LATENCY TOLERANCE FOR DYNAMIC PROCESSORS

James E. Bennett
Michael J. Flynn

Technical Report No. CSL-TR-96-687

January 1996

The research described herein has been supported by NASA-Ames under grant NAGW 419, using equipment supplied by Silicon Graphics, Inc.

LATENCY TOLERANCE FOR DYNAMIC PROCESSORS

by

James E. Bennett

Michael J. Flynn

Technical Report No. CSL-TR-96-687

January 1996

Computer Systems Laboratory

Departments of Electrical Engineering and Computer Science

Stanford University

Stanford, California 94305-4055

pubs@shasta.stanford.edu

Abstract

While a number of dynamically scheduled processors have recently been brought to market, work on hardware techniques for memory latency tolerance has mostly targeted statically scheduled processors. This paper attempts to remedy this situation by examining the applicability of hardware latency tolerance techniques to dynamically scheduled processors. The results so far indicate that the inherent ability of the dynamically scheduled processor to tolerate memory latency reduces the need for additional hardware such as stream buffers or stride prediction tables. However, the technique of victim caching, while not usually considered as a latency tolerating technique, proves to be quite effective in aiding the dynamically scheduled processor in tolerating memory latency. For a fixed size investment in microprocessor chip area, the victim cache outperforms both stream buffers and stride prediction.

Key Words and Phrases: Dynamic scheduling, Memory latency, Stream buffer, Stride prediction, Victim cache, Nonblocking cache

Contents

List of Figures

List of Tables

1 Introduction

The current crop of microprocessors includes a number of dynamically scheduled machines, such as the Intel P6 and the PA–8000[Gwe95, Gwe94]. However, most recent studies of hardware techniques for tolerating memory latency, such as stride prediction[CB95] and stream buffers[KF95, Jou90], have included only statically scheduled processors. In this paper we study the effectiveness of these hardware techniques when applied to dynamically scheduled processors.

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. This aspect of dynamically scheduled processors may limit the effectiveness of other hardware schemes for latency tolerance. This study tends to confirm this suggestion, but at the same time, shows that there are other techniques that can enhance the effectiveness of dynamic processors in tolerating memory latency. In particular, victim caches[Jou90] work well in conjunction with a dynamic processor.

This paper is organized as follows. In section 2, the experimental procedure is outlined, with descriptions of the processor model, the simulation environment and the benchmarks. Section 3 describes the techniques used to improve the processor's latency tolerance and the results of the simulation runs. Section 4 discusses these results and related work on techniques for tolerating memory latency. Finally, section 5 presents some conclusions and plans for future work.

2 Experimental Procedure

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.

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].

2.1 The benchmarks

A set of benchmarks was chosen from the SPEC 92 benchmark suite, together with 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. Cache sizes were fixed for each benchmark to maintain miss rates of around 5–10% for the level

Benchmark	*Instructions*	*Miss rate [size]* *L1 cache*	*Miss rate [size]* *L2 cache*
Compress	2.3M	9.14% [8K]	2.29% [128K]
Uncompress	1.7M	10.55% [1K]	0.99% [16K]
Espresso	37.3M	3.53% [4K]	1.19% [8K]
Sc	37.8M	4.93% [4K]	2.71% [16K]
Xlisp	11.8M	6.55% [1K]	1.66% [8K]
Linpacks	66.9M	7.33% [8K]	2.15% [32K]
Spice	91.9M	9.39% [8K]	2.39% [64K]
Wave	32.7M	9.51% [8K]	2.29% [32K]

Table 1: The Benchmarks

one cache, and 1–2% for the level two cache. Table 1 shows the benchmark length, in instructions executed, and the level one and two cache sizes and miss rates for each benchmark.

2.2 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].

2.3 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.

Benchmark	*Stream buffer on L1 cache*	*Stream buffer on L2 cache*
Compress	99.76%	92.45%
Uncompress	98.54%	65.15%
Espresso	66.14%	66.57%
Sc	80.46%	81.58%
Xlisp	94.36%	86.13%
Linpacks	12.23%	64.48%
Spice	86.00%	75.22%
Wave	93.70%	77.37%

Table 2: Stream Buffer Miss Rates

In many current systems, the miss penalties are considerably greater than this (private communication, Larry McVoy), so that the results presented here actually understate the speedup that might be achieved by these various latency tolerating techniques.

3 Results

3.1 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. In this study, four stream buffers were allocated, each eight cache lines long.

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. The succeeding buffers in the stream 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. Each cycle when the bus is idle, the stream buffers are checked to see if they have an available buffer, and if so a prefetch is issued. The stream buffers are serviced in a round robin fashion when more than one is active.

Figure 1 shows the performance of the dynamic processor with and without stream buffers. The "Performance Ratio" is the number of cycles it took to execute the benchmark on the given processor model, divided by the number of cycles it took to execute the same benchmark on a system with perfect memory (0-cycle cache miss latency). A performance ratio of 1.0 indicates that all the memory latency was effectively hidden, while a ratio of 1.2 indicates that there was a 20% degradation in performance due to cache misses.

