Effective Cache Prefetching on Bus-Based Multiprocessors

DEAN M. TULLSEN and SUSAN J. EGGERS
University of Washington

Compiler-directed cache prefetching has the potential to hide much of the high memory latency seen by current and future high-performance processors. However, prefetching is not without costs, particularly on a shared-memory multiprocessor. Prefetching can negatively affect bus utilization, overall cache miss rates, memory latencies and data sharing. We simulate the effects of a compiler-directed prefetching algorithm, running on a range of bus-based multiprocessors. We show that, despite a high memory latency, this architecture does not necessarily support prefetching well, in some cases actually causing performance degradations. We pinpoint several problems with prefetching on a shared-memory architecture (additional conflict misses, no reduction in the data-sharing traffic and associated latencies, a multiprocessor's greater sensitivity to memory utilization and the sensitivity of the cache hit rate to prefetch distance) and measure their effect on performance. We then solve those problems through architectural techniques and heuristics for prefetching that could be easily incorporated into a compiler: (1) victim caching, which eliminates most of the cache conflict misses caused by prefetching in a direct-mapped cache, (2) special prefetch algorithms for shared data, which significantly improve the ability of our basic prefetching algorithm to prefetch invalidation misses, and (3) compiler-based shared-data restructuring, which eliminates many of the invalidation misses the basic prefetching algorithm does not predict. The combined effect of these improvements is to make prefetching effective over a much wider range of memory architectures.

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles—*cache memories*; *shared memory*; C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures

General Terms: Design, Experimentation, Performance

Additional Key Words and Phrases: Bus-based multiprocessors, cache prefetching, false sharing, memory latency hiding

1. INTRODUCTION

Several factors contribute to the increasing need for processors to tolerate high memory latencies, particularly in multiprocessor systems. Certainly the widening gap in speed between CPUs and memory increases memory latencies in uniprocessors and multiprocessors alike [Hennessy and Jouppi 1991].

Fast processors also increase contention in multiprocessors, lengthening the actual latency seen by CPUs, because of CPU queuing for the interconnect. Second, parallel workloads exhibit more interconnect operations, caused by data sharing among the processors, resulting in more delays and greater memory subsystem contention. Finally, as processors and memory become more physically distributed, memory latencies necessarily increase.

Software-controlled cache prefetching is a technique that is designed to make processors more tolerant of memory latency. In software-controlled cache prefetching, the CPU executes a special prefetch instruction for data that is to be loaded at some point in the near future. In the best case, the data arrives at the cache before it is needed by the CPU, and the CPU sees its load as a hit. Lockup-free caches [Kroft 1981; Motorola 1990; Scheurich and Dubois 1991; Sohi and Franklin 1991], which allow the CPU to continue execution during the prefetch, hide the prefetch latency from the CPU.

In this article we address the issue of prefetching in bus-based, shared-memory multiprocessors. The goal of our work is to gauge its impact on the performance of these architectures and to pinpoint the factors responsible. Our experiments simulate parallel workloads on a bus-based multiprocessor, coupled with a prefetching algorithm that represents the "ideal" for current compiler-directed prefetching technology, in that it has an "oracle" to predict cache misses (apart from misses caused by data sharing). We use this to identify architectures and workloads where prefetching improves performance and where performance degrades. These results give us insight into the particular problems multiprocessors pose to prefetching and allow us to introduce changes to the memory subsystem, the prefetching algorithm, and the shared-data allocation to solve them. Although our studies most closely model a bus-based system, they should extend to other multiprocessor architectures for which memory contention is an issue.

We show that prefetching on a bus-based multiprocessor, unlike on a uniprocessor, is not a universal win. Issues such as increased pressure on the cache, a parallel machine's greater sensitivity to memory subsystem utilization, and the interaction of prefetching with data-sharing effects can cause the performance improvements from prefetching to be less than expected or even nonexistent. With our most basic prefetching scheme, we observe speedups no higher than 29% and degradations as high as 6%. We address these issues through various architectural and compiler-based techniques: victim caching, compiler-based shared-data restructuring, and two special prefetch algorithms for shared data. We show that these techniques make prefetching effective over a much wider range of memory architectures, and significantly more effective in those regions where it is already viable. Some of these techniques are particularly effective in combination; one combination achieves speedups from 6% to 83% across the range of simulated memory architectures.

The remainder of the article is organized as follows. Section 2 describes related work in compiler-directed prefetching and multiprocessor prefetching. Section 3 describes our methodology and justifies our choice of simulation environment. Section 4 presents the results of a basic prefetch strategy used

in conjunction with a highly efficient cache miss predictor, and highlights some of the drawbacks of prefetching with that strategy. The following three sections explore in detail issues that have prevented better multiprocessor prefetching performance and present architectural and compiler techniques that make prefetching more effective. Section 5 examines the problem of prefetches that do not complete on time, Section 6, cache conflict misses caused by prefetching, and Section 7, data-sharing issues, principally the difficulty of predicting invalidation misses. Section 8 shows the effect of combining these techniques, and the conclusions appear in Section 9.

2. RELATED WORK

This work builds on a previous study [Tullsen and Eggers 1993] in which we pinpointed the problems with prefetching on a shared-memory machine (additional conflict misses, data-sharing traffic, a multiprocessor's greater sensitivity to memory utilization, and determining the best prefetch distance) and measured their effect on performance. In that paper, we showed that, despite high memory latencies, many bus-based multiprocessors do not support prefetching well, and in some cases prefetching strategy actually causes a performance degradation. This article extends that work in several ways. First, we examine conflict misses in more detail and use a new memory architecture alternative (a victim cache) to eliminate virtually those caused by prefetching (Section 6). Second, we present a new prefetching that makes more-effective use of exclusive prefetching (Section 7.3). Third, we make use of more-sophisticated shared-data restructuring techniques (Section 7.2) that make prefetching more viable. Fourth, we have improved the methodology of the previous study in several respects. For example, we trace a different region of the Pverify application to capture more parallelism. Also, we now model hardware barriers more accurately; this increased the accuracy of short-term sharing activity in Topopt (the only application that makes frequent use of barriers).

Although the need to make processors tolerant of high memory latency is much more severe in multiprocessors than in uniprocessors, most other studies of cache prefetching have concentrated on uniprocessor architectures [Callahan et al. 1991; Chen and Baer 1992; Chen et al. 1991; 1992; Mowry et al. 1992]. DASH [Lenoski et al. 1993] has hardware support for cache prefetching, but to date they have only published the results of micro-benchmark throughput tests. A noteworthy exception is Mowry and Gupta [1991], in which simulations were driven by three parallel programs, providing analysis of potential speedups with programmer-directed cache prefetching. However, the multiprocessor architecture they examined (16 DASH clusters connected by a high-throughput interconnection network, with only one processor per cluster) avoids the types of contention and interference that we wish to study. As a result, they did not include the full effects of contention for a shared bus. We found this effect to be crucial to prefetching performance for the architectures we examined. In addition, we provide more-detailed analysis of multiprocessor cache misses, identifying key compo-

nents that affect performance. Their scheme deals with programmer-directed prefetching, while ours emulates compiler-directed prefetching. They simulate only shared-memory references, while we simulate both shared and private, and the interference between the two in the caches is a key element of this study.

Mowry et al. [1992] details a compiler-based prefetching algorithm for a uniprocessor, which is the model that our simulated prefetching algorithms emulate. It uses several techniques to do very selective prefetching, targeting only those memory accesses that are very likely to miss in the cache.

Among processors either currently available, or expected to be available this year, three (Hewlett-Packard PA-7200 [Microprocessor 1994a], Digital Equipment Corporation Alpha 21164 [Edmondson and Rubinfield 1994], and Sun Ultrasparc [Microprocessor 1994b]) have some kind of support for prefetching. Of those, only two (Alpha and Ultrasparc) support exclusive prefetching, and all three have direct-mapped first-level caches, although the PA-7200 has a victim cache.

3. SIMULATION ENVIRONMENT

Our prefetching studies use trace-driven simulation. Traces are generated from coarse-grain, explicitly parallel workloads, and prefetch instructions are inserted into the traces. We simulate several types of bus-based multiprocessors, which differ in the extent to which contention affects memory latency, i.e., we vary bus speeds. We also examine cache architectures both without and (in Section 6) with a victim cache. Several prefetching strategies are used that differ in when, how, and how often prefetching is done. This section details the simulation environment.

3.1 Prefetching Algorithms

Software-directed prefetching schemes either cache prefetch (which brings data into the data cache closest to the processor, also known as nonbinding prefetch) or prefetch data into a separate prefetch buffer (binding prefetch). Our prefetching study concerns cache-based prefetching.

Our baseline prefetching algorithm contains an optimized prefetcher for nonshared, i.e., "uniprocessor," data misses (those that only depend on the cache configuration). It very accurately predicts nonsharing cache hits and misses and never prefetches data that is not used. We emulate this algorithm by adding prefetch instructions to the address traces after they are generated on a shared-memory multiprocessor. The candidates for prefetching are identified by running each processor's address stream through a uniprocessor cache filter and marking the data misses. The prefetch instructions are then placed in the instruction stream some distance ahead of the accesses that miss. The number of CPU instructions between the prefetch and the actual access is referred to as the *prefetch distance*. Since it is an off-line algorithm, the technique represents the "ideal" for current prefetching algorithms, i.e., one that prefetches both scalars and array references, and accurately identifies leading references (first access to a cache line) and capacity and conflict

misses. Mowry et al. [1992] have shown that compiler algorithms can approximate this very well already, predicting compulsory and capacity misses on array references; and, as existing algorithms improve, they will get closer to this ideal. Using a prefetcher that is ideal with respect to nonsharing misses enables us to pinpoint the exact cause of each remaining miss observed by the CPU after prefetching, as explained in Section 4.

With our baseline algorithm, we strive to emulate a compiler-based algorithm (of which Mowry et al. [1992] is the best example) rather than a programmer-directed approach (such as Mowry and Gupta [1991]), because we feel that prefetching will predominantly be the domain of compilers rather than programmers. Mowry and Gupta show that prefetching inserted by a programmer intimately familiar with the application can be effective, but such a methodology does not necessarily indicate what compiler-directed prefetching would do. Mowry et al. [1992], despite being targeted to a uniprocessor, represents the best available compiler-directed prefetching algorithm, for any architecture. It is not unreasonable to expect that it would perform well for a large number of multiprocessor architectures without enhancement.

The overhead associated with each prefetch in our simulations is relatively low, a single instruction and the prefetch access itself, as we continue to assume the existence of effective and efficient prefetching algorithms. Mowry et al. report overheads for their uniprocessor compiler algorithm as typically less than 15% (in increased instruction count; the impact on total execution time is typically about half that), and Mowry and Gupta's programmer-directed scheme experienced overheads between 1% and 8% of total execution time. Chen and Baer's [1994] implementation of Mowry et al.'s compiler algorithm on a multiprocessor (including two of our benchmarks, Mp3d and Water) experienced prefetch instruction overhead of 2–4% of total execution time. In our simulations, the overheads are never more than 4% of the instruction count or 2% of the total execution time.

