Prefetching and memory system behavior of the SPEC95 benchmark suite

by M. J. Charney
T. R. Puzak

This paper presents instruction and data cache miss rates for the SPEC95™ benchmark suite. We have simulated the instruction and data traffic resulting from 500 million instructions of each of the 18 programs. Simulation results show that only a few of the applications place more than modest demands on the memory system. This was noticed for instruction caches, where only a few workloads required more than a 32Kb cache to achieve miss rates of less than one miss every 1000 instructions. We also analyze two prefetching algorithms using the SPEC95 workload: next-sequential prefetching and shadow-directory prefetching. Each prefetching algorithm is evaluated using three performance metrics: coverage, accuracy, and traffic. Variations in each prefetching algorithm involve the use of a confirmation mechanism that receives feedback information about the quality of each prefetch. With confirmation, the prefetching algorithm is able to enhance the accuracy of prefetching decisions. The results show that shadow-directory prefetching averages miss coverage about ten percent higher than next-sequential prefetching when used in prefetching instructions (about 60 percent coverage for next-sequential prefetching versus 70 percent for shadow-directory prefetching). The prefetching accuracy for both algorithms is more than 90 percent when a confirmation mechanism is used. In general, data prefetching is shown to be less accurate and to provide less coverage than instruction prefetching. Shadow-directory prefetching averaged about a 40 percent miss coverage versus a 25 percent miss coverage for next-sequential prefetching. Prefetching accuracy is over 70 percent when confirmation is applied.

1. Introduction

We examine the memory referencing behavior of the Standard Performance Evaluation Corporation CPU benchmark suite (SPEC95**) [1] for a large set of cache

0018-8646/97/$5.00 © 1997 IBM

Table 1 SPEC95 benchmark programs.*

Floating-point benchmarks: CFP95	
Application	*Description*
101.tomcatv	A mesh-generation program
102.swim	Shallow-water model with 513 × 513 grid
103.su2cor	Quantum physics; Monte Carlo simulation
104.hydro2d	Astrophysics; hydrodynamical Navier–Stokes equations
107.mgrid	Multigrid solver in 3D potential field
110.applu	Parabolic/elliptic partial differential equations
125.turb3d	Simulates isotropic, homogeneous turbulence in a cube
141.apsi	Solver regarding temperature, wind, velocity, and distribution of pollutants
145.fpppp	Quantum chemistry
146.wave5	Plasma physics; electromagnetic particle simulation
Integer benchmarks: CINT95	
Application	*Description*
099.go	Artificial intelligence; plays the game of "Go"
124.m88ksim	Motorola 88K chip simulator; runs test program
126.gcc	New version of Gcc; builds SPARC code
129.compress	Compresses and decompresses file in memory
130.li	LISP interpreter
132.ijpeg	Graphic compression and decompression
134.perl	Manipulates strings (anagrams) and prime numbers in Perl
147.vortex	A database program

*Adapted from *http://www.specbench.org*.

Table 2 Trace characteristics (all numbers in millions).

Benchmark	*Instructions*	*Memory references*	*Loads*	*Stores*
099.go	524	218	165	53
124.m88ksim	524	181	126	55
126.gcc	426	159	105	53
129.compress	524	200	139	61
130.li	524	240	144	95
132.ijpeg	524	166	114	52
134.perl	524	248	149	98
147.vortex	524	255	157	97
101.tomcatv	524	205	144	60
102.swim	524	212	141	71
103.su2cor	524	234	160	74
104.hydro2d	524	195	143	52
107.mgrid	524	246	188	58
110.applu	524	212	146	66
125.turb3d	524	247	135	112
141.apsi	524	218	136	81
145.fpppp	524	295	211	83
146.wave5	524	273	170	102

memory organizations. The SPEC** benchmarks are popular for comparing the performance of workstations. They attempt to measure processor and memory system performance under different compiler optimization strategies. In this study we focus on the memory system behavior with the goal of identifying the programs and cache organizations that may pose performance problems.

The SPEC95 CPU benchmarks are divided into two groups: floating-point (CFP95) and integer (CINT95). The CFP95 group reflects "numeric–scientific applications," while the CINT95 group is "system or commercial." SPEC has tried to isolate the parts of the application that are CPU-intensive and involve little operating system overhead or other I/O. The individual benchmarks are summarized in **Table 1**. The trace lengths are described in **Table 2**.

- *Related work*

In addition to describing a simple variant of next-sequential prefetching, this paper studies a more sophisticated shadow-directory prefetching algorithm described in a patent by Pomerene et al. [2]. The shadow-directory prefetching algorithm is described in Section 4.

Next-sequential prefetching is a very simple and common form of prefetching [3]. In Section 4, we describe trade-offs one faces when implementing a particular kind of next-sequential prefetching.

Gee et al. carried out a similar study [4] for the early release of the SPEC benchmarks, the so-called SPEC89**. Their work focused on the miss ratios and does not include an analysis of prefetching. In our work, we attempt to use the metric of *misses per instruction* (MPI) rather than miss ratios. A cache miss ratio is the number of loads or stores that result in cache misses compared to the total number of memory operations. On the other hand, the number of misses per instruction reflects the memory bandwidth that must be supported for each instruction. MPI is useful for another reason. To obtain the memory component of cycles per instruction (CPI), one need only multiply misses per instruction by the average number of memory cycles to service a cache miss. Our graphs actually use "misses per 1000 instructions," a scaling that allows us to work with convenient integers when describing miss rates. We use the abbreviation MP1000I to denote misses per 1000 instructions.

Compilers can also be relied upon to insert prefetch instructions into the programs to prefetch data or even other instructions [5–12]. In general, the compiler algorithms attempt to predict which of the program memory references will cause cache misses. For those memory references, prefetch instructions are inserted. As part of this study, we have identified the loads and stores in the SPEC95 benchmarks that are responsible for misses

in the programs. These data can be used to tune compiler prefetching algorithms by pointing out opportunities for additional prefetching and areas where no prefetching is necessary.

• *Methodology*

Each of the 18 benchmarks was compiled using Version 3.1 of the IBM CSet compiler running on AIX* 3.2.5. The only optimization option used was O3. The benchmarks were run using the reference inputs. The compiler benchmark, Gcc, is the only exception. For Gcc, only one of the reference inputs was run through the compiler and simulated.

We used an internal tracing tool to capture the memory references. To help avoid initialization behavior of the programs, we skipped the references at the beginning of each execution. For 14 of the benchmarks, using this tool, we skipped the first billion instructions in the program and traced the next 500 million instructions. For Gcc, since it only ran for about 500 million instructions, we skipped only 100 million references. Only nonprivileged references were captured.

To help validate our segment traces, we examined the hot-blocks in the execution profiles. (Hot-blocks are the top ten blocks, which account for most of the instructions in the execution.) The hot-block execution profile for the 500 million instructions matches very closely the hot-block execution profile for the full runs of the workloads.

For the three remaining programs, the hot-block profile from the 500 million instructions (after the one billion that were skipped) did not match the hot-block profile of the full run. For these three (Su2cor, M88ksim, and Compress), we took ten uniformly spaced contiguous samples of 50 million instructions. The execution profiles for the sampled traces matched those of the full profiles.

• *Overview of this paper*

In Section 2 we describe the differing instruction and data cache miss rates for the benchmarks. We show how the largest portion of the memory system references are biased toward the data cache miss traffic. In Section 2, we also examine the trade-offs involved for splitting the first level of instruction and data caches into two separate caches or using one unified cache of the same total capacity.

In Section 3, we define three metrics that are used to evaluate prefetching algorithms: coverage, accuracy, and traffic. In Section 4, we describe two prefetching algorithms: next-sequential and shadow-directory prefetching. Simulation results are presented in Section 5. Several graphs are shown describing the coverage, accuracy, and traffic for each prefetching algorithm. Finally, our conclusions and future work are presented in Section 6.

Table 3 Cache organizations for the basic study.

L1	*Capacity* (Kb)	8, 16, 32, 64, 128, 256, 512
	Associativity	1, 2, 4, 8
	Bytes per line	32, 64, 128, 256
	Policy	LRU write-back write-allocate

2. Cache behavior of SPEC95

In **Table 3** we show the basic cache organizations simulated for this paper.

Throughout this paper we refer to cache organizations using the following notation: *capacity/associativity/line size*. Cache capacity is in kilobytes; cache line size (or block size) is measured in bytes. For example, a 32Kb cache with four-way set-associativity and 128-byte cache lines would be described as 32/4/128. When we vary capacity, we denote it with an asterisk: */4/128 describes the collection of four-way set-associative caches with 128-byte lines but differing capacity.

• *Instruction cache*

In this section we examine the miss rates for an instruction cache as cache capacity, associativity, and line size are varied.

Varying capacity

The SPEC95 benchmarks were compiled for the POWER Architecture*, which is modeled in the compiler as having a 32Kb instruction cache. As a result, the compiler unrolls loops and eliminates subroutine-call overhead for small functions with this cache capacity in mind. For the CINT95 instruction references, a 32Kb eight-way set-associative instruction cache with 32-byte lines produces, on average, fewer than one miss per 1000 instructions. For the CFP95 instruction references, a 64Kb cache is required to reach such minimal miss rates. At 32Kb, there are approximately three misses per 1000 instructions. Basically, the compiler does a good job of reaching minimal miss ratios for the intended instruction cache.

Figure 1 shows the instruction cache miss rates for several one-way set-associative caches with 32-byte lines. The *Y*-axis on these graphs is in a log scale and represents the number of misses per 1000 instructions for an application on a particular cache. Each curve on the graph represents a particular cache capacity in the range from 8 Kb to 512 Kb. Large vertical distances between curves indicate that the larger cache (represented by the lower curve) captured a significantly larger piece of the working set. Similar graphs are used to describe the variation of other parameters and to examine the data cache.

For */1/32 caches, the benchmarks with the largest instruction miss rates are Li, Gcc, Go, Perl, M88ksim,

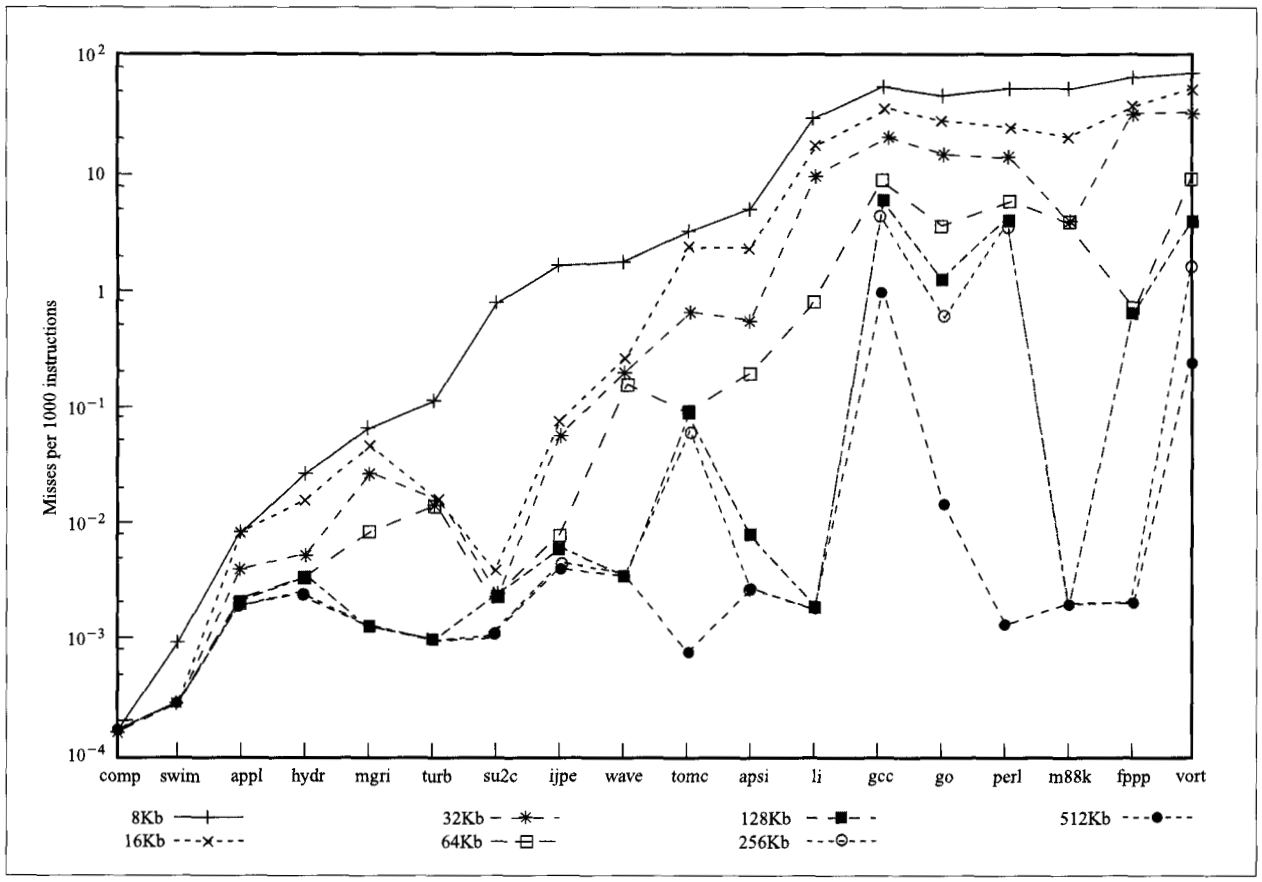

Figure 1

Instruction cache miss rate as cache capacity is varied from 8Kb to 512Kb (one-way set-associative, 32-byte lines).