In this case we see that the performance is little changed in most cases. Only Linpack, with its highly sequential access pattern, shows a dramatic improvement. The effectiveness of stream buffers

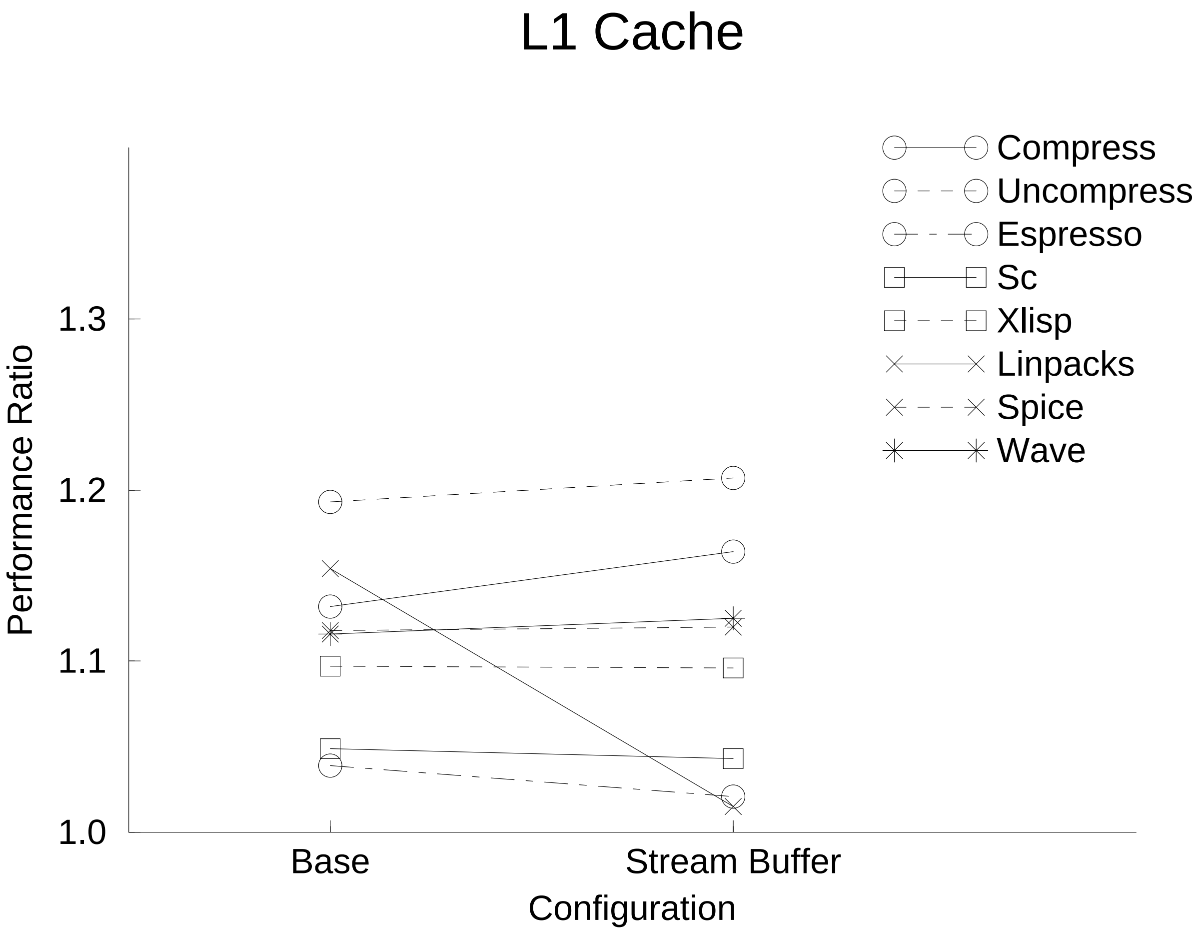

Figure 1: Stream Buffers - L1 Cache

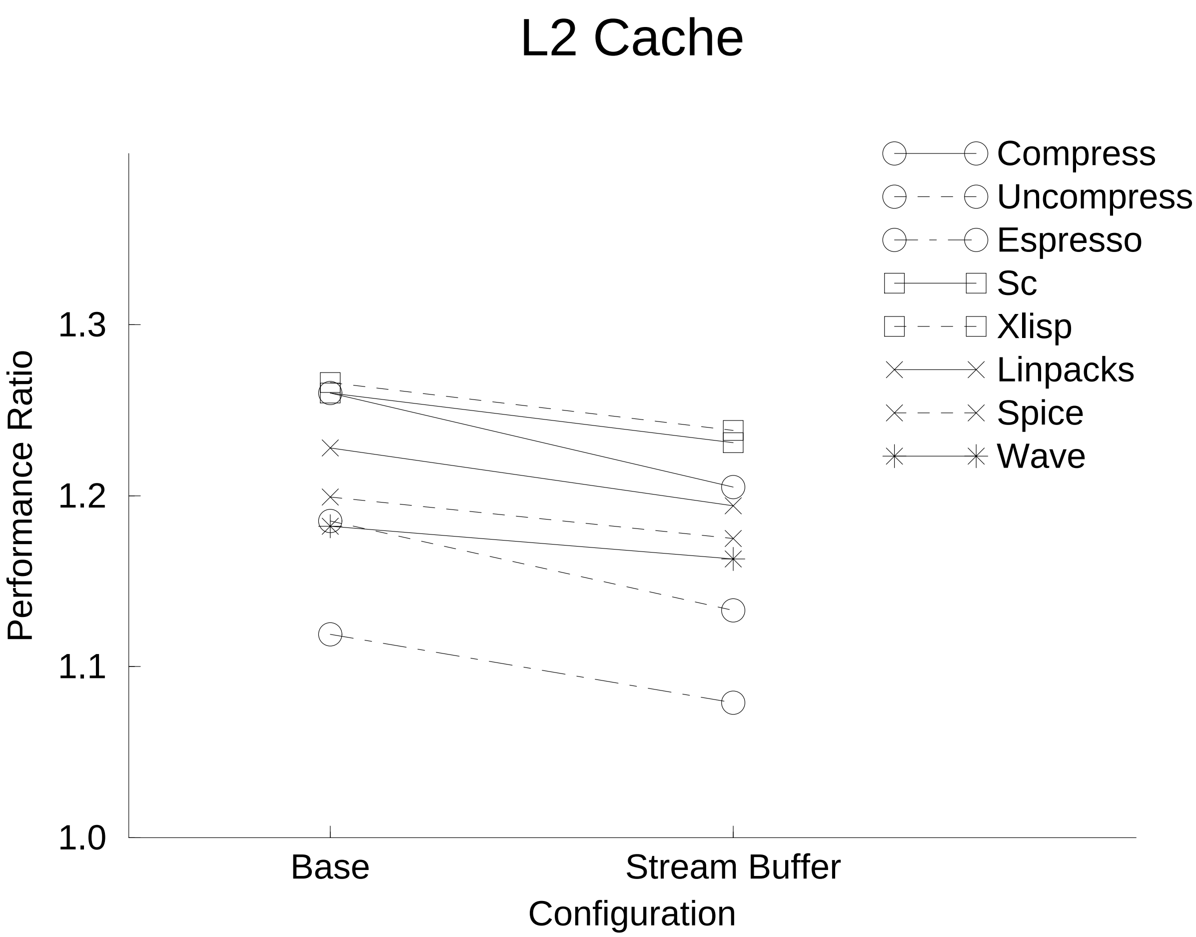

Figure 2: Stream Buffers - L2 Cache

varied widely with the benchmark, as can be seen by comparing miss rates (see table 2, under "L1 cache"). The miss rate given here is the local miss rate, the percent of misses out of all references to the stream buffer.

In cases where the miss rate was especially high, such as Compress and Uncompress, the performance got worse when stream buffers were added. This degradation is caused by the extra bus traffic generated by stream buffers. Even though stream buffer prefetches were issued only when the bus was idle, once on the bus they take up several cycles, and may delay the handling of subsequent cache misses.

In the case of a level two cache (figure 2), the addition of stream buffers always improved performance, although the improvement isn't particularly impressive. By looking at the stream buffer miss rates in table 2 for the L2 cache, we see that miss rates have improved for many of the benchmarks, but in the case of Linpacks the miss rate got much worse.

Because of the larger size of the level two cache, most of the capacity and conflict misses have been satisfied, and the remaining misses are largely compulsory[HP90] or start up misses. Stream buffers can exploit the spatial locality of these start up misses, which for most of the benchmarks is greater than spatial locality of the misses from the level one cache.

Linpacks has a high degree of sequentiality in its cache misses from the level one cache. In the case of the level two cache, however, the sequential accesses get satified from the cache, and the cache misses have much less locality.

3.2 Stride prediction

The basic idea of using stride information to predict future memory references, and collecting this information dynamically, was reported by several researchers[CB91, Skl92, FPJ92]. A stride prediction table, indexed by the program counter, records the address of the memory operation at that location. When the instruction at that location is executed a second time, the new memory address is compared to the previous address, and if they are different, then their difference is the stride.

