REDUCING CACHE MISS RATES USING PREDICTION CACHES

James E. Bennett
Michael J. Flynn

Technical Report No. CSL-TR-96-707

October 1996

The research described herein was performed using equipment supplied by Silicon Graphics, Inc.

REDUCING CACHE MISS RATES
USING PREDICTION CACHES

by

James E. Bennett

Michael J. Flynn

Technical Report No. CSL-TR-96-707

October 1996

Computer Systems Laboratory

Stanford University

Gates Building 3A, Room 332

Stanford, California 94305-9030

pubs@shasta.stanford.edu

Abstract

Processor cycle times are currently much faster than memory cycle times, and the trend has been for this gap to increase over time. The problem of increasing memory latency, relative to processor speed, has been dealt with by adding high speed cache memory. However, it is difficult to make a cache both large and fast, so that cache misses are expected to continue to have a significant performance impact.

Prediction caches use a history of recent cache misses to predict future misses, and to reduce the overall cache miss rate. This paper describes several prediction caches, and introduces a new kind of prediction cache, which combines the features of prefetching and victim caching. This new cache is shown to be more effective at reducing miss rate and improving performance than existing prediction caches.

Key Words and Phrases: Dynamic scheduling, Memory latency, Stream buffer, Victim cache, Prediction cache

Copyright © 1996
by
James E. Bennett
Michael J. Flynn

Contents

1 Introduction **1**

2 Predictive Caches **2**

2.1 Stream buffers 2

2.2 Victim caches 3

2.3 Comparing victim caches to stream buffers 3

3 Using the Cache Miss History **3**

3.1 Miss history driven victim cache 6

3.2 Miss history driven prefetching 7

4 Simulation Methodology and Benchmarks **7**

4.1 The benchmarks 8

4.2 The processor model 8

4.3 The memory subsystem 9

5 Results **9**

5.1 The importance of lookahead 9

5.2 The final variation 12

5.3 Tolerating memory latency 12

6 Related Work **17**

7 Implementation Issues **17**

8 Conclusion **18**

List of Figures

1 Impact of Memory Latency on Dynamic Processor 1

2 Stream Buffers and Victim Cache on L1 Cache 4

3 Stream Buffers and Victim Cache on L2 Cache 5

4 Predictive Victim Cache on L1 Cache 6

5 Predictive Caches on L1 Cache 10

6 Predictive Caches on L2 Cache 11

7 Predictive Caches on L1 Cache 13

8 Predictive Caches on L2 Cache 14

9 Save Ratio and Latency Tolerance on L1 Cache 15

10 Save Ratio and Latency Tolerance on L2 Cache 16

List of Tables

1 The Benchmarks 8

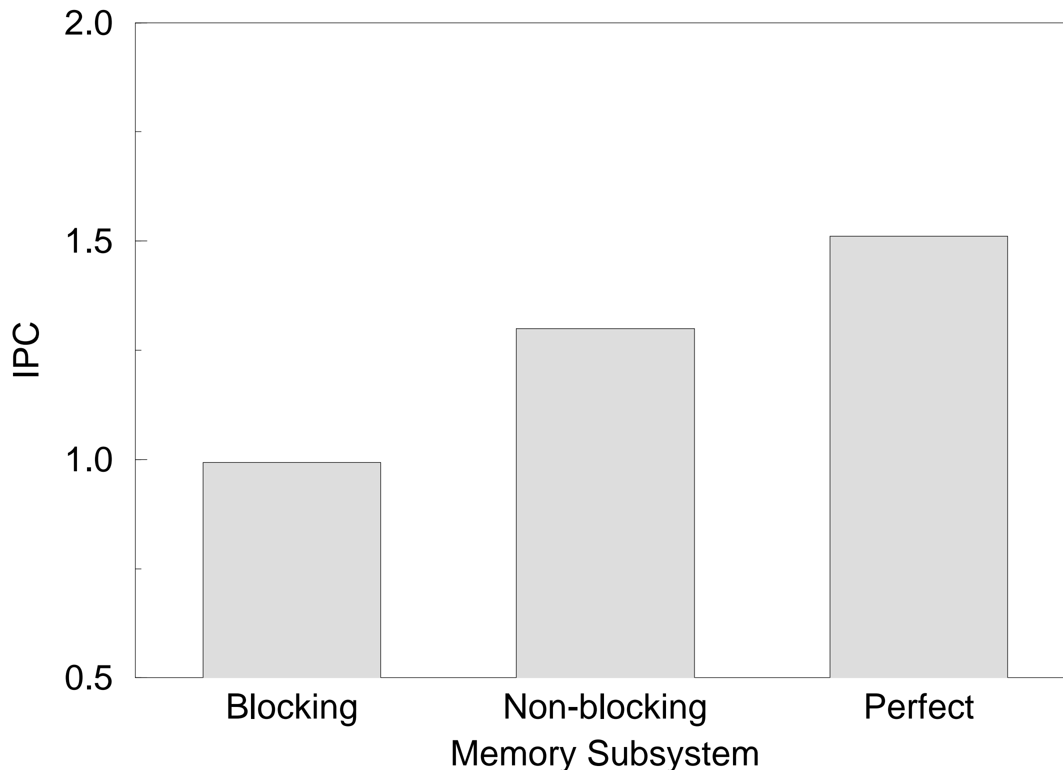

Figure 1: Impact of Memory Latency on Dynamic Processor

1 Introduction

Processor cycle times are currently much faster than memory cycle times, and the trend has been for this gap to increase over time. The problem of increasing memory latency, relative to processor speed, has been dealt with by adding high speed cache memory. However, it is difficult to make a cache both large and fast, so that cache misses are expected to continue to have a significant performance impact.

Dynamic scheduling has been proposed as a technique for tolerating memory latency[CCMH91, BP91, GGH92]. By speculating through branches, fetching load instructions, and issuing these as soon as the address is available, the dynamic processor can effectively prefetch data. A non-blocking cache[Kro81] enables this prefetching to be quite effective[BP91].

Figure 1 demonstrates this effect, but also shows that there is still a significant impact on performance due to memory latency. This graph summarizes the performance in instructions per cycle of a group of benchmarks, run on a dynamically scheduled processor, with three different memory subsystems. The details of the processor model and benchmarks are provided in section 4. The memory subsystem labeled “Perfect” assumes that all cache misses are handled in one cycle.

