An Automated Method for Software Controlled Cache Prefetching

Daniel F. Zucker*, Ruby B. Lee, and Michael J. Flynn
Computer Systems Laboratory
Department of Electrical Engineering
Stanford University
Stanford, CA 94305

Email: {zucker,rblee,flynn}@umunhum.stanford.edu

Abstract— **As the gap between cycle time and main memory access time increases, memory system performance becomes increasingly important. The trend to higher instruction level parallelism with superscalar processors puts even higher demands on the memory system. Prefetching is a common strategy to tolerate this increased memory latency. This paper presents a software only technique to prefetch data to the CPU cache before it is needed in order combat this problem.**

The software prefetching technique presented is motivated by emulation of a hardware stride prediction table (SPT). Performance similar, and in some cases superior, to the hardware based technique is achieved with no additional hardware costs. In the first step, a simulation of the hardware SPT is conducted to identify where useful prefetches are best added. In the next step, software prefetches are added to the executable code. The technique is automated and could be implemented by a compiler as a two phase optimization of a profile step followed by an optimization step.

Data is presented for both SPEC95 and multimedia benchmarks. In the best case, a performance improvement of 2.78X is observed over the same code with no prefetching at no extra hardware costs.

Keywords— **prefetching, cache, SPEC95, software prefetching, stride based prefetching, mpeg**

I. Introduction

As the gap between cycle time and main memory access time increases, memory system performance becomes increasingly important. The trend to higher parallelism with superscalar processors puts even higher demands on the memory system. Cache prefetching is a technique used to hide the latency of a main memory access by predicting in advance what data will be need.

Both hardware and software cache prefetching techniques have been proposed. Hardware prefetching techniques typically perform prefetching dynamically based on run time information, while software prefetching techniques are based on static analysis done at compile time. Software prefetching often requires additional information in the form of hints added by the programmer. This paper presents an automated software prefetching technique based on emulation of a hardware prefetching system. Benefits of hardware prefetching are obtained without the added cost of dedicated hardware. Performance similar, and in some cases superior, to the hardware based technique is achieved with no additional hardware costs.

In the first step, a simulation of the hardware stride prediction table (SPT) is conducted to identify where useful prefetches are best added. In the next step, software prefetches are added to the executable code. The technique is automated and could be implemented by a compiler as a two phase optimization of a profile step followed by an optimization step. No special programmer input is required.

In this paper data is presented for two SPEC95 benchmarks and for an MPEG decode application across a range of cache sizes and both direct mapped and 4 way cache associativity. Compared to the same cache configuration with no prefetching, execution time is reduced to 0.36 of unenhanced execution time in the best case. In the worst case, execution time is degraded such that execution is slower than the unenhanced code with no prefetching.

II. Related Work

A number of techniques exist for cache prefetching. The idea of prefetching is to predict data access needs in advance so that a specific piece of data is loaded from the main memory before it is actually needed by the application.

The earliest hardware prefetching work was reported by Smith [1] who proposed a one-block-lookahead (OBL) scheme for prefetching cache lines. That is, when a demand miss brings block i into the cache, block

*Presently at TriStrata Security Inc., 3 Lagoon Drive, Suite 110, Redwood Shores, CA 94065

i+1 is also prefetched. Jouppi [2] expanded this idea with his proposal for stream buffers. In this scheme, a miss that causes block i to be brought into the cache also causes prefetching of blocks i+1, i+2, ..., i+n into a separate stream buffer. Jouppi also recognized the need for multi-way stream buffers so that multiple active streams can be maintained for a given cache. He reported significant miss rate reduction. Palacharla and Kessler [3] proposed several enhancements to the stream buffer. They have developed both a filtering scheme to limit the number of unnecessary prefetches, and a method for allowing variable length strides in prefetching stream data.

Another hardware approach to prefetching differs from the stream buffer in that data is prefetched directly to the main cache. In addition, some form of external table is used to keep track of past memory operations and predict future requirements for prefetching. This has the advantage of efficiently handling variable length striding, that is data accesses that linearly traverse a data set by striding through in non-unit steps. Fu and Patel [4] proposed utilizing stride information available in vector processor instructions to prefetch relevant data. They later [5] expanded the application to scalar processors by use of a cache-like look-up table called the stride prediction table (SPT).

Chen and Baer [6] have proposed a similar structure called the reference prediction table. Their scheme additionally includes state bits, so that state information can be maintained concerning the character of each memory operation. This is then used to limit unnecessary prefetching. Further analysis of this scheme [7] investigate the timing issues of prefetching by use of a cycle-by-cycle processor simulation. Sklenar [8] presents a third variation on the same theme of the use of an external table to predict future memory references.