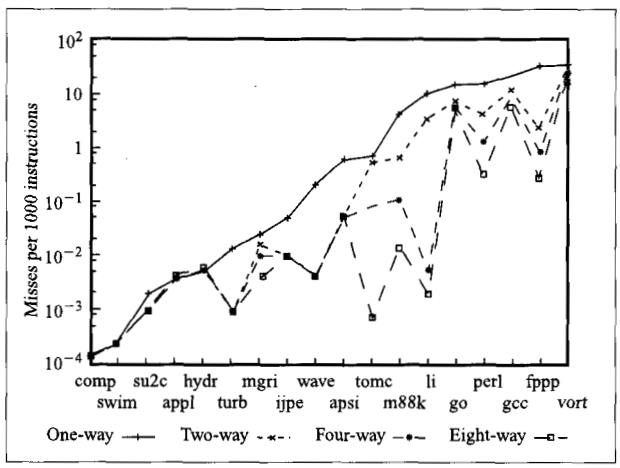

Figure 2

Instruction cache miss rate as cache associativity is varied from one-way to eight-way (32Kb caches, 32-byte lines).

Fpppp, and Vortex. Most of these required at least a 32Kb cache to have fewer than ten misses per 1000 instructions. However, even with the smallest 8/1/32 caches, the cache miss rate never exceeded 100 misses per 1000 instructions—one miss every ten instructions.

Most applications dropped down to one miss every 100000 to one million instructions when run on a 512Kb instruction cache. Gcc and Vortex are the exceptions here. Even with a 512Kb instruction cache, these two programs generated miss rates of approximately one per 1000 and one per 5000 instructions, respectively. The use of a four-way set-associative cache significantly reduces the miss rate for Vortex but not for Gcc.

Varying associativity

Figure 2 shows the cache family 32/*/32; in the figure, each curve on the graph represents a particular associativity. By looking for the large vertical gaps, we can see which benchmarks benefit by increasing associativity. However, increasing associativity does not always reduce

miss rates; in some cases, it can lead to increases in the miss rate for an application. Usually this is a problem only when the cache lines are long compared to the total cache capacity. For example, in Figure 1, Applu and Hydro2d displayed very small increases in miss ratio as the cache associativity was increased from one-way to eight-way. For these two applications, the increased associativity and the resulting decrease in number of congruence classes (sets) caused cached data to displace one another. With more congruence classes, data are more isolated from one another, and better replacement decisions can often be made.

For the 32/*/32 caches, a two-way cache resulted in significant gains over the direct-mapped alternative. Li experienced a reduction in miss rate by three orders of magnitude when a four-way cache was used (in place of a two-way cache). Wave5 also showed large gains with a four-way cache. All of the benchmarks other than Gcc, Go, and Vortex have fewer than one miss per 1000 instructions with a four-way cache. With the four-way caches, all benchmarks except Vortex generate fewer than ten misses per 1000 instructions. Vortex generated the most misses for the 32/*/32 caches, approximately 25 per 1000 (one in 40 instructions).

Varying line size

Figures 3(a) and **3(b)** respectively show the cache families 32/1/* and 32/4/*. Each curve on the graphs represents a different line size. As with the associativity graphs, increasing line size can result in increased miss rates when there are not enough cache congruence classes. Figure 3(b) shows such anomalies for Li, Tomcatv, M88ksim, and Perl. For these four benchmarks, as the line size increased and the capacity and associativity were held constant, the miss rate increased. It is well known that miss ratio can increase as line size increases; our experiments show many instances of such behavior.

The effects of line size are not as dramatic as those for increasing capacity or associativity of the instruction cache. For the one-way caches, it is generally good to increase the line size. For the four-way caches, the four programs mentioned above do not benefit from the longest line size—256 bytes. With the exception of Tomcatv, however, Li, M88ksim, and Perl benefited from 64-byte lines (compared to 32-byte lines) and Li benefited only slightly from 128-byte lines. These effects are produced by reducing the number of congruence classes in the more set-associative caches. For the direct-mapped caches, increasing line size appears to be beneficial; for the more associative caches, the long lines are generally beneficial and only rarely harmful. However, any harmful effects produced by increasing line size should not outweigh the benefits delivered to the other applications.

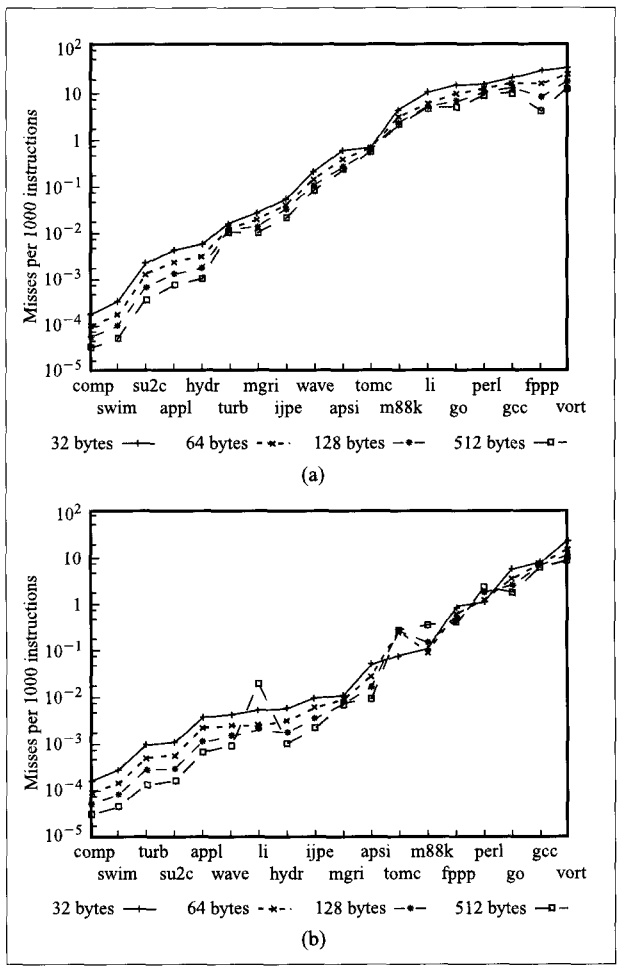

Figure 3

Instruction cache miss rate as cache line size is varied from 32 bytes to 256 bytes; 32Kb caches, (a) one-way and (b) four-way set-associative.

- *Data cache*

In the previous section we analyzed the instruction miss rates for the SPEC95 benchmarks. In this section we carry out similar analysis for the data references. In contrast to the instruction side, the data side poses more of a challenge to the memory system when the caches are small.

Varying capacity

Figures 4(a) and **4(b)** respectively show the miss rates for the cache families */1/32 and */4/32. The data side clearly has higher miss rates than the instruction side. Wave5, Tomcatv, and Swim generated the most data misses—more than one every ten instructions—with the 8Kb and 16Kb caches for both the one-way and four-way caches. A 32Kb

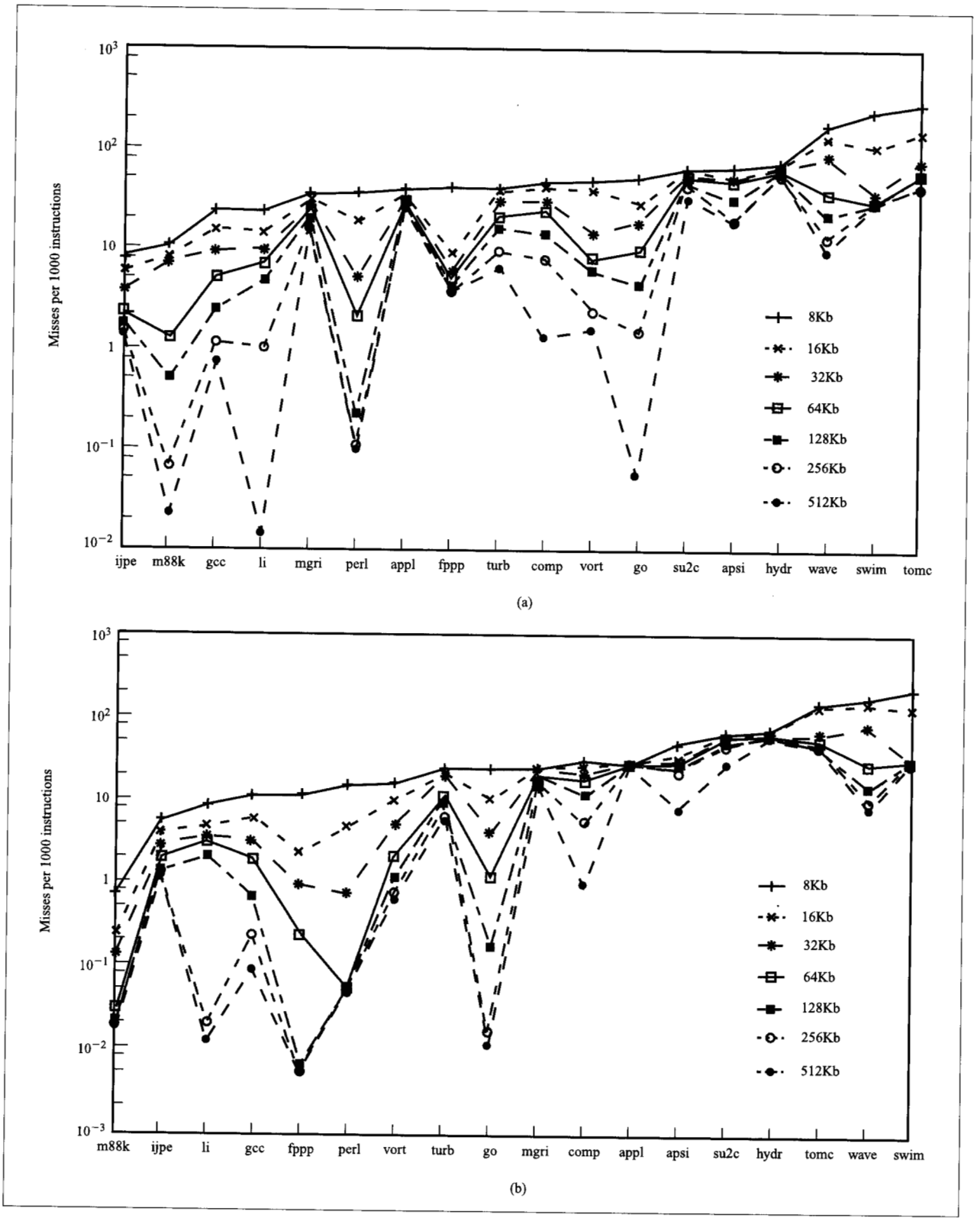

Figure 4

Data cache miss rate as cache capacity is varied from 8Kb to 512Kb; (a) one-way and (b) four-way caches, 32-byte lines.

cache is required to reduce the miss rates for these applications to less than one miss every ten instructions (100 MP1000I).