The gap between the performance in the case of a non-blocking cache and the performance in the

perfect memory case represents the impact of memory latency on the performance of a modern dynamically scheduled processor, such as the Intel P6 and the PA–8000[Gwe95, Gwe94]. This paper studies a class of techniques, predictive caches, that have been proposed to help close this gap.

Predictive caches use a history of recent cache misses to predict future misses, and to reduce the overall cache miss rate. Stream buffers[Jou90] use cache misses to drive prefetching, and so are an example of a predictive cache. Stream buffers predict that future cache misses follow in sequential order from current cache misses.

Victim caches, also introduced in [Jou90], are another example of a predictive cache. The victim cache predicts that cache lines that were recently replaced in the cache will be referenced again.

In this paper we introduce a new kind of predictive cache, which combines the features of prefetching and victim caching. By adapting dynamically to the program behaviour, as revealed in the miss history, this cache is more effective at reducing miss rate and improving performance than stream buffers or victim caches.

2 Predictive Caches

This section looks at the performance of two kinds of predictive caches, stream buffers and victim caches. For the purposes of this study, only data references are considered, and the memory references due to instruction fetch are ignored.

2.1 Stream buffers

Stream buffers were first proposed by Jouppi[Jou90] as an extension to the older idea of prefetching on a cache miss[Smi82]. The idea is to allocate room for a series of sequential fetches when a cache miss occurs.

On a cache miss, the stream buffers are checked to see if the data is present. If so, then the data is fetched from the stream buffer into the cache, and removed from the stream buffer. Succeeding lines in the stream buffer are moved forward to take its place.

If the data is not present in any stream buffer, then the cache line is fetched from the next level in the memory hierarchy. In addition, the least recently used stream buffer is allocated to service this new (potential) data stream. Once a stream buffer is allocated, it fetches data sequentially from memory using idle bus cycles. It continues to fetch ahead as long as there is room in the buffer. The stream buffers are serviced in a round robin fashion when more than one is active.

In this study, four stream buffers of eight entries each were allocated. This configuration was chosen to achieve most of the benefit attainable with stream buffers, based on Jouppi's study[Jou90].

2.2 Victim caches

A victim cache is a small, fully associative cache that holds data recently pre-empted from the main cache. On a cache miss, the victim cache is checked to see if the data is present. If so, then the data is fetched from the victim cache, and that line in the victim cache is freed.

On every cache miss, the line in the main cache being replaced is copied to the victim cache. If the victim cache is full, the least recently referenced line in the victim cache is replaced. In this study, we used a cache size of 32 lines, to match the number of cache lines present in the stream buffers.

2.3 Comparing victim caches to stream buffers

The performance of stream buffers and victim caches are compared in the figures 2 and 3. The benchmarks used for these graphs and the simulation methodology are described in section 4.

These graphs show the save ratio, which is the percentage of cache misses that hit in the stream buffer or the victim cache, respectively. In the case of stream buffers, partial hits can also occur, when the data requested isn't yet in the stream buffer, but is already on the way. The middle column for each benchmark counts these partial hits as a save.

In comparing the performance of stream buffers and victim caches, we see that sometimes one technique is preferable, and sometimes the other. The preferred technique varies with the cache size as well as with the benchmark. These graphs also show that stream buffers and victim caches are complementary. These observations motivated the development of an adaptive technique that combines the prefetching behaviour of stream buffers with victim caching.

3 Using the Cache Miss History

Both stream buffers and victim caches use only the most recent cache miss to drive their actions. If we keep a history of recent cache misses, we can use this to affect the caching algorithm. For example, the stream buffer filter[PK94] uses a history table of the cache line addresses of recent misses to drive stream buffer allocation. This technique, however, doesn't help in deciding between prefetching new lines and caching old lines (victims).

In our proposal, a history of the cache line *indices* of recent misses is maintained. The cache line index is the bit field from the data address that is used to select the set of lines to be checked for a tag match. The simulations in this study all assume that the main cache is four way set associative, so the cache line index in this case selects a set of four cache lines.

