Hardware and Software Cache Prefetching Techniques for MPEG Benchmarks

Daniel F. Zucker, *Member, IEEE,*, Ruby B. Lee, *Member, IEEE*, and Michael J. Flynn, *Member, IEEE*

***Abstract*—With the popularity of multimedia acceleration instructions such as MMX, MPEG decompression is increasingly executed on general purpose processors instead of dedicated MPEG hardware. The gap between processor speed and memory access means that a significant amount of time is spent in the memory system. As processors get faster—both in terms of higher clock speeds and increased instruction level parallelism—the time spent in the memory system becomes even more significant.**

Data prefetching is a well-known technique for improving cache performance. While several studies have examined prefetch strategies for scientific and commercial applications, this paper focuses on video applications. Data is presented for three types of hardware-prefetching schemes: the stream buffer, the stride prediction table (SPT), and the stream cache, as well as a new software-directed prefetching technique based on emulation of the hardware SPT. Up to 90% of the misses that would otherwise occur with no prefetching are eliminated. The stream cache can cut execution time by more than half with the addition of a relatively small amount of additional hardware. Software prefetching achieves nearly equal performance with minimal additional hardware. Techniques presented in this paper can be used to improve performance in a general-purpose CPU or an embedded MPEG processor. Performance gains achieved for MPEG benchmarks apply equally effectively to similar multimedia applications.

***Index Terms*—Cache, MPEG, multimedia, prefetching, software prefetching, stride-based prefetching.**

I. INTRODUCTION

WHILE there has been much work studying memory performance for scientific and general-purpose applications, this research focuses on the needs of multimedia applications. Relatively simple prefetching techniques can significantly improve the memory hit rates for MPEG applications. As processors become faster and utilize increasing instruction level parallelism, memory performance has a dominating effect on overall processor performance. Improvements in memory performance can eventually result in performance increases of up to $2\times$, with relatively little additional hardware.

Manuscript received June 16, 1997; revised April 12, 2000. This paper was recommended by Associate Editor S. Panchanathan.

D. F. Zucker was with is with the Computer Systems Laboratory, Department of Electrical Engineering, Stanford University, Stanford, CA 94305 USA. He is now with ePocrates, Inc., c/o P.O. Box 61036, Palo Alto, CA 94306 USA (e-mail: zucker@stanfordalumni.org).

R. B. Lee was with the Computer Systems Laboratory, Department of Electrical Engineering, Stanford University, Stanford, CA 94305 USA. She is now with the Department of Electrical Engineering, Princeton University, Princeton, NJ 08540 USA (e-mail: rblee@ee.princeton.edu).

M. J. Flynn is with the Computer Systems Laboratory, Department of Electrical Engineering, Stanford University, Stanford, CA 94305 USA (e-mail: flynn@arith.stanford.edu).

Publisher Item Identifier S 1051-8215(00)06563-0.

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. While a number of papers have been written studying both hardware and software-prefetching techniques, relatively little work has looked specifically at the memory behavior of MPEG applications.

The earliest hardware-prefetching work was reported by Smith [2] 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 $i+1$ is also prefetched. Jouppi [3] 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 $i+1, i+2, \ldots, i+n$ 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 [4] 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 nonunit steps. Fu and Patel [5] proposed utilizing stride information available in vector processor instructions to prefetch relevant data. They later [6] expanded the application to scalar processors by use of a cache-like look-up table called the stride prediction table (SPT).

Chen and Baer [7] 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 [8] investigate the timing issues of prefetching by use of a cycle-by-cycle processor simulation. Sklenar [9] 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 [10] proposed a technique for prefetching certain types of array data. Mowry *et al.* [11] proposed an early practical software-prefetch scheme based on information obtained at compile time. While software prefetching clearly has a cost

1051–8215/00$10.00 © 2000 IEEE

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 may result in negative side effects such as increased register usage. [12] 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* [13] 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.

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.

II. Methodology

A. Simulation Methods

A cache simulator was linked into memory address traces generated by RYO, an instruction instrumentation tool for the Hewlett Packard PA-RISC architecture [14]. Only data references are modeled and instruction accesses are ignored. The simulator provides data over a wide range of data cache sizes and associativities. 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.

Miss rates for these applications run in a baseline cache with no enhancements are shown in Fig. 1. Characteristics of the movies used in the benchmark executions are summarized in Table I. Complete miss-rate data for a range of cache line sizes and associativities is given in [15].