On a multiprocessor with a write-invalidate cache coherency protocol, data can be prefetched in either shared mode (in which case a subsequent write might require an extra invalidating bus operation) or in exclusive mode (which would cause all other cached copies of that cache line to be invalidated). The latter is referred to as an *exclusive prefetch*. Our simulations support both types of prefetches, as in Mowry and Gupta. The base algorithm will prefetch in shared mode; exclusive prefetching will be considered as an extension in Section 7.3. We do this for two reasons. Our base prefetcher strives to emulate Mowry et al. [1992], which does not support exclusive prefetching, being a uniprocessor algorithm. Also, as discussed in Section 2, not all processors that support prefetching support exclusive prefetching.

3.2 Workload

The address traces were generated with MPTrace [Eggers et al. 1990] on a Sequent Symmetry [Lovett and Thakkar 1988], running the following coarse-grained, explicitly parallel applications, all written in C (see Table I). Topopt

Table I.   Workload Used in the Experiments

Program	Data Set	Shared Data	Number of Processes	Dynamic Data Set Size	Data References	Percent Reads	Percent Private
Pverify	C880.21 berk1/2	130 KB	12	128 KB	5.4 million	82%	59%
Topopt	apla lomim	20 KB	9	20 KB	5.8 million	85%	69%
LocusRoute	Primary1	1.6 MB	12	709 KB	7 2 million	75%	87%
Mp3d	10,000 molecules	1.9 MB	12	459 KB	8 2 million	69%	75%
Water	343 molecules	227 KB	12	237 KB	6.7 million	76%	94%

[Devadas and Newton 1987] performs topological optimization on VLSI circuits using a parallel simulated annealing algorithm. Pverify [Ma et al. 1987] determines whether two boolean circuits are functionally identical. Statistics on the amount of shared data for these programs can be found in Eggers [1991]. LocusRoute is a commercial quality VLSI standard cell router. Mp3d solves a problem involving particle flow at extremely low density. Water evaluates the forces and potentials in a system of water molecules in liquid state. The latter three are part of the Stanford SPLASH benchmarks [Singh et al. 1991] which, in contrast to the other two applications, have been optimized by the programmers for processor locality.

Restricted by the practical limit on trace lengths in multiprocessor trace-driven simulations, a balance must be struck between the desire for large data sets that do not fit in the cache and tracing a reasonable portion of the program. With a very large data structure, one could easily end up tracing only a single loop, which may or may not be indicative of the rest of the program. We attempt to solve this by scaling back both the data sets and the local cache sizes by about a single order of magnitude relative to what might be considered a reasonable configuration on current moderately parallel multiprocessors. Thus we maintain a realistic ratio of data set to off-chip cache size, ensuring that in most cases neither critical data structures nor the dynamic cache working sets fit into the simulated cache. The exception is Topopt, which is still interesting because of its high degree of write sharing and the large number of conflict misses, even with the small shared-data-set size. For each application, we begin collecting the traces right after parallel execution begins. We initially simulate (without collecting statistics) about 500,000 data references to avoid cold-start effects; we then simulate at least 5 million more data references for each application. In Table I, the column Shared Data gives the total amount of shared data allocated by the application, while the column Dynamic Data Set Size shows the total amount of both private and shared data touched by the traced portion of the program. The number of references simulated for which statistics were kept is given in the references column. The last two columns show the percent of the data references that are reads and the percent that are to private data for each benchmark.

3.3 Multiprocessor Simulations

After the prefetch accesses were added, the traces were run through Charlie [Eggers 1989], a multiprocessor simulator that was modified to handle

prefetching, lockup-free caches, a split-transaction bus protocol, and victim caches. Besides modeling the CPU, cache, and bus hardware at a low level, Charlie carries out locking and barrier synchronization; therefore, as the interleaving of accesses from the different processors is changed by the behavior of the memory subsystem, Charlie ensures that a legal interleaving is maintained. So, for instance, processors vie for locks and may not acquire them in the same order as the traced run; but they still will acquire each in some legal order and will enter the critical sections one processor at a time.

We have modeled direct-mapped, write-back data caches, assuming instructions caches with insignificantly low miss rates. For all of the simulations presented here the caches are 32KB, with a 32-byte block size.[1] Our simulations include both private and shared data, in order to include the effects of interference between the two in the cache.

The cache coherency scheme is the Illinois coherency protocol [Papamarcos and Patel 1984], an invalidation-based protocol. Its most important feature for our purposes is that it has a private-clean state for exclusive prefetches. We simulate a 16-deep buffer to hold pending prefetches on each processor, which is sufficiently large to prevent almost always the processor from stalling because the buffer is full. The bus uses a round-robin arbitration scheme that favors blocking loads over prefetches. We model a sequentially consistent memory system, consistent with our approach of concentrating on low-end multiprocessor systems.

We only consider systems with high memory latency. (Prefetching is less useful and possibly harmful if there is little latency to hide.) The processors execute a single instruction per cycle, with 2-cycle data loads. This is coupled with a memory subsystem latency of 100 cycles. Given this latency we examine a spectrum of memory architectures from high to low memory bandwidth, resulting in a range of bus utilizations. Our memory/interconnect model is a split-transaction bus protocol, where the system has enough parallelism in the memory banks/controllers to make the address bus and the memory access relatively conflict free and the data bus transfer the bottleneck. By varying the speed of the data bus, we can vary the maximum throughput of the memory subsystem. In this way we are able to model a spectrum of values for the ratio of the bus latency to bus bandwidth, since that is the factor to which many of the phenomena we describe are most sensitive. We do this without varying the minimum latency, and thus prevent our results from being dominated by large changes in the memory latency, since the effectiveness of prefetching is obviously highly sensitive to that parameter. While the specific model is a split-transaction bus, our results should be reflective of any memory architecture that has the potential to saturate.

---

[1] We also ran simulations with larger block sizes. The results are not presented here, because they did not add a great deal of insight to those already presented. Essentially, larger block sizes increased the amount of sharing traffic (due to false sharing) and thus increased the importance of the techniques we use to deal with sharing, but did not otherwise impact the effectiveness of prefetching.

In the simulations in this article, the "data transfer" portion of the memory latency is varied from 4 to 32 cycles out of the total 100 cycles. In the split-transaction bus architecture described above, a data transfer latency of 4 cycles would make the address transmission and memory lookup 96 cycles; if the processor cycle speed was 400 MHz, this would model a memory subsystem with a memory latency of 250 nanoseconds, and a peak throughput of 3.2 GB/second. The 32-cycle latency corresponds to a memory throughput of 400 MB/second.

Also, in Section 6, we add victim caches [Jouppi 1990] to the architecture. A hit in the victim cache takes 4 cycles longer than a hit in the main cache, but is much less than a memory access and saves at least one bus operation.

4. BASIC PREFETCHING

We simulate each memory architecture with a basic prefetching algorithm. *No prefetching* serves as our baseline for calculating speedups for all prefetching algorithms. Execution times for all experiments are given relative to no prefetching with the same memory architecture and cache configuration. In the basic prefetch algorithm, prefetch instructions are inserted into the traces for each potential cache miss identified by the cache filter. For this algorithm, the prefetch distance is 100 instructions, which means that, barring any CPU stalls, the code between the prefetch and the associated load would execute in about 140 cycles, given our processor model. This gives the memory subsystem, with a minimum latency of 100 cycles, time to complete the prefetch when contention delays are not large or when the processor is otherwise slowed down. This, our most basic prefetching strategy, is identified in all figures as PREF. We compare the basic prefetch strategy to no prefetching, which will be identified as NP.

Results for a 32KB cache with a 32-byte cache line are shown in Figures 1 and 2 and Tables II and III. The effect of prefetching on the miss rates is in Figure 1. Because some terminology becomes ambiguous in the presence of prefetching, we will use the following terms. *Misses* (or *total miss rate*) refer to both prefetch and nonprefetch accesses that do not hit in the cache. *CPU misses* (*CPU miss rate*) are misses on nonprefetch accesses and thus are observed by the CPU. *Prefetch misses* occur on prefetch accesses only. Because accesses of prefetches still in progress (since the CPU must stall, we count these as CPU misses and refer to them as *prefetch-in-progress* misses) often comprise a nonnegligible portion of the CPU miss rate, the *Adjusted CPU Miss Rate* does not include them. Therefore, the adjusted CPU miss rate includes all accesses that cause the CPU to stall for an entire memory latency. The CPU miss rate is the adjusted CPU miss rate plus the prefetch-in-progress miss rate, and the total miss rate is the CPU miss rate plus the prefetch miss rate. The miss rates are cumulative in Figure 1, so the total miss rate is the combined height of the three bars, and the CPU miss rate is the combined height of the black and diagonally striped bars. Without prefetching, the total miss rate, the CPU miss rate, and the adjusted CPU miss rate are all identical. The data in Figure 1 is for an 8-cycle data transfer
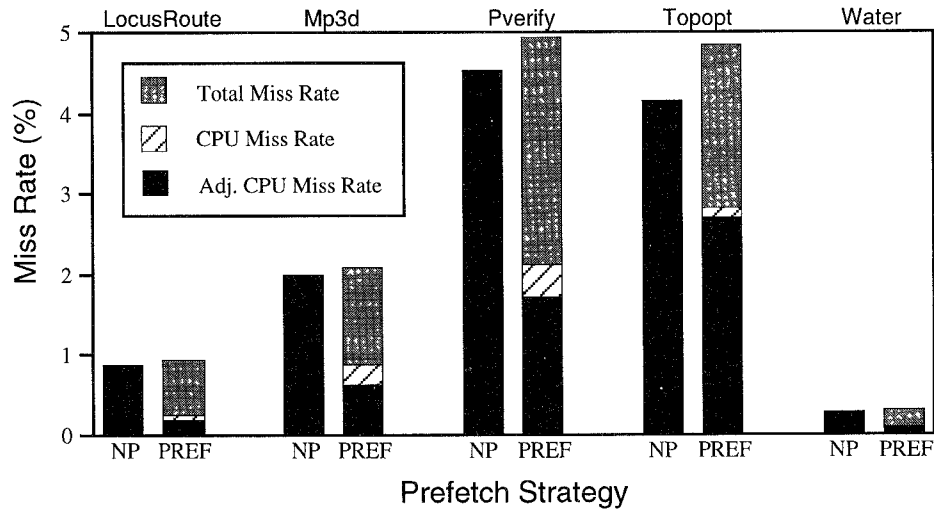
Fig. 1.   Total and CPU miss rates for the five workloads with an 8-cycle data bus latency.

latency. The only component of the miss rate that varies significantly across memory throughputs is the prefetch-in-progress misses (the difference between the CPU miss rate and the adjusted CPU miss rate), which rises as the data bus gets slower.

Several observations can be made from the miss rate results. First, CPU miss rates fell significantly (32–71%, or 35–77% for adjusted) for the results shown in Figure 1. Because we use an oracle prefetcher, one might naively expect even more misses to have been covered. There are three reasons why this did not happen. First, the prefetch-in-progress misses account for a significant part of the CPU miss rate in some of the applications. Second, prefetching introduces actually additional cache conflict misses. Last, and most importantly, data sharing among processors produces invalidation misses, which in most cases are the largest single component of the CPU miss rate. Our oracle prefetcher does not predict misses that are the result of invalidations.