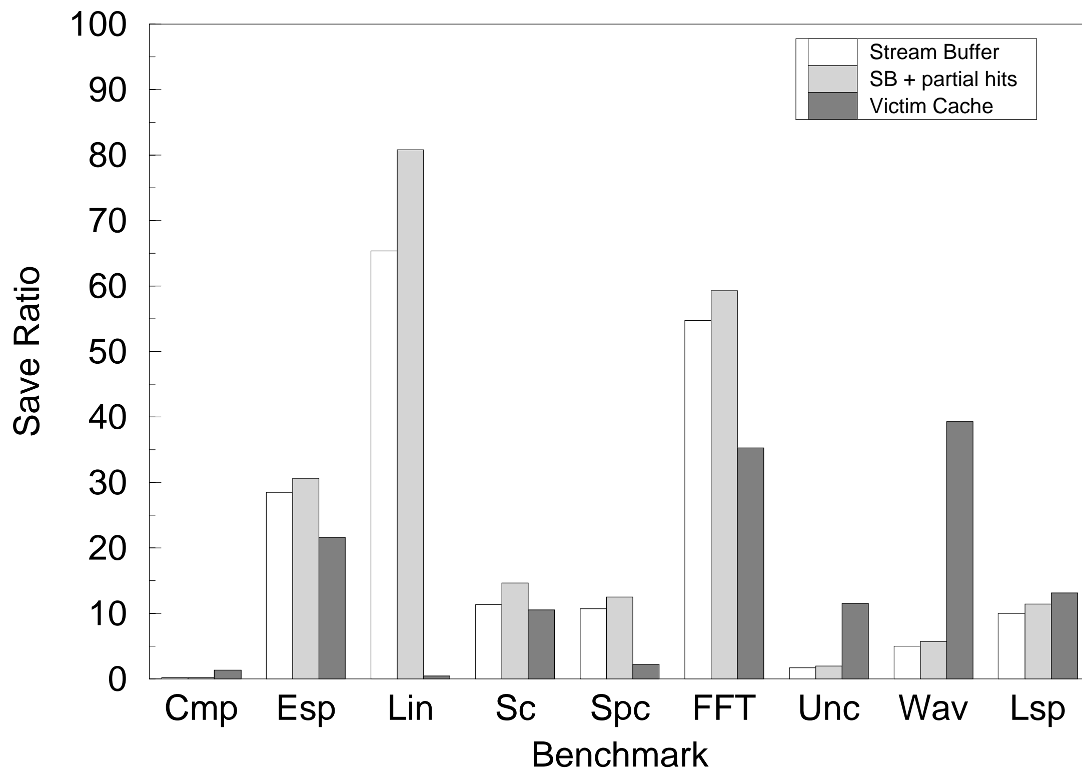

Figure 2: Stream Buffers and Victim Cache on L1 Cache

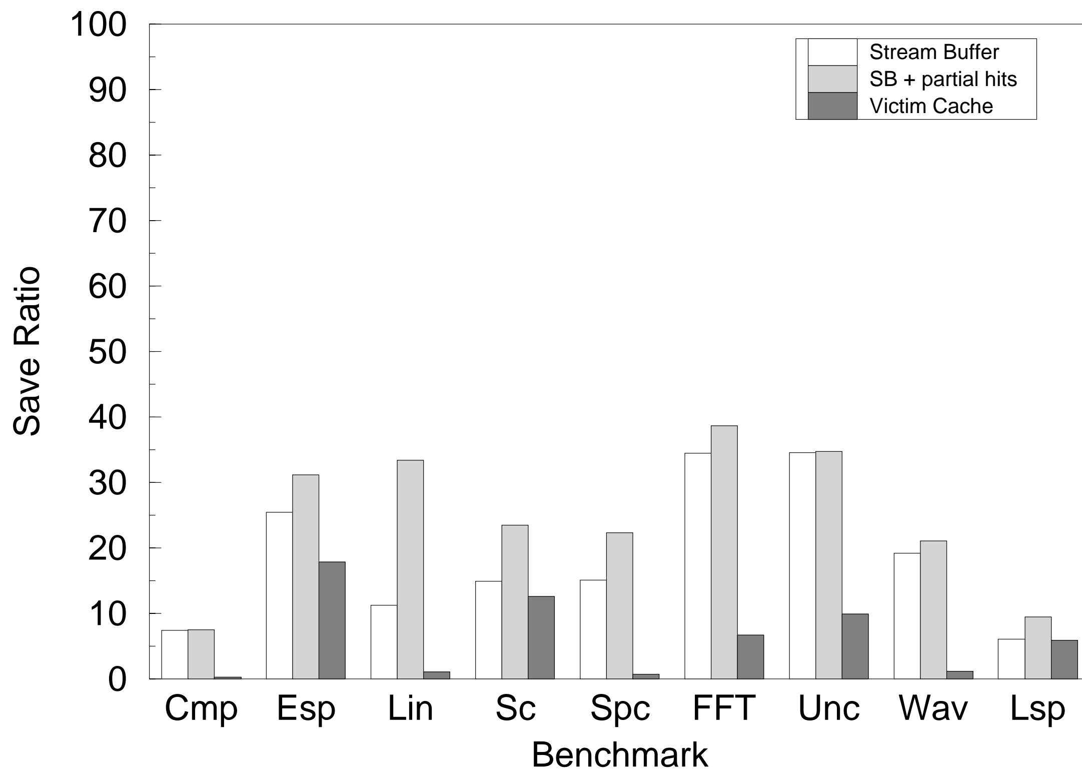

Figure 3: Stream Buffers and Victim Cache on L2 Cache

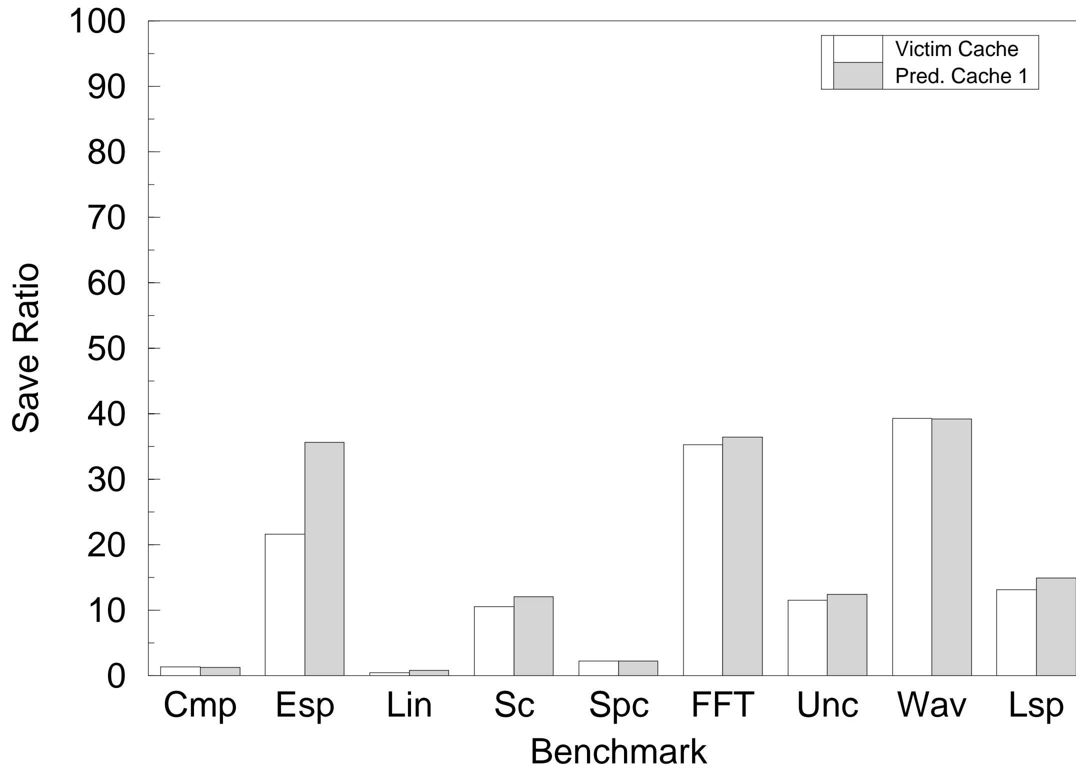

Figure 4: Predictive Victim Cache on L1 Cache

3.1 Miss history driven victim cache

To decide whether victim caching is likely to be effective, we use the idea of *hot spots* in the cache. When cache misses are uniformly spread out over all the cache lines, then a victim cache is unlikely to be effective. In this case the victims get pushed out of the victim cache before they are referenced. On the other hand, if cache misses are clustered in a hot spot consisting of a few cache lines, then the victim cache has a better chance of success. A hot spot is indicated whenever the same cache line index occurs more than once in the history of recent misses.