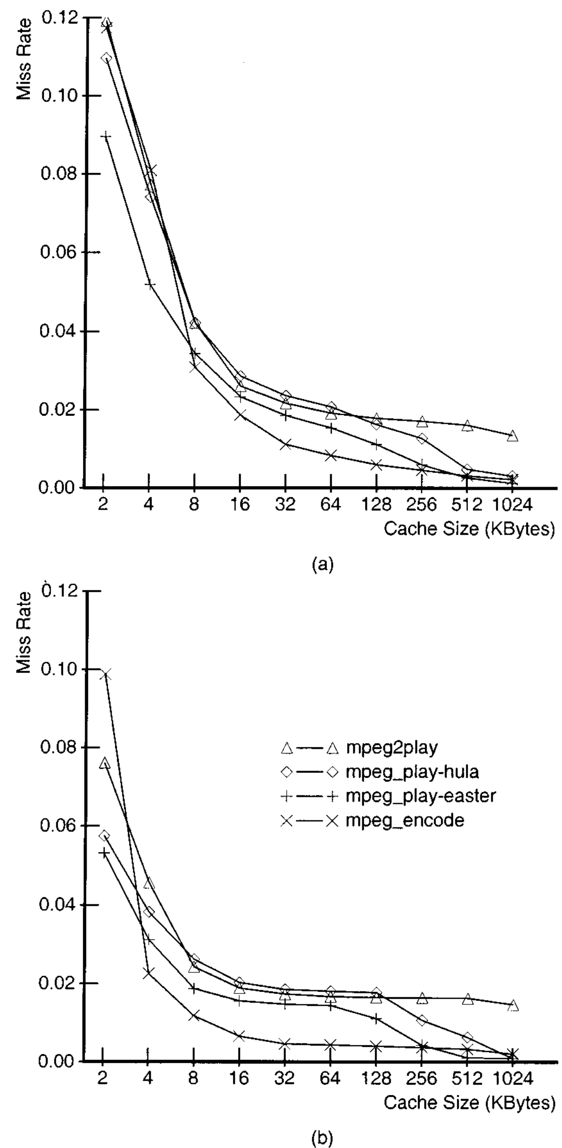


Fig. 1. Baseline miss rates. (a) Data for a direct-mapped cache. (b) Data for a 4-way associative cache.

B. Performance Metrics

Fraction of misses eliminated is the primary performance 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 the case of a second-level cache, the fraction of misses eliminated metric is identical to the hit rate for the second-level cache. Of all the misses that occur in the first-level cache, the fraction of those that hit in the second-level cache is, by definition, equal to the fraction of misses eliminated. The reason for using a fraction of misses eliminated instead of a second-level hit rate is for those configurations such as the SPT and parallel-stream cache where no discrete second-level cache exists. In this way, comparisons with a common metric can be used across all cache configurations in the study.

This metric is desirable for a number of other reasons as well. 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,

TABLE I
BENCHMARK IMAGE CHARACTERISTICS

application	image	frame size	number of frames	frame pattern	data memory references
mpeg	hula_2.mpg	352x240	40	IPPIPPI	6e+07
mpeg	flower.mpg	352x240	148	IBBPBBPBB	2e+08
mpeg	easter.mpg	240x176	49	IBBPBB	6e+07
mpeg	bicycle.mpg	352x240	148	IBBPBBPBB	2e+08
mpeg2	tennis.m2v	576x704	7	IBBPBBPP	8e+07
mpeg_encode	tennis.yuv	352x240	10	IBBPBBPBB	3e+08

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 nonmemory operation in a single cycle. Therefore, the only limit to computation time is the number of memory operations. 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 25 cycles for both misses and prefetches is assumed. Furthermore, a fully interleaved memory is assumed such that multiple outstanding prefetch requests are allowed. 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 Section IX, 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.

C. Memory Bandwidth

For the purposes of this study, memory bandwidth is assumed to be large enough such that this 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.

All architectures studied in this paper rely on significantly increasing accesses to main memory in the form of increased prefetching. Techniques for filtering ([4] and [8]) exist to address this issue. The software prefetching proposed in this paper eliminates a significant number of unneeded prefetches. [11] and [13] describe additional analytic techniques to limit the number of prefetches issued.

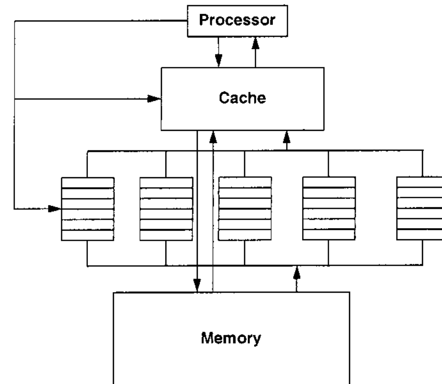


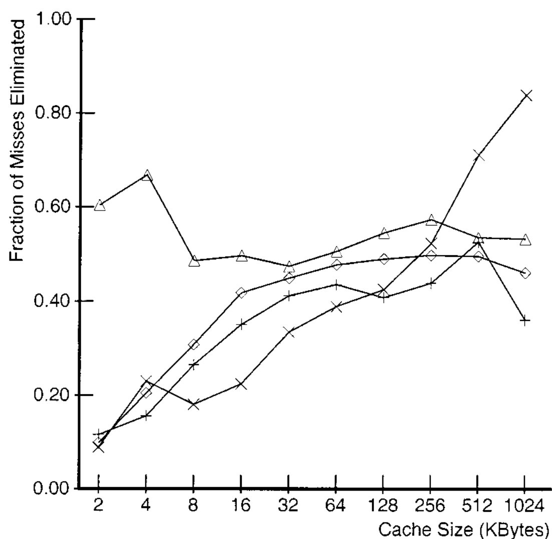
Fig. 2. A 5-way stream buffer architecture.

III. STREAM BUFFERS

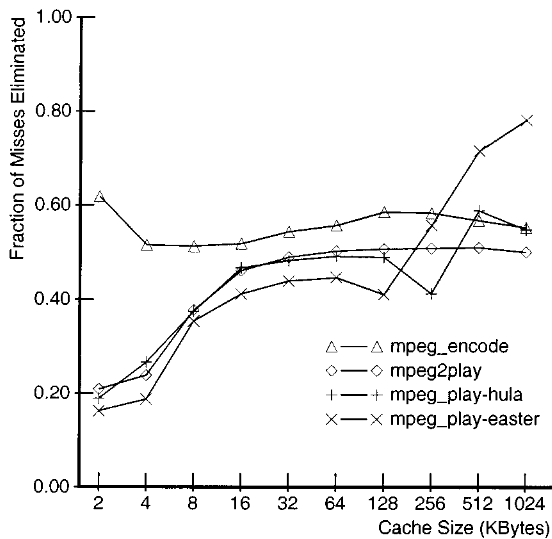
The stream buffer architecture simulated is shown in Fig. 2. As proposed by Jouppi, the stream buffer is a FIFO-type queue that sits on the refill path to the main cache. A new stream is allocated at each successive data cache miss. When all stream buffers are allocated, the next data cache miss replaces the stream least recently accessed (LRU replacement). A memory access that misses in the main cache, but hits in the stride buffer is counted as a hit. Since the refill time from the stream buffer can be an order of magnitude faster than a refill from main memory, this assumption does not significantly affect the reported results.