Total miss rates increase in all simulations with prefetching. Previous studies for both uniprocessors and multiprocessors have focused on CPU miss rates. On a uniprocessor it is likely that the memory interconnect has the bandwidth to absorb extra memory traffic, so the increase in total miss rate is not significant relative to the decrease in the CPU miss rate. But for some multiprocessor systems total miss rate is a more important metric, indicative of the demand at the bottleneck component of the machine. This can be particularly true for a bus-based multiprocessor, but also for any multiprocessor where contention for memory or the interconnect is significant. As the bus becomes more and more saturated, system performance will track the throughput of the bus; speeding up the CPUs has no beneficial effect. Since bus demand is a function of the total miss rate rather than the CPU miss

Table II.   Selected Bus Utilizations

Program	Pref. Alg.	Data Transfer Latency			
		4 cycles	8 cycles	16 cycles	32 cycles
Locus	NP	.21	.33	.56	.89
	PREF	.27	.42	.70	.97
Mp3d	NP	.48	.65	.90	1.00
	PREF	.64	.83	.99	1.00
Pverify	NP	.46	.68	.96	1.00
	PREF	63	.88	1.00	1.00
Topopt	NP	.13	.20	.33	.51
	PREF	.17	.26	.41	.61
Water	NP	.10	.14	.22	.38
	PREF	.11	.16	.25	.43

rate, the total miss rate will be a better indicator of performance on these architectures. In a bus or memory bottlenecked system, prefetching, which reduces the CPU miss rate at the expense of the total miss rate, may hurt performance.

In Table II we see how the miss rates affect data bus utilization as the memory architecture is varied. Bus utilization is the number of cycles the bus was in use divided by the total cycles of the simulation. Bus utilization results can be misleading if not interpreted correctly. There are two reasons why bus utilization can increase: one is that the same workload produces more bus operations; the second is that the same number of bus operations occur but over a shorter time period. So, for instance, bus utilization increased with prefetching for the 4-cycle Pverify simulation, both because the total miss rate increased and because the execution time was reduced as a result of prefetching successfully.

In order to see the bus demand independent of execution speed, Table III gives the bus demand per access (BDPA), which is the total number of bus cycles used divided by the total number of memory accesses (both cache hits and misses). BDPA, unlike bus utilization, is independent of execution time, and thus gives a better indication of the additional demand placed on the memory subsystem by prefetching. Results in Tables II and III indicate that, for all applications, the bus demand increased with prefetching, as expected, given the total miss rates from Figure 1. The data in Table III is for the 8-cycle data transfer latency. The data for the other bus speeds is not shown, because for all practical purposes it scales linearly with bus speed.

In Figure 2, we see the effects of prefetching on execution time for the different memory subsystems. In this figure, the execution time with prefetching for each bus speed is normalized to the execution time with no prefetching for that same memory architecture. If we examine Table II and Figure 2, we see that whenever the bus utilization is greater than 90% without prefetching, the use of prefetching resulted in an increase in execution time. In this region, there is not enough spare bus bandwidth to absorb the extra demand prefetching has placed on it. The increases in execution time when the bus is saturated are not too dramatic, however, because total miss rates rise by small amounts.

Table III. Bus Demand Per Access (BDPA) for the 8-Cycle Data Bus Latency

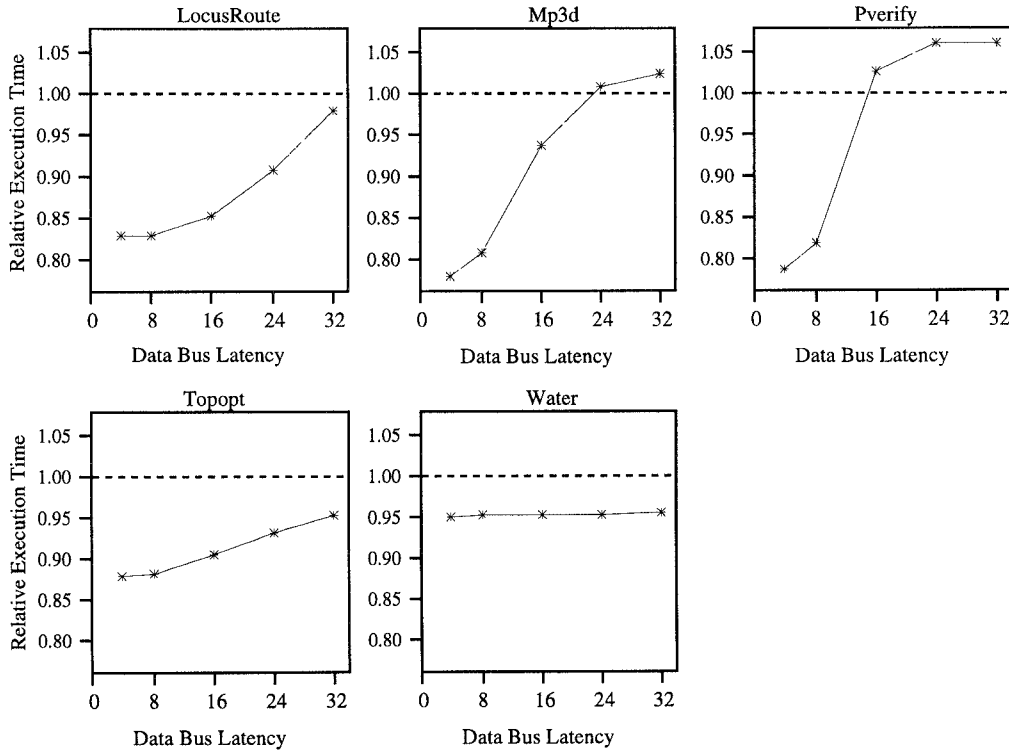| Workload	NP	PREF
LocusRoute	.127	.137
Mp3d	.424	.438
Pverify	.577	.620
Topopt	.524	.599
Water	.051	.055

Fig. 2. Relative execution time for the five workloads with prefetching (PREF), normalized to no prefetching (NP).

Prefetching has an increasingly positive effect on execution time as bus loads become lighter (as the bus gets faster), but in general the performance improvements are not large. There are two reasons for this. First, as already discussed, prefetching causes an increase in memory latency due to increased contention between processors for the bus. In addition, there is an overhead for prefetching in CPU execution time, although it is relatively small in our experiments (no more than 1% of execution time for the PREF scheme). In summary, the execution time results indicate that, even using a highly efficient cache miss predictor, the benefits of prefetching in a bus-based multiprocessor can be marginal, except in the case of a very high bandwidth

memory subsystem. The largest gain in execution time observed is a 29% speedup, and the largest degradation is 6%.

In order to gain insight into how much improvement was actually possible with prefetching, we can look at processor utilization without prefetching. For instance, the average processor utilization for Water is 0.82 with the fastest bus and 0.81 with the slowest bus. Since the best any memory-latency-hiding technique can do is to bring processor utilization to 1, the best speedup that could have been achieved for Water is about 1.2. On the other hand, the processor utilization for Mp3d ranged from 0.39 to 0.22, so it had room for a speedup of 2.5 with the fast bus and 4.5 with the slow bus, ignoring the overhead of prefetch instructions. So, while Mp3d had some of the best speedups, it still falls far short of its maximum potential. For the other workloads, the average processor utilization (without prefetching) for Locus-Route ranged from 0.64 to 0.54; Pverify ranged from 0.36 to 0.17, and Topopt from 0.13 to 0.11. In Topopt, which makes use of hardware barriers, much of the low processor utilization is due to synchronization delays.

It should also be noted that predicting the effectiveness of prefetching for a particular workload can be difficult, because the same workloads achieve both the largest improvement and the largest degradation, depending upon the memory subsystem architecture. This is because applications that put a heavy load on the memory system will see a larger memory latency due to contention and will benefit more from hiding that latency. Those same applications, however, are the first to enter bus saturation and begin degrading with prefetching. In our simulations, this is exemplified by Mp3d and Pverify.

High bus utilization is one reason that results with prefetching are disappointing. Another is the large number of CPU misses despite the use of an oracle miss predictor. In order to understand the various sources and magnitudes of the remaining CPU misses, we analyze the different types of CPU misses. Figure 3 shows the breakdown of the CPU misses for three of the applications—Topopt, Pverify and Mp3d—with an 8-cycle data transfer latency. The misses shown fall into the following categories. They are either invalidation misses (the tags match, but the state has been marked invalid) or nonsharing misses (this is a first use, or the data has been replaced in the cache). A miss of each type is either prefetched (and disappeared from the cache before use) or not prefetched (the miss was not predicted). The fifth type of miss is prefetch-in-progress, which means that the prefetch access was presented to the memory subsystem but did not complete by the time the CPU requested the data. The sum of these five types of misses (the combined height of the five bars) is the CPU miss rate. The other component of the total miss rate (for PREF) is prefetch misses, not shown here. Prefetch misses are those misses that are effectively hidden (turned into hits from the perspective of the CPU) by a prefetch access. The goal of prefetching is to turn as many of the misses in NP into prefetch misses as possible, hopefully without incurring many additional cache misses. It is our use of an oracle for predicting nonsharing misses that allows us to categorize nonsharing misses so precisely. With an imperfect prefetcher, it would not necessarily be clear whether
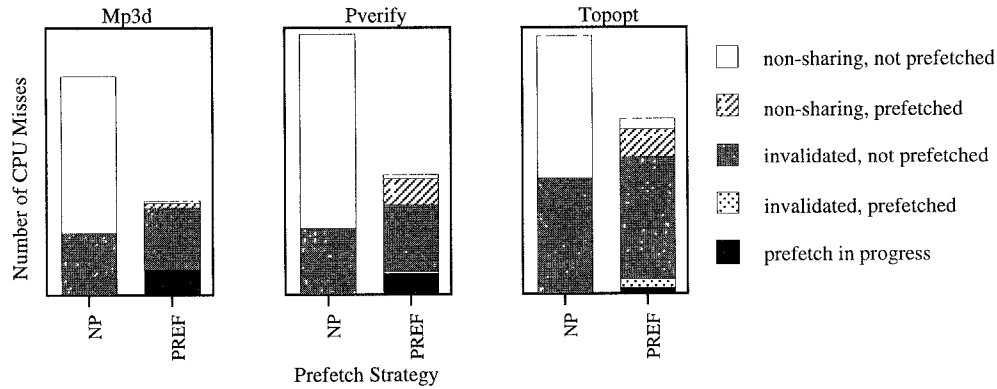
Fig. 3. Sources of CPU misses for Mp3d, Pverify, and Topopt with an 8-cycle data bus latency.

a miss was caused by prefetches permuting the memory reference pattern or by imperfections in the prefetcher.

From these results, we observe that there remain a significant portion of nonsharing CPU misses that are not covered by prefetching. Since our "oracle" prefetcher predicts nonsharing misses perfectly in the absence of prefetches, those that remain are either caused by a prefetch access replacing data that is still being used (*nonsharing, not prefetched* in the figure) or a prefetched cache line being replaced before its use (*nonsharing, prefetched* in the figure). This implies that the degree of conflict between the prefetched data and the current working set can be significant. These two components of the CPU miss rate are particularly important, because they not only represent cache misses not covered by prefetching, but also because they result in bus accesses that were not necessary without prefetching. The *nonsharing, prefetched* misses require an extra bus access because the prefetch access was wasted, and the *nonsharing, not prefetched* misses because without prefetching the access was a cache hit. Therefore, each remaining nonsharing miss represents an increased demand on the bus due to prefetching.