In the scheme of Chen and Baer[CB95], the stride prediction table is indexed by a "lookahead PC", which is the predicted value of the program counter some number of cycles in the future. This number can be tuned to reflect the expected latency of the memory subsystem. Chen and Baer propose associating some additional state with the stride prediction table, and to make predictions only when the stride is non-zero and the stride prediction has been confirmed. That is, wait to make a prediction until that entry is accessed a third time, and the second stride calculation matches the first. The purpose of this is to reduce the number of incorrect or useless predictions.

In this study, we followed the approach of Chen and Baer, except for the lookahead PC. For a dynamically scheduled processor, the program counter is not such a well defined concept. In this work, we have defined the program counter as the address of the instruction currently being fetched. This PC is used to index the stride prediction table, and a predicted address is generated by taking the address found at that entry and adding the stride to it. We also experimented with adding a multiple of the stride in order to fetch data several loop iterations ahead of the current PC.

In order to compare the effectiveness of stride prediction to stream buffers, a 64 entry stride prediction table was chosen, as being roughly equivalent in area to the 32 cache lines of the stream buffers. Each cache line was 16 bytes long, whereas each entry in the stride prediction has three pieces of information: an address, a stride, and a valid bit. This information can be packed into about 6 bytes, but the bit density will be lower for the table than for the cache line.

The results are shown in figure 3. The different configurations "SP - 1", "SP - 2", "SP - 4", and "SP - 8" correspond to using multiples of 1, 2, 4, and 8 times the stride, respectively. These multiples were chosen to minimize the hardware required to compute the predicted address. The performance was generally worse for a system with stride prediction than for the base system. The reason for this is that the predicted addresses compete with regular memory operations for scarce cache bandwidth. When the stride multiple is one, the predicted address is the same as the actual address, which will be computed by the processor within a few cycles of when it is fetched. In this case, the performance penalty due to stealing cache cycles to handle the predictions outweighs the benefit of fetching the data a few cycles early.

As the stride multiple increases, the benefit from providing the addresses early is greater, and it starts to compensate for the lost cache access cycles. The results for the level two cache (not shown) are similar.