A number of techniques also exist to do software prefetching. Porterfield [9] proposed a technique for prefetching certain types of array data. Mowry, et al [10] proposed an early practical software prefetch scheme based on information obtained at compile time. While software prefetching clearly has a cost advantage, it does introduce additional overhead to the application. Extra cycles must be spent to execute the prefetch instruction, and the code expansion that is often required from loop unrolling may result in negative side effects such as increased register usage. [11] compares Mowry's software prefetching scheme to their hardware prefetching scheme. They determine that while the software approach can use compile time information to perform more complex analysis, hardware prefetching has the advantage of dynamic information. Furthermore, they determine that while both methods improve the miss rate, the overhead of adding software prefetch instructions is significant. Santhanam et al. [12] present a sophisticated compile time algorithm to insert software prefetch instructions for the HP PA-8000. The HP compiler, furthermore, has the capability to add prefetch instructions based on execution profile data. This is an improvement over Mowry's algorithm. Speedups of up to 100% are reported for SPECfp95.

III. Methodology

The software prefetch scheme presented here is based on emulating the hardware SPT by adding software prefetch instructions. Data on prefetch usefulness is obtained from profiling simulated SPT execution. This technique is unique in that it relies completely on profile information to do prefetching. This low complexity means that the technique can easily add prefetch instructions to an existing executable or be used as a profile based optimization for an existing compiler.

A. Prefetch Methodology

The software prefetching technique works by gathering execution profile information from a simulation of the hardware SPT. A prefetch hint file is generated based on tracing which instructions caused the most useful prefetches in the hardware SPT simulation. The hint file is then used to insert software prefetch instructions. To do this automatically in a compiler would be possible by first profiling, then inserting prefetch instructions into the code in two separate steps.

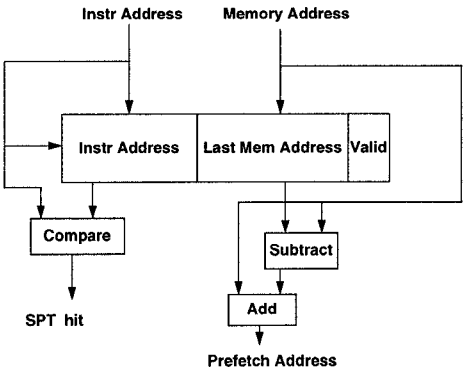

Fig. 1. Stride prediction table architecture.

The profile step simulates a hardware SPT prefetching into a series stream cache. The SPT architecture is shown in figure 1 and the series stream cache architecture is shown in figure 2. These architectures are described in detail in [13]. The SPT is used to determine

what data will be needed by a given instruction based on what data it has accessed previously. An attempt is made to calculate a stride value as if the memory access is made in a regular stride through the data. A table, indexed by instruction address, is maintained for all memory operations executed and holds the address of the last memory address accessed. When a memory instruction is executed, its address is compared to the instruction addresses stored in the SPT. When the instruction does not match an instruction stored in the SPT, an SPT miss occurs. On an SPT miss the new entry, composed of the instruction address and data memory address, is added to the SPT replacing the least recently used entry (LRU). When a memory access is made by an instruction already contained in the stride prediction table, an SPT hit occurs. The current memory access address is subtracted from the previously stored last memory address to calculate a data stride value. If this value is non-zero, a prefetch is issued. The prefetch address is calculated by adding the stride value and the current memory address. The data is prefetched into the series stream cache.

The series stream cache lies on the refill path to the main cache. The prefetch data is prefetched not directly to the main cache, but to the series stream cache. On a main cache miss, the series stream cache is searched for the miss data before it is retrieved from main memory. The series stream cache acts a filter to the prefetch data so that only data that is actually needed in the main cache is fetched to the main cache. This prevents excessive amounts of unnecessary prefetched data from swamping the main cache. The stream cache simulated is fully associative.

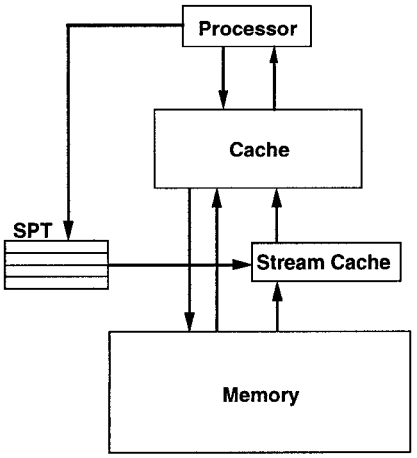

Fig. 2. Series stream cache architecture.

By tracking which instructions caused which cache lines to be prefetched, and then keeping track of which prefetch data is actually used by the application, we determine which instructions were useful in prefetch data that is subsequently used by the application. Furthermore, by keeping track of the stride value that is used to prefetch the data, we can determine what the best value is to use for a static stride prediction.

After obtaining data describing which prefetches are useful, we selectively insert software prefetch instructions into the executable code using a static stride prediction. The data we have obtained thus far gives a list of memory instructions that caused useful prefetches as well as a static stride value associated with each instruction. At this point the best strategy might seem to add a software prefetch instruction for every memory operation that caused a useful prefetch instruction. This, however, is not the case.