Our simulations assume up to 16 parallel-stream buffers. This number is selected to be large enough so that the number of stream buffers is not a limiting factor in performance. We also simulate a stream buffer depth of five entries. Palarcharla [4] proposed an enhancement to the stream buffer to filter unnecessary excess prefetches. However, because memory bandwidth is not a limiting factor in our model, this could only potentially hurt performance and is not included.

Performance data across a range of cache sizes and with direct mapped and 4-way associativities is shown in Fig. 3. For most applications, the stream buffer tends to peak out at eliminating about 50% of the misses. Mpeg_play playing easter.mpg is the single exception and can eliminate 80% of the misses for very large caches. This seems to be an outlying data point, however.



(a)



(b)

Fig. 3. Fraction of misses eliminated for 16-way stream buffers. (a) 16-way stream buffer with a direct-mapped main cache. (b) 16-way stream buffer with a 4-way associative main cache.

Thus, for most cases, only 50% of misses are eliminated with the stream buffer. This is because the relatively complicated algorithms involved tend to access the data in a nonunit strides, and the stream buffer is designed to aid only in cases of unit strides. Even with a 16-way stream buffer, approximately 50% of misses are eliminated, and therefore the stream buffer is not a totally effective prefetching technique.

IV. SPT

The structure of the SPT simulated is shown in Fig. 4. 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,

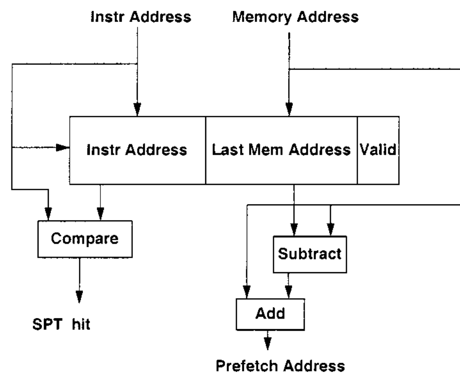


Fig. 4. SPT architecture.

is added to the SPT replacing the LRU entry. When a memory access is made by an instruction already contained in the SPT, 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 nonzero, 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 main cache.

The SPT requires access to the program counter (PC) and may therefore be slightly disadvantageous compared to the stream buffer. The stream buffer, since it relies only on external data requests, may be added more easily than the SPT to an existing commercial processor.

Data obtained from simulations using a 128- and 1024-entry stride table is shown in Fig. 5. All applications perform very well with a stride cache of 128 entries. For large main cache sizes, between 70%–90% of misses are eliminated relative to a cache with the same size and associativity, but no stride prediction mechanism. A knee in the curve appears, however, at a cache size of approximately 32 kB, below which the SPT rapidly becomes less effective.

Surprisingly, this is not as major of a factor for mpeg_encode, for which the performance does not appreciably decay for small cache sizes. This is due to the memory intensive motion estimation that must be done for mpeg encoding. The fairly large search space required for motion-vector encoding must be repeated for each 16×16 macroblock in the image. Although this requires a very large total number of memory references, the memory locality is quite good, and the traditional cache structure performs well, even for very small caches. Therefore, the smaller number of remaining misses that are not captured by the traditional main cache are handled more easily by the stride prediction mechanism.

Finally, an interesting effect is observed for SPT's of greater than 128 entries. In these cases, the stride prediction actually harms memory performance for relatively small cache sizes. The large number of nonuseful prefetches begins to remove useful data from the cache. This problem could potentially be solved by the use of filtering techniques.

The SPT works very well for middle and large cache sizes. Indeed, it would be difficult to do better than eliminating 90% of the misses. In this range, the SPT is an effective means of prefetching.

Fig. 5. Fraction of misses eliminated for SPT's. (a) 128-entry SPT with a direct-mapped main cache. (b) 128-entry SPT with a 4-way associative main cache. (c) 1024-entry SPT with a direct-mapped main cache. (d) 1024-entry SPT with a 4-way associative main cache.

V. Stream Cache

A. *Series-Stream Cache*

The stream cache overcomes the problems of the SPT by improving performance for the small cache sizes. The stream cache is an independent cache into which data is prefetched on an SPT hit. On an SPT miss, no data is prefetched to the stream cache, but the instruction that missed in the SPT is added to the SPT as described in Section IV. The SPT does a good job of predicting which data to prefetch, but fails for smaller cache sizes because it prefetches a large amount of unnecessary data. With the stream cache, data is prefetched not to the main cache, but to an independent stream cache. Because the data is not prefetched directly to the main cache, polluting the main cache is not a problem.