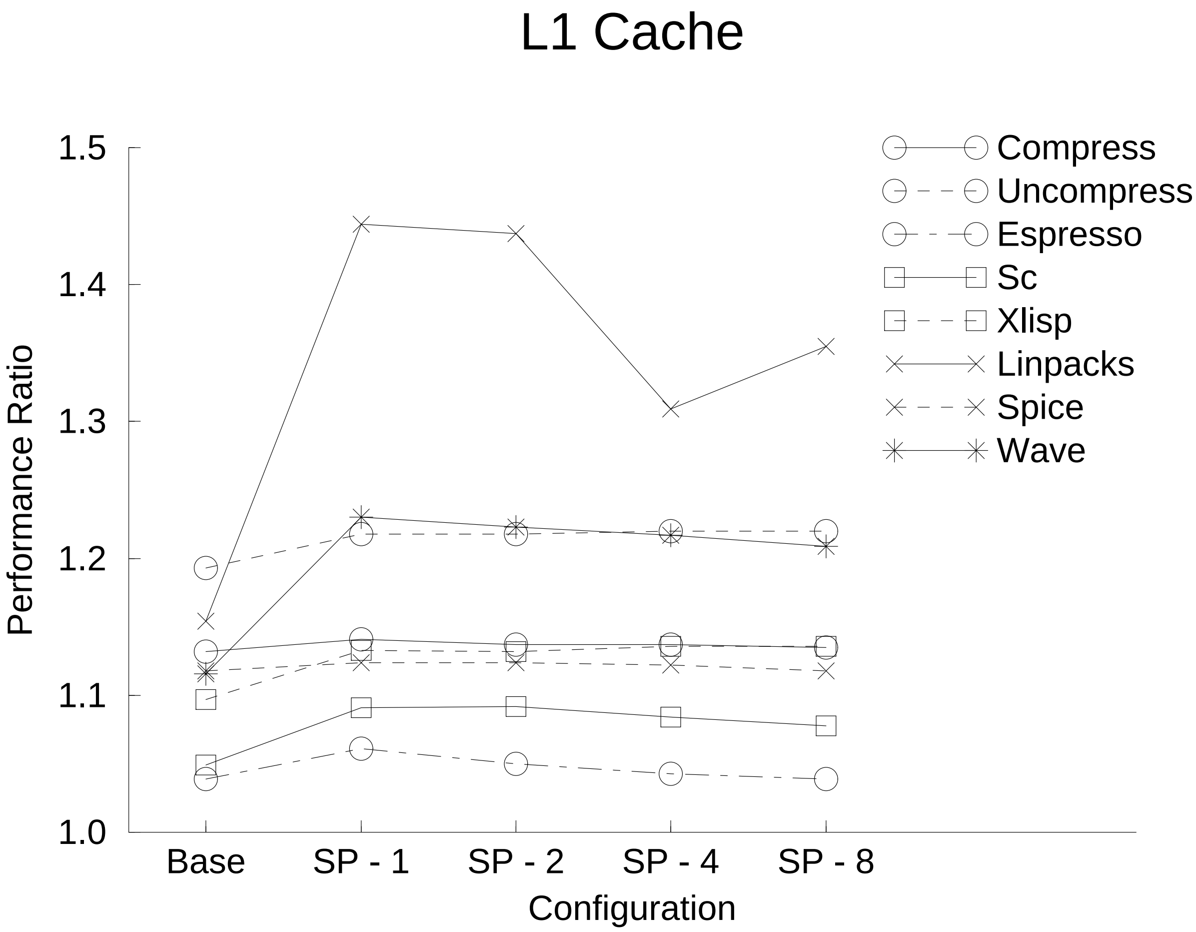

Figure 3: Stride Prediction - L1 Cache

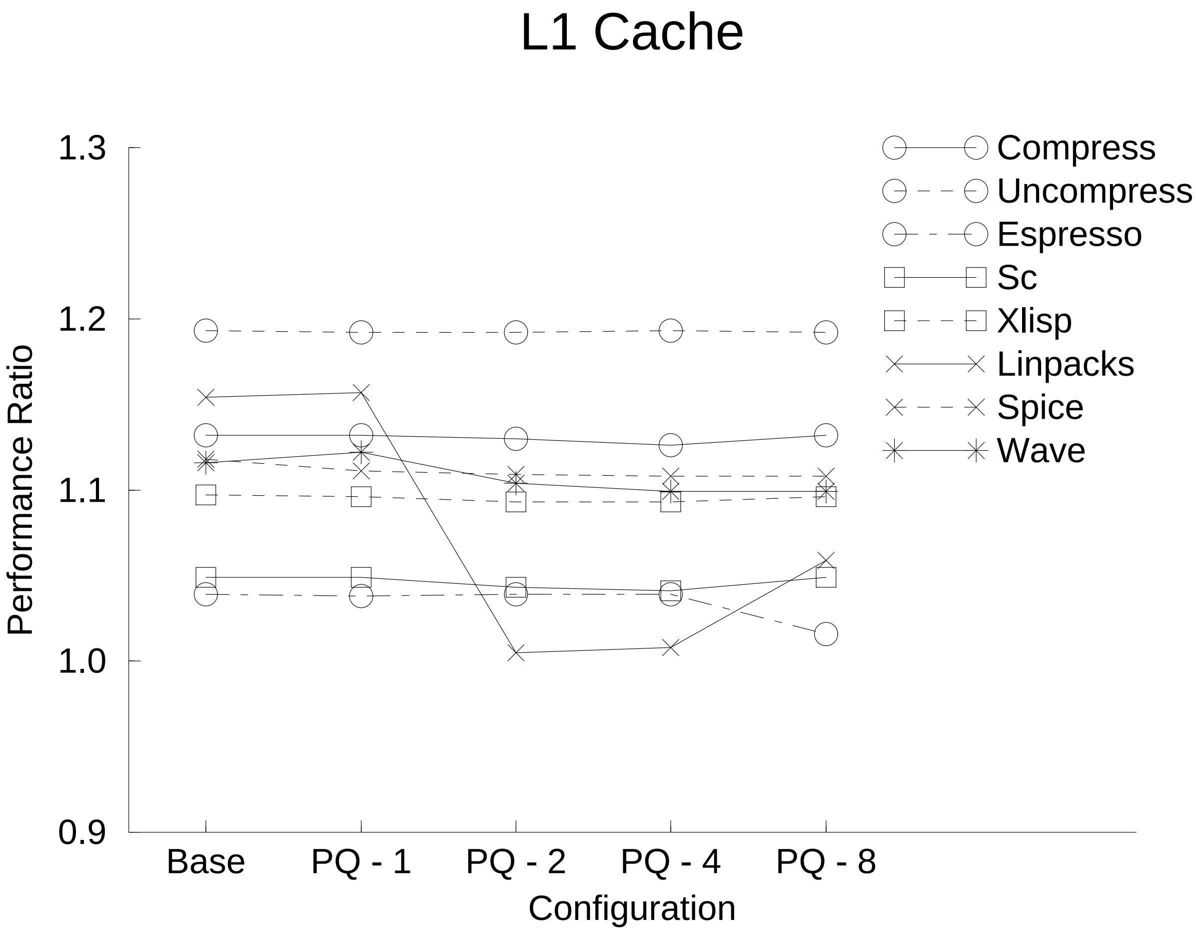

Figure 4: Prediction Queue - L1 Cache

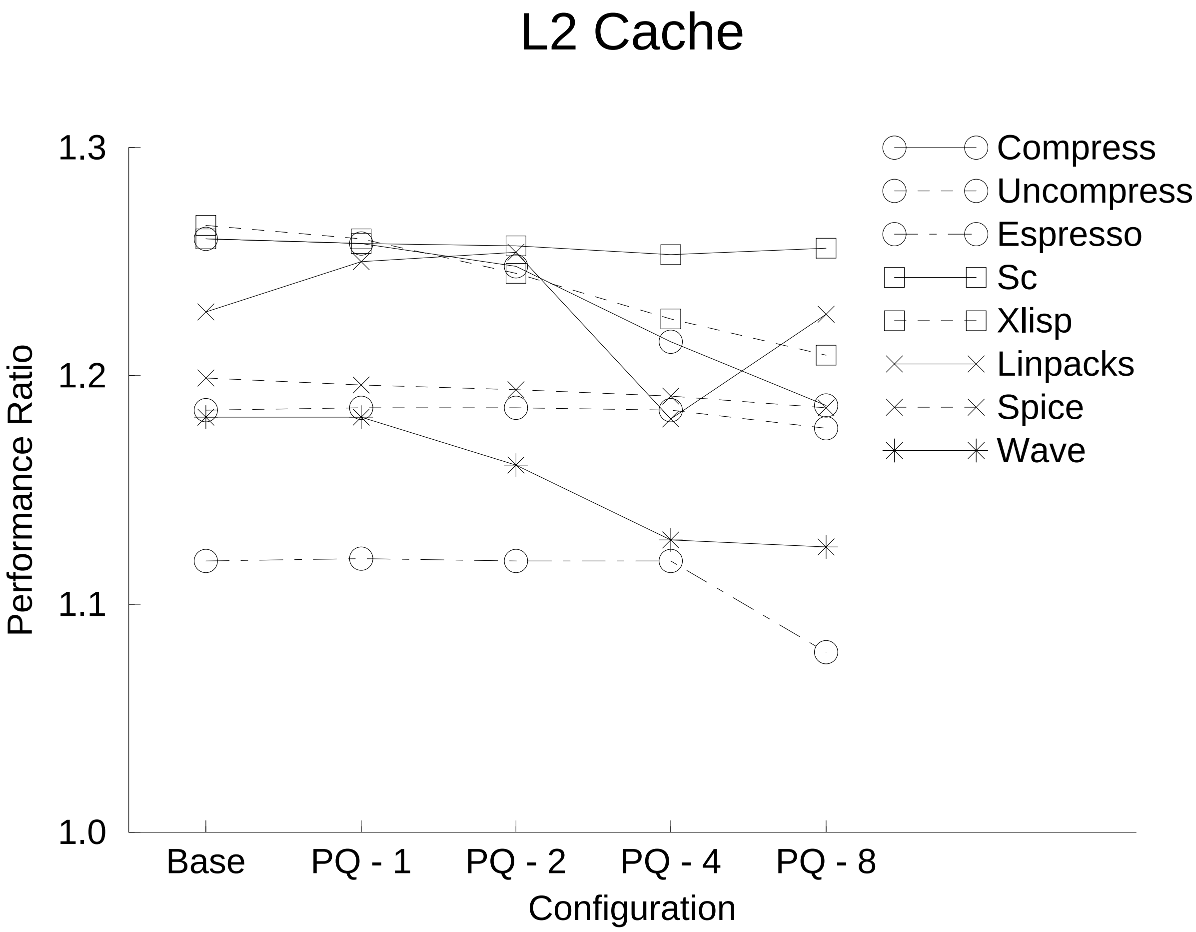

Figure 5: Prediction Queue - L2 Cache

Benchmark	*Prediction ratio*	*Prefetch ratio*
Compress	5.36%	0.18%
Uncompress	7.01%	1.48%
Espresso	24.69%	1.16%
Sc	15.05%	1.54%
Xlisp	12.97%	0.95%
Linpacks	77.43%	15.24%
Spice	7.56%	1.08%
Wave	50.12%	6.60%