Adding additional prefetch instructions increases the number of instructions that must be executed in the benchmark. A trade-off is made between the number of new cycles that must be executed in prefetch instructions and the number of cycles saved from eliminating cache misses. In [14] it was determined that approximately 15% of all instructions causing useful prefetches cause over 90% of useful prefetches. It was also determined that inserting instructions to cause only 90% of useful prefetches resulted in the best trade-off between increased overhead and decreased cache misses. The benchmarks in this paper are therefore executed adding instructions to cause only 90% of useful prefetches.

The results in this paper are generated by simulating a discrete software prefetch instruction. The particular prefetch instruction could be implemented in a variety of ways. For these simulations, an atomic prefetch-by-stride instruction is assumed. The instruction prefetches directly into the main cache. This prefetch-by-stride instruction is invoked with an immediate stride value. The last executed load or store address is added to the stride value and a prefetch from this new address is initiated. The stride value is available at compile time and is derived from the hint file generated at the profiling step.

B. Simulation Methodology

Trace driven simulations are used to model memory behavior in order to determine performance results. SPEC95 benchmarks were executed on a DEC Alpha. Application code was compiled using the standard DEC version 4.0 C compiler. Optimization was set according to the SPEC guidelines. The executable was then instrumented using ATOM [15] so that library functions were called for every memory access. The MPEG code was executed on an HP 9000/725 workstation and compiled with the HP version A.09.75 C Compiler. The source code was similarly instrumented using RYOLS [16], an instruction instrumentation tool

written for the PA-RISC architecture.

To save both execution time and disk space, discrete traces are not written to disk files. Instead, the cache simulator is executed concurrently with the instrumented executable so that address references are dynamically simulated. Because new traces are dynamically generated every execution, variables returned from system calls may cause slightly different traces at each run and may result in some run-to-run variation. The simulator provides data over a wide range of data cache sizes and associativities. A write allocate policy is assumed. A line size of 16 bytes was chosen for all simulations. This line size was chosen so as to better expose the potential benefits of prefetching. Because only a single process was simulated for each cache configuration, it is expected that the performance for the cache sizes reported corresponds to a larger cache size in a real system. Instruction memory accesses are not modeled.

C. Performance Metrics

Fraction of misses eliminated is one metric reported. This metric judges the performance of a given prefetch scheme independent of the particular cache implementation. A perfect prefetching scheme would eliminate all memory misses. This would have a fraction of misses eliminated value of 1.0 since all misses have been eliminated. Similarly, an architecture that eliminates half of all the misses of a cache with similar size and associativity would have a fraction of misses eliminated value of 0.5.

In this way, performance improvement can be judged independently of other cache design considerations such as main cache size and associativity. The size of the main cache will have a dominating effect on miss rate, so that if results were simply compared in terms of absolute miss rates, the variation due to cache size would tend to mask out the variation due to prefetching scheme. Furthermore, performance can also be judged independently of memory implementation parameters such as time to access main memory. If this were not the case, varying memory parameters such as cycles to fill a main cache line could have a significant impact on results.

In deriving fraction of misses eliminated, a memory access is counted as a hit as long as a prefetch to that address has been issued. This means data in the process of returning from memory is counted as a hit. This is done to compare prefetching performance of the schemes under the best conditions. Counting incomplete prefetches as misses, furthermore, has little effect on the resulting data.

Results are also reported for execution time in numbers of cycles. For these results, an aggressive memory-limited processor model is assumed. An n-way superscalar processor is assumed such that there are sufficient resources to perform any non-memory operation in a single cycle. Therefore, only memory operations are counted as instructions executed. The benchmark execution time is composed only of loads, stores, prefetches, and time spent in the memory system due to cache misses. Relative execution time reports execution time compared to an identical cache configuration with no prefetching. A relative execution time of less than one indicates a performance improvement from prefetching.

A constant memory access time of 50 cycles for cache misses is assumed. Furthermore, a fully interleaved memory is assumed such that multiple outstanding prefetch requests are allowed. Demand misses block program execution, while prefetch misses are non-blocking. For software prefetching, execution time incorporates the cost of the additional prefetch instructions executed.

These assumptions are made to show the effect of prefetching on a processor unlimited by other resource constraints. In [14], the effect of modifying these assumptions is investigated. Execution time is calculated considering the cost for partially completed prefetches. Furthermore, an instruction mix in which memory operations make up only a fraction of total instructions is considered. Finally, different bus models in which only a fixed number of simultaneous outstanding memory requests are allowed are also considered.

The data in this case shows the same general trends, but both the benefits and costs of prefetching appear less pronounced. The overhead of prefetching is less of a penalty since additional prefetch instructions add a smaller percentage of additional overhead, while the benefits from prefetching are also less since the time spent in the memory system is a smaller percentage of total execution time.

D. Memory Bandwidth

For the purposes of this study, memory bandwidth is assumed to be large enough such that it is not a limiting factor on performance. This assumption is made to study the effects of differing prefetch strategies independent of memory bus architectures. It is recognized that this assumption may not be valid in terms of today's architectures. However, the trend for wider bandwidth to memory indicates that this may not be a problem in the future.

The software prefetching proposed in this paper eliminates a significant number of unneeded prefetches compared to the hardware prefetching emulated. [10] and [12] describe additional analytic techniques to limit the number of prefetches issued.