Another difference from the instruction side is that many of the benchmarks have miss rates in the 10–100-MP1000I range even with 128Kb caches (one-way). For some, the miss rate does not drop below 10 MP1000I even when 512Kb caches are used. In general, such behavior is caused by either very large working sets or a touch-the-data-once pattern in the data references of the trace. The touch-the-data-once pattern would be expected during program initialization. However, since the first billion instructions are skipped for all of the programs except Gcc, and since the execution profiles from the traces matched the profiles from the full executions, we felt that this behavior is not necessarily caused by programs initializing data structures before the "real work" starts. Since we know which instructions are responsible for the misses in the executions, we can correlate this information with the application source to verify our hypothesis. This remains as future work.

Varying associativity

Figure 5 shows a family of 32/*/32 data caches. The associativity is varied from one-way (direct-mapped) to eight-way set-associative. The graph shows that increasing associativity can have large effects on the miss rates for certain benchmarks. For example, M88ksim, Fpppp, Perl, Go, Li, Vortex, and Gcc experienced large reductions in miss rate with the two-way cache compared to the direct-mapped cache. For the workloads with the larger miss rates (Applu, Swim, Su2cor, Hydro2d, Tomcatv, and Wave5), the increased associativity is of little benefit.

Figure 5 also presents one anomaly where increased associativity slightly increases the miss rate for Compress. The two-way set-associative cache has a slightly larger miss rate than the one-way cache. However, the four-way and eight-way caches improve upon both the one-way and two-way cache miss rates. Here, the benefits of four- or eight-way associativity made up for the decrease in the number of congruence classes (cache capacity is held constant).

The graph in Figure 5 also shows the diminishing returns from four-way and eight-way caches compared to two-way set-associative caches. This is observed in many of the graphs that we have examined. However, Perl is the exception. A large drop in miss rate still occurs for the eight-way cache, but it can be argued that the miss rate is so low already with a four-way cache that any further reductions enabled by the eight-way cache would be of little benefit.

Varying line size

Longer cache lines help applications with good spatial locality. **Figure 6** presents the data cache miss rates as the line size is varied in a family of 32/4/* caches. The line sizes are 32, 64, 128, and 256 bytes, with each represented by a separate curve. This graph is very different from Figure 3, which showed a more uniform reduction in miss rate as line size is increased. Here, in Figure 6, we see that as line size increases, the miss rate often

- Decreases (Ijpeg, Li, Mgrid, Applu, Hydro2d);
- Increases (Perl, Fpppp, Go, Vortex, Tomcatv);
- Is relatively unchanged.

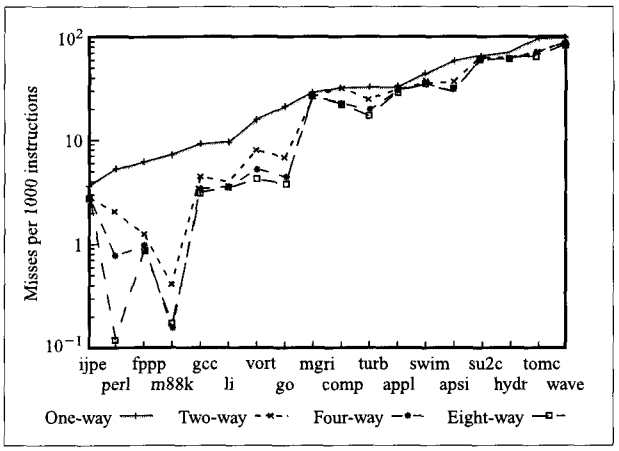

Figure 5

Data cache miss rate as cache associativity is varied from one-way to eight-way; 32Kb caches, 32-byte lines.

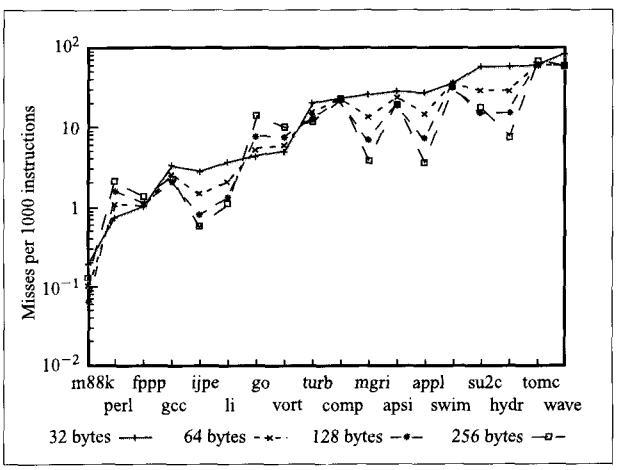

Figure 6

Data cache miss rate as cache line size is varied from 32 byte 256 bytes (32Kb caches, four-way set-associative).

Table 4 Cumulative percent of all data misses caused by the top ten instructions causing data misses.

Benchmark	*Cumulative percent of data misses*
li	88.84
turb3d	54.40
ijpeg	47.15
fpppp	44.67
mgrid	32.01
perl	30.02
apsi	24.51
swim	23.22
applu	22.20
hydro2d	20.71
vortex	13.51
go	12.89
gcc	5.56

Table 5 Number of load or store instructions required to account for 90 percent of all the data-miss traffic in each program.

Benchmark	*Number of instructions*	*Cumulative percent of data misses*
gcc	983+	56.30
go	760+	84.68
li	11	90.06
ijpeg	38	90.43
turb3d	46	90.10
fpppp	47	90.16
swim	57	90.18
mgrid	66	90.09
perl	66	90.02
apsi	112	90.21
applu	194	90.04
hydro2d	228	90.04
vortex	1233	90.00

Note: Since we kept track of only those instructions responsible for at least 500 misses, and since Gcc and Go are so "flat," those two did not reach 90 percent.

For many of those applications the increase (or decrease) in miss rate is quite large. The benefits of increased line size are larger than the effects of increased associativity. This graph suggests that the larger 256-byte lines might be useful for aggregate reductions in miss rate on a 32Kb four-way cache. It would not help all of the workloads, but it does help some significantly. To verify this hypothesis, one would need to look at the increased memory bandwidth requirements and bus queuing effects that would result from transferring longer cache lines into the primary cache.

Stride in the data misses

We measured the stride in the cache misses in two ways. One we call *last-miss*, or casual stride, and the other is called *stride with respect to an instruction*. In measuring casual stride, cache misses are watched as they occur, and the difference in their address is recorded as the program executes. Stride with respect to a particular instruction is somewhat different. In this case, stride is calculated based on the last cache miss caused by the same load or store instruction when it executed previously. One key difference between the two kinds of stride is that the stride with respect to instruction can be (and very often is) zero cache lines. This would happen if a cache line were displaced from the cache in the time between two accesses by the same instruction.

Casual stride would be the sort of stride covered by a simple next-sequential prefetcher. Stride with respect to instruction is useful for analyzing more sophisticated table-based prefetchers, which use instruction information in an attempt to refine their guess of the stride in the reference pattern.

We now analyze the stride in the data references with respect to instructions for the SPEC95 benchmark suite as a whole when run with 32/4/32 primary data caches. We compiled histograms of the frequency of each stride. The stride is measured in cache lines.

As we increase cache line size, the dominant (most frequent) strides shift. For 32-byte lines (**Figure 7**), stride 1 is most frequent (43 percent)—these are next-sequential misses by the recurrences of the same instruction. Stride 3 accounts for the next most frequent stride with ten percent. Strides 6 (four percent), 2 (three percent), 256 (three percent), and 0 (three percent) follow. For 64-byte cache lines (figure not shown), stride 1 is still dominant with 43 percent, but now stride 0 is second, accounting for 14 percent of all strided cache misses. Stride 3 still accounts for ten percent of the misses. With 256-byte cache lines (figure not shown), Stride 0 is dominant with 44 percent, and stride 1 accounts for only 15 percent. We see that as cache line size increases, next-sequentiality drops off and second misses to the same cache line become more frequent. This behavior is caused by what we call *congruence-class starvation*; that is, there are not enough sets in the long-line caches compared to caches of the same capacity with shorter lines.

Number of instructions causing data misses

Tables 4 and **5** show which instructions contribute the most misses in the benchmarks. These data are collected from a 32/4/32 data cache. Table 4 shows the percent of all data misses contributed by the top ten load or store instructions causing data misses. To measure the tail of the distribution of instructions, Table 5 shows the number of instructions required to reach 90 percent of all data misses. In our simulator, we report only the instructions for which more than 500 misses occurred. Consequently, we see in Table 5 that the "long flat tails" of Gcc and Go

do not reach 90 percent when one considers only the instructions causing more than 500 misses. Gcc and Go appear to have many instructions responsible for a few misses, in sharp contrast to programs such as Li or Ijpeg, where 11 and 38 instructions respectively contribute over 90 percent of the misses.

- *Comparing instruction and data miss rates*

To better distinguish the applications with more instruction or more data traffic, we directly compare the instruction and data miss rates for the SPEC95 benchmarks. **Figure 8** shows the ratio of data misses to instruction misses on equal-size primary caches. The ratio is plotted using a log scale, and a horizontal line is drawn where the data and instruction miss rates are equal. The figure shows the ratio for each benchmark and for */4/32 caches. Each curve represents a different cache capacity: 32Kb, 64Kb, and 128Kb.

Figure 8 shows that most of the benchmarks have at least 100 times more data-miss traffic than instruction-miss traffic. At one extreme, Tomcatv, Swim, and Compress can have 100000 times more data-miss traffic than instruction-miss traffic. At the other extreme, the 32Kb caches Vortex, Gcc, Perl, and Go can have more instruction-miss traffic than data-miss traffic from their primary caches.

- *Split vs. unified caches*

Figure 9 compares the miss rates of split and unified primary caches. In the figure the split instruction and data caches are half the size of the unified primary cache. The four curves represent

- Data-miss rate from a 16/4/32 cache ("data 16").
- Instruction-miss rate from a 16/4/32 cache ("instr 16").
- Sum of the previous two lines ("split 16/16").
- Miss rate from a unified 32/4/32 cache ("unified 32").

We see that the unified cache occasionally reduces the aggregate miss rate delivered to the L2. For Swim, Wave5, and Tomcatv, the extra capacity of the unified cache significantly reduces the miss rate (70, 41, and 54 percent, respectively). Since those three have very little instruction-miss traffic, the gains are purely from accommodating the larger working set of the data.

Fpppp behaves differently. Since the applications are compiled and optimized for 32Kb primary instruction caches, Fpppp has a very large instruction miss rate with the 16Kb primary caches. In this case, the unified cache outperforms the split caches because of its very large, frequently executed inner basic block that contains 4138 32-byte instructions—a little over 16Kb. For the cases we have examined (beyond what is presented here), the unified cache outperforms the split caches only occasionally when there is significant traffic from both the instructions and the data.

Perl is the only application where the unified cache has a larger miss rate than the separate half-size instruction and data caches. However, even this difference is small: about one miss per 1000 instructions. In general, the unified caches perform as well as the split caches or, as in the several caches mentioned above, much better.

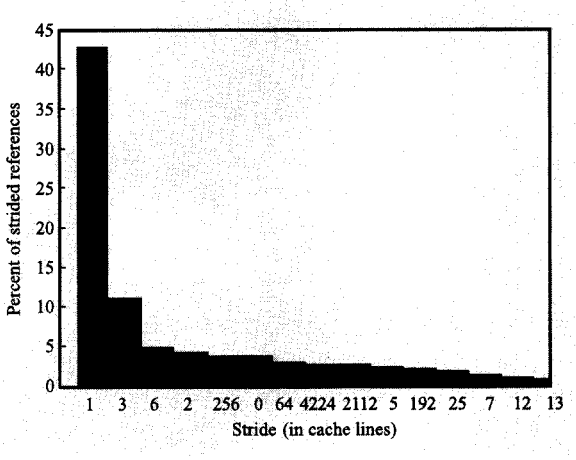

Figure 7

Histogram of stride with respect to instruction for 32/4/32 data caches in SPEC95.

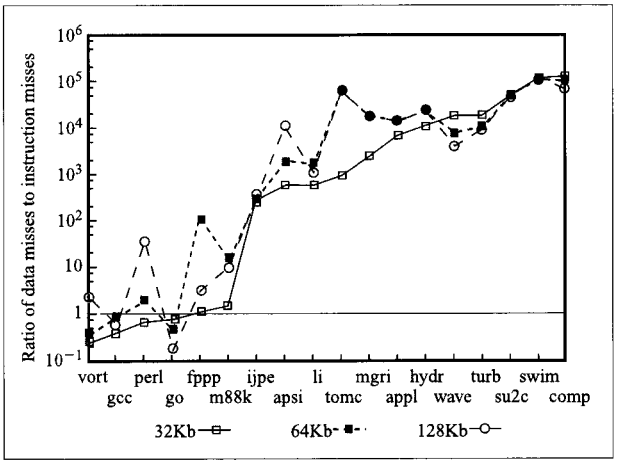

Figure 8

Ratio of data-miss rates to instruction-miss rates for */4/32 caches.