Table 3: Stride Prediction Effectiveness

In order to deal with this problem, Chen and Baer[CB95] recommend dual porting the cache tags. Then the prediction mechanism can determine if the predicted value is present in the cache, without stealing cycles from regular memory operations. If the data isn't present in the cache, a prefetch is issued to fetch the data into the cache. We simulated this case, and the results are shown in figures 4 and 5. We refer to this case as the prediction queue, because a separate queue, distinct from the load/store buffer, was used to handle the predicted addresses.

In the L1 cache case, we see that stride prediction with a stride multiple of two effectively hides the memory latency of the Linpacks benchmark, and slightly improves the performance of the Wave benchmark. These two benchmarks have a regular access pattern which is missing from the other benchmarks, as can be seen in table 3. This table shows the ratio of valid stride predictions to the total number of load instructions fetched ("Prediction ratio"). The column labeled "Prefetch ratio" shows the number of predictions that generated a cache line prefetch to the L1 cache divided by the total number of loads. By either measure, Linpacks and Wave score more highly than the other benchmarks.

For a level two cache, stride prediction again improves the performance of Linpacks and Wave, although in this case the best performance is achieved with a stride multiple of four. Linpacks performance actually declines with stride multiples of one and two. The remaining source of performance degradation in the prediction queue scheme is that prefetches can knock data out of the cache that is still needed. Only when the stride multiple is increased to four does the benefit from prefetching outweigh the performance impact of this pre-empted data.

Another interesting result is that some of the benchmarks with irregular access patterns benefit from stride prediction in the level two cache case. This results from the increased locality of level two cache misses, as we observed in section 3.1.

3.3 Victim caches

Finally, we consider victim caches, proposed by Jouppi in [Jou90]. A victim cache is a small, fully associative cache that holds data recently pre-empted from the main cache. We used a cache size of 32 lines, the same number of cache lines as were used for stream buffers. The results are shown in figures 6 and 7.