The miss history driven victim cache keeps a history of the ten most recent cache misses. On each cache miss, it checks to see if the cache line index of the miss matches an entry in the miss history. If a match occurs, then it copies the cache line about to be replaced (the victim) into the victim cache. Otherwise it doesn't cache the victim. On each cache miss, the cache line index of the miss is written to the miss history. The oldest entry in the miss history is discarded to make room, i.e. the miss history is a queue.

The effectiveness of this technique is shown in figure 4. This graph compares a standard victim cache to a victim cache driven by the miss history, labeled “Pred. Cache 1”. The predictive victim cache is generally as effective as the victim cache (and more effective in one case). The results for the L2 cache (not shown) also show that the predictive victim cache matches the save rate of an

ordinary victim cache.

The predictive victim cache outperforms the victim cache in the case where a large number of misses occur between reuse. If these misses are on different cache lines, then the predictive victim cache decides not to cache them, preserving potentially useful data present in the cache.

3.2 Miss history driven prefetching

This technique can be extended to support prefetching as follows. A series of sequential misses that would be captured by a stream buffer show up in the miss history as a series of misses to adjacent cache line indices. So a prefetch is indicated whenever the cache line index of the miss is adjacent to a cache line index present in the miss history.

On each cache miss, the history buffer is searched for a match with the cache line index of the current miss, and with the two values adjacent to it. This requires three associative lookups in parallel, but the table to be searched is quite small, as it only has ten entries. The table width is small as well; for example, a 32K cache with 16 byte lines has a nine bit cache line index.

These lookups allow forward strides, backward strides and cache hot spots to be detected. If a match occurs with the current cache line index, a hot spot in the cache is indicated. If a match occurs with the previous cache line index, then a forward stride is indicated, and backward strides are indicated by a match with the cache line index following the current index.

When a stride is detected, instead of copying the victim line into the victim cache, a spot is reserved in the victim cache and a prefetch is issued. The victim cache is acting as a prefetch buffer in this case. The address for the prefetch is computed by adding (or subtracting, for a backward stride) the size of the cache line to (from) the address of the current cache miss. All other operations proceed as described previously.

If more than one match occurs, then a decision has to be made about which course of action to take. In this study a fixed priority was assumed, in which forward strides take precedence over backward strides, and backward strides take precedence over victim caching.

In the following section we describe the simulation methodology used in this study, and the benchmarks. The performance of the predictive caching scheme just described is presented in section 5.

4 Simulation Methodology and Benchmarks

In order to accurately model a dynamically scheduled processor, an execution based simulation method was chosen. In this way the effect of cache misses on the instruction schedule can be correctly modeled, as well as memory accesses that are generated along incorrectly speculated paths. This information (speculative memory accesses) isn't available to a trace based simulator.

Benchmark	*Instructions*	*Miss rate [size] L1 cache*	*Miss rate [size] L2 cache*
Compress	2.1M	8.83% [8K]	2.32% [128K]
Espresso	35.6M	7.75% [2K]	1.12% [8K]
Linpacks	67.3M	7.39% [8K]	2.21% [32K]
Sc	39.9M	6.41% [2K]	0.62% [32K]
Spice	88.5M	9.45% [8K]	2.55% [64K]
FFT	3.1M	6.30% [16K]	0.19% [32K]
Uncompress	1.6M	6.66% [2K]	0.96% [16K]
Wave	30.8M	10.23% [8K]	2.35% [32K]
Lisp	12.4M	4.11% [2K]	1.67% [8K]

Table 1: The Benchmarks

The disadvantage of an execution based simulation is that the speed of the simulation limits the number of cycles that can be reasonably simulated. To allow the simulation of complete programs, rather than an initial subset, the input data had to be reduced in some cases. The cache size was then fixed for each benchmark to obtain miss rates comparable to those observed in real world applications[MDO94].

4.1 The benchmarks

A set of benchmarks was chosen from the SPEC 92 benchmark suite, together with a one dimensional FFT implementation and the Linpack benchmark, a collection of linear algebra routines. This set was chosen in order to provide a variety of reference patterns and programming styles, and includes both integer and floating point intensive benchmarks. All benchmarks were run to completion, in some cases on a reduced problem size. Table 1 shows the benchmark length, in instructions executed, and the level one and two cache sizes and miss rates for each benchmark.

4.2 The processor model

The processor model was selected to represent the current generation of dynamically scheduled processors, such as the Intel P6 and the PA-8000[Gwe95, Gwe94]. It is a four issue, dynamically scheduled processor, with register renaming, branch prediction, speculative execution, and precise interrupts[HP90]. Instructions issue out-of-order, as their operands become available, and a reorder buffer is used to restore the precise state after an interrupt[SP85, Joh91]. The load/store buffer has 32 entries, and the reorder buffer is 64 entries long. For comparison, the P6 has 40 reorder buffer entries, and the PA-8000 has 56.

More detailed information on the benchmarks, processor model, and simulation environment are available in [BF95].

4.3 The memory subsystem

The simulator supports only a single level cache. In order to investigate the impact of both level one and level two cache misses, each benchmark was run with two different cache sizes, one modeling the first level cache, and the other modeling the second level cache. The cache is single ported, so only one load or store instruction can access the cache each cycle. It is four way set associative with an LRU replacement policy and a fixed line size of 16 bytes. The cache is write back with write miss allocate. The same line size was used for the L1 and L2 caches, so that the effects due to cache size and memory latency could be isolated from the effects due to varying the line size.

Both memory latency and memory bus traffic were modeled. An L1 cache miss has a latency of 8 cycles and consumes 4 bus cycles. An L2 cache miss has a latency of 50 cycles and consumes 8 bus cycles. The bus activity due to instruction cache misses and other system activities, for example disk accesses, was not modeled.

Figure 1 shows the equally weighted average of the IPC of all the benchmarks, for the L2 cache case. In many current systems, the miss penalties are considerably greater than the values assumed in this study (private communication, Larry McVoy). In this case, there is even more performance to be gained by reducing cache miss rates.

5 Results

The performance of three predictive caches are compared in the figures 5 and 6. These graphs show the save ratio, with partial hits not included. The miss history driven prefetching cache, labeled "Pred. Cache 2", is the cache described in section 3.2.

In the case of the L1 cache, we can see that the predictive cache performs as desired, outperforming both the victim cache and the stream buffer. Unfortunately, in the case of the L2 cache, the predictive cache doesn't perform as well as the stream buffer or the victim cache in some cases. What has gone wrong?