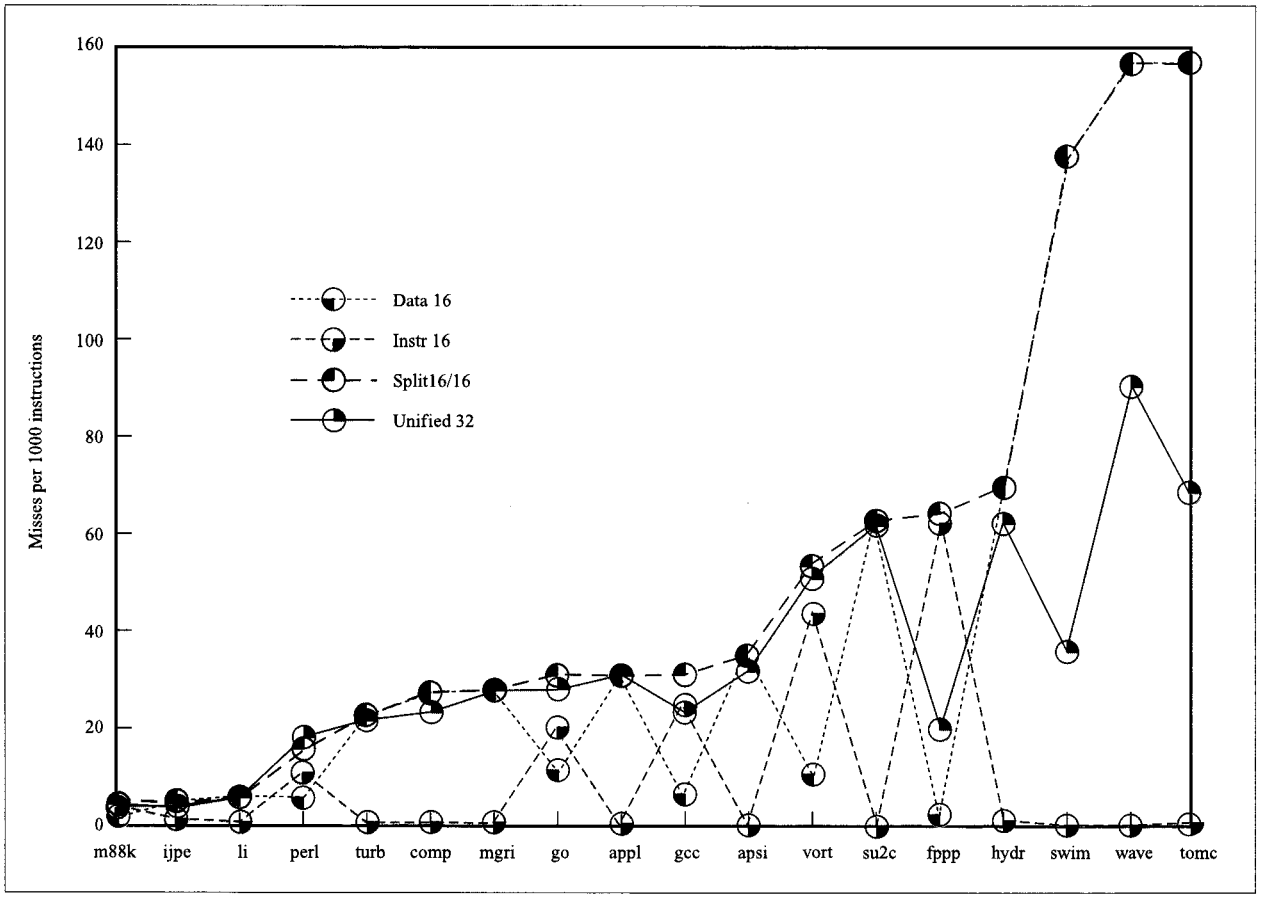

Figure 9

Comparison of split 16/4/32 caches to unified 32/4/32 primary cache.

3. Prefetching terminology

In this section, we discuss three fundamental aspects of memory hierarchy prefetching: coverage, accuracy, and traffic. They measure, respectively, the number of misses removed by the prefetching algorithm, the quality of the prefetching decision, and the amount of extra bandwidth required of the memory system to support prefetching. *The goal in prefetching is to have high coverage, high accuracy, and minimal extra traffic.* Prefetching traffic can be simply subdivided into two categories: "good" prefetches and "bad" prefetches:

$$Prefs = Pref_{\text{Hits}} + Pref_{\text{Bad}}$$

The good prefetches are used by the program before they are displaced from the buffer or cache in which they are held. The bad prefetches are not used by the program before they are displaced. There are finer subdivisions based on the latency covered by the prefetches, but they are not important at this time.

Accuracy is a measure of the skill of the prefetching mechanism in choosing what to prefetch:

$$Acc = Pref_{\text{Hits}}/Prefs = Pref_{\text{Hits}}/(Pref_{\text{Hits}} + Pref_{\text{Bad}}) .$$

Coverage is a measure of the number of misses that are accurately prefetched. BaseCaseMisses are misses that would have occurred with no prefetching. When (imperfectly) prefetching into the cache, new cache misses are produced because prefetched data displace other live cached data:

$$Cov = Pref_{\text{Hits}}/BaseCaseMisses$$
$$= (Acc \times Prefs)/BaseCaseMisses.$$

Coverage and accuracy are in the range (0, 1). Theoretically, however, coverage can exceed 1 with this definition, for the following reasons. The prefetches (good or bad) which are automatically placed into the cache can displace live data from the cache, creating more

opportunities for prefetching. With these extra, or "redundant," opportunities for prefetching, there may be more prefetch hits than base case misses. This does not occur in practice. Consider this example: A program references only one line, A. In a normal execution it would have one cache miss. Now consider a prefetching algorithm that could prefetch B, displacing A, and then prefetch A (displacing B) before the processor refers to it again. When the processor references A, it will have a prefetch hit. The prefetcher prefetches B and then A in sequence as before. When the processor references A again, that will be a second prefetch hit in a program that originally only had one miss. In this unusual case, coverage exceeds 1. Here it is equal to 2.

ResidueMisses are those that remain after prefetching because of imperfect coverage of all cache misses. These are the ones the prefetcher missed:

$$ExtraTraffic = (Prefs + ResidueMisses)/BaseCaseMisses.$$

Extra traffic is always greater than or equal to 1.

4. Prefetching algorithms

In this section we describe two prefetching algorithms, next-sequential prefetching (NSP) and shadow-directory prefetching (SDP), and a process that avoids unnecessary prefetches. (A prefetch is unnecessary if it is not used by the processor while in the cache.)

- *Next-sequential prefetching*

In NSP, whenever line L is referenced an attempt is made to prefetch line $L + 1$. Many designers find NSP an attractive prefetching algorithm because of its simplicity. A prefetch of line $L + 1$ is initiated on the basis of an access to line L. NSP relies on the spatial and temporal locality properties of a program to predict which line to prefetch. These two properties account for much of the success of caches in attempting to contain those portions of a program requested by a processor. The effectiveness of NSP depends on the cache size and, in particular, the line size used in the cache. Typically, smaller cache line sizes result in better NSP performance. For example, consider NSP when applied to instruction prefetching. If an instruction-fetched line does not contain a taken branch, the next-sequential line must be referenced. Also, most branches are short forward jumps in a program. This short forward jump is frequently contained in the original line or the next-sequential line.

- *Shadow-directory prefetching*

In SDP, a history of the referencing pattern of a processor is recorded in a table and a prefetch is attempted whenever it is determined that the referencing pattern is repeating. For example, consider **Figure 10**. The figure shows a two-level memory system consisting of a cache and directory at level one (L1) backed up by a second-level cache (L2) and directory. Typically the L2 cache is several times larger than the L1 cache. Each directory entry for the L2 identifies a line currently contained in the L2, and the L2 directory represents a history of the lines that were in the L1. Now, let each directory entry in the L2 contain a second address, a shadow address. Let the shadow address identify the line that was referenced after the line identified by the directory entry was referenced. Thus, each directory entry in the L2 contains a pair of addresses that represent a parent–successor (follower) sequence of references made by the processor. The parent address identifies the line normally contained in the L2, while the shadow address identifies a following address to the parent address and represents a logical choice for a prefetch candidate. By integrating the prefetch address into the structure of the L2, SDP relies on the L2 to maintain a history of lines that were in the L1 at an earlier point in time, and would still be in the L1 if it were larger. The effectiveness of SDP depends upon the persistence of the referencing pattern as captured by the L2. In those applications where the L2 can contain a large portion of an application's working set, and if the referencing pattern of the application is repetitive, SDP should be able to prefetch L1 misses so that the miss ratio of the L1 approaches that of the L2.

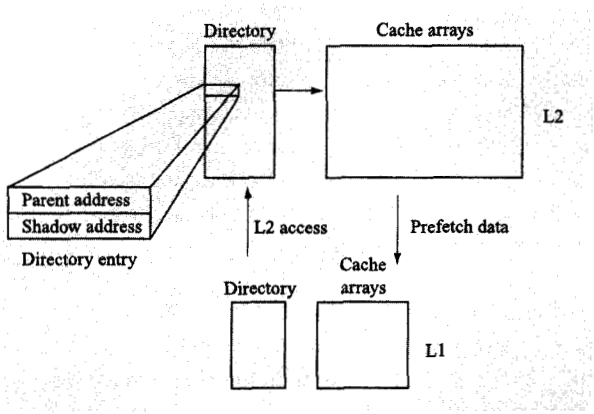

Figure 10

Shadow-directory prefetching.

An immediate advantage of SDP over NSP is that it can prefetch lines that are not just next-sequential referencing patterns. For example, consider a sequence of six cache misses L, $L + 1$, $L + 3$, $L + 6$, $L + 7$, $L + 8$. If the sequence is repeated, NSP can prefetch only lines $L + 1$, $L + 7$, and $L + 8$, whereas SDP can prefetch all of the misses. Recall that NSP can prefetch line $L + 1$ only if line L is referenced. However, the advantage of SDP

prefetching over NSP is not without cost. NSP can be implemented with relatively little hardware cost, whereas SDP requires each directory entry in the L2 to contain a second address, a prefetch address. Also, SDP must see a reference pattern repeated before it can accurately generate prefetches, whereas NSP can successfully prefetch unseen or first-time-referenced information when the referencing pattern is next-sequential in nature.

- *Prefetch buffers*

Besides determining which line to prefetch, a prefetch algorithm must decide where to put the prefetch and the frequency with which a prefetch can be attempted. We consider two prefetch placement policies in this paper: prefetch into the cache, and prefetch into buffers.

When prefetching into the cache, the replacement algorithm chooses the least-recently-used (LRU) line in a cache congruence class and overwrites it with the prefetched line. LRU is used as the replacement policy for all prefetching experiments in this paper. The prefetched line is then given the newest or most-recently-used (MRU) status of all the lines in the congruence class. When prefetching into buffers, each prefetch is initially transferred to a buffer. When a cache miss occurs, the buffers are searched to see whether the miss was prefetched. If the miss was prefetched, the line is transferred from the buffer to the cache, and the buffer is marked free and available for another prefetch. The number of prefetch buffers studied in this paper varies among 1, 2, 4, and 8. If there are no free buffers when a prefetch occurs, the buffer containing the oldest prefetched line is chosen for replacement, and the current prefetch overwrites the existing line in the prefetch buffer. LRU is the replacement policy used to select a prefetch buffer.

The prefetch buffers can be implemented in the same technology as the cache and placed near the cache. Thus, an access that is found in the prefetch buffer can be satisfied in approximately the same amount of time as a cache hit. There are several advantages to having lines sent to a buffer and not loaded directly into the cache.

First, prefetches are a guess or prediction that a line will be used by the processor. If a prefetched line is copied directly into the cache, a line already in the cache must usually be discarded. If the prefetched line is not used while in the cache, the cache has been contaminated with a useless line, thus wasting valuable cache space. The prefetch buffer acts as a filter for all prefetches and allows only the prefetched lines that are used by the processor to be placed into the cache. Typically a small number of buffers are required to keep lines that are not used by the processor from entering the cache.