A series-stream cache architecture is shown in Fig. 6. The term, series, is used since the stream cache is connected in series with the main cache. The series-stream cache is queried after a main cache miss, and is used to fill the main cache with the desired data. If the data missed in the main cache is not in the stream cache, it is brought from main memory directly to the

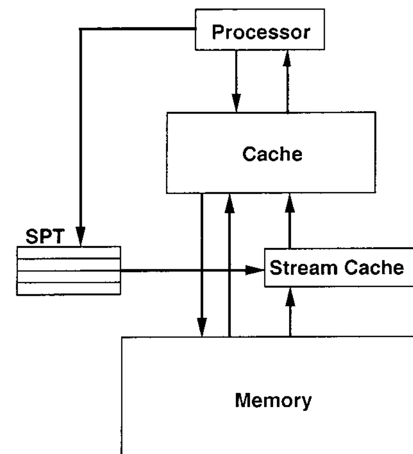


Fig. 6. Series-stream cache architecture.

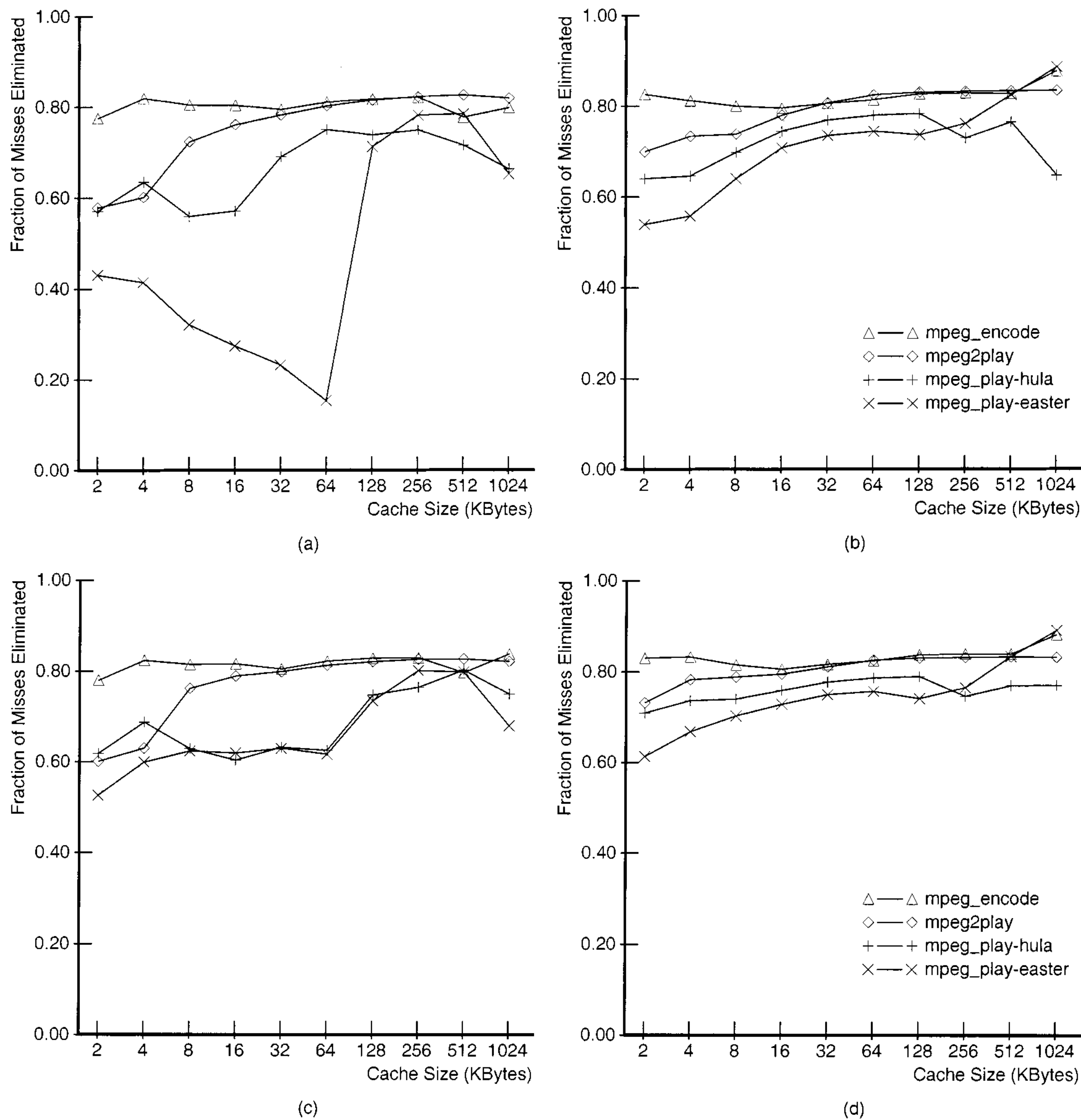


Fig. 7. Fraction of misses eliminated for 256- and 512-entry series-stream cache using a 128-entry SPT. (a) 256-entry series-stream cache with a direct-mapped main cache. (b) 256-entry series-stream cache with a 4-way associative main cache. (c) 512-entry series-stream cache with a direct-mapped main cache. (d) 512-entry series-stream cache with a 4-way associative main cache.

main cache. The series-stream cache simulated is fully associative. Because it is unlikely the data will be reaccessed from the series-stream cache once it is copied from the stream cache to the main cache, a most recently used (MRU) replacement policy is used when fetching new data into the stream cache. Using an LRU replacement policy would cause the data that has most recently been copied to the main cache to linger in the stream cache. This data is now in the main cache and keeping a copy in the stream cache is an inefficient use of stream cache storage. New data is fetched into the stream cache only on an SPT hit. The prefetch is not completed if the data at the prefetch address is already contained in the main cache.