The prefetch-in-progress misses represent as much as 27% of the CPU misses in this figure, and for the slowest data bus (32 cycles), as much as 62%. Their total contribution to the observed memory latency is less than that, however, because a prefetch-in-progress miss latency is typically much less than an entire memory access latency (see Section 5).

Perhaps the most conspicuous result from Figure 3 is that the prefetch algorithm has not affected the invalidation misses. One concern that motivated this study is the hypothesis that, by increasing the interval over which each cache seeks to hold a cache line, prefetching would exacerbate the data-sharing problem, resulting in more invalidate operations and more invalidation misses. Our results do not bear that out for the PREF prefetching strategy. But what is seen is that prefetching has not reduced the number of invalidation CPU misses at all. Clearly there is a limit to how effective prefetching can be if it does not address invalidation misses. While it may be
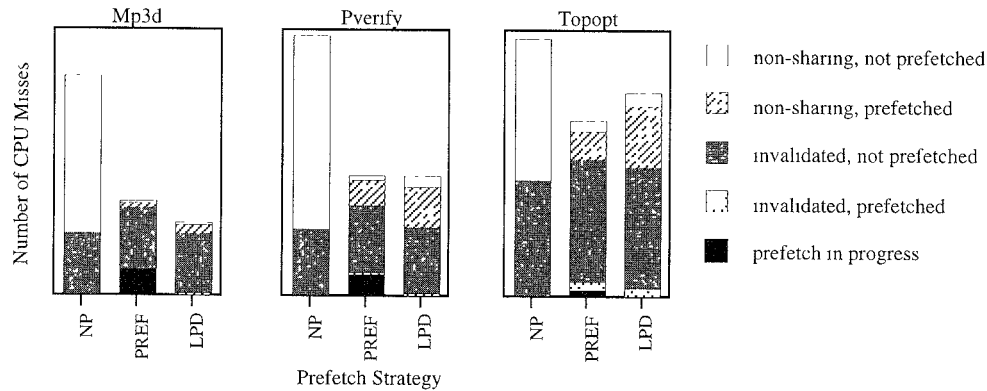
Fig. 4. Sources of CPU misses in Mp3d, Pverify, and Topopt with the Long Prefetch Distance strategy (LPD) and a data bus latency of 8 cycles.

true that prefetching did not exacerbate the data-sharing problem, it has exposed the performance of the applications to the effects of data sharing to a much greater extent.

The results presented so far suggest three primary opportunities for improvement in our multiprocessor prefetcher: the large number of prefetch-in-progress misses, the conflict misses (and additional bus load) introduced by prefetching, and the inability to hide or reduce the latencies due to data sharing. These areas will be investigated in the following sections.

5. REDUCING PREFETCH-IN-PROGRESS MISSES

In PREF, the prefetch distance is relatively close to the best-case memory latency of 100 cycles. Contention can cause the real latency to be much higher, however. For this reason, Mowry et al. [1992] suggest using a larger prefetch distance to ensure that the prefetched data has time to arrive. In this section, we examine the effect of increasing the prefetch distance to 400 instructions, and we will label this prefetching strategy LPD. For the LPD strategy, we wanted a prefetch distance that was high enough to cause the prefetch-in-progress misses to have only an insignificant impact on execution time for most of the bus speeds, without necessarily removing them all. Lower values for the prefetch distance did not quite achieve this. Figure 4 presents the effect of the longer prefetch distance on individual components of the miss rate, and Figure 5 shows the effect on execution time. Figure 4, for the sake of consistency, is given for a data bus latency of 8 cycles, where the prefetch-in-progress misses are not as serious a problem. With the 32-cycle bus, the number of prefetch-in-progress misses is increased in each application by at least a factor of four over those shown.

Increasing the prefetch distance from 100 to 400 will successfully eliminate the majority of prefetch-in-progress misses (for the bus speed shown, they are virtually eliminated; in the worst case, the 32-cycle bus, the LPD strategy eliminates on average 63% of the prefetch-in-progress misses), but at the cost
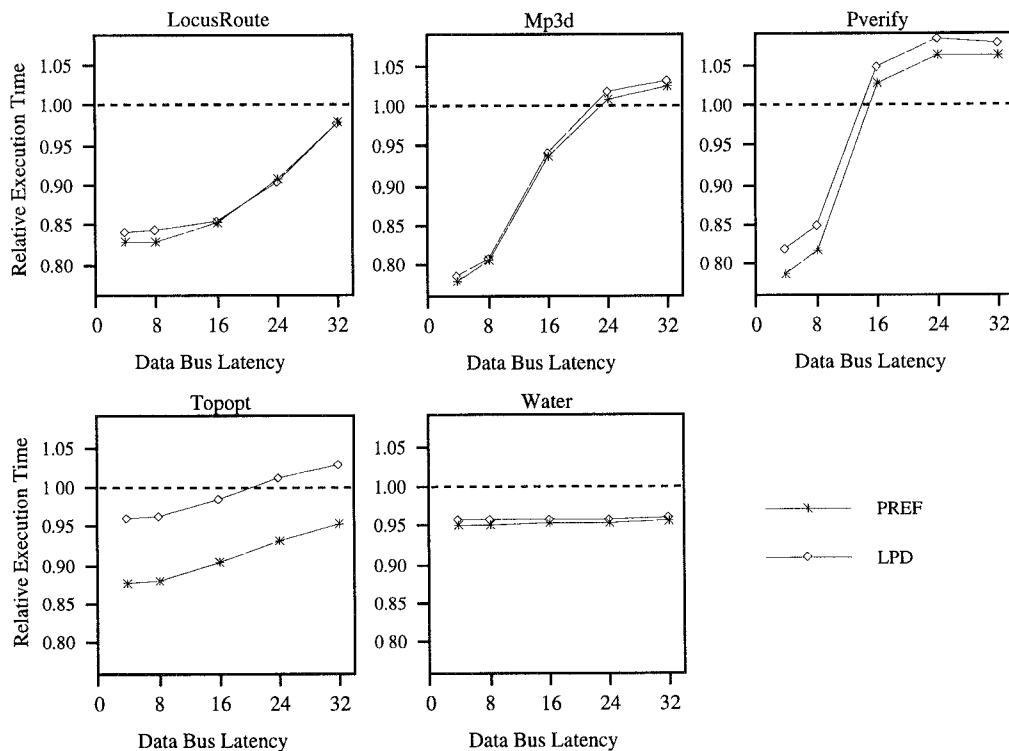
Fig. 5.   Relative execution time, with the Long Prefetch Distance strategy.

of more conflict misses. The earlier the prefetch is begun, the more likely it is to replace data that is still being used. Also, the longer the prefetched data sits in the cache before it is used, the more likely it is to be replaced. The numbers for Pverify and Topopt indicate that the latter is particularly critical.

Trading prefetch-in-progress misses for conflict misses is not wise. Prefetch-in-progress misses are the cheapest type of miss, because the processor has only to wait for an access in progress to complete instead of the entire access time. In Mp3d, for example, while the prefetch-in-progress misses represent 27% of the CPU misses for the 8-cycle bus, they only add 2% to the total execution time. Also, incurring a prefetch-in-progress miss does not increase the load on the bus, while a prefetch-induced conflict miss represents an extra bus operation (as compared to no prefetching). Even with Mp3d, where LPD adds the least number of conflict misses, there is no improvement in execution time for LPD over PREF. Our results indicate that increasing the prefetch distance to the point that virtually all prefetches complete does not pay off. This argues that prefetching algorithms should strive to receive the prefetched data exactly on time. The penalty for being just a few cycles late is small, because the processor must stall only those few cycles, much less than the time of a full access. The penalty for being a few

cycles early is also small, because the chances of losing data before its use are slight over that period. Mowry et al. [1992] also studied prefetch distance, noting that only one of their programs degraded with increasing prefetch distance, but they manually restructured four others to avoid the conflicts that are causing this phenomenon.

The inability of the long prefetch distance to improve prefetching performance is closely tied to the more general problem of prefetch-induced conflict misses; therefore, we will reexamine the LPD strategy in the next section, where we look at one cache architecture that is designed to reduce conflict misses.

6. VICTIM CACHING TO REDUCE CONFLICT MISSES

We have shown that prefetching can increase the number of replacement misses in the cache due to the conflict between the current working set and that part of the future working set that is being prefetched. Those results are based on a direct-mapped cache. In this section, we investigate how a modified cache organization can reduce the magnitude of that conflict and, consequently, whether that improves the effectiveness of prefetching.

In order to see the effect of an alternate cache organization on the magnitude of the prefetch-induced conflicts, we here simulate the same configuration with the addition of a small (8-entry) fully associative victim cache [Jouppi 1990]. Although caches with higher levels of associativity (greater than one) should also lessen the impact of prefetch-induced conflicts, we chose to examine victim caches, because they seem to represent a less-costly approach (in terms of critical-path cache access time, if not size and complexity) that appears to be ideally suited to the problem of prefetch-induced conflicts. Chen [1993] showed that a victim cache matched the performance of a 2-way set-associative cache in the context of hardware prefetching. Our tests showed actually that while an 8-line victim cache was, in general, somewhat less effective than a 2-way set-associative cache at reducing the number of conflict misses, it all but eliminated the particular problem of prefetch-induced conflicts, as this section will show.

A victim cache is a small cache of the blocks most recently replaced in the main cache. It is a good match for prefetch-induced conflicts, because it targets conflict misses on data recently replaced in the cache. When we incur a prefetch-induced conflict miss, the number of instructions between the time the block is replaced in the cache and is then accessed (resulting in a conflict miss) is bounded by the prefetch distance; therefore, prefetch-induced conflicts occur over a very short interval.

A victim cache may in some cases require more hardware than adding 2-way set-associativity to a cache. For example, in our case an 8-line victim cache requires 8 lines of data, tags, and state (on the order of $8 \times 285 = 2280$ bits for a 32-byte line size and 32-bit addresses), while a 2-way set-associative cache will require one extra tag bit per cache line and an LRU bit per set (1536 bits in our configuration). However, caches of the same configuration that are 64KB or larger would require (increasingly) more storage than an

8-line victim cache. In a cache-coherent multiprocessor, the victim caches are slightly more complex than on uniprocessors, because they need hardware support for snooping on the victim cache tags.

A victim cache has the distinct advantage that it does not add any delay to the critical path of a cache lookup in a direct-mapped cache, which is not true for a conventional associative cache. A victim cache lookup occurs only on a main cache miss. Access to the victim cache takes longer than an access to the main cache and ties up the cache longer (for a swap between the main and victim caches), but is much less costly than a main memory access and does not require a bus operation.