Second, if the prefetched line is copied into the cache, the replacement algorithm must choose a line currently in the cache to be overwritten by the prefetched line. If the replaced line is re-referenced before the prefetched line is referenced, an additional cache miss occurs; that is, the line just discarded from the cache must be re-accessed before the cache request can be satisfied.

Third, if the prefetched line is transferred directly into the cache, normal references made by the processor may be blocked during the line transfer cycles.

The advantages of transferring prefetches directly to the cache are simplicity and cost. Transferring prefetches directly into the cache is simpler to implement than transferring them to a buffer. Also, prefetch buffers consume valuable chip area, and any value would be lost if they adversely influence the critical cycle time of the processor.

The frequency with which a prefetch is attempted is an important design parameter in any prefetching algorithm because each prefetch attempt requires a directory lookup to see whether the line is already in the cache or prefetch buffers (if used). Prefetching too frequently can saturate the cache directory and delay necessary cache accesses. Conversely, an infrequent prefetching policy can reduce the miss coverage that would otherwise be obtainable by more aggressive prefetching schemes. For example, prefetching on every cache reference can double the directory traffic, while prefetching on only cache misses has an upper bound of removing only 50 percent of misses. For the prefetching studies in this paper, a prefetch is attempted on each MRU change within a cache congruence class. An MRU change occurs whenever a line other than the most-recently-used line in a cache set is referenced. It is noted that all cache misses are MRU changes. Prefetching on MRU changes removes most of the unnecessary directory traffic associated with prefetching on every reference while still providing ample prefetching opportunity to reduce the number of overall misses.

- *Confirmation*

A technique known as confirmation is used to improve the accuracy of each prefetching scheme [2]. Confirmation improves the accuracy of a prefetching algorithm by keeping track of what has been prefetched and what has been used, and tries to avoid making not-used prefetches in the future. For example, **Figure 11** shows next-sequential prefetching with confirmation using a memory hierarchy consisting of an L1, an L2, and a single prefetch buffer.

A confirmation bit is added to each directory entry in the L2 indicating whether the line was used (*1*) or not-used (*0*) when it was last prefetched. Assume that the confirmation bit is initially set to *1* for each line that

enters the L2. This initial setting is arbitrary, but will bias the prefetching mechanism to prefetch on first-time references. Now consider NSP and a referencing sequence of three lines L, $L + 1$, and $L + 4$, each causing a cache miss; let each generate a prefetch request for the L1. Ignoring startup effect, we assume that lines L, $L + 1$, and $L + 4$ are all in the L2 with their confirmation bit set to *1* and that the prefetch buffer is empty. The reference to line L generates a prefetch request for line $L + 1$. The L2 directory is then searched for line $L + 1$ and its confirmation bit examined. If the confirmation bit is *1*, line $L + 1$ is prefetched. If the bit is *0*, line $L + 1$ is not prefetched. Assume that the confirmation bit is *1* and that the line is prefetched to the prefetch buffer.

The reference to line $L + 1$ causes a cache miss that hits in the prefetch buffer. Line $L + 1$ is then transferred to the cache, and the prefetch buffer is marked free. The reference to line $L + 1$ also causes a prefetch for line $L + 2$. Assuming that line $L + 2$ is in the L2 and its confirmation bit is *1*, it is prefetched and sent to the prefetch buffer.

The reference to line $L + 4$ misses in both the cache and prefetch buffer and generates a prefetch request for line $L + 5$. The confirmation bit for line $L + 5$ is examined, and it is prefetched. However, when $L + 5$ is prefetched, the prefetch buffer is not free. Line $L + 2$ is still in the prefetch buffer and not-used. The confirmation mechanism recognizes this condition and notifies the L2 that line $L + 2$ was not-used when last prefetched. The confirmation bit for line $L + 2$ is then reset to zero.

If line $L + 5$ is not-used by the processor, it is eventually overwritten by the next prefetch. This causes its confirmation bit to be set to *0* and inhibits further prefetching of this line. Thus, for the sequence of L, $L + 1$, and $L + 4$, three lines were prefetched ($L + 1$, $L + 2$, and $L + 5$), but only one line was used ($L + 1$).

If the sequence of lines L, $L + 1$, and $L + 4$ is repeated, only line $L + 1$ is prefetched. Lines $L + 2$ and $L + 5$ are not prefetched because their confirmation bits are set to *0*.

Once a line in the L2 has its confirmation bit set to *0*, it can be reset to *1*. The confirmation mechanism coupled with next-sequential prefetching monitors the referencing pattern sent to the L2. Each pair of consecutive references are compared to determine whether they represent a sequential referencing pattern. If one is detected, the confirmation bit for that pair of references is examined and reset to *1* if it is currently *0*.

For example, if the previous referencing pattern changes and becomes L, $L + 1$, $L + 2$, $L + 4$, the reference to line $L + 2$ following the reference to line $L + 1$ is detected as a sequential referencing pattern. The confirmation bit for line $L + 2$ is then examined and, if *0*, reset to *1*. This allows future references of line $L + 1$ to prefetch line $L + 2$.

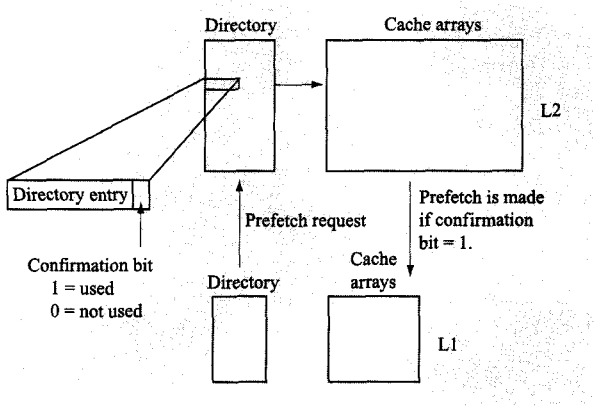

Figure 11

Next-sequential prefetching with confirmation.

Implementing a confirmation mechanism with SDP requires slightly more hardware than described for NSP. As before, a confirmation bit is added to each shadow address saved in the L2 to indicate whether the line identified by the shadow address was used (*1*) or not-used (*0*) when last prefetched. Recall that a prefetch of the shadow address is attempted in SDP whenever the parent address is referenced. However, when a prefetch is made, the prefetch address (shadow address) is saved along with the parent address in the prefetch buffers. The shadow address identifies the line that was prefetched, and the parent address identifies the L2 entry that caused the prefetch. This address is needed to notify the L2 whenever it is detected that a prefetch was made and not-used. The confirmation mechanism then uses the parent address to locate the L2 entry that caused the prefetch, and the confirmation bit for the shadow address is set to *0*, inhibiting future prefetches until it is reset to *1*.

A slight modification to the prefetching algorithm occurs when there are no prefetch buffers. Here prefetches are sent directly to the cache. A used bit is added to each line in the L1 directory indicating whether the line was used or not-used while in the cache. A *1* indicates that the line was used, and a *0* indicates not-used. The used bit assists the confirmation algorithm and helps to initiate prefetch attempts. All prefetches are loaded into the cache with their used bit set to *0*. The bit is set to *1* whenever the line is referenced by the processor. Demand misses are loaded into the cache with their used bit set to *1*. A miss is a demand miss if it is not a prefetch miss. These misses represent the normal requests generated by the processor while executing a

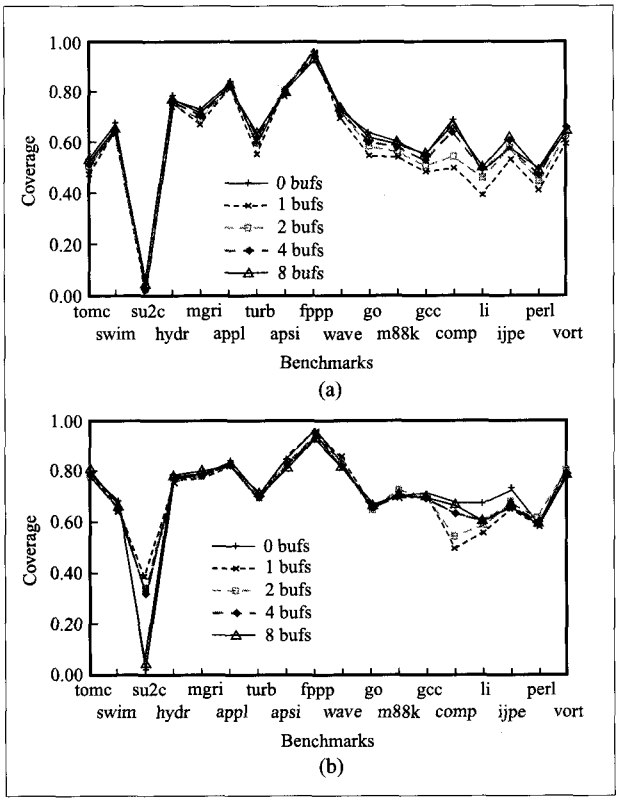

Figure 12

Miss coverage for instruction prefetching (8/1/32 cache): (a) next-sequential prefetching; (b) shadow-directory prefetching.

program. A prefetch is attempted whenever a reference causes an MRU change or it is detected that the processor has made a reference to a line with a used bit set to *0*. Here the line was prefetched and used by the processor for the first time. If the prefetching algorithm uses a confirmation mechanism to eliminate not-used prefetches, the prefetch algorithm must reset the appropriate confirmation bit whenever it is detected that a line is chosen for replacement and its used bit is still *0*. This occurs when a prefetch is made into the cache and the line gets replaced from the cache before it is used.

5. Simulation results

The effectiveness of each prefetching algorithm was evaluated against the SPEC benchmark suite. We begin by showing graphs for coverage, accuracy, and traffic for NSP and SDP with and without confirmation for an 8Kb cache. Separate graphs for instruction and data prefetching are shown and analyzed in detail. We end this section by showing average performance characteristics for 8Kb, 16Kb, and 32Kb caches that are one-way and four-way set-associative.

- *Instruction prefetching*

Comparisons between NSP and SDP for instruction prefetching are shown in **Figure 12**. The size for each cache is 8Kb, one-way set-associative, with a 32-byte line size. No confirmation mechanism was used, and the number of prefetch buffers varied from zero to eight. Zero prefetching buffers indicates that we are placing each prefetched line directly into the cache. The SPEC floating-point applications are listed first (leftmost workloads in the graph), followed by the SPEC integer applications. We found that SDP averages a miss coverage about ten percent higher than NSP for instruction prefetching when coverage is averaged over all of the SPEC applications, about 60 percent for NSP versus 70 percent for SDP. Some interesting features of the two graphs include the following:

- The increased prefetching coverage for SDP is produced by the ability to prefetch jumps or branches in the instruction stream. Recall that NSP cannot prefetch jumps in the instruction stream that go beyond the next sequential line.
- Prefetching into the cache generally outperformed prefetching into prefetch buffers. This is possible when the prefetching accuracy is high. For example, NSP averaged a prefetching accuracy of about 75 percent, while SDP averaged an amazingly high accuracy of more than 90 percent. With high prefetching accuracy, the need for prefetch buffers is diminished because each line prefetched into a buffer eventually ends up in the cache.
- The variation in coverage for prefetching into the cache or into prefetch buffers is generally smaller for SDP than NSP. By being able to prefetch taken-branch misses and next-sequential misses, SDP remembers the miss order more accurately than NSP. This effect also accounts for SDP's higher accuracy.
- NSP was able to prefetch very few misses for Su2cor because it had relatively few instruction misses (see Figure 1), and most of its misses were caused by two taken branches fetching lines into the same set of the cache. Each branch target would displace the other branch's line and cause another miss. These misses are not prefetchable with NSP because the previous line was not referenced.