The stream buffer works well only for unit strides and is inherently configured for a fixed number of streams. If a 16-way stream buffer is used, there should be 16 separate streams of application data for the cache memory to be efficiently utilized. The stream cache solves the problems of the stream buffer by uniting the separate FIFO's of multiple stream buffers into one relatively small fully associative stream cache. The SPT is used as before to predict which data to prefetch, but the data is prefetched to the stream cache instead of the main cache. Because the stream cache is unified, the specific number of streams in the application is irrelevant.

Performance data for a 256- and 512-entry stream cache are shown in Fig. 7. The 512-entry stream cache appears large enough to give a fairly uniform performance improvement of between 60%–80% across most cache sizes and both associativities. Performance for main cache sizes of less than approximately 32 kB is significantly improved over the same cache configurations using only a SPT.

This region on the left part of the graph is significant, since this is where the smaller main caches are not performing as efficiently and memory performance is a much higher percentage of execution time.

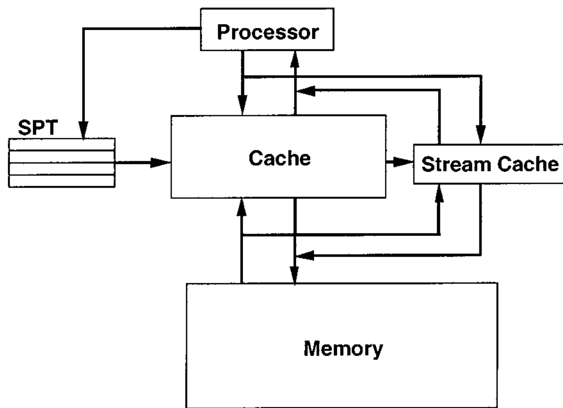


Fig. 8. Parallel-stream cache architecture.

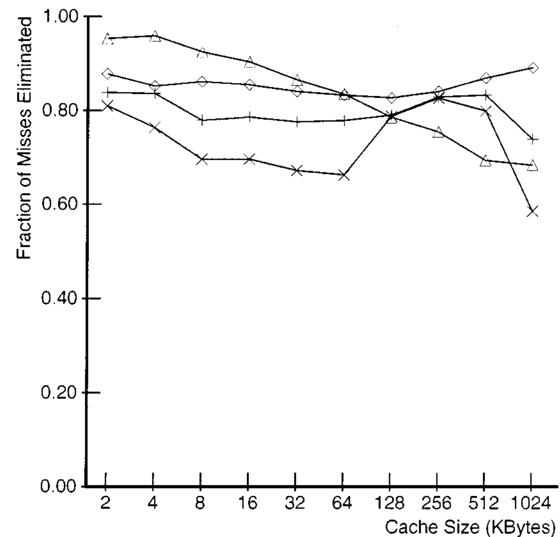
B. Parallel-Stream Cache

The parallel-stream cache is similar to the series-stream cache except the location of the stream cache is moved from the refill path of the main cache to a position parallel to the main cache. This is shown in Fig. 8, and is based on the hypothesis that multimedia applications tend to operate on a relatively small workspace of data that marches through the movie. The data in this workspace is operated on for a short time, but then is not frequently reused. The goal of the modified stream cache is to isolate this local workspace to the stream cache. Prefetched data is brought into the stream cache, but is not copied into the main cache. A cache access must search both the main cache and the stream cache in parallel. On a cache miss that cannot be satisfied from either the main cache or the stream cache, the data is fetched from main memory directly to the main cache. Because the data in the stream cache can be accessed multiple times before it becomes stale, an LRU replacement scheme is now employed. An MRU replacement scheme would prematurely discard recently accessed data before it becomes stale. Like the series-stream cache, data is prefetched to the stream cache on an SPT hit. The prefetch is not completed if the data is already contained in the main cache. On an SPT miss no data is prefetched to the stream cache, but the instruction that missed in the SPT is added to the SPT as described in Section IV.

Miss-rate data for a 256-entry parallel-stream cache with a 128-entry SPT is shown in Fig. 9. The smaller caches show a greater enhancement than mid sized caches, since there is a greater benefit from keeping less frequently used data out of the main cache. For small cache sizes, performance is better than the 128-entry series-stream cache described previously in Section V-A. Furthermore, this is the region where the main cache is suffering from high miss rates, so that this improvement is particularly beneficial.

VI. Time/Area Tradeoffs

In the previous sections, significant improvements in miss rates were reported. This increase did not come free, however. There is a cost in the additional die area required for the SPT and stream cache. In this section, performance comparisons are presented after considering this additional area.



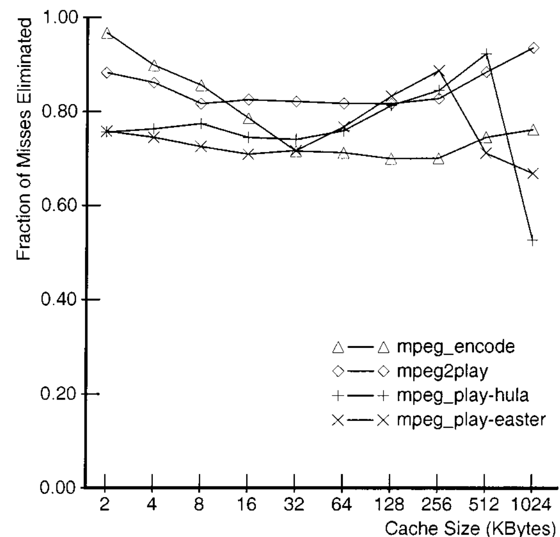
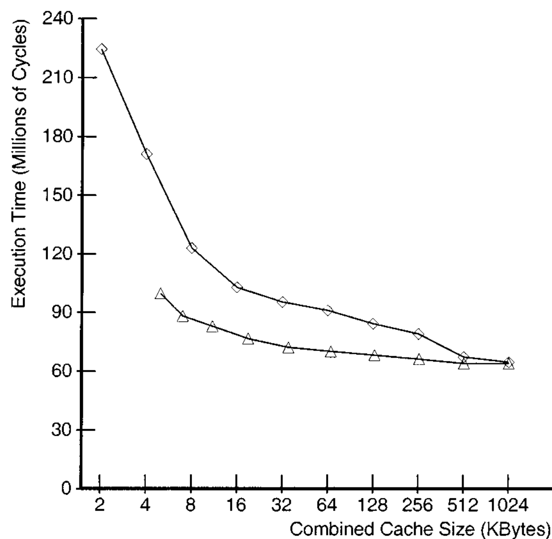