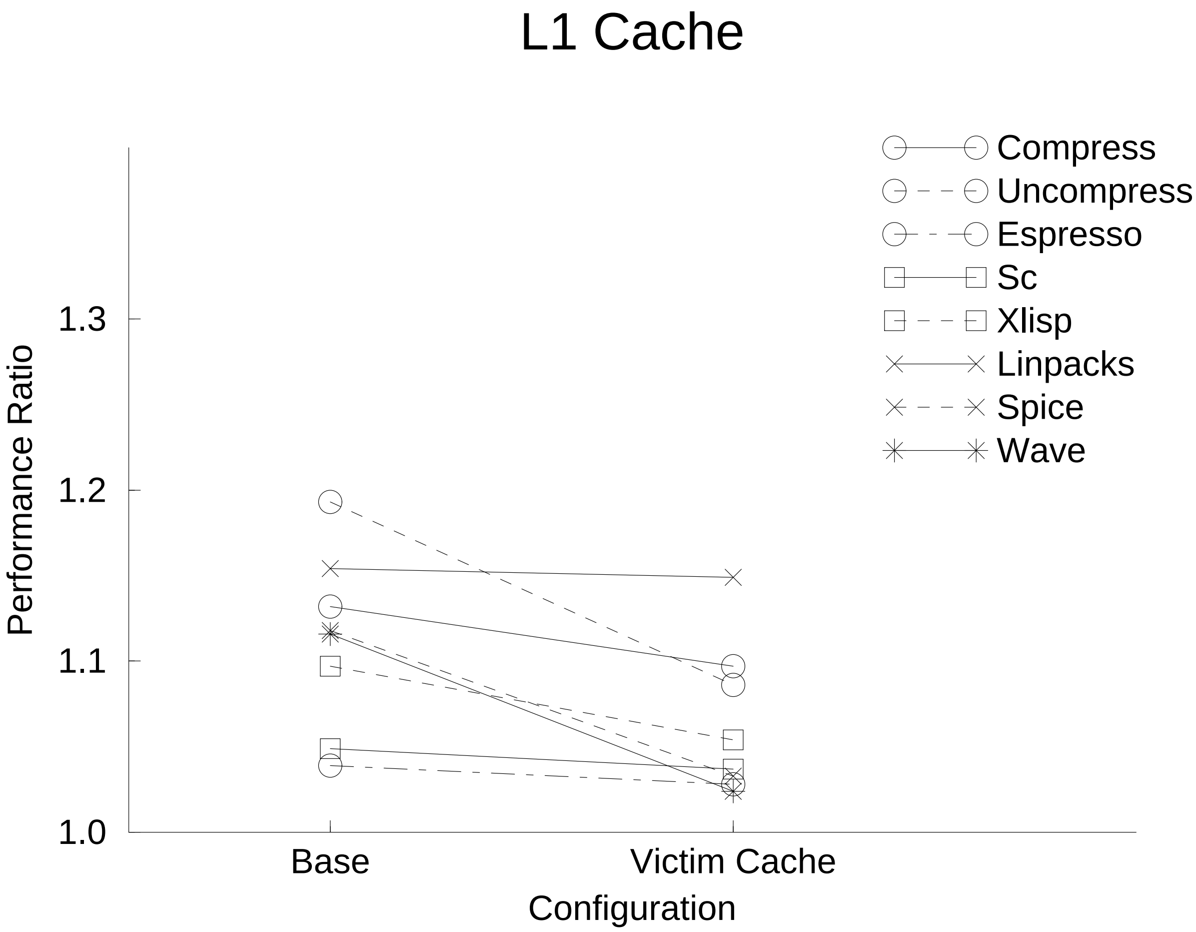

Figure 6: Victim Cache - L1 Cache

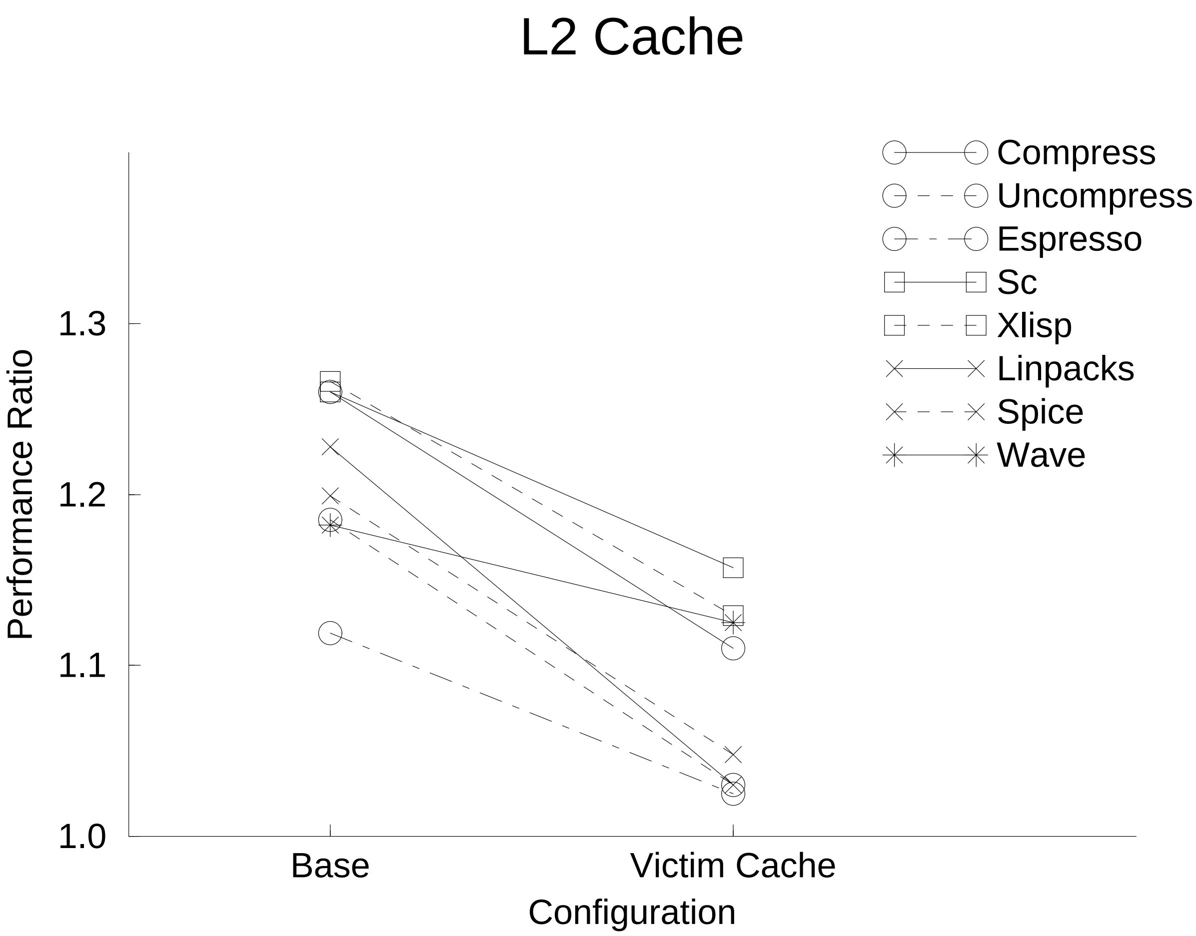

Figure 7: Victim Cache - L2 Cache

Benchmark	*Victim cache on L1 cache*	*Victim cache on L2 cache*
Compress	61.60%	71.10%
Uncompress	39.82%	20.65%
Espresso	65.41%	22.03%
Sc	73.97%	54.57%
Xlisp	48.35%	55.52%
Linpacks	96.77%	15.11%
Spice	36.59%	21.50%
Wave	26.56%	69.46%

Table 4: Victim Cache Miss Rates

By comparing these graphs with the results in sections 3.1 and 3.2, we see that the victim cache is effective in improving performance for more of the benchmarks than either stream buffers or stride prediction. In the level two cache case, the addition of a victim cache significantly improved the performance of all the benchmarks.

Table 4 demonstrates one reason for the effectiveness of victim caches. The miss rates are better (lower) for the victim cache than for the stream buffer (see table 2) for all of the benchmarks for both cache sizes, with the sole exception of Linpacks on the level one cache.