- *Data prefetching*

Miss coverage for SDP and NSP prefetching data is shown in **Figure 13**. When prefetching data, both SDP and NSP had a lower miss coverage than was observed for instruction prefetching. The average prefetch miss

coverage for SDP was only 32 percent versus 20 percent for NSP. The graphs contain many interesting features that show when a prefetching algorithm is working well or poorly. Some interesting features of the two prefetching algorithms are the following:

- Prefetching into the cache is no longer the best strategy. For NSP, prefetching into the cache was the worst prefetching policy for Tomcatv and Swim and inferior to eight prefetch buffers for Hydro2d, Mgrid, Turb3d, Wave5, and M88ksim. Two factors contributed to this result. First, the prefetching accuracy for NSP was much lower for data prefetching than for instruction prefetching, approximately 30 percent when averaged over all of the workloads. Prefetching a line that will not be used still requires that a line be discarded from the cache. (The accuracy graphs for NSP and SDP are discussed below.) Second, for a prefetching algorithm to be successful, it must decide not only which line to prefetch, but also which line to discard from the cache. If a line is discarded before its last use, a new miss can result when the line is referenced again. Any miss reduction caused by prefetching a line might be offset by an additional miss caused by discarding the wrong line from the cache. The prefetching algorithm must discard a dead line from the cache for each prefetch. A line is dead if it will be discarded from the cache before it will be re-referenced without considering prefetching effects. Prefetching into a direct-mapped cache can produce additional misses caused by displacing lines too soon from the cache.
- When varying the number of prefetch buffers from zero to eight, sensitivity in prefetching coverage is much higher than was observed for instruction prefetching. For example, when NSP was applied to the Hydro2d workload, the prefetch coverage varied from three percent with one prefetch buffer to over 80 percent coverage with eight prefetch buffers. When only one prefetch buffer is used, the order in which lines are prefetched is extremely important. A prefetch must be used before the next line is prefetched in order for it to be successful. When the number of prefetch buffers is increased to eight, the time between prefetching a line and using it is not as critical as with one prefetch buffer. With eight prefetch buffers, a prefetch can exist in the prefetch buffers for a much longer time before it is used by the processor. The same effect is seen for SDP, but to a lesser degree.
- Miss coverage for NSP on Compress and Go was relatively low (ten percent or less), regardless of the number of prefetch buffers used. These two applications lack a next-sequential miss pattern and yield poor prefetching results. SDP improved the miss coverage to 20–40 percent for these applications.

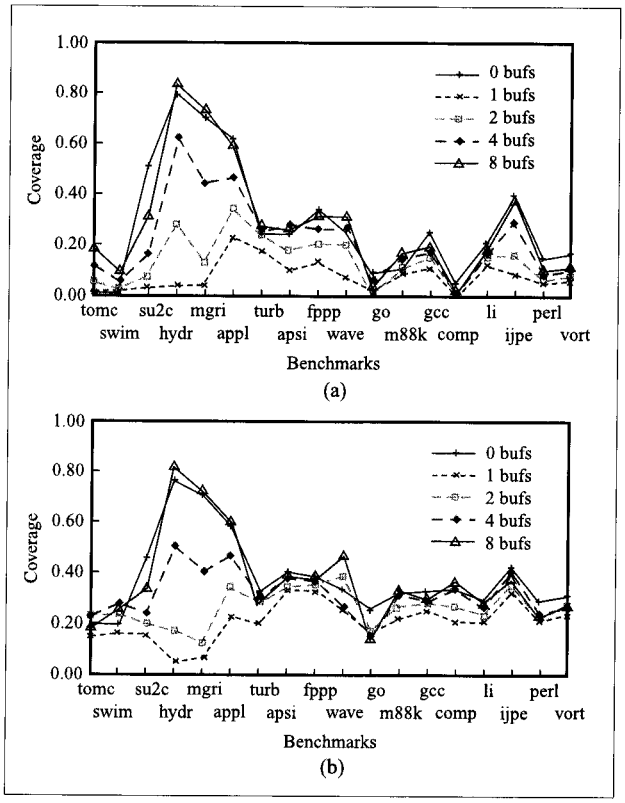

Figure 13

Miss coverage for data prefetching (8/1/32 cache): (a) next-sequential prefetching; (b) shadow-directory prefetching.

Figure 14 shows the prefetching accuracy for NSP and SDP corresponding to the data-prefetching graphs shown in the previous figure.

The prefetching accuracy of NSP varied widely depending on the application and number of prefetch buffers used. For example, the prefetching accuracy of Hydro2d varied from near zero percent for one prefetch buffer to nearly 100 percent for eight prefetch buffers, while Compress and Go had an accuracy of ten percent or less for all buffer policies. The large variation in prefetching accuracy of Hydro2d indicates that several different blocks of data are being prefetched simultaneously, and that several buffers are needed to hold these prefetch lines before their use by the processor. The low prefetching accuracy for Compress and Go shows that few misses are next-sequential in nature. This can also be seen in Figure 12, where the miss coverage was low.

It is also possible for the prefetching accuracy to be high and still have a low miss coverage. For example, NSP for the Tomcatv workload had a prefetching accuracy of

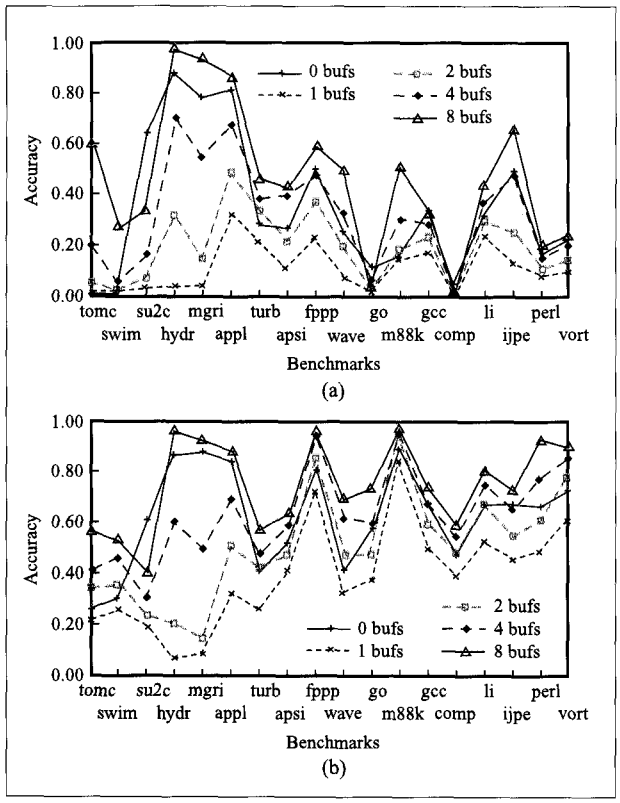

Figure 14

Data-prefetching accuracy (8/1/32 cache): (a) next-sequential prefetching; (b) shadow-directory prefetching.

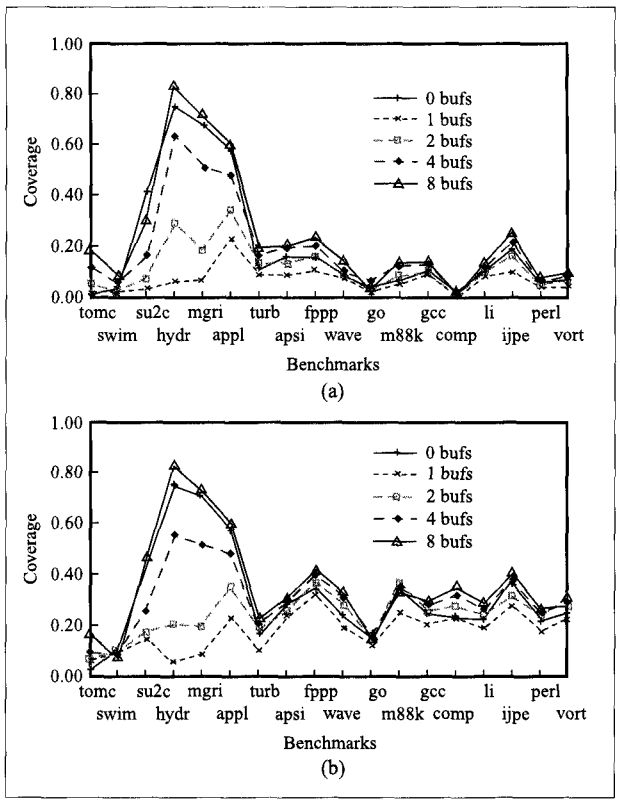

Figure 15

Data-prefetching miss coverage with confirmation (8/1/32 cache): (a) next-sequential prefetching; (b) shadow-directory prefetching.

60 percent with eight prefetch buffers, while the prefetching coverage was only 20 percent. This can occur when most of the prefetch attempts are already in the cache and the majority of the misses are to lines that are not next-sequential to the previously accessed lines. Other workloads where accuracy exceeded coverage are Ijpeg (60 to 40 percent) and M88ksim (60 to 20 percent).

The prefetching accuracy for SDP was higher than that achieved for NSP because it can prefetch across breaks in the miss pattern. For example, consider a miss sequence of line L followed by line $L + 2$. SDP can accurately prefetch $L + 2$ after line L is referenced, whereas NSP makes an incorrect prefetch attempt to line $L + 1$.

Using SDP, the average prefetching accuracy was 60 percent (over all workloads); however, seven workloads achieved greater than 90 percent accuracy with eight prefetch buffers: Hydro2d, Mgrid, Applu, Fpppp, M88ksim, Perl, and Vortex. In fact, all but one had a 50 percent or higher prefetching accuracy for eight prefetch buffers, the exception being Su2cor with a 40 percent accuracy. SDP can also have a high prefetching accuracy and still have a low prefetching coverage. For example, Fpppp and M88ksim had nearly 100 percent prefetching accuracy with four or eight prefetch buffers, while the miss coverage was only 30 to 40 percent. This is possible when a cache line has more than one unique miss-follower. For example, consider that line L is followed by line $L + 4$ in one reference sequence; later, line L is followed by line $L + 2$. SDP remembers an L-to-$L + 4$ parent–follower pair. The SDP prefetching mechanism modeled in this paper can only remember one unique follower per line; lines having multiple followers were not allowed. If a line has two or more different miss-followers, only one of the lines is prefetchable. By modifying the shadow-directory to record multiple miss-followers per parent address, it is possible for SDP to attempt two or more prefetches when an MRU change is detected.

- *Confirmation in data prefetching*

The next set of graphs shows the effectiveness of confirmation to improve prefetch accuracy and effects on miss coverage. **Figure 15** shows prefetching coverage, and

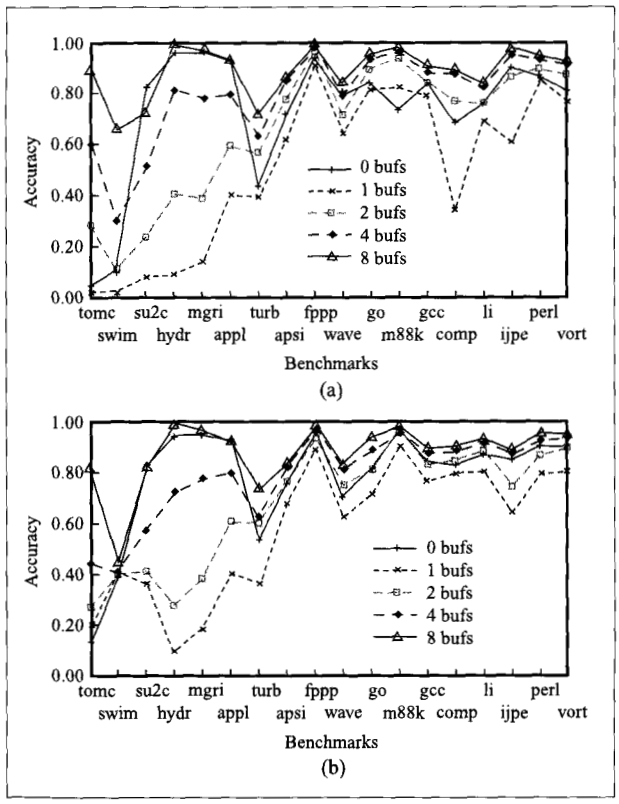

Figure 16

Data-prefetching accuracy with confirmation (8/1/32 cache): (a) next-sequential prefetching; (b) shadow-directory prefetching.