Fig. 9. Fraction of misses eliminated for 256-entry parallel-stream cache using a 128-entry SPT. (a) Direct-mapped main cache. (b) 4-way associative main cache.

A. Additional Area Requirement

Actual die area is highly implementation dependent and is difficult to model accurately. Each SPT entry must hold a complete instruction address, data address, and a valid bit for each entry. The SPT area is modeled assuming two 32-bit words, or 8 bytes, per entry. Additional area is also required for associated logic such as adders and comparators. This is not considered in the model. The additional area for the stream cache is calculated assuming 16 bytes per entry. This considers only the 16-byte data line size per entry and neglects the tag bits.

It may be possible to reduce the size of the SPT by storing only the lower 16 bits of the instruction address. This will cause some aliasing between unlike instructions, but is based on the hypothesis that the effect is not significant. Furthermore, it may be possible to store only the lower 16 bits of the address based on the assumption that data strides of more than 64 kB are unlikely.



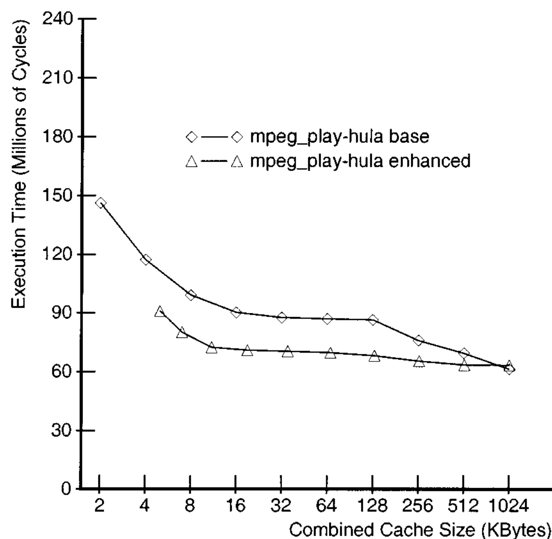


Fig. 10. Execution times for mpeg_play-hula with 128-entry parallel-stream cache and 128-entry stride table adjusted for extra area required. (a) Direct-mapped main cache. (b) 4-way associative main cache.

Again, this will cause some address aliasing. These techniques are not considered in the area model employed.

B. Execution Time

Absolute execution times for a single application are shown in Fig. 10. Execution time is calculated assuming a main memory latency for both the main and stream caches of 25 cycles. If data is needed while it is in the process of being loaded to the cache, then the balance of cycles remaining is counted in total execution time. Memory latency is always charged a constant latency of 25 cycles and conflicts between requests are not simulated. The horizontal axis is adjusted such that total area, including both the main cache and stream cache, is shown for the enhanced cache. For very small cache sizes, the stream cache can cut the execution time in half. For cache sizes of up to about 256 kB, less than 80% of the original time is required for execution. For very large cache sizes, the traditional cache design does

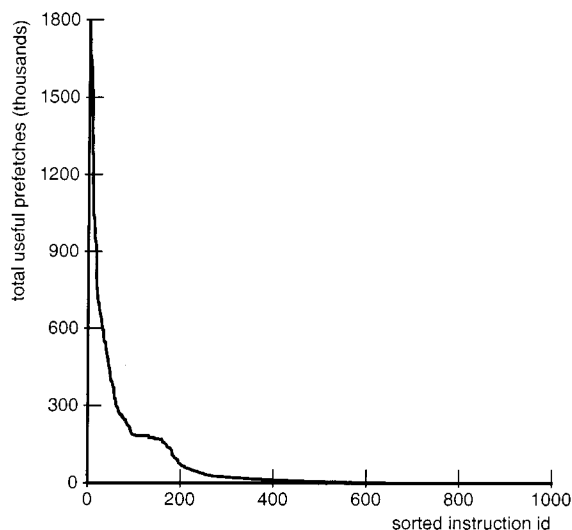


Fig. 11. Histogram of useful prefetches for movie flower using a 32-kB direct-mapped cache and parallel-stream cache.

a fairly good job of capturing the working set and the stream cache is proportionately less beneficial or even detrimental in some cases. In the case of large caches, then, the SPT alone is an effective means of prefetching. As image sizes become larger, however, this break point shifts to the right and the stream cache is useful over a larger range of caches.

This data suggests that the stream cache is effective in improving execution time for either a very small on chip cache or a low-cost multimedia system using only a small cache. A 128-entry SPT with a 128-entry stream cache adds only about 3-kB extra area, but cause the 2-kB main cache to perform as a baseline 16-kB cache or a 4-kB cache to perform as a baseline 128-kB cache for the application shown.