In Figure 6 we see that victim caching does indeed have a significant impact on the number of replacement misses caused by prefetching. For example, with the PREF strategy and no victim cache, the prefetch-induced conflict misses (that is the combination of the prefetched and nonprefetched nonsharing misses) represent 22% of the CPU miss rate for Topopt (8-cycle results); with the same strategy and a victim cache, they account for only 1% of the CPU miss rate.

Figure 7 shows the execution time results with victim caches. These results are normalized to the NP results without victim caches. From these results, we make several observations. First, although we still see some small performance degradations with prefetching, they are much smaller than without the victim caches. In fact, in the worst case PREF.vict is no more than 1% above NP.vict, perhaps small enough to allow us to ignore the prefetch-induced conflict problem when the architecture supports an appropriate cache configuration (e.g., set-associative cache or direct-mapped with a victim cache) at each level.

Second, we observe that in nearly all the cases the combined effect on speedup of prefetching and the victim cache together is greater than the sum of their individual contributions. The effect is that prefetching has more benefit with a victim cache than without. For example, with Topopt on a 4-cycle bus, PREF provides a 14% speedup over NP, while PREF.vict provides a 22% speedup over NP.vict. There are two reasons why this occurs. As already discussed, the victim cache eliminates a negative side-effect of prefetching (the additional conflict misses), which previously detracted from the potential speedups with prefetching. Also, because the victim cache lowers the overall miss rate, it decreases the load on the bus. This means that the same configuration will be less sensitive to bus contention effects, which we have shown to limit the effectiveness of prefetching. For example, a region that without the victim cache saturated the bus (Pverify at 16 cycles is a case in point) and thus saw no benefit from prefetching may with a victim cache no longer be in bus saturation. We would expect that the increase in prefetching effectiveness due to the victim cache would be even greater in a real prefetching system without an oracle to determine potential cache misses, since conflict misses are more difficult for a compiler to identify than capacity misses.

The third observation is that increasing the prefetch distance is no longer clearly harmful. In only one of the applications (Topopt) is LPD.vict notice-
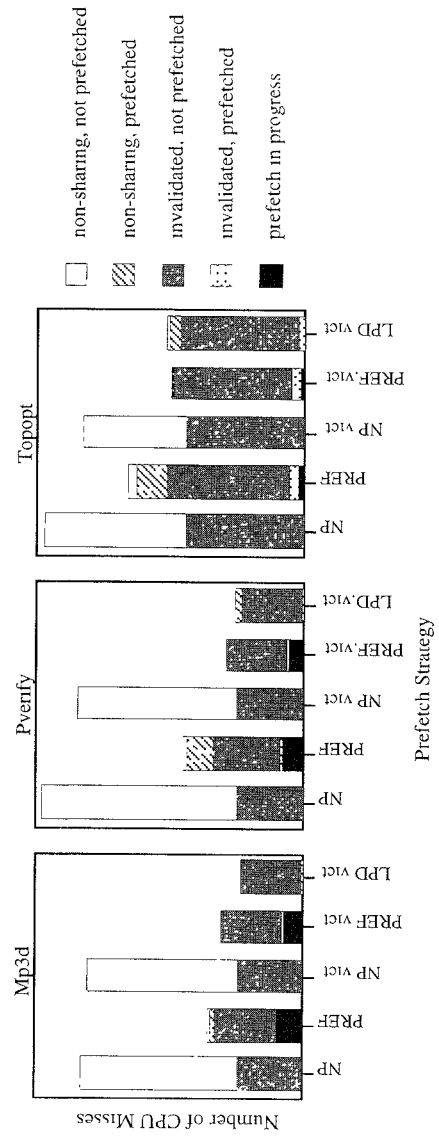
Fig. 6   Sources of CPU misses in Mp3d, Pverify, and Topopt with a victim cache.
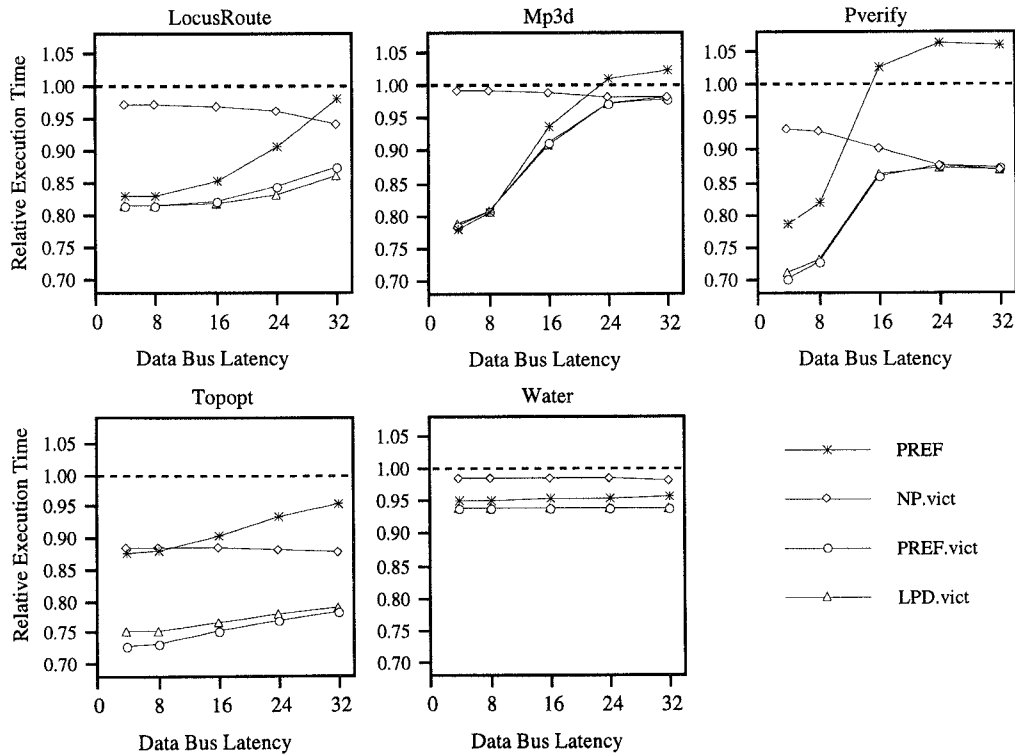
Fig. 7.   Execution times (relative to no prefetching and no victim cache) for the five workloads with a victim cache.

ably inferior to PREF.vict. The victim cache now catches most of the additional conflicts caused by the increased prefetch distance. In the one case (LocusRoute) where increasing the prefetch distance is effective (comparing LPD.vict to PREF.vict), it is most effective at high bus utilization (but short of saturation), where the number of and the delays associated with prefetch-in-progress misses are greatest. But in most cases, LPD.vict is not clearly better or worse that PREF.vict, because the performance loss due to the prefetch-in-progress misses was never very great. Perhaps the most important aspect of this result, then, is not that it allows the compiler to place the prefetches earlier, but that the performance of prefetching is less sensitive to the exact placement of prefetches when the cache is not strictly direct mapped. This should allow the compiler much more flexibility in prefetch placement. These results are for an 8-line victim cache. A 4-line victim cache was not found to be sufficient to eliminate enough of the prefetch-induced conflicts for these applications in our studies, as PREF still visibly outperformed LPD for most of the applications.

Finally, we observe that although we have eliminated some of the drawbacks (which resulted in occasional performance degradations) of prefetching with the victim cache, prefetching still does not provide significant speedups

when the bus is saturated (the bus is more than 90% utilized with NP.vict for Pverify at 16–32 cycles and Mp3d at 24 and 32 cycles). So, although a cache organization that is more forgiving of cache conflicts mitigates some of the drawbacks of prefetching, in a memory-bottlenecked system prefetching still does not attack the problem at the bottleneck component, which is the total number of interconnect/memory operations.

A victim cache also fails to help with the largest single component of the CPU miss rate, which is the invalidation misses. These are dealt with in the following section.

7. REDUCING SHARED-DATA LATENCIES

We currently know of no available compiler-based prefetching algorithms for dealing with invalidation misses. We show in this section that some simple heuristics (simpler, for instance, than dataflow analysis across processes), such as recognizing write-shared data and blindly prefetching it more often, prefetching writes differently than reads, and recognizing read-modify-write patterns, can improve performance.

There are several opportunities to reduce the impact of sharing traffic observed with the PREF scheme. Certainly, if we can do a better job of predicting and prefetching invalidation misses, we can achieve much better miss coverage. It is clear from Figures 3, 4, and 6 that our current miss predictor, although it is extremely efficient at predicting nonsharing misses, is inadequate for predicting invalidation misses. This is because the prefetch algorithm we emulate is tailored for a uniprocessor. But even as better algorithms appear, predicting invalidation misses will remain a much more difficult problem than predicting nonsharing misses due to the nondeterministic nature of invalidation traffic. We investigate three mechanisms for making prefetching more effective in the presence of data-sharing traffic. Section 7.1 examines a better heuristic for prefetching invalidation misses, and Section 7.2 studies the effect on prefetching performance of a compiler algorithm that reduces invalidation misses by restructuring shared data.

In addition, prefetching can increase sharing traffic in ways not obvious from the data shown so far. A successful prefetch of a write miss to shared data can increase the bus traffic, even if it causes no CPU misses, by causing an unnecessary invalidate operation. In Section 7.3 we show that exclusive prefetching can solve not only this problem, but also the more general problem of unnecessary invalidate operations.

7.1 Prefetching Invalidation Misses

We saw in Section 4 and Figure 3 that a clear limit to the effectiveness of prefetching is invalidation misses on shared data. None of the traditional (uniprocessor-based) prefetching strategies we have looked at so far successfully reduces the less-predictable invalidation misses, which are the largest component of CPU misses in each of the workloads. In fact, the more effectively we prefetch nonsharing misses, the more critical invalidation misses become to the performance of the application with prefetching. For
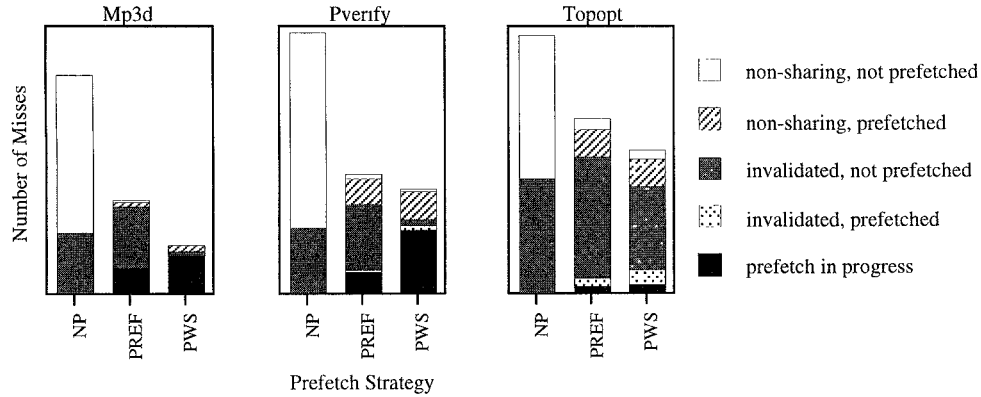
Fig. 8.   Sources of CPU misses with enhanced write-shared data prefetching.

example, as seen in Figure 3, without prefetching, invalidation misses represent 29% of the CPU miss rate for Mp3d, but with prefetching they represent 70% of the CPU miss rate and 96% of the adjusted CPU miss rate. In other words, prefetching has made the applications much more sensitive to the data-sharing problem.