5.1 The importance of lookahead

One advantage that stream buffers have over the predictive cache is that stream buffers can fetch far ahead of the computation during idle bus cycles. This is particularly useful in the L2 cache case, when memory latency is so large. The predictive cache, on the other hand, never gets more than one cache line ahead of the computation. If the results for partial hits are included, however, then the picture changes. The predictive cache in this case outperforms both stream buffers and victim caches. So the predictive cache is correctly issuing prefetches, but it isn't doing it soon enough.

The predictive cache fetches one cache line ahead of the computation, by adding the cache line size to the address of the current cache miss. If we instead add twice the cache line size to the address, this doubles the amount of lookahead. When the cache misses from the application occur at

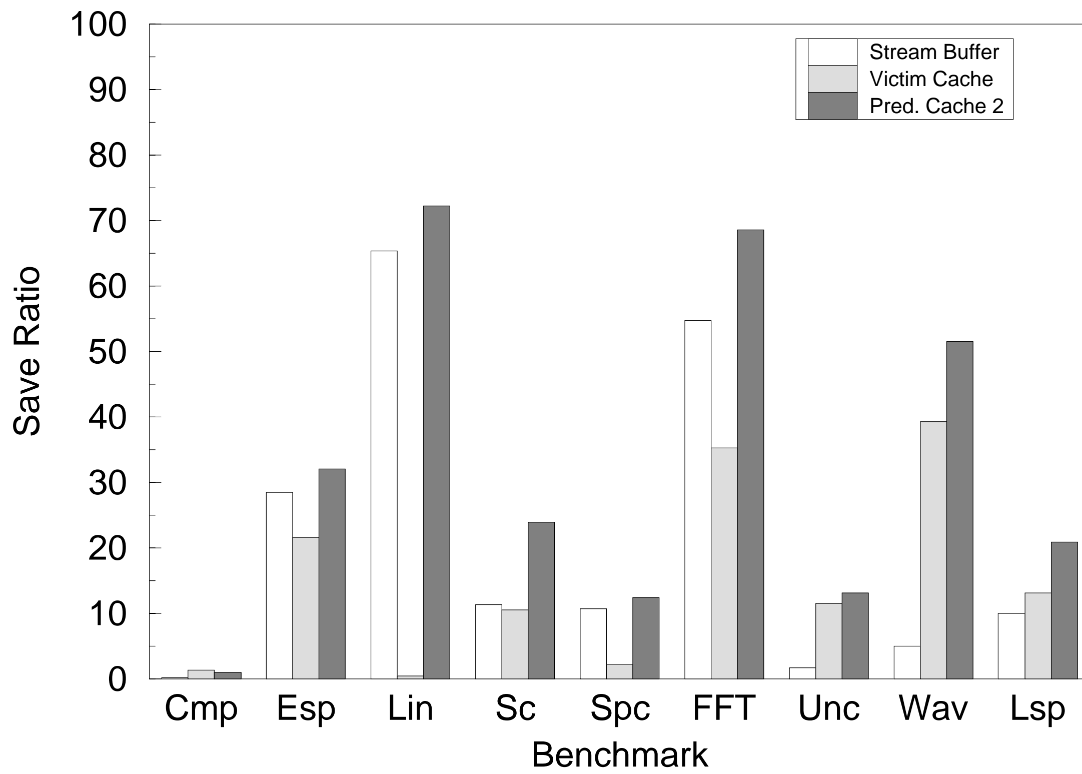

Figure 5: Predictive Caches on L1 Cache

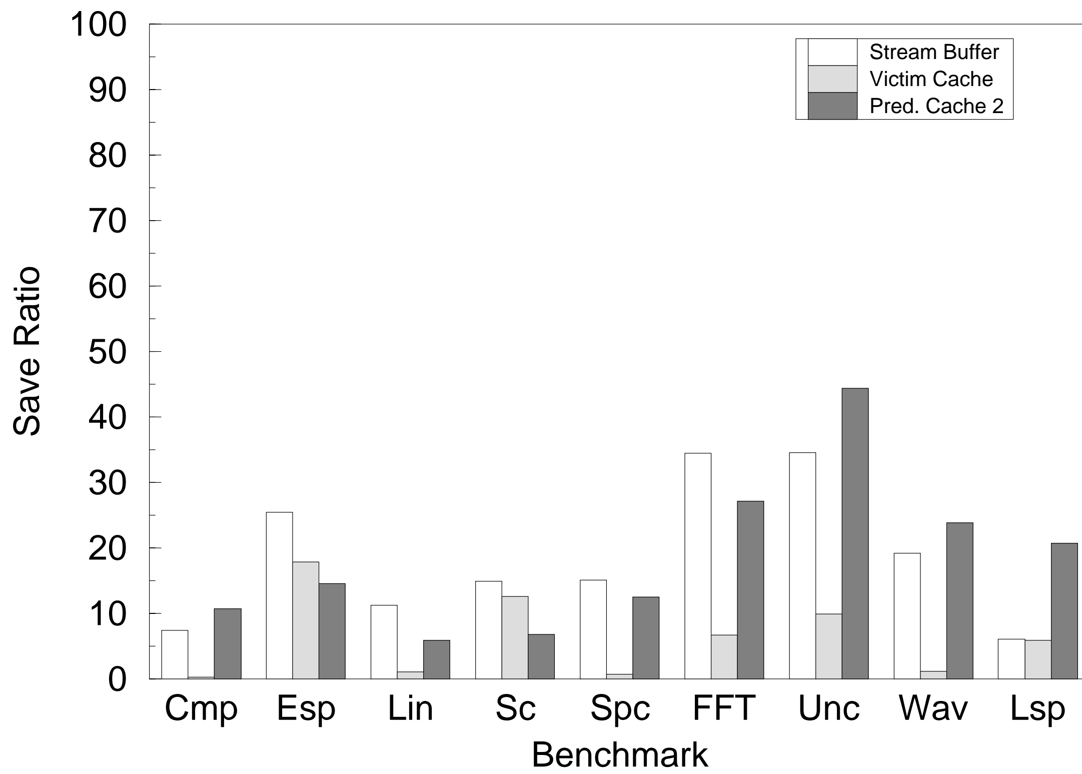

Figure 6: Predictive Caches on L2 Cache

consecutive addresses, the predictive cache still prefetches consecutive addresses, but the addresses are offset by twice the cache line size from the miss addresses.