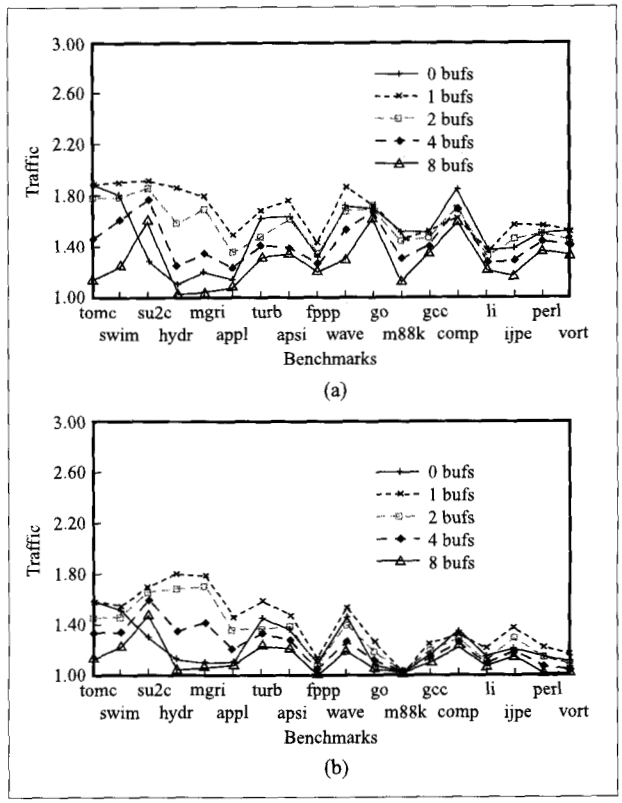

Figure 17

Data-prefetching traffic without confirmation (8/1/32 cache): (a) next-sequential prefetching; (b) shadow-directory prefetching.

Figure 16 shows prefetching accuracy for NSP and SDP with confirmation when prefetching data. The cache modeled is the same size as above, 8Kb by one-way set-associative, with a 32-byte line size.

The figures show that the miss coverage for NSP dropped only slightly. For example, the overall miss coverage is now 16 versus 20 percent without confirmation. However, the prefetching accuracy has improved significantly. The average prefetching accuracy for NSP is now 70 versus 30 percent (from Figure 14). Confirmation can remove unnecessary or not-used prefetches from a prefetch algorithm to a significant extent.

Similar improvements in prefetching accuracy were observed for SDP with confirmation, with only a modest drop in prefetching coverage. The overall miss coverage is now 28 versus 32 percent without confirmation, and the prefetch accuracy is now 75 percent when averaged over all of the workloads. Only two workloads have a prefetching accuracy of less than 80 percent with eight prefetch buffers (Swim and Turb3d).

We note that it is still possible to have a very high prefetching accuracy and low miss coverage. A prefetching mechanism can attempt very few prefetches, all of which are correct, and have little if any overall effect on the miss coverage. It is important for a confirmation process to eliminate any incorrect prefetches without removing those prefetches that are correct most of the time.

The next set of graphs shows the bus traffic ratio of each prefetching algorithm. **Figure 17** shows the bus traffic for NSP and SDP prefetching data without confirmation for an 8/1/32 cache. **Figure 18** shows the bus traffic when confirmation is used.

The traffic ratio for NSP without confirmation varied from 1.0 to 1.9, with an average of 1.5. A bus traffic ratio of 1.5 means that the prefetching algorithm generated 50 percent more bus traffic than the normal bus traffic associated with a cache that is processing base case misses without prefetching. Most of the extra bus traffic is due to incorrect or not-used prefetches. Confirmation removed most of the extra traffic for NSP. With confirmation, the average bus traffic ratio fell to 1.07, where only seven percent more traffic is generated by prefetching over the normal miss traffic without prefetching.

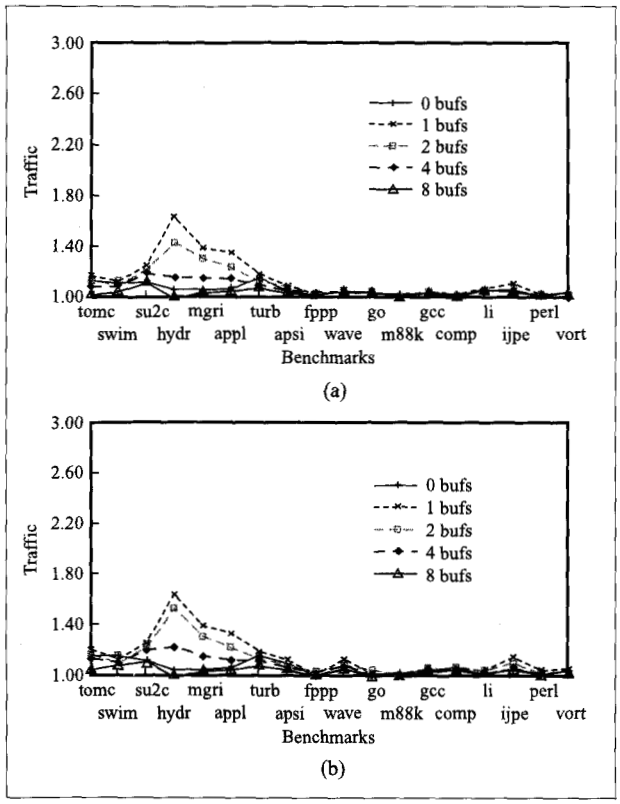

Figure 18

Data-prefetching traffic with confirmation (8/1/32 cache): (a) next-sequential prefetching; (b) shadow-directory prefetching.

When reviewing the results from the coverage, accuracy, and traffic graphs, we see that three applications did particularly well with NSP and confirmation: Hydro2d, Mgrid, and Applu. When prefetching into the cache or when eight prefetch buffers were used, each application prefetched 60 percent or more of the misses with less than five percent more bus traffic.

Similar results were obtained for SDP. Here the average bus traffic ratio was 1.3 without confirmation and 1.09 with confirmation.

- *Equivalent caches*

It is useful to compare the caches modeled in the previous graphs to caches, without prefetching, that produced an equivalent miss ratio. **Figure 19** compares the miss ratios of caches without prefetching to the miss ratios of four SPEC applications with prefetching: (a) Vortex and (b) Gcc with instruction prefetching, and (c) Compress and (d) Wave5 with data prefetching.

In each graph an 8/1/32 cache was modeled and the miss ratio plotted (in misses per 1000 instructions) for NSP and SDP, with and without confirmation. Each horizontal line represents the miss ratio for a cache without prefetching. The legend identifies each cache modeled. The graph clearly shows the effect on performance of varying the number of prefetch buffers. For example, the miss ratio for Vortex using NSP with confirmation was approximately 27 misses per 1000 instructions when prefetching into the cache (zero buffers); it increased to 30 misses per 1000 instructions with one prefetch buffer, and improved to nearly 27 misses per 1000 instructions with eight prefetch buffers. These effects can also be seen in Figure 12, where the coverage for prefetching into the cache was better than the coverage achieved with one prefetch buffer and nearly the same as with eight prefetch buffers.

For Vortex using NSP, prefetching into the cache was the best policy; adding prefetch buffers reduced performance. With only one prefetch buffer, several prefetches were overwritten by the next prefetch before they could be used by the processor. Increasing the number of prefetch buffers to eight increased the amount of time between prefetching of a line and its availability for use by the processor. However, if the accuracy of each prefetch is high, placing the line directly into the cache allows the largest amount of time between its prefetch and eventual use.

The miss ratios for Vortex using SDP were much flatter when plotted against the number of prefetch buffers. SDP with confirmation had a miss ratio of approximately 16 misses per 1000 instructions, regardless of the number of prefetch buffers, and about 15 misses per 1000 instructions without confirmation. Generally, a flat miss-ratio curve is the result of a high accuracy, even when prefetching into one prefetch buffer.

The miss ratios for four caches without prefetching are shown ranging in size from 32Kb, two-way set-associative with a 32-byte line (32/2/32) to 64Kb, one-way set-associative with a 32-byte line (64/1/32). The figure shows that NSP was able to reduce the miss ratio of an 8Kb, one-way cache to approximately a 32Kb, two-way set-associative cache without prefetching. In fact, NSP without confirmation and prefetching into the cache was slightly better than the 32Kb, two-way set-associative cache. This represents a factor of 4 in cache size performance.

Similarly, SDP was able to achieve a miss ratio of slightly better than a 64Kb, one-way set-associative cache, a factor of 8 in cache size performance.

We see similar results in the graph for Gcc. NSP without confirmation achieved a miss ratio of approximately 22 misses per 1000 instructions when prefetching into the cache, then grew to about 27 misses per 1000 instructions with one prefetch buffer, and nearly returned to 22 misses per 1000 instructions with eight prefetch buffers. Comparing the miss ratios of the 8Kb cache using NSP to a cache without prefetching, we see a

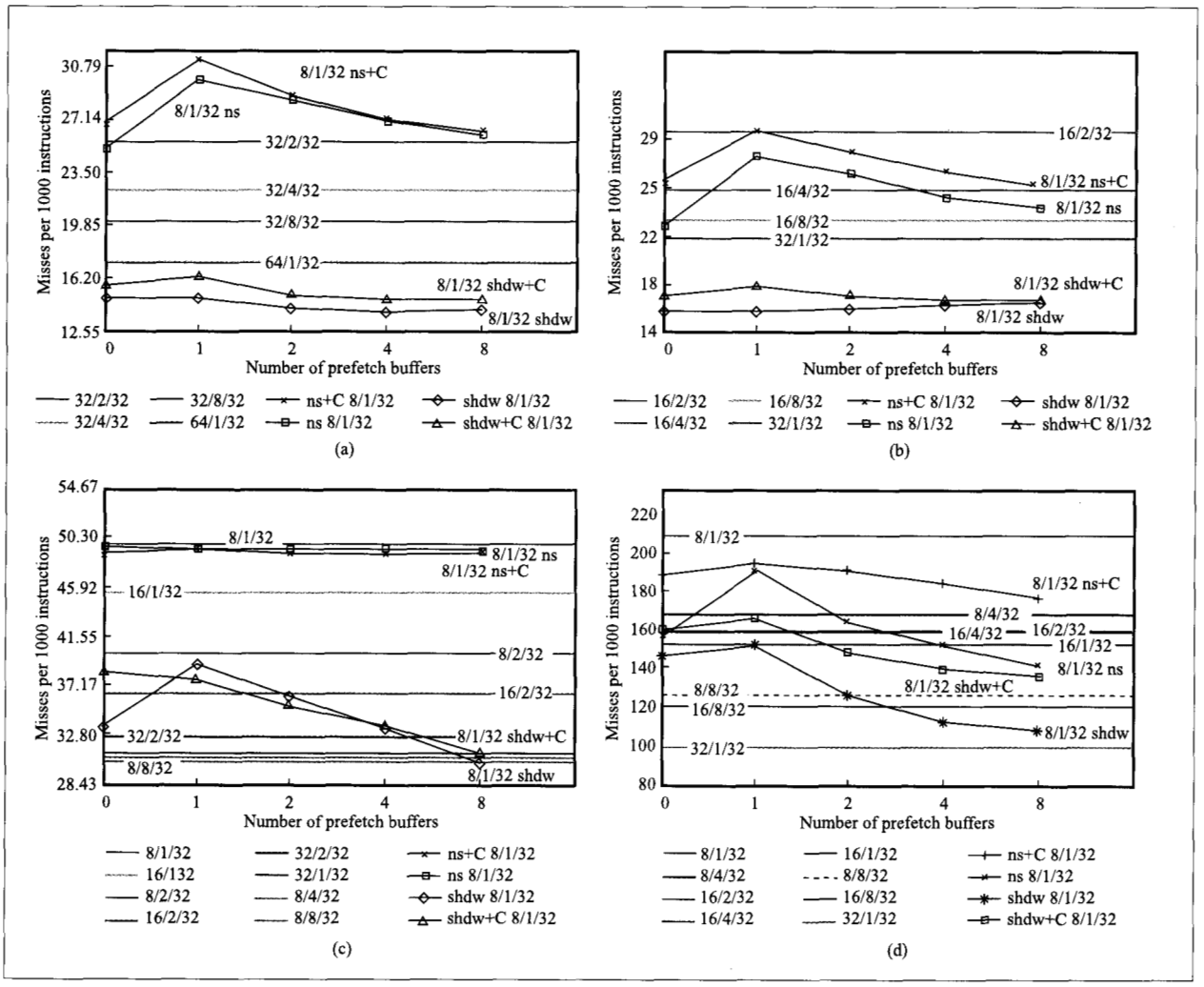

Figure 19

Equivalent cache miss ratios for prefetching and nonprefetching caches: (a) Vortex with instruction prefetching; (b) Gcc with instruction prefetching; (c) Compress with data prefetching; (d) Wave5 with data prefetching.

cache doubling. Each point lies between a 16Kb, two-way set-associative cache and a 16Kb, eight-way set-associative cache, regardless of the number of prefetch buffers.

