. Finding and exploiting parallelism in an ocean simulation program: Experience, results and implications. *Journal of Parallel and Distributed Computing*, 15(1):27–48, 1992.

[72] 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.

[73] B. J. Smith. Architecture and applications of the HEP multiprocessor computer system. *SPIE*, 298:241–248, 1981.

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

[75] M. D. Smith. *Support for Speculative Execution in High-Performance Processors*. PhD thesis, Stanford University, November 1992.

[76] L. Soule and A. Gupta. Parallel Distributed-Time Logic Simulation. *IEEE Design and Test of Computers*, 6(6):32–48, December 1989.

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

[78] G. L. Steele. Proposal for alignment and distribution directives in HPF. Draft presented at HPF Forum meeting, June 1992.

[79] P. Stenstrom, F. Dahlgren, and L. Lundberg. A lockup-free multiprocessor cache design. In *Proceedings of the 1991 International Conference on Parallel Processing*, volume I, pages 246–250, 1991.

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

[81] J. Torrellas, M. S. Lam, and J. L. Hennessy. Shared data placement optimizations to reduce multiprocessor cache miss rates. In *Proceedings of the 1990 International Conference on Parallel Processing*, volume II, pages 266–270, August 1990.

[82] P.-S. Tseng. *A Parallelizing Compiler for Distributed Memory Parallel Computers*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1989.

[83] D. M. Tullsen and S. J. Eggers. Limitations of cache prefetching on a bus-based multiprocessor. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 278–288, May 1993.

[84] W.-D. Weber. *Scalable Directories for Cache-Coherent Shared-Memory Multiprocessors*. PhD thesis, Stanford University, January 1993.

[85] W.-D. Weber and A. Gupta. Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: Preliminary results. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 273–280, June 1989.

[86] M. E. Wolf. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Stanford University, August 1992.

[87] 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.

[88] H. Zima, H.-J. Bast, and M. Gerndt. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.