How far ahead to fetch depends on both the application and the memory latency. However, the number of partial hits provides feedback as to whether we are prefetching sufficiently far ahead. When there are a large number of partial hits, this indicates that we need to prefetch further ahead. When there are no partial hits, then we are (possibly) prefetching too far ahead. We found that the following feedback technique successfully maintained the correct degree of prefetching.

5.2 The final variation

This version of the predictive cache has three counters and a register to hold the prefetch amount, which is the number that is added to the cache miss address to generate the prefetch address. The counters are a tick counter, a miss counter, and a partial hit counter. The tick counter is incremented every time the predictive cache is referenced, the miss counter and the partial hit counter are incremented whenever the references misses or partially hits in the predictive cache, respectively.

Every 20 ticks (i.e. when the tick counter reaches 20), the prefetch amount is re-evaluated, and all the counters are set to zero. If there are two or more partial hits, then the prefetch amount is multiplied by two. If there is one partial hit, then the prefetch amount is left unchanged. If there are no partial hits, and more than ten misses, then the prefetch amount is divided by two. The prefetch amount is initialized to equal the cache line size, and is never allowed to fall below this amount.

The results of this scheme are shown in figures 7 and 8. The adaptive scheme just described is labeled "Pred. Cache 3". This final variation outperforms both the victim cache and stream buffers in both cache configurations.

5.3 Tolerating memory latency

One way to measure how good a job the predictive cache is doing is to look at how well it closes the gap between the base performance and the performance of the perfect memory subsystem. The "latency tolerated" statistic (figures 9 and 10) expresses this as a percentage, where 0% means the performance of the predictive cache was the same as the base performance, and 100% means the performance was the same as that of perfect memory.

These graphs show that the save ratio and the relative improvement in performance are well correlated. For the L1 cache case, the average reduction in miss rate was 34%, and the average amount of latency tolerated was 30%. In the L2 cache case, the average reduction in miss rate was 29%, and the average amount of latency tolerated was 28%.