The results for SDP show a factor of 4 improvement compared to caches with equivalent miss ratios. The 8Kb cache, with and without confirmation, achieved a miss ratio better than the 32Kb, one-way cache.

The equivalent cache graphs for Wave5 and Compress with data prefetching are analyzed next. NSP showed very little improvement when applied to Compress; the miss ratio for the 8Kb, one-way set-associative cache was only slightly improved over a cache of the same size without prefetching. This is expected because of the low miss coverage for Compress shown in Figure 13.

SDP did show a miss ratio reduction when applied to Compress. Figure 19(c) shows that the equivalent cache miss ratios range in size from 8/8/32 to 32/2/32.

The Wave5 application shows a miss-ratio reduction for both NSP and SDP. Typically, the order of miss-ratio reduction is from NSP with confirmation, then NSP without confirmation, to SDP with confirmation, to SDP without confirmation. Equivalent cache miss ratios vary from 8Kb, four-way set-associative to 32Kb, one-way set-associative.

We must now add a word of caution regarding the use of the equivalent cache graphs. It is important to know the cache size needed (without prefetching) to produce a miss ratio equal to the miss ratio achieved with prefetching, but it is not necessarily correct to assume

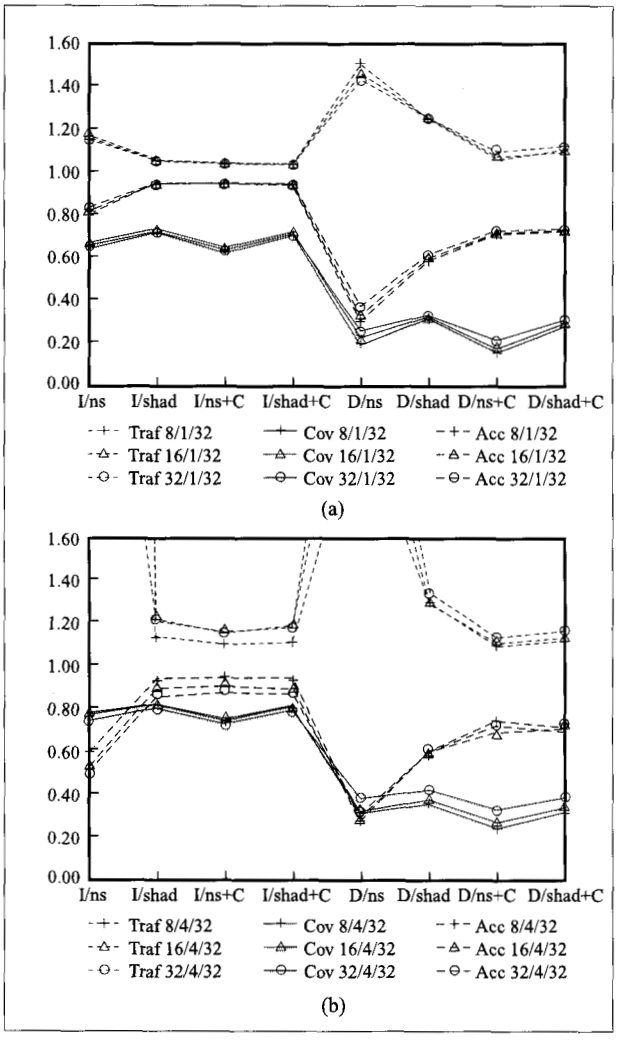

Figure 20

Average coverage, accuracy, and traffic for next-sequential prefetching and shadow-directory prefetching: (a) one-way; (b) four-way.

that each cache can achieve equal performance. The performance of a memory system can be obtained by multiplying a miss rate (misses per instruction) by a miss penalty (cycles per miss). A prefetching algorithm can initiate the fetch of a miss earlier than the processor, but may not be able to eliminate all of the delay (penalty) associated with the miss. To fully evaluate the performance of a prefetching algorithm, we must also know the timeliness of each prefetch. That is, we must know the cycles of delay associated with each prefetch. If the prefetch is started far enough in time ahead of the miss, it is possible to avoid all of the penalty associated with the miss. However, if the prefetch is started only one or two cycles ahead of the miss, there may still be an appreciable amount of delay associated with the prefetch. To accurately evaluate this delay, we must have a more detailed model of the processor and the memory system. This is clearly beyond the scope and intent of this paper.

• *Average coverage, accuracy, and traffic*

We conclude this section by showing in **Figure 20** the average coverage, accuracy, and bus-traffic ratios for 8Kb, 16Kb, and 32Kb caches that are one-way and four-way set-associative. The figure shows that instruction prefetching coverage varies from 60 to 70 percent for the one-way set-associative caches to 70 to 80 percent for the four-way set-associative cache. Typically, SDP is ten percent higher than NSP. The accuracy for instruction prefetching varied from 80 to 90 percent for the one-way caches and from 50 to 90 percent for the four-way caches. Typically, confirmation, when applied to NSP or SDP, is able to produce prefetching accuracy of 90 percent or higher for all cache configurations. The bus-traffic ratio, for instruction prefetching, ranged from 1.2 to 1.05 for all cache configurations except NSP without confirmation in a four-way set-associative cache. Here the graph is cropped at 1.6, with the traffic ratio continuing to more than 3.5. The bus-traffic ratio is large for two reasons. First, several of the SPEC applications fit into the L1 cache and have very few instruction misses. Second, the prefetching accuracy is low, approximately 45 percent. These two effects combine to produce a high bus-traffic ratio. If an application fits in the cache, every prefetch is unnecessary or incorrect. A prefetching strategy such as NSP without confirmation attempts to prefetch the next-sequential line after every taken branch. The majority of these prefetches are not-used, since the working set of the application is already in the cache. Each prefetch eventually ages out of the cache and is not used. When confirmation is added to NSP, these prefetches are eliminated and the bus-traffic ratio is reduced. Clearly, implementing a prefetching scheme such as NSP without confirmation (which always tries to prefetch the follower address) is very dangerous, since it is doubtful that the bus linking different levels in a memory hierarchy could handle the additional memory traffic due to prefetching over the normal bus traffic that exists to process base-case misses.

Miss coverage for data prefetching varies from 20 to 35 percent for a one-way set-associative cache and from 25 to 40 percent for the four-way set-associative caches. Typically, SDP produces a 10 to 15 percent higher miss coverage than NSP. Data prefetching accuracy varied from 30 percent (for NSP and SDP without confirmation) to more than 70 percent. The bus-traffic ratio for the one-way set-associative caches varied from 1.5 (for NSP without confirmation) to approximately 1.1 (NSP and SDP with confirmation). For the four-way set-associative caches, the bus-traffic ratio ranged from 1.3 to 1.1, with

the exception being NSP without confirmation. Here the bus-traffic ratio climbs to more than 3 and is again cropped at 1.6. Again low accuracy, combined with a few SPEC applications fitting into the cache, accounts for the high traffic ratios.

6. Conclusions

One of the goals in the creation of SPEC95 as a replacement for SPEC92** was to increase the demands placed on the memory system. This paper shows that only a few of the applications place more than modest demands on the memory system. This is especially true for instruction caches where only a few applications required more than a 32Kb cache to achieve miss rates of less than one per 1000 instructions.

This paper first analyzed the cache miss rates of the SPEC95 benchmarks on a large family of cache configurations. The largest reductions in cache miss rate are the result of increasing cache capacity. For several applications, increasing associativity and line size helped. The effects of increased line size are more uniform for the instruction caches. The effects of increasing associativity and line size are generally not as great as those from increasing cache capacity. For most benchmarks, a small number of instructions are responsible for the majority of the data cache misses.

Second, this paper analyzed two prefetching algorithms using the SPEC95 benchmark suite. Shadow-directory prefetching generally results in ten percent better miss coverage than next-sequential prefetching for instruction caches. On the data caches, shadow-directory prefetching generally results in 15 percent better miss coverage than next-sequential prefetching.

Confirmation significantly increased prefetching accuracy for both NSP and SDP while only slightly reducing miss coverage. For instruction prefetching, prefetching accuracy was generally 90 percent or more for NSP and SDP when confirmation was employed. Data-prefetching accuracy was approximately 70 percent or better when confirmation was used with NSP and SDP.

When prefetching accuracy is high, the extra prefetching traffic generated by prefetching must be low. Generally prefetching with confirmation produced less than ten percent extra miss traffic. When prefetching accuracy is high, there is no need for prefetching buffers. This is especially true for instruction caches. In several cases, prefetching into the cache is the best strategy for instruction prefetching.

This paper has presented a limited view of the many cache simulations we have run to analyze the memory system behavior of the SPEC95 benchmark suite. Future work remains to analyze the instructions causing the more problematic data cache misses and attempt to characterize and possibly reduce or remove them.

Acknowledgments

We thank Dan Prener, Christos Georgiou, and Eric Kronstadt for supporting this study, Ravi Nair for his program *xtrace* and related tracing software, Pradip Bose for his work in setting up the PAID'96 conference, Timothy Dinger and his group for providing supercomputer time, and the anonymous reviewers for offering many valuable suggestions.

References

1. SPEC; Standard Performance Evaluation Corporation, *http://www.specbench.org*, 1996.
2. J. Pomerene, T. Puzak, R. Rechtschaffen, and F. Sparacio, "Prefetching System for a Cache Having a Second Directory for Sequentially Accessed Blocks," U.S. Patent 4,807,110, February 1989.
3. A. J. Smith, "Cache Memories," *ACM Computing Surv.* **14,** No. 3, 473–530 (September 1982).
4. Jeffrey D. Gee, Mark D. Hill, Dionisios N. Pnevmatikatos, and Alan Jay Smith, "Cache Performance of the SPEC Benchmark Suite," *Technical Report 1049*, Computer Sciences Department, University of Wisconsin, September 1991.
5. A. Porterfield, "Software Methods for Improvement on Cache Performance on Supercomputer Applications," Ph.D. thesis, Rice University, Houston, TX, May 1989.
6. A. Klaiber and H. Levy, "An Architecture for Software-Controlled Data Prefetching," *Proceedings of the 18th Annual International Symposium on Computer Architecture*, 1991, pp. 43–53.
7. D. Callahan, K. Kennedy, and A. Porterfield, "Software Prefetching," *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991, pp. 40–52.
8. T. Mowry and A. Gupta, "Tolerating Latency Through Software-Controlled Prefetching in Shared-Memory Multiprocessors," *J. Parallel & Distributed Computing* **12,** 87–106 (June 1991).
9. T. Mowry, M. Lam, and A. Gupta, "Design and Evaluation of a Compiler Algorithm for Prefetching," presented at the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, 1992.
10. T. C. Mowry, "Tolerating Latency Through Software-Controlled Data Prefetching," Ph.D. thesis, Department of Electrical Engineering, Stanford University, March 1994.
11. Mikko H. Lipasti, William J. Schmidt, Steven R. Kunkel, and Robert R. Roediger, "SPAID: Software Prefetching in Pointer- and Call-Intensive Environments," *Proceedings of the 28th Annual International Symposium on Microarchitecture*, November 1995, pp. 231–236.
12. Z. Zhang and J. Torrellas, "Speeding Up Irregular Applications in Shared-Memory Multiprocessors: Memory Binding and Group Prefetching," *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995, pp. 188–199.

Received August 8, 1996; accepted for publication February 7, 1997

Mark J. Charney *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (charney@watson.ibm.com).* Dr. Charney received his B.S. from Princeton University in 1990 and his Ph.D. from Cornell University in 1995. He now works as a Research Staff Member at the IBM Thomas J. Watson Research Center in Yorktown Heights, New York. His current research interests are cache memories, prefetching, and simulation.

Thomas R. Puzak *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (puzak@watson.ibm.com).* Dr. Puzak received a B.S. and M.S. in mathematics from the University of Pittsburgh and a Ph.D. in electrical and computer engineering from the University of Massachusetts. Since joining IBM in 1970, he has spent nearly twenty years working as a Research Staff Member. Dr. Puzak's areas of interest include processor design, concentrating in cache and pipeline performance. While at IBM he has received Outstanding Achievement and Innovation Awards and holds several patents in high-end processor design.