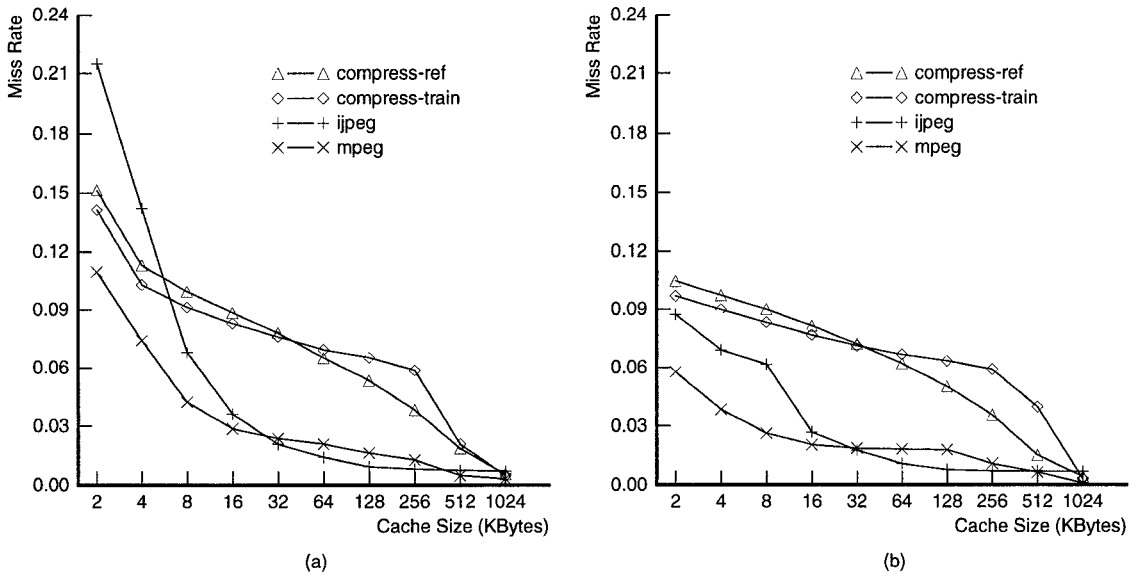

Fig. 3. Base miss rates. (a) is for a direct mapped cache, and (b) is for a 4 way associative cache.

IV. Data

In the following sections, data is presented for two SPEC95 benchmarks, compress and ijpeg, and for an implementation of MPEG decode. In all cases data is shown for three cases: the hardware case, the training data set, and the reference data set. The hardware curve is labeled hw and reports the simulated performance of the hardware SPT and series stream cache used to generate the hint file. The hardware performance is determined using the training data set. The training curve is labeled train and reports the simulated performance for software prefetching using the SPEC95 training data. Finally, the reference curve is labeled ref and reports the SPEC95 reference performance using the reference data set. Since the MPEG application is not part of the SPEC95 benchmark suite, two input data sets are arbitrarily chosen as reference and training data sets. A fourth data set is shown for compress and is described in that section. Base miss rates for these applications with no prefetching is presented in figure 3.

A. Ijpeg

Performance data for ijpeg is shown in figure 4. Figures 4a and 4c show that hardware results in the greatest fraction of misses eliminated for most cache sizes. This is true for the applications to follow as well. This metric, however, does not take into account the additional overhead of executing software prefetch instructions. In the hardware case it is assumed that prefetch instructions are executed concurrently with the application code so there is no added overhead charged in performing as many prefetches as possible. In the software case, however, it is assumed that every software prefetch must be executed as a discrete instruction and is added to the execution time of the application. It is for this reason only instructions causing 90% of effective prefetches are added to the code. If 100% of the effective prefetches were added, the fraction of misses eliminated curve would match the hardware fraction of misses eliminated curve more closely, but the relative execution time would be hurt by the greatly increased overhead. Relative execution time would not be as good as the data reported in figures 4b and 4d. Relative execution time, then, is the proper metric for which to optimize.

For both the direct mapped and 4 way associative caches, the best relative execution time is achieved at cache sizes of 16KB and 32KB at less than 80% of base execution time. The curve stays relatively flat to the left of this region for the 4 way associative cache, but creeps upward for the very small caches in the the direct mapped case. For both cases, performance is degraded compared to the base case for cache sizes of 256KB and larger. At caches of this size, the base cache architecture captures the working set of the application well and additional prefetch instructions only add to execution overhead with no performance benefit.