A second reason why victim caches perform well is that victim caches reduce the demands of the processor on the memory bus. The victim cache intercepts and services requests that would otherwise be sent to the next level in the memory hierarchy, which decreases the amount of bus traffic. Also victim caches never fetch data from memory; they copy the data from the cache before it is overwritten.

Both stride prediction and stream buffers, on the other hand, always increase the load on the memory bus. They generate prefetches in an attempt to get data into the cache before it is needed. Whenever they are successful, the memory bus traffic is unchanged, and whenever they are unsuccessful (e.g. fetching data that isn't referenced), the memory bus traffic increases.

4 Discussion

4.1 Software techniques

This study has compared various hardware techniques for tolerating memory latency. There has also been work on software techniques for for tolerating memory latency, such as prefetching[MLG92, TE95] and balanced scheduling[KE93]. Hardware and software techniques are compared in [BP92], [CCMH91], and [CB94]. In general, it appears that software and hardware techniques are complementary. Compile time optimizations for memory latency tolerance can include large scale code motion, such as loop transformations, that are beyond the scope of hardware techniques. On the other hand, hardware techniques have access to dynamic information such as memory addresses and recent reference history that is unavailable to the compiler.

4.2 Stream buffers with filters

Palacharla and Kessler[PK94] propose that filters be associated with stream buffers to reduce the amount of bus bandwidth that is consumed by the stream buffer. This proposal resembles the stride prediction mechanism described in section 3.2. A history of cache misses is kept, and a new stream is started only when a series of sequential misses is observed. They also propose extending this mechanism to allow prefetching of strided data. The comparative study of [FJ94] concludes that the main benefit of filters is to reduce the bus bandwidth consumed by stream buffers, and that performance declined slightly when filters were introduced.

In this study, the use of filters might have improved the performance of stream buffers, because bus bandwidth is a significant performance issue, as noted in section 3.1. To see if this would affect the relative merits of the different latency tolerating techniques, the benchmarks were re-run with an infinite bandwidth memory subsystem. Even after eliminating effects due to bus bandwidth, the victim cache still outperformed the other two techniques.

4.3 Cache associativity

The effectiveness of the victim cache is due in part to the low degree of associativity of the main cache. That is, if the main cache were fully associative, then the addition of a small number of additional fully associative cache lines would be unlikely to have a significant affect on performance. For all of these simulations, the main cache was taken to be four way set associative. Since the victim cache was effective for this degree of associativity, making the main cache eight way set associative might also be an effective way to increase performance.

Taking this argument in the other direction, systems with a direct mapped cache should benefit even more from the addition of a victim cache.

4.4 Future directions

In comparing the effectiveness of stream buffers, stride prediction, and victim caches for the different benchmarks, we see that stream buffers and stride prediction were both more effective for Linpacks than for other benchmarks. For the victim cache (when added onto the level one cache), the Linpacks benchmark had the worst miss rate of all the benchmarks. This suggests that the victim cache and either stream buffers or stride prediction might work well in cooperation.

One possible extension of this work is to consider a combined stream buffer/victim cache. A short history of cache misses would be kept, as in [PK94], and when a stream of consecutive or strided misses is detected, some of the cache lines could be stolen from the victim cache to serve as a stream buffer. We are currently investigating this possibility.

5 Conclusion

In this study, we have compared three hardware techniques, to see which was more effective at helping a dynamically scheduled processor tolerate cache misses. These techniques, stride prediction, stream buffers, and victim caches, have been previously shown to be effective in the case of statically scheduled processors. We have looked at the question of whether these techniques are still effective for dynamically scheduled processors, and also which of these techniques provides the greatest benefit for a fixed investment in processor die area.

All three techniques continue to be effective for certain classes of applications. Stream buffers and stride prediction offer a benefit for applications with a regular access pattern, especially when the data accessed doesn't fit in the cache (as in the case of Linpacks on the level one cache). Victim caches, however, are more effective in improving performance for most of the benchmarks used in this study. Because victim caches and stream buffers or stride prediction are effective for different benchmarks, combinations of these techniques are a promising area for future research.

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.

[BP92] M. Butler and Y. Patt. An investigation of the performance of various dynamic scheduling techniques. In *Proc. of the 25th International Symposium on Microarchitecture*, pages 1–9, December 1992.

[CB91] T-F. Chen and J-L. Baer. An effective on-chip preloading scheme to reduce data access penalty. In *Supercomputing 91*, pages 176–186, November 1991.

[CB94] T-F. Chen and J-L. Baer. A performance study of software and hardware data prefetching schemes. In *Proc. of the 21st International Symposium on Computer Architecture*, pages 223–32, April 1994.

[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.

[FJ94] K. I. Farkas and N. P. Jouppi. Complexity/performance tradeoffs with non-blocking loads. In *Proc. of the 21st International Symposium on Computer Architecture*, pages 211–22, April 1994.

[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.

[KE93] D. R. Kerns and S. J. Eggers. Balanced scheduling: Instruction scheduling when memory latency is uncertain. In *ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 278–89, June 1993.

[KF95] P. Chow K.I. Farkas, N.P. Jouppi. How useful are non-blocking loads, stream buffers and speculative execution in multiple issue processors? In *First IEEE Symposium on High-Performance Computer Architecture*, pages 78–89, January 1995.

[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.

[MLG92] T. Mowry, M. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *SIGPLAN Notices*, pages 62–73, September 1992.

[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.

[TE95] D. Tullsen and S. Eggers. Effective cache prefetching on bus-based multiprocessors. *ACM Transactions on Computer Systems*, 13:57–88, February 1995.