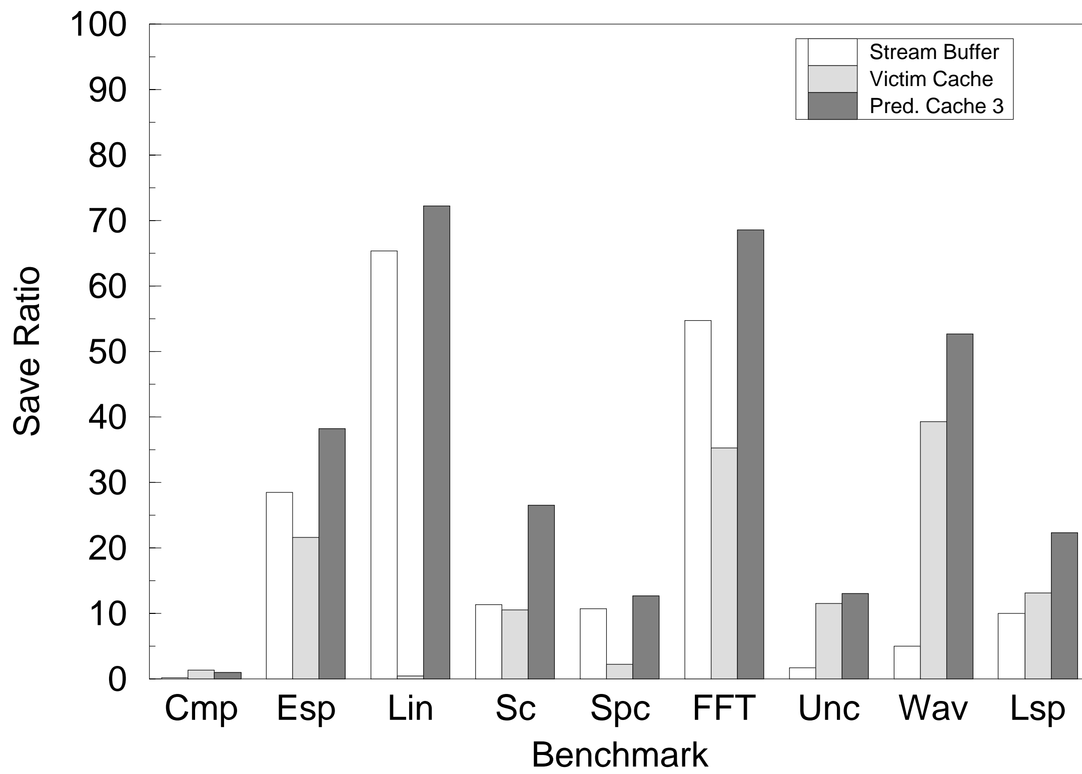

Figure 7: Predictive Caches on L1 Cache

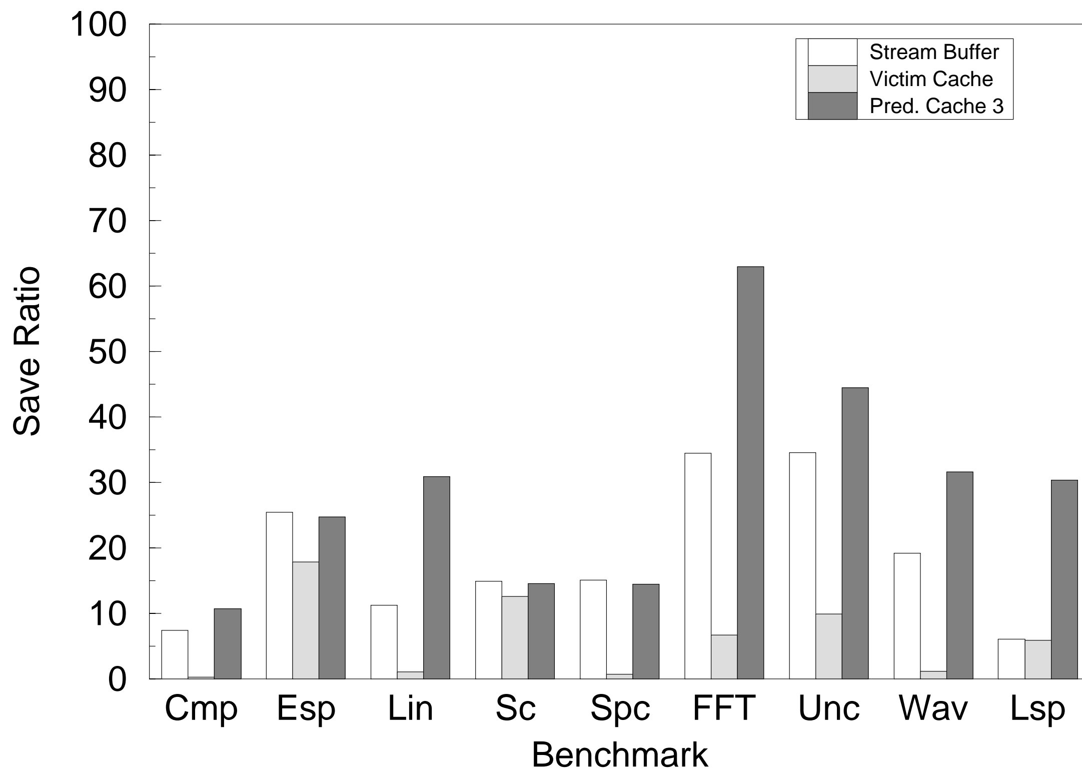

Figure 8: Predictive Caches on L2 Cache

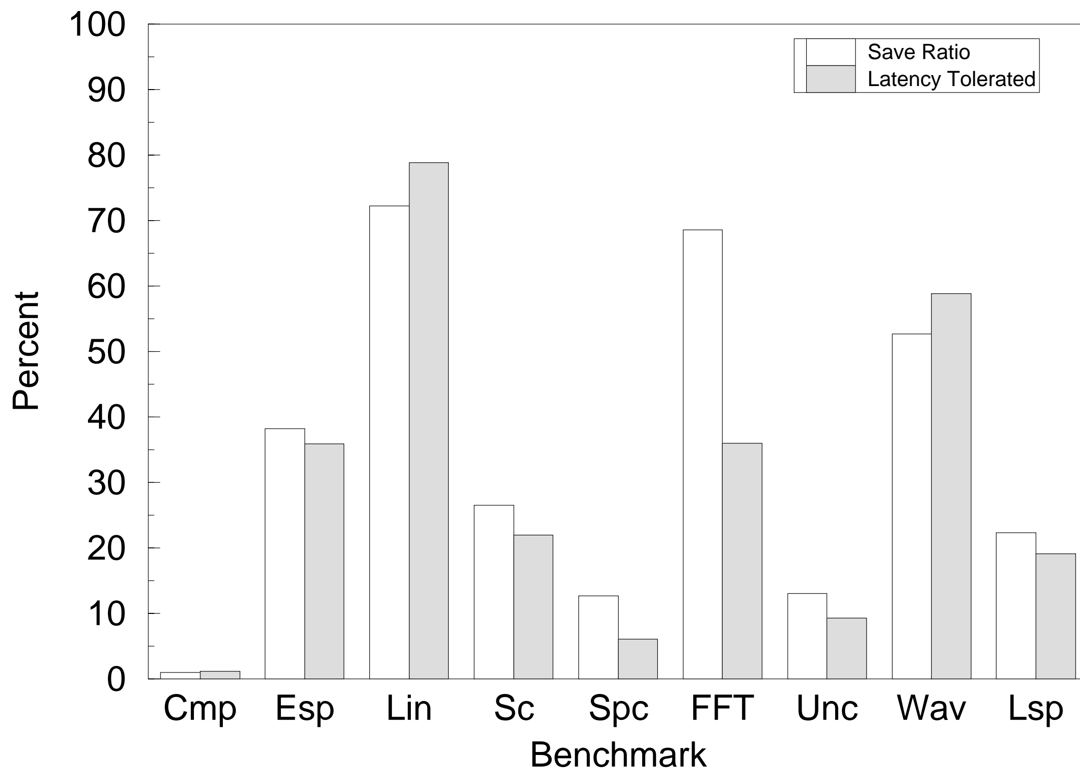

Figure 9: Save Ratio and Latency Tolerance on L1 Cache

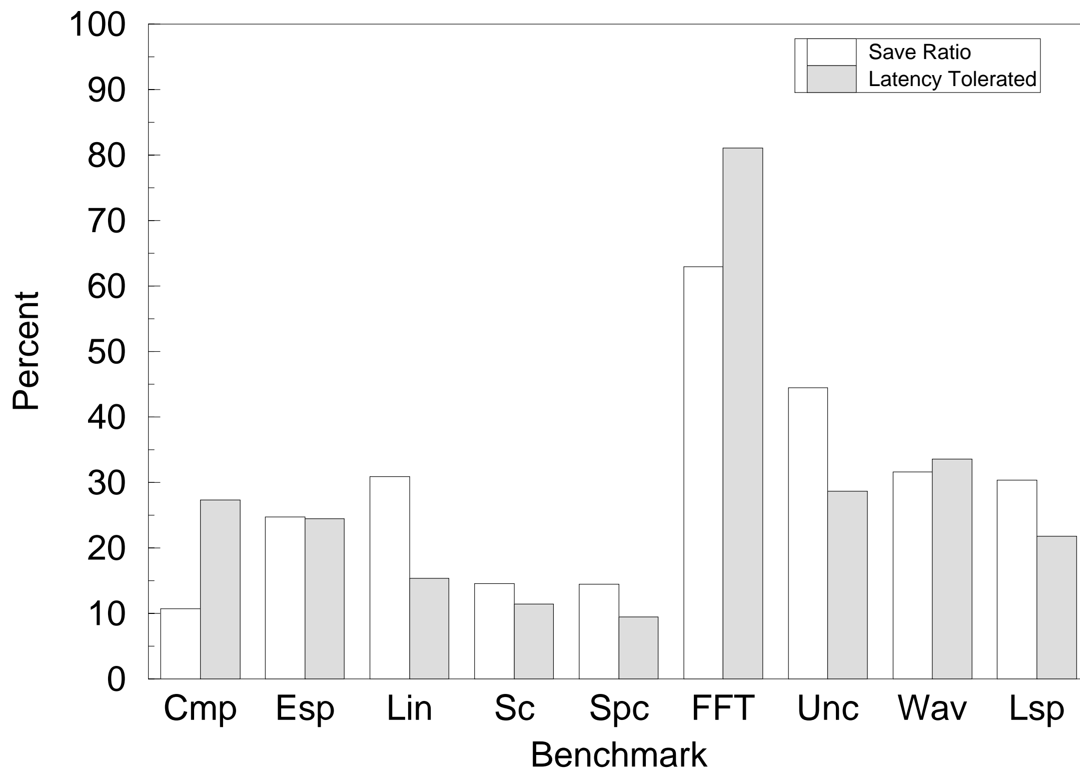

Figure 10: Save Ratio and Latency Tolerance on L2 Cache

6 Related Work

Palacharla and Kessler[PK94] proposed stream buffer filters, which reduce the memory bandwidth requirements of a stream buffer by screening out useless prefetches. In general, stream buffer filters cause a slight reduction in the hit ratio of the stream buffer, whereas the predictive cache scheme proposed here has a better hit ratio than a stream buffer. Also the miss history table proposed here is much smaller than the stream buffer filter, since only the cache line index is stored.