B. Compress

Performance data for compress is presented in figure 5. For compress the reference data performance is significantly different from the training data. This is because of the size difference between the two data

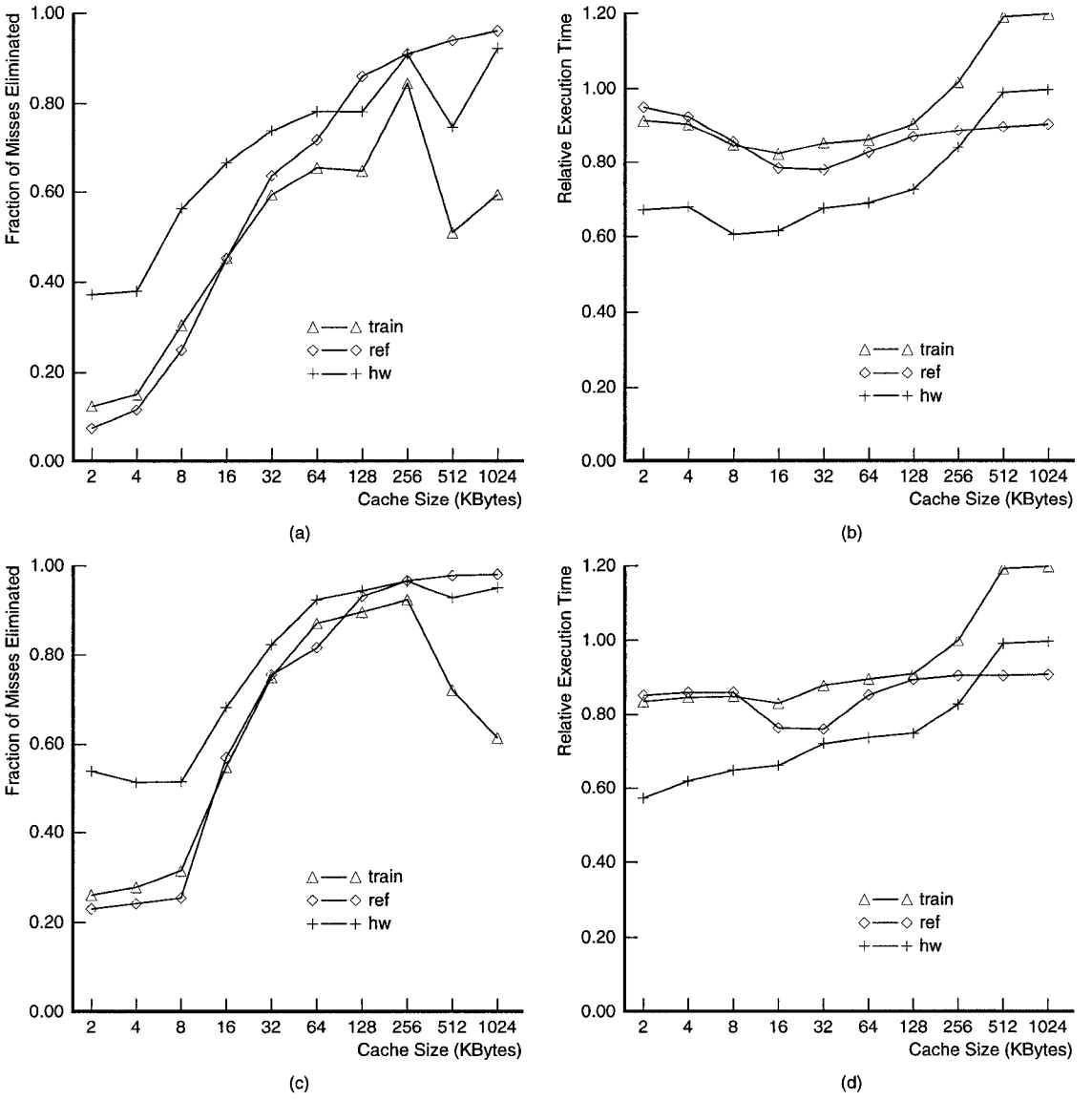

Fig. 4. Performance data for ijpeg. (a) is fraction of misses eliminated for a direct mapped cache, (b) is relative execution time for a direct mapped cache, (c) is fraction of misses eliminated for a 4 way associative cache, and (d) is relative execution time for a 4 way associative cache.

sets and is explained with the fourth performance curve labeled small-ref.

The compress application is really composed of two parts. In the first part, the data to be compressed is generated at random from a given seed value. In the second part the data is actually compressed. The amount of data to be generated is determined by the input data set. When a small amount of data is generated, the training data set for example, the data generation part is the dominant part of the program. The data generation is a highly regular algorithm and stride prediction is very effective in predicting its data usage. In this case the overall performance is significantly improved by prefetching. For the reference data, however, the dominant part of the program is the data compression part. The data compression part is irregular and is not significantly improved by stride-based prefetching. The curve labeled small-ref illustrates this point. This data set is different from the training data, but generates a data file the same size as the training data set. The performance is similar to the training data set.