VII. Motivation for a Software-Directed Prefetching Technique

The advantage of the hardware-based stride prediction is that the stride value can change dynamically. A single instruction can prefetch on several different stride values throughout the duration of the program. The SPT described in Section IV, however, must be accessed for every load and store executed in order to determine when to prefetch. Perhaps it is necessary to keep track of only a small subset of load and store instructions.

Fig. 11 shows a histogram of total useful prefetches for a 32-kB direct-mapped main cache and 128-entry parallel-stream cache and movie flower. Total useful prefetches are shown on the y axis for a given instruction on the x axis. We define a useful prefetch as one for which the prefetched data is subsequently used by the application. Prefetches are counted for the instruction that predicted a given stride. The graph further sorts the instructions from those causing the most prefetches to those causing the least number of prefetches. From this histogram, it is observed that only a relatively small number of instructions cause most of the useful prefetches.

Fig. 12 shows the same data cumulatively for four cache sizes and a 128-entry parallel-stream cache. Thus, when the line indicating total prefetches becomes level, the instructions at that

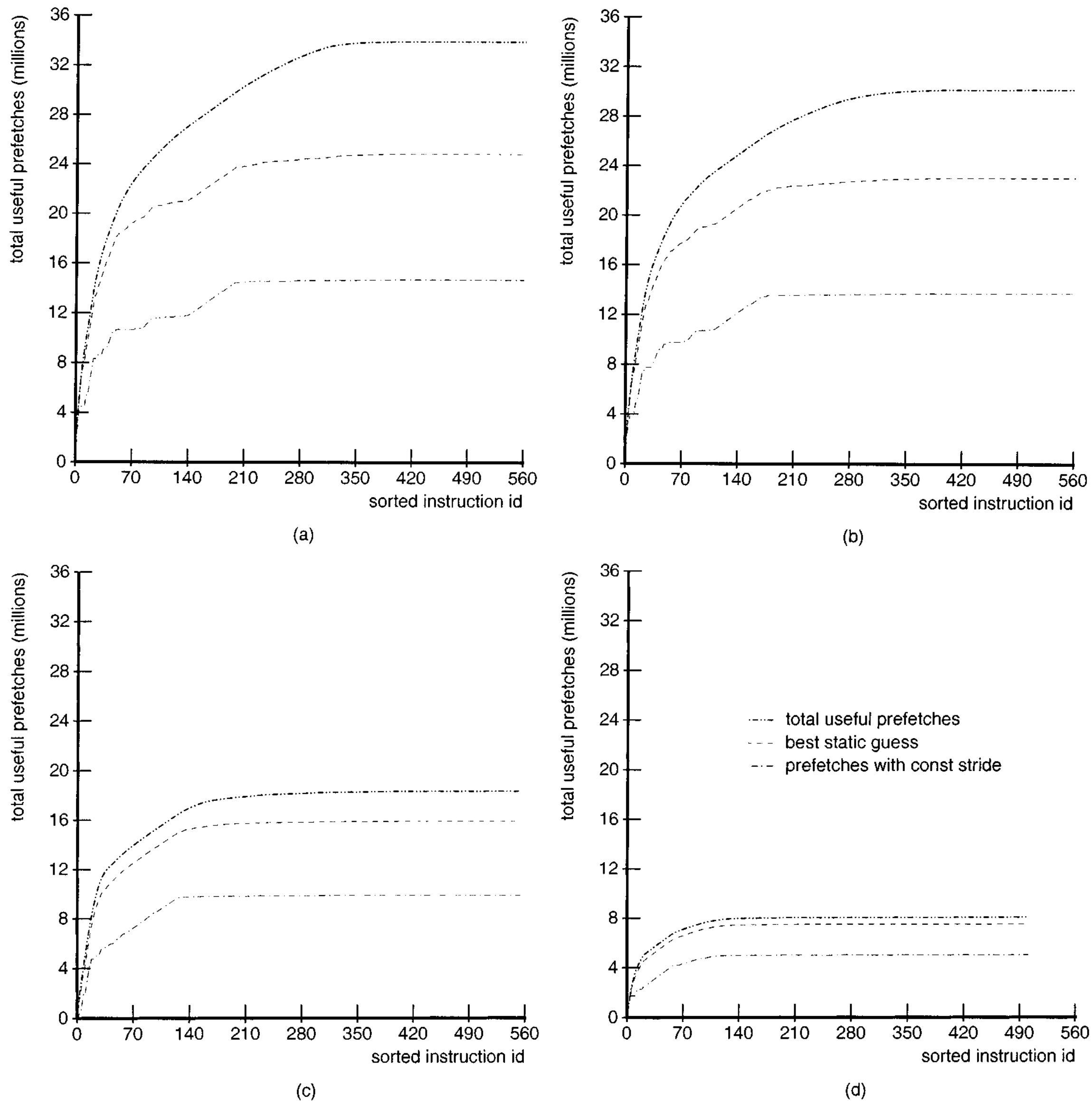


Fig. 12. Effect of main cache size. All graphs are for image hula with a direct-mapped main cache and a parallel-stream cache. (a) 2-kB main cache. (b) 4-kB main cache. (c) 32-kB main cache. (d) 1-MB main cache.

point are causing no significant prefetches. A relatively small number of instructions, on the order of 150, is all that is needed to cause most of the prefetches. These graphs, furthermore, divide the prefetches into three separate categories indicated by the three separate lines on the graphs. The top-most line is the total number of useful prefetches. The bottom-most line counts only those prefetches that had constant strides. For these, the SPT seems unnecessarily complex, since these strides do not change dynamically. The capability to change strides dynamically is one of the key features of the SPT. Finally, the middle line shows all the prefetches that result if each instruction has only a static stride associated with it. A single stride value is selected based on the most common stride value from the run. The static prediction performs only slightly worse than the dynamic prediction.