The predictive cache reduces memory bandwidth requirements in three ways. First, no prefetching is performed until a stream is confirmed by two adjacent cache misses. Second, since the prefetching is driven by cache misses, it stops as soon as the sequence of cache misses is complete. Third, victim caching, when successful, provides data without consuming any bandwidth at all.

On the other hand, stream buffer filters can detect non-unit strides, where the predictive caches discussed here don't. So for applications that access their data using large strides, the stream buffer filter may provide better performance. The results reported in [PK94] didn't distinguish between partial hits and hits, so they can't be directly compared to the results reported here.

Phalke and Gopinath[PG95] also propose a technique for prefetching based on the miss history. Their idea is to model the addresses of the cache misses as a Markov chain. Their implementation stores an approximate Markov model in main memory, and uses this to direct the prefetching. This is considerably more costly than the simple techniques discussed here, but it also has a better miss ratio than sequential prefetching.

Dahlgren, Dubois, and Stenstrom[DDS95] propose an adaptive scheme for determining how far to lookahead when doing sequential prefetching. They use an estimate of prefetch effectiveness, based on counting the number of prefetches and comparing it to the number of useful prefetches, to update the degree of prefetching. This is similar to our lookahead proposal in section 5.2, except that we use the frequency of partial hits for this purpose. Their paper studies the effectiveness of sequential prefetching in the context of shared memory multi-processors.

Stride-directed prefetching[Skl92, FPJ92, CB95] has been proposed as another technique for hardware prefetching. Stride-directed prefetching, like stream buffer filters, has the ability to detect non-unit strides. However, to implement these techniques requires access to the instruction stream, which may not always be available.

7 Implementation Issues

Although the miss history table is quite small, there is an approximate implementation that may take up even less area. The idea is to keep a single bit per cache line index, stored adjacent to the cache tags. When this bit is set, it indicates that there was a cache miss at that index. These bits get cleared periodically, so that the set bits represent only the most recent misses.

When a cache miss occurs, the determination of whether to issue a prefetch or cache the victim

line can be made by examining the three bits at the current, next, and previous cache line indices. These three bits can be driven out to the cache control logic using a precharge technique, just like the bit lines in the cache.

In addition to this prediction mechanism, an adder is needed to compute the prefetch address, a register is needed to hold the prefetch amount, and a shifter is needed to multiply or divide the prefetch amount by two. The shift amounts allowed are only +1, -1, or 0, so the shifter is small. Three small counters are needed to implement the prefetch amount feedback mechanism.

8 Conclusion

In this paper, we have examined a class of techniques, predictive caches, that improve performance by reducing the overall cache miss rate. The most effective of these techniques combines the features of a victim cache and a stream buffer, and adaptively determines the degree of lookahead. Using this scheme, 28% to 30% of the memory latency was tolerated, over a range of cache sizes and miss penalties, for this set of benchmarks.

We also showed that the save ratio (the percent of cache misses that hit in the predictive cache) correlates well with the amount of latency tolerated. Note that the save ratio as computed here does not include partial hits, which are accesses to cache lines that have been fetched but have not yet arrived in the cache.

References

[BF95] J. E. Bennett and M. Flynn. Performance factors for superscalar processors. Technical Report CSL-TR-95-661, Stanford University, Computer Systems Laboratory, February 1995.

[BP91] M. Butler and Y. Patt. The effect of real data cache behavior on the performance of a microarchitecture that supports dynamic scheduling. In *Proc. of the 24th International Symposium on Microarchitecture*, pages 34–41, November 1991.

[CB95] T-F. Chen and J-L. Baer. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers*, 44:609–23, May 1995.

[CCMH91] P. Chang, W. Chen, S. Mahlke, and W. Hwu. Comparing static and dynamic code scheduling for multiple-instruction-issue processors. In *Proc. of the 24th International Symposium on Microarchitecture*, pages 25–33, November 1991.

[DDS95] F. Dahlgren, M. Dubois, and P. Stenstrom. Sequential hardware prefetching in shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 6:733–46, July 1995.

[FPJ92] J. Fu, J. Patel, and B. Janssens. Stride directed prefetching in scalar processors. In *Proc. of the 25th International Symposium on Microarchitecture*, pages 102–110, December 1992.

[GGH92] K. Gharachorloo, A. Gupta, and J. Hennessy. Hiding memory latency using dynamic scheduling in shared-memory multiprocessors. In *19th International Symposium on Computer Architecture*, pages 22–33, May 1992.

[Gwe94] L. Gwennap. PA-8000 combines complexity and speed. *Microprocessor Report*, 8:5–9, November 1994.

[Gwe95] L. Gwennap. Intel's P6 uses decoupled superscalar design. *Microprocessor Report*, 9:9–15, February 1995.

[HP90] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., Palo Alto, CA, 1990.

[Joh91] Mike Johnson. *Superscalar Microprocessor Design*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1991.

[Jou90] N. P. Jouppi. 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*, pages 364–73, May 1990.

[Kro81] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proc. Eighth Symposium on Computer Architecture*, pages 81–87, May 1981.

[MDO94] A. Maynard, C. Donnelly, and B. Olszewski. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. In *Proc. of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 145–56, October 1994.

[PG95] V. Phalke and B. Gopinath. A miss history-based architecture for cache prefetching. In *International Workshop IWMM 95 Proceedings*, pages 381–98, September 1995.

[PK94] S. Palacharla and R.E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *Proc. of the 21st International Symposium on Computer Architecture*, pages 24–33, April 1994.

[Skl92] I. Sklenar. Prefetch unit for vector operations on scalar computers. *Computer Architecture News*, 20:31–37, September 1992.

[Smi82] A. J. Smith. Cache memories. *Computing Surveys*, 14:473–530, September 1982.

[SP85] J. E. Smith and A. R. Pleszkun. Implementation of precise interrupts in pipelined processors. In *12th International Symposium on Computer Architecture*, pages 36–44, June 1985.