For the reference data, execution time is never reduced below approximately 0.9 of what it is without prefetching. On the other hand, performance is never degraded by prefetching. The relative execution time is

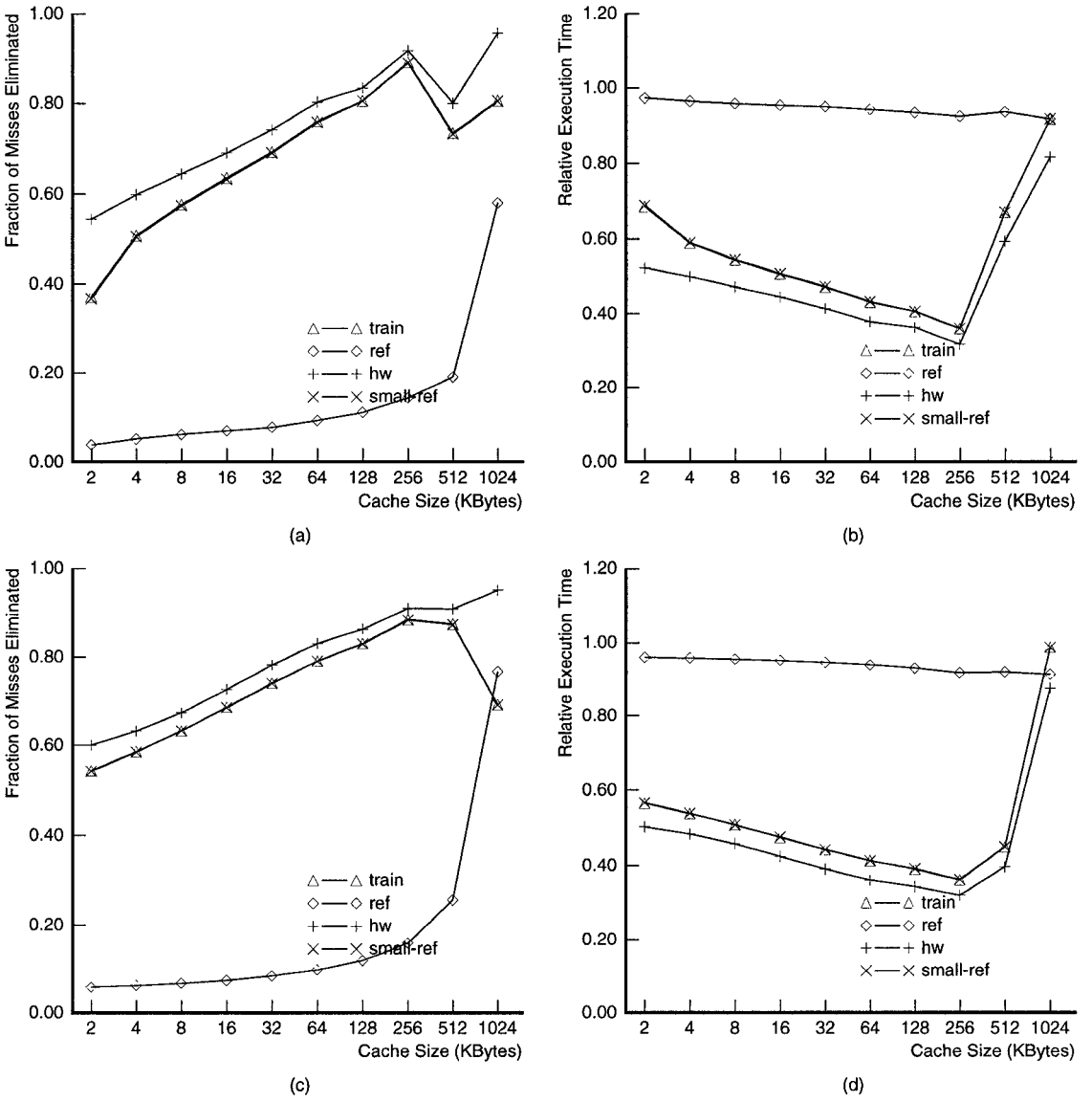

Fig. 5. Performance data for compress. (a) is fraction of misses eliminated for a direct mapped cache, (b) is relative execution time for a direct mapped cache, (c) is fraction of misses eliminated for a 4 way associative cache, and (d) is relative execution time for a 4 way associative cache.

approximately flat across the range of cache sizes. For the other cases, performance is significantly improved with a relative execution time minimum for software prefetching of 0.36 for the 256KB cache size. Again, performance is never degraded by applying software prefetching.

C. Mpeg

Finally, data for Berkeley's mpeg_play [17] is presented in figure 6. The movie easter.mpg is chosen as the training data, and the movie hula.mpg is chosen as the reference data. In the range from approximately 16KB to 256KB, relative execution time is approximately 0.70 of what it would be with no prefetching. A thorough investigation of software prefetching for MPEG benchmarks across a range of execution parameters is presented in [18].

V. Analysis

Aside from the compress reference data, the results show that the proposed software prefetching technique is effective in improving performance of certain SPEC95 and MPEG applications. Even for compress, although not greatly improved, performance was improved over no prefetching.