To improve the coverage of these externally caused and therefore less-predictable misses, we introduce some redundant prefetches to cache lines known to be write-shared. They are redundant in the uniprocessor sense, i.e., they are issued for data that would reside in the cache, were it not for invalidations. To emulate a prefetch algorithm that prefetches write-shared data that exhibits poor temporal locality (under the premise that the longer a shared cache line has resided in the cache without being accessed, the more likely it is to have been invalidated), we ran the write-shared data from each trace through a 16-line associative cache filter to get a first-order approximation of temporal locality, selecting the misses for prefetching. These are prefetched in addition to all prefetches identified by PREF. This strategy, labeled PWS, increases the prefetching instruction overhead (but is still less than 4%), but improves our coverage of invalidation misses. A compiler algorithm could obtain the same effect by using the Mowry et al. [1992] algorithm, but assuming a much smaller cache size when dealing with data known to be write-shared.

We can see from Figure 8 that the coverage of invalidation misses improves considerably with PWS, as the invalidation portion of the CPU miss rate drops significantly, betweem 20% and 91% (on average a 56% drop). As a result, over the range of bus utilizations (data transfer latencies) for which prefetching is already viable, improvements in execution time are achieved for most of the workloads, as seen in Figure 9. The fastest bus (4-cycle) results allow us to see the benefit of the improved prefetching of write-shared data most clearly in isolation from the memory contention effects. For that architecture, the speedup of PWS relative to PREF ranged from 0% (Water) to 15% (Pverify, where PREF is 27% faster than no prefetching, while PWS
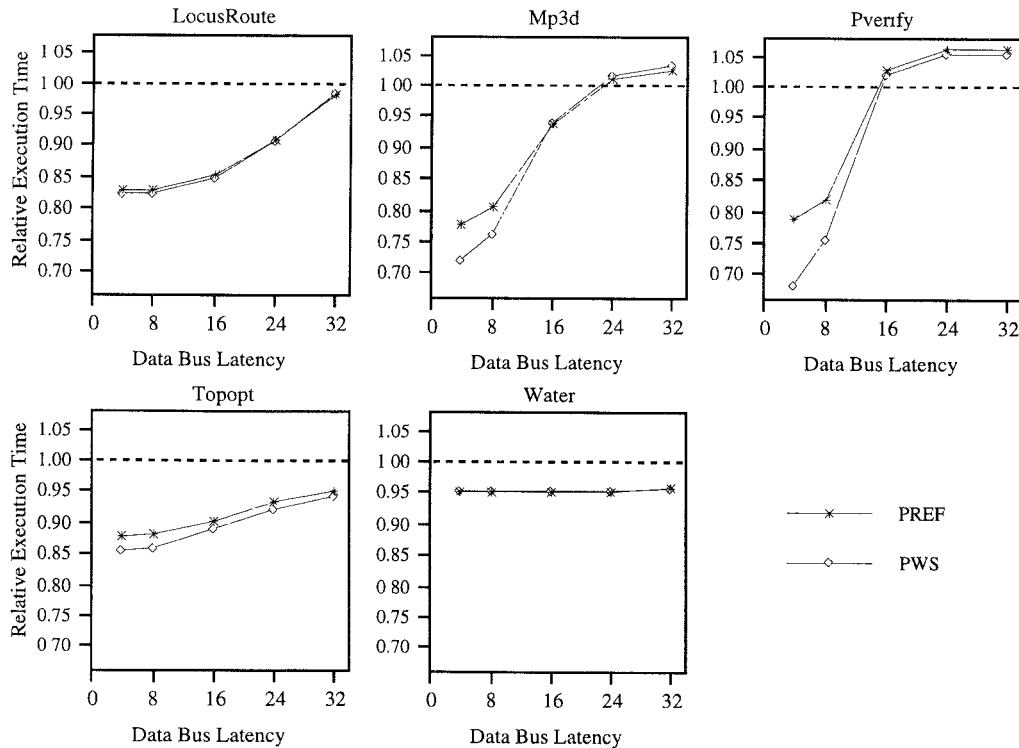

Fig. 9. Execution times (relative to no prefetching) for the five workloads with the enhanced write-shared data prefetching.

achieves a 47% speedup over no prefetching), and CPU miss rates for PWS are 11% to 64% lower than PREF. One reason for the consistent reduction in CPU misses by the write-shared algorithm is that, although PWS increases the number of prefetches, it does not significantly increase the number of prefetch-induced CPU conflict misses.

7.2 Restructuring of Shared Data

Because of the nondeterministic behavior of interprocessor sharing, predicting invalidation misses on multiprocessors is more difficult than predicting nonsharing misses, where the algorithm will be the same for both uniprocessors and multiprocessors. In this section, we investigate the extent to which reducing sharing traffic through compiler-based shared-data restructuring can eliminate or reduce the need for multiprocessor-specific prefetching algorithms.

Sharing traffic consists of both true and false sharing. While the amount of true sharing is inherent to the algorithm used by the program, false sharing can be reduced by improved processor locality of shared data. False sharing occurs when a cache line is shared between two processor caches, but each is

Table IV. Total Invalidation and False-Sharing Miss Rates with No Prefetching

Workload	Total Miss Rate	Total Invalidation Miss Rate	Total False Sharing Miss Rate
Pverify	4.52	1.18%	1.11%
Topopt	4.16	1.90%	1.39%
Locus	0.88	0.14%	0.08%
Mp3d	2.00	0.58%	0.19%
Water	0.30	0.06%	0.04%

accessing different data in it. When one processor modifies a data location, it causes an invalidation in the other's cache, because cache coherency is maintained on a cache block basis. We record a false-sharing miss if an invalidation miss is caused by a write from another processor to a word in the local cache line that the local processor has not accessed.[2] Table IV shows that for most of the benchmarks, over half of the invalidation misses could be attributed to false sharing, even for the SPLASH benchmarks, which have been hand-tuned for processor locality, although the total amount of false sharing in those benchmarks is rather low. We show results for a 32-byte cache line; previous work [Eggers and Jeremiassen 1991; Torellas et al. 1994] demonstrates that false sharing goes up significantly with larger block sizes. In Jeremiassen and Eggers [1992; 1994], an algorithm is presented for restructuring shared data to reduce false-sharing. While the technique improves overall performance, for the purpose of this study we are only interested in whether doing so makes prefetching more viable. Table V and Figure 10 show the result of some of the prefetching strategies on restructured Topopt and Pverify. The other programs are improved less significantly, because they have already been optimized for processor locality by programmer-based restructuring.

The restructured programs experience a 6% decrease in the total miss rate for Pverify and a 67% decrease for Topopt. These reductions are achieved for the most part by significant reductions in the false-sharing miss rate. In Pverify that reduction is offset somewhat by an increase in the nonsharing miss rate, while in Topopt an increase in data locality is achieved as a side effect of the data restructuring, causing the nonsharing miss rate to decrease also. Because we use trace-driven simulation, it is difficult to compare execution times of different traces accurately, since it is not always clear exactly what fraction of the total execution time was captured. This means that we cannot calculate the raw performance improvement of restructuring. However, we can measure the incremental performance of prefetching on a program that has been restructured. This is exactly what makes restructuring interesting for this research, since the invalidation misses were shown to

---

[2] Dubois et al.'s definition of false sharing [Dubois et al. 1993], in that it calculates false sharing over the lifetime of a cache line, is more accurate than the definition we use. However, we have measured only small differences between that definition and ours. Here, at any rate, we are less concerned with exactly how much false sharing exists and how it is measured than with how many sharing misses we can eliminate.

Table V.   Miss Rates for Restructured Programs with a Data Transfer Latency of 8 Cycles

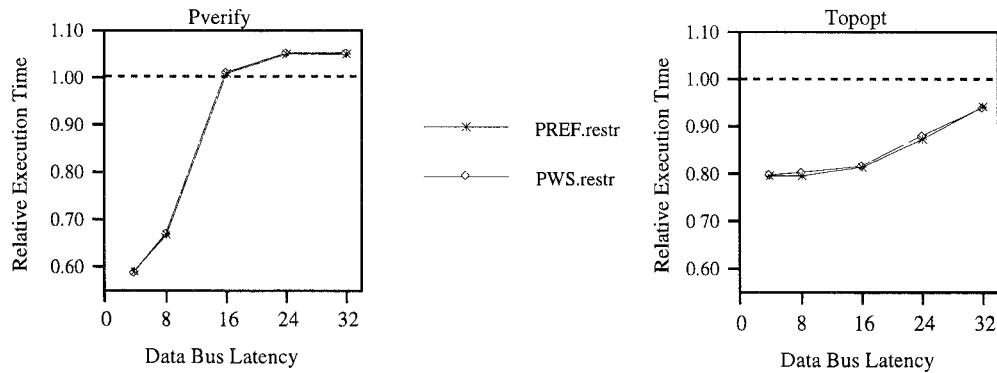| Workload	Prefetch Discipline	CPU MR	Total MR	Total Inval MR	Total FS MR
Pverify	NP	4.26%	4.26%	0.12%	0.09%
	PREF	1.91%	4.54%	0.18%	0.09%
	PWS	1.91%	4.59%	0.18%	0.10%
Topopt	NP	1.37%	1.37%	0.15%	0.05%
	PREF	0.37%	1.48%	0.17%	0.07%
	PWS	0.35%	1.48%	0.18%	0.07%

Fig. 10.   Execution times (relative to no prefetching on the restructured program) for Pverify and Topopt, applying two prefetching strategies to restructured programs.

be a limiting factor in the performance of our prefetching strategies, particularly PREF. Figure 10 shows the performance of the PREF and PWS strategies applied to the two restructured programs. In this figure, the results are normalized to the execution times of the restructured programs without prefetching.

Despite the fact that both restructured programs are less sensitive to memory latencies (due to a lower total miss rate), they experienced, in general, greater improvements from prefetching than the original programs. This can be seen by comparing Figure 9 with Figure 10; for example, Pverify at 4 cycles experiences speedups with the PREF strategy of 27% without restructuring (PREF, relative to NP) and 69% with (PREF.restr, relative to NP.restr), and with the PWS strategy, 47% without and 70% with restructuring. This validates our assertion that invalidation misses limit the effectiveness of our prefetching algorithms.

Although PREF.restr and PWS.restr both show more improvement over NP.restr than PREF and PWS showed over NP for both programs, the improvement is much more significant in PREF.restr. Consequently, the distinction between PREF and PWS is almost nonexistent in the restructured programs. This is not surprising in that both the PWS algorithm and the restructuring are attempting to attack the same problem. We conclude that when restructuring is effective at significantly reducing the invalidation miss

rate, a simpler, uniprocessor-based prefetch algorithm can be used in place of one tuned for multiprocessor data sharing.