The figures indicate that although the hardware mechanism allows us to fully exploit a dynamically changing stride value, using a best-guess static stride value works almost equally effectively. Furthermore, only a few hundred individual instructions, not the thousands that are in the executable, cause most of the useful prefetches. A software-directed stride-based prefetching technique that replaces the hardware SPT is now proposed.

VIII. A Trace-Based Software-Prefetching Technique

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.

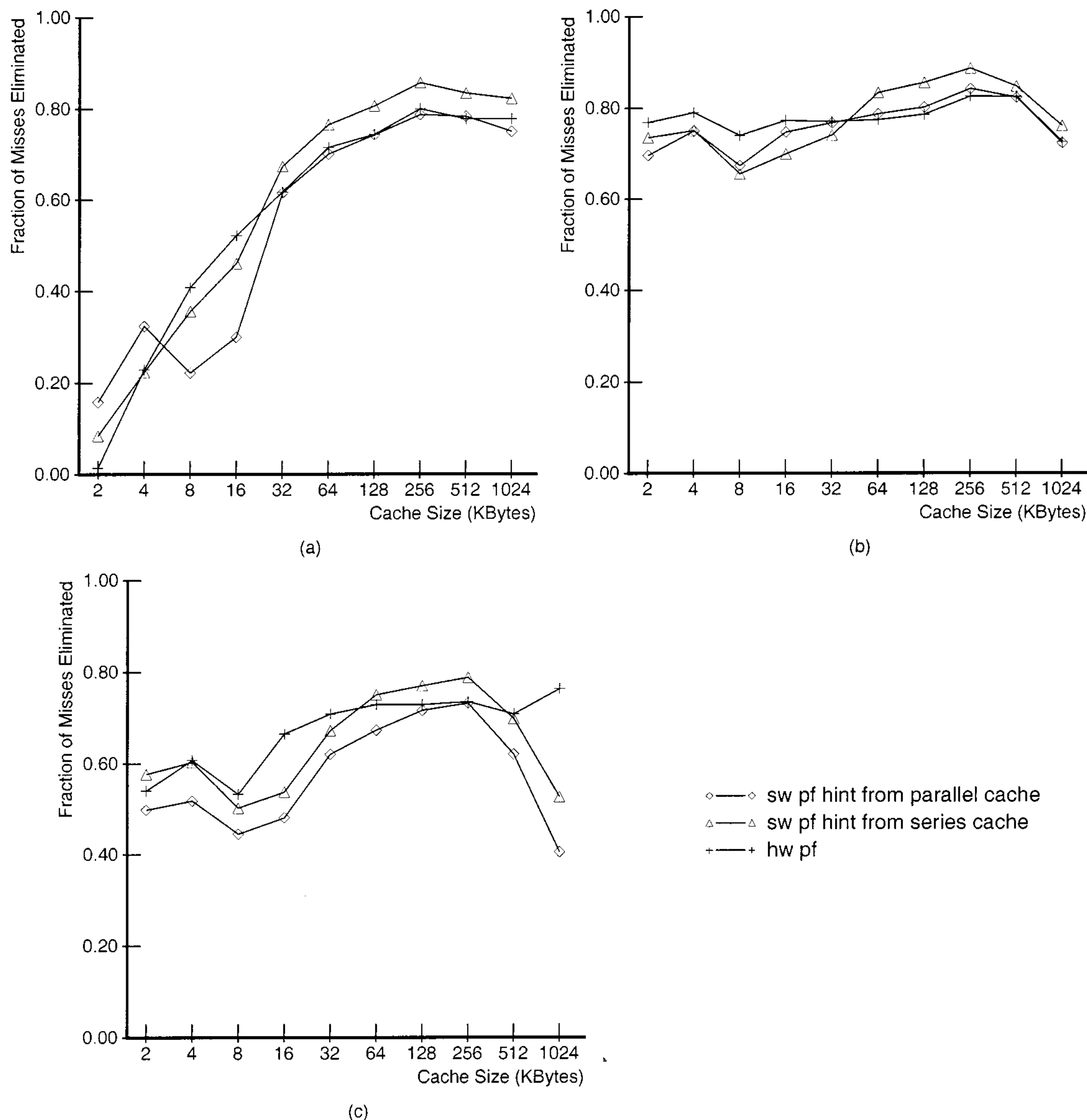


Fig. 13. Effect of varying cache type used for generating the prediction file. All graphs compare the effects of predictions generated using a parallel-stream cache, a series-stream cache, and hardware-directed prefetching using the movie hula and a 2-kB direct-mapped main cache. Only the 200 most effective instructions issue prefetches in the software case. Fraction of misses eliminated is reported for execution performed with a direct-mapped main cache and in a: (a) no stream cache; (b) parallel-stream cache; and (c) series-stream cache.

The profile step simulates a hardware SPT. 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 can selectively insert software-prefetch instructions into the executable code using a static stride prediction.

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 into either the special-purpose stream cache or prefetches directly into the main cache depending on the cache configuration simulated. 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.

Fig. 13 shows the fraction of misses eliminated for three different cache configurations using the software-prefetch technique described. The 200 most useful prefetch instructions are added to the executable code. Fig. 13(a) shows data prefetched directly to the main cache, Fig. 13(b) shows data prefetched to a parallel-stream cache, and Fig