The compress reference data illustrates an impor-

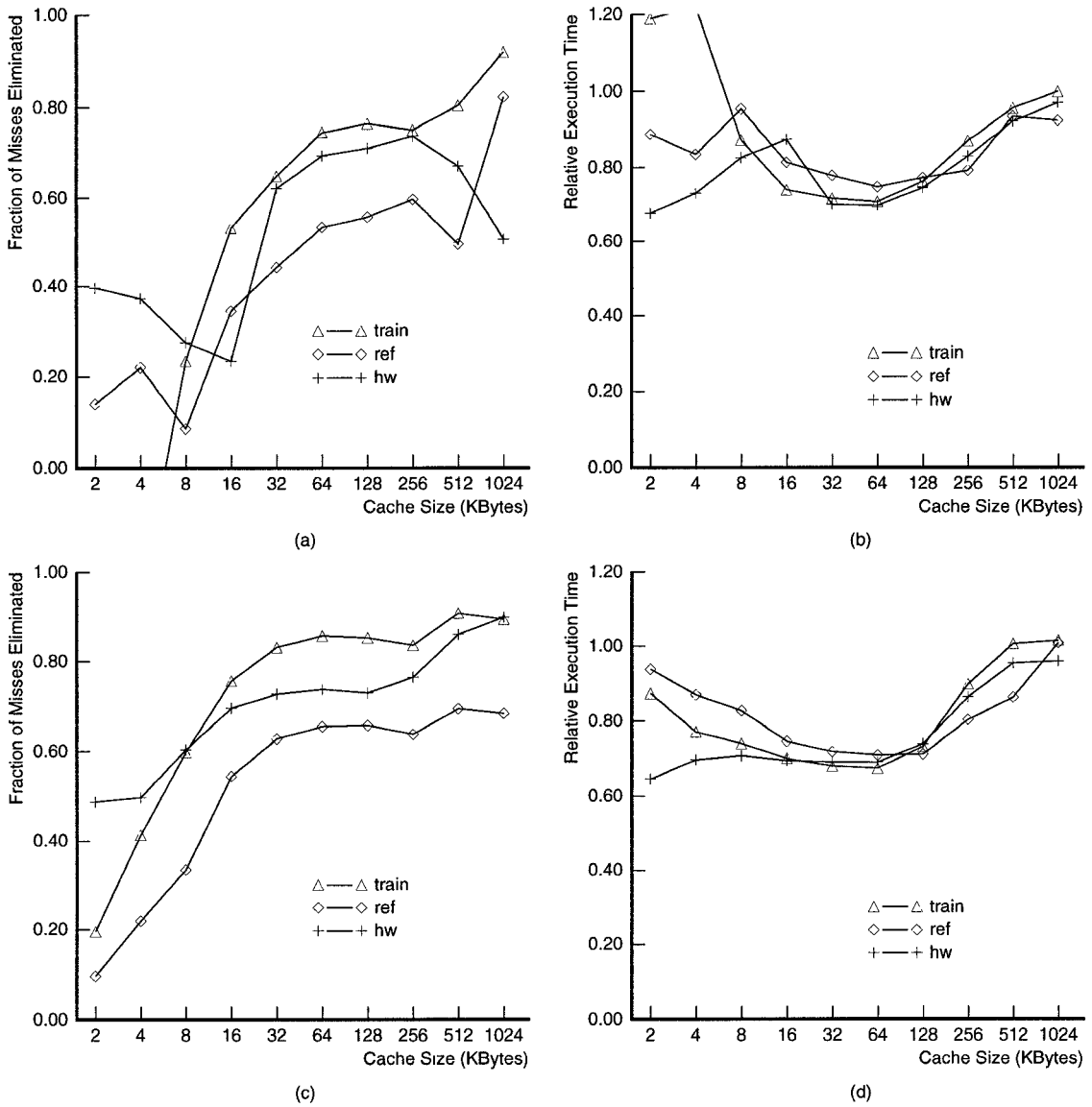

Fig. 6. Performance data for mpeg_play. (a) is fraction of misses eliminated for a direct mapped cache, (b) is relative execution time for a direct mapped cache, (c) is fraction of misses eliminated for a 4 way associative cache, and (d) is relative execution time for a 4 way associative cache.

tant point. Because the proposed software prefetching technique relies on emulating a hardware stride prediction scheme, the underlying stride prediction methodology must be effective for the software prefetching to perform well. Compress is an example of an application that is not amenable to stride prediction, while ijpeg and mpeg_play exhibit regular data access behavior that is amenable to stride prediction.

[19] and [20] investigate hardware stride prediction for a number of common benchmarks on a dynamically scheduled processor model. That work confirms similarly poor performance for compress. Based on this work we hypothesize that scientific applications such as linpack, wave, or fft will perform well using the software prefetching technique described here. These applications exhibit regular memory access behavior similar to ijpeg and mpeg_play.

The data also shows that prefetching is generally not effective at improving performance for large cache sizes. This is because when the main cache is large enough to capture the working set of the application, the total number of misses is fairly low. In this case, adding prefetch instructions increases the time that must be spent executing the application, since additional instructions must now be executed, but does little to reduce misses since there are so few misses to begin

with.