7.3 Exclusive Prefetching

In the prefetching strategies simulated thus far, prefetches have looked to the bus like reads. In the Illinois coherency protocol a line is read into the cache in exclusive mode if no other cache holds that line currently; otherwise it is cached in shared mode on a read. In that case a prefetch of a write miss would fetch (shared) data in shared mode; the write (now likely a hit) would then require an invalidate operation on the bus. This turns one bus operation (read with intent to modify, which loads the data and invalidates in one operation) into two. This can be avoided by an exclusive prefetch, which prefetches data into the cache in exclusive mode, invalidating copies in other caches. In migratory sharing (where only one processor at a time is typically accessing cache line), exclusive prefetching saves a bus operation; however, when there is interprocessor contention for cache lines, an exclusive prefetch to write-shared data can cause many more invalidation misses.

The problem of unnecessary invalidate operations is not limited to shared-mode prefetching, but is a more general problem in parallel applications. The WREX prefetching strategy targets the extra invalidates caused by shared-mode prefetching, while the RDEX strategy also targets the general problem.

In the WREX prefetching strategy, if the expected miss is a write, the algorithm issues an exclusive prefetch for that line. If the prefetch misses, the line is brought into the cache in exclusive mode, invalidating any copies in other caches. If the prefetch hits in the cache, no bus operation is initiated, even if the cache line is in the shared state. With the Illinois protocol, all reads of cache lines that are not currently in another cache enter the exclusive state immediately, so there is only a difference between PREF and WREX when a miss occurs for a line that is shared among caches.

In Figure 11, we see that the WREX prefetching strategy is ineffective at reducing execution time over PREF for four of the five applications (with Topopt, execution time dropped about 2%). The primary reason is that there are few write misses, since many writes are preceded over a short distance by a read to the same location. In other words, the WREX strategy only targets unnecessary invalidate operations caused by prefetching write accesses, a problem which is only evident in one of the applications.

With more aggressive use of exclusive prefetching, we can attack the more general problem of unnecessary invalidates caused by both prefetch and nonprefetch accesses. An unnecessary invalidate operation occurs when a read-modify-write pattern results in a read miss followed by a write hit for a cache line currently shared by another cache. The result is two bus operations (a shared read, followed by an invalidate). If preceded by an exclusive prefetch, the write hit will not require a second bus operation, as long as there are no intervening accesses by another processor to the cache line between the prefetch and the write. If the compiler can recognize a read-modify-write pattern over a short span of instructions, it can issue an exclusive
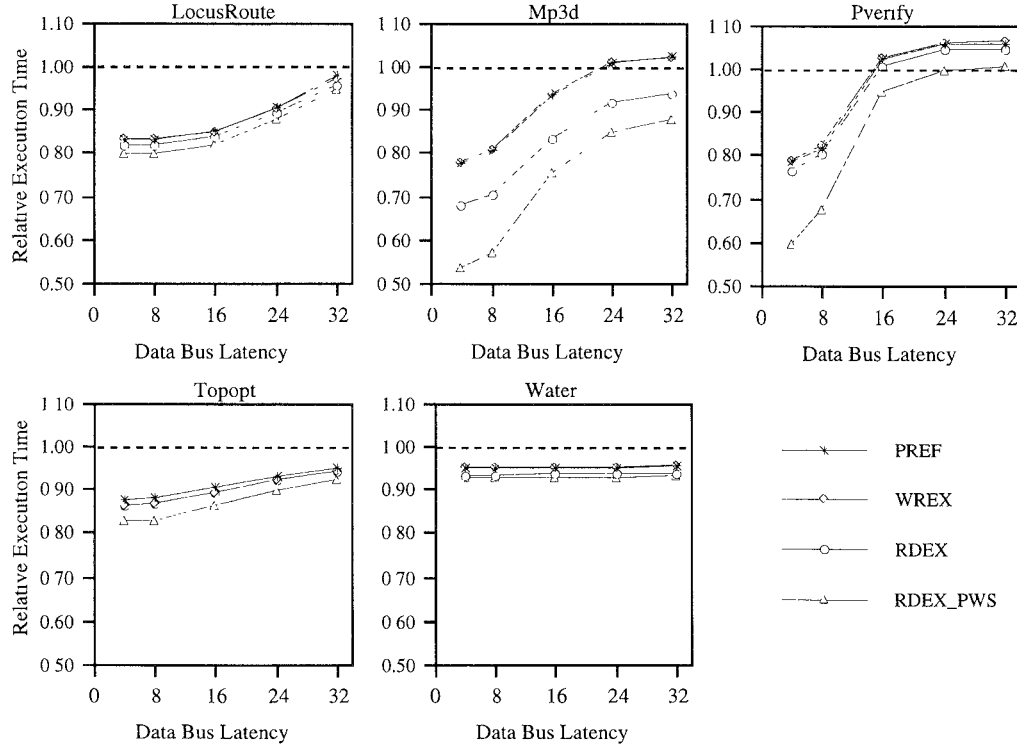
Fig. 11.   Execution times (relative to no prefetching) for the five workloads with exclusive prefetching.

prefetch for the leading read miss. Mowry and Gupta [1991] take advantage of this in their programmer-directed prefetching study.

Therefore, in the RDEX prefetching strategy, we modify the WREX algorithm to do exclusive prefetching also when a read miss is followed by a write to the same word within 100 instructions. We chose to recognize the read-modify-write pattern for words rather than cache lines, because we felt that when access patterns were complex, it would be difficult for a compiler to recognize these patterns on a cache line basis, and when the access patterns were simple (e.g., uniform access to array elements), the results (analysis by word or by cache line) would be identical.

In Figure 12 we see that RDEX, like WREX, does not reduce miss rates. This is not surprising since the goal was to eliminate invalidate operations (which are not shown in the miss rate graphs). The effect of the reduced invalidation traffic can be observed in that execution times decrease without any reduction in the miss rate. The RDEX strategy can potentially increase the invalidation miss rate, but in our measurements those increases were very small, so there was no significant cost to issuing invalidations early. This is not guaranteed to always be true, however, so exclusive prefetching
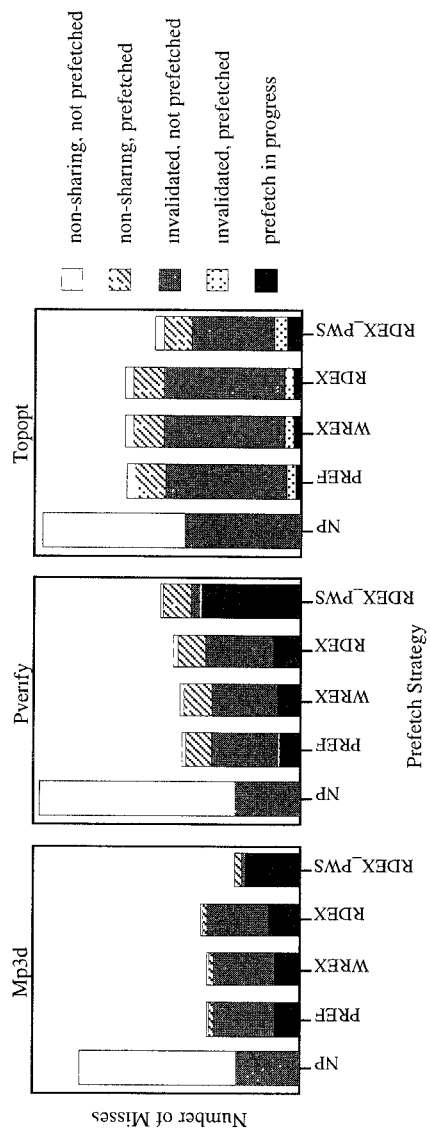
Fig. 12. Sources of CPU misses with exclusive prefetching.

Table VI.   Bus Demand Per Access for the 8-Cycle Data Bus Latency

Workload	NP	PREF	WREX	RDEX	RDEX_PWS
LocusRoute	.127	.137	.136	.131	.130
Mp3d	.424	.438	.438	.376	.333
Pverify	.577	.620	.620	.605	.563
Topopt	.524	.599	.594	.594	.598
Water	.051	.055	.054	.048	.047

needs to be done with some caution; in particular, it should be avoided when interprocessor contention for a cache line is expected to be high.

This is the first application of prefetching in this article that can actually decrease the number of bus operations relative to no prefetching. We can see in Table VI and Figure 11 that bus demands have improved with the more aggressive exclusive prefetching strategy; the consequence is improved execution time for all bus speeds. The significance of this result is that it shows that prefetching *can* be a win, even on a memory-saturated multiprocessor.

We also see from Figure 11 that the RDEX strategy, when used in conjunction with PWS, made the write-shared algorithm significantly more effective. The two work well together for two reasons. First, the exclusive prefetching lowers bus demands, thus making PWS (which is not useful when the bus is saturated) effective over a wider range of bus speeds. Second, PWS allows RDEX to attack the component of misses responsible for the vast majority of unnecessary invalidate operations—invalidation misses. So, for instance, we see that RDEX_PWS provides speedups as high as 28% over RDEX alone (up to 34% over PWS alone), and the combination provides speedups as high as 45% over PREF. In fact, in all five applications RDEX_ PWS outperforms all other strategies noticeably, even with the slowest memory subsystem.

8. PUTTING IT ALL TOGETHER

We have demonstrated several architectural and compiler-oriented techniques which increase the effectiveness of prefetching. In this section, we want to see how far we have come. In other words, when we use these techniques together, how effective can prefetching be over the range of memory architectures and applications that we have been studying? In Figure 13, we have applied several (but not all) of these techniques. In particular, we have applied a combination of the PWS and RDEX prefetching strategies to an architecture with victim caches. In this figure, the PREF result is normalized to NP, and the first RDEX_PWS.vict result is normalized to NP.vict. This allows us to see the overall increase in effectiveness of the enhanced prefetching strategy, independent of the benefits of the victim cache.

With this combination (RDEX_PWS.vict), we achieved speedups due only to prefetching that were much more significant than our original basic prefetcher (PREF), as high as 83%. In addition, we no longer experienced any performance degradations, the minimum speedup being 6%. These results

indicate that, with the right cache architecture and with careful application of prefetching by the compiler, performance improvements can be both extensive (covering a wide range of memory bandwidths) and, in some cases, very significant.

The third line in Figure 13 shows the absolute performance gain achieved from the combination of using a victim cache and the composite prefetching algorithm. Applying both architectural and prefetching techniques achieved speedups between 9% and 95% over the base architectures, as opposed to a maximum speedup of 29% and slowdowns as much as 6% with the base uniprocessor-style prefetcher.

What we see, then, is that while none of the individual solutions provided very dramatic improvements, taken together, the total solution is significant. This is due to the fact that the different techniques attack different aspects of the prefetching problem, and in some cases there was synergy among the different approaches.

In interpreting these results, it should be remembered that our oracle-based prefetch algorithm is likely to underestimate prefetch instruction overhead and overestimate the ability to identify nonsharing misses. The latter will have a mixed effect; more prefetching minimizes the CPU miss rate but also maximizes bus demand due to cache conflicts. Nonetheless, these results give us high confidence that, with a combination of techniques, prefetching can be made profitable across a very wide array of multiprocessor memory architectures.

9. SUMMARY AND CONCLUSIONS

In a multiprocessor system with limited memory bandwidth, compiler-directed prefetching algorithms are not guaranteed to improve performance, even if they reduce the CPU-observed miss rate successfully. They can increase the load on the memory subsystem (through prefetch-induced cache conflicts and unnecessary invalidate operations) and have difficulty hiding invalidation misses. Not only are slowdowns possible, but performance can be unpredictable. We found that the applications that benefited most from prefetching were the same applications that suffered the most when the architecture was varied by changing bus speeds. We show also that when prefetching is effective at reducing the effect of nonsharing cache misses, it makes the applications more sensitive to the data-sharing problem. With an assortment of architectural and compiler techniques, however, each of the drawbacks of prefetching can be alleviated.

For the systems we simulated, we have shown the following techniques to be successful in increasing the effectiveness of prefetching. (1) A cache design that is more forgiving of cache conflicts than a direct-mapped cache, in this case an additional victim cache, eliminates most of the prefetch-induced conflict misses that both increase the CPU miss rate and the load on the bus and memory. This both increases the effectiveness of prefetching, and makes the application less sensitive to prefetch distance, allowing the compiler more freedom in prefetch placement. (2) A prefetch algorithm that is targeted at invalidation misses as well as nonsharing misses can greatly increase the
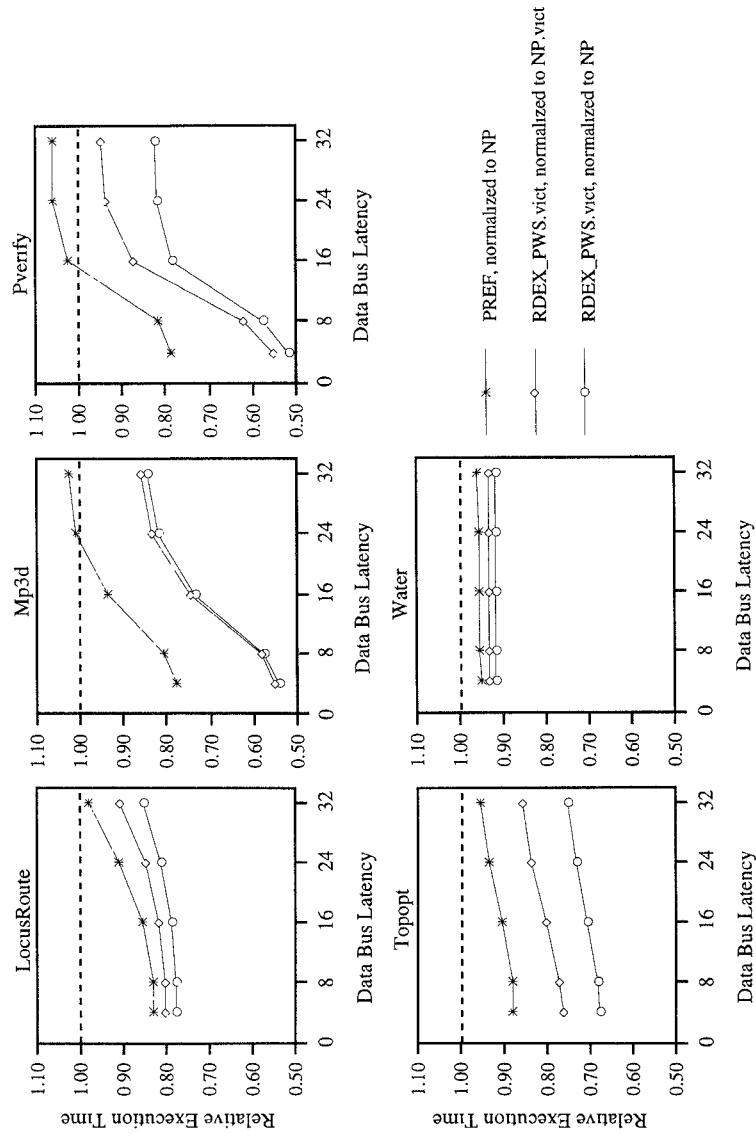
Fig. 13. Execution times, applying both the RDEX and PWS prefetching strategies to a system with victim caches.

coverage of prefetching on a shared-memory multiprocessor. (3) A prefetch algorithm that makes effective use of exclusive prefetching can significantly reduce the number of invalidate operations, and thus reduce the load on the memory subsystem. (4) When restructuring shared data is effective at increasing processor locality (and thus reducing the number of invalidation misses), it enhances the performance of prefetching and allows the use of a simpler, uniprocessor-oriented prefetching algorithm. Two of these techniques (a victim cache as an architectural improvement and restructuring of shared data as a compiler algorithm) that have been shown elsewhere to improve performance independent of prefetching also make prefetching more effective after they have been applied.

Although the individual contribution of any one of these techniques is not very dramatic, the combined effect of several of these techniques can be very significant. With a combination of these techniques, then, prefetching can be made viable across a much wider range of parallel applications and memory subsystem architectures, even when the memory subsystem represents the bottleneck in the multiprocessor system.

ACKNOWLEDGMENTS

REFERENCES

CALLAHAN, D., KENNEDY, K., AND PORTERFIELD, A. 1991. Software prefetching. In *The 4th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, 40–52.

CHEN, T.-F. 1993. Data prefetching for high-performance processors. Tech. Rep. No. UW TR-93-07-01, Ph.D. thesis, Univ. of Washington, Seattle, Wash. July.

CHEN, T.-F. AND BAER, J.-L. 1994. A performance study of software and hardware data prefetching schemes. In *21st Annual International Symposium on Computer Architecture*. ACM/IEEE, New York, 223–232.

CHEN, T.-F. AND BAER, J.-L. 1992. Reducing memory latency via non-blocking and prefetching caches. In *The 4th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, 51–61.

CHEN, W. Y., BRINGMANN, R. A., MAHLKE, S. A., HANK, R. E., AND SICOLO, J. E. 1992. An efficient architecture for loop based data preloading. In *25th International Symposium on Microarchitecture*. ACM/IEEE, New York, 92–101.

CHEN, W. Y., MAHLKE, S. A., CHANG, P. P., AND HWU, W. W. 1991. Data access microarchitectures for super-scalar processors with compiler-assisted data prefetching. In *24th International Symposium on Microarchitecture*. ACM/IEEE, New York, 69–73.

DEVADAS, S. AND NEWTON, A. R. 1987. Topological optimization of multiple level array logic. *IEEE Trans. Comput. Aid. Des.* (Nov.), 915–941.

DUBOIS, M., SKEPPSTEDT, J., RICCIULLI, L., RAMAMURTHY, K., AND STENSTROM, P. 1993. The detection and elimination of useless misses in multiprocessors. In *20th Annual International Symposium on Computer Architecture*. ACM/IEEE, New York, 88–97.

EDMONDSON, J. AND RUBINFIELD, P. 1994. An overview of the 21164 AXP microprocessor. In *Hot Chips VI*, 1–8.

EGGERS, S. J   1991. Simplicity versus accuracy in a model of cache coherency overhead. *IEEE Trans. Comput. 40*, 8 (Aug.), 893–906

EGGERS, S. J. 1989. Simulation analysis of data sharing in shared memory multiprocessors. Tech. Rep No UCB/CSD 89/501 Ph.D. thesis, Univ of California, Berkeley Mar.

EGGERS, S. J. AND JEREMIASSEN, T. E. 1991 Eliminating false sharing In *International Conference on Parallel Processing*. Vol. 1. IEEE, New York, 377–381.

EGGERS, S. J., KEPPEL, D. R., KOLDINGER, E. J., AND LEVY, H. M. 1990. Techniques for inline tracing on a shared-memory multiprocessor. In *Proceedings of the 1990 ACM Sigmetrics* ACM, New York, 37–47.

HENNESSY, J. L. AND JOUPPI, N. P. 1991. Computer technology and architecture: An evolving interaction. *IEEE Comput. 24*, 9 (Sept.), 18–29.

JEREMIASSEN, T. E AND EGGERS, S. J. 1994. Static analysis of barrier synchronization in explicitly parallel programs. In *International Conference on Parallel Architectures and Compilation Techniques*. ACM, New York, 171–180

JEREMIASSEN, T. E. AND EGGERS, S. J. 1992. Computing per-process summary side-effect information. In *5th International Workshop on Languages and Compilers for Parallel Computing*. Lecture Notes on Computer Science, vol. 757. Springer-Verlag, New York, 175–191.

JOUPPI, N. P. 1990. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *17th Annual International Symposium on Computer Architecture*. ACM/IEEE, New York, 364–373.

KROFT, D. 1981. Lockup-free instruction fetch/prefetch cache organization. In *8th Annual International Symposium on Computer Architecture*. ACM/IEEE, New York, 81–87.

LENOSKI, D., LAUDON, J., JOE, T., NAKAHIRA, D., STEVENS, L., GUPTA, A., AND HENNESSY, J. 1993. The dash prototype: Logic overhead and performance. *IEEE Trans. Parallel Distrib. Syst. 4*, 1 (Jan.), 41–61.

LOVETT, R. AND THAKKAR, S. 1988. The symmetry multiprocessor system. In *International Conference on Parallel Processing*. IEEE, New York, 303–310.

MA, H.-K. T., DEVADAS, S., WEI, R., AND SANGIOVANNI-VINCENTELLI, A. 1987. Logic verification algorithms and their parallel implementation. In *24th Design Automation Conference*. 283–290.

MICROPROCESSOR. 1994a. *Microprocess. Rep. 8*, 2, 1–8.

MICROPROCESSOR. 1994b. *Microprocess. Rep. 8*, 13, 1–9.

MOTOROLA. 1990. *MC88100 RISC Microprocessor User's Manual*. Prentice-Hall, Englewood Cliffs, N J.

MOWRY, T. C. AND GUPTA, A. 1991. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *J. Parallel Distrib. Comput. 12*, 2 (June), 87–106.

MOWRY, T. C., LAM, M. S., AND GUPTA, A. 1992. Design and evaluation of a compiler algorithm for prefetching. In *The 5th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, 62–73.

PAPAMARCOS, M. S. AND PATEL, J. H. 1984. A low-overhead coherence solution for multiprocessors with private cache memories. In *11th Annual International Symposium on Computer Architecture*. ACM/IEEE, New York, 348–354.

SCHEURICH, C. AND DUBOIS, M. 1991. Lockup-free caches in high-performance multiprocessors. *J. Parallel Distrib. Comput. 11*, 1 (Jan.), 25–36.

SINGH, J. P., WEBER, W., AND GUPTA, A. 1991. SPLASH: Stanford parallel applications for shared-memory. Tech Rep. CSL-TR-91-469. Comput. Syst. Lab., Stanford Univ., Stanford, Calif.

SOHI, G. S. AND FRANKLIN, M. 1991. High-bandwidth data memory systems for superscalar processor. In *The 4th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, 53–62

TORELLAS, J., LAM, M. S., AND HENNESSY, J L. 1994. False sharing and spatial locality in multiprocessor caches. *IEEE Trans. Comput. 43*, 6 (June), 651–663.

TULLSEN, D. M. AND EGGERS, S. J. 1993. Limitations of cache prefetching on a bus-based multiprocessor. In *20th Annual International Symposium on Computer Architecture*. ACM/IEEE, New York, 278–288.