The data reported uses an execution model which makes simplifying assumptions that an access to main memory returns in a constant time of 50 cycles. [14] investigates execution models taking into account variable latencies based on bus contention as well as models for access times ranging from 10 to 100 cycles. It is found that bus contention has little effect on performance when demand loads and stores are given priority over prefetches. Increasing the memory access penalty beyond 50 cycles has the effect of improving the maximum benefit from prefetching while keeping the general shape of the performance curve relatively constant. In this case, the application spends a greater fraction of execution time in the memory system so that a greater performance benefit is achieved from prefetching. Similarly, decreasing memory access cost below 50 cycles tends to flatten the performance curves. In this case, the application spends less total time in the memory system from the same number of cache misses and prefetching performance is less beneficial.

Finally, the data indicates that for smaller caches a higher degree of associativity improves software prefetching effectiveness. Prefetching speculatively brings much data into the cache, and for small cache sizes this cause data to be prematurely eliminated from the cache before it becomes stale. A higher degree of cache associativity removes many of the mapping conflicts and prevents the prefetched data from knocking useful data out of the cache.

VI. Conclusions

In this paper we presented a software only prefetch technique that was shown to improve execution time by 278% in the best case. This scheme takes advantage of the dynamic nature of hardware based prefetching without incurring the cost of adding additional hardware. This is achieved by emulating a hardware based prefetching scheme with the addition of software prefetch instructions to existing executable code. Data was reported for two SPEC95 and an MPEG decode application.

The software prefetching technique presented is fully automated and can be added as an optimization to an existing compiler. The technique, furthermore, does not require any programmer input and can be used to add prefetching to existing binaries.

References

[1] Alan Jay Smith, "Cache memories," *ACM Computing Surveys*, vol. 14, pp. 473–530, September 1982.

[2] Norman P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Proc. of the 17th Annual International Symposium on Computer Architecture*, May 1990, pp. 364–373.

[3] Subbarao Palacharla and R.E. Kessler, "Evaluating stream buffers as a secondary cache replacement," in *Proc. of the 21st Annual International Symposium on Computer Architecture*, April 1994, pp. 24–33.

[4] John W. C. Fu and Janak H. Patel, "Data prefetching in multiprocessor vector cache memories," in *Proc. of the 18th Annual International Symposium on Computer Architecture*, May 1991, pp. 54–63.

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

[6] Jean-Loup Baer and Tien-Fu Chen, "An effective on-chip preloading scheme to reduce data access penalty," in *Proceedings of Supercomputing '91*, November 1991, pp. 176–186.

[7] Tien-Fu Chen and Jean-Loup Baer, "Effective hardware-based data prefetching for high-performance processors," *IEEE Transactions on Computers*, vol. 44, pp. 318–328, May 1995.

[8] Ivan Sklenar, "Prefetch unit for vector operations on scalar computers," *ACM Computer Architecture News*, vol. 20, pp. 31–37, September 1992.

[9] A.K. Porterfield, "Software methods for improvement of cache performance on supercomputer applications," Technical Report COMP TR 89-93, Rice University, May 1989.

[10] T. Mowry, M. Lam, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching," in *SIGPLAN Notices*, September 1992, pp. 62–73.

[11] Tien-Fu Chen and Jean-Loup Baer, "A performance study of software and hardware data prefetching schemes," in *Proc. of the 21st International Symposium on Computer Architecture*, April 1994, pp. 223–32.

[12] Vatsa Santhanam, Edward H. Gornish, and Wei-Chung Hsu, "Data prefetching on the HP PA-8000," in *The 24th Annual International Symposium on Computer Architecture*, June 1997.

[13] Daniel F. Zucker, Michael J. Flynn, and Ruby B. Lee, "A comparison of hardware prefetching techniques for multimedia benchmarks," in *Proceedings of the International Conference on Multimedia Computing and Systems*, Hiroshima, Japan, June 1996, pp. 236–244.

[14] Daniel F. Zucker, *Architecture and Arithmetic for Multimedia Enhanced Processors*, Ph.D. thesis, Stanford University, June 1997.

[15] A. Srivastava and A. Eustace, "Atom: A system for building customized program analysis tools," in *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, June 1994, pp. 196–205.

[16] Daniel F. Zucker and Alan H. Karp, "RYO: a versatile instruction instrumentation tool for PA-RISC," Technical Report No. CSL-TR-95-658, Computer Systems Laboratory, Stanford University, January 1995.

[17] K. Patel, B.C. Smith, and L.A. Rowe, "Performance of a software MPEG video decoder," in *Proceedings ACM Multimedia 93*, August 1993, pp. 75–82.

[18] Daniel F. Zucker, Ruby B. Lee, and Michael J. Flynn, "Hardware and software cache prefetching techniques for mpeg benchmarks," *IEEE Transactions on Circuits and Systems for Video Technology*, submitted June 1997.

[19] James E. Bennett and Michael J. Flynn, "Latency tolerance for dynamic processors," Tech. Rep. CSL-TR-96-687, Stanford University, Computer Systems Laboratory, January 1996.

[20] James E. Bennett and Michael J. Flynn, "Reducing cache miss rates using prediction caches," Tech. Rep. CSL-TR-96-707, Stanford University, Computer Systems Laboratory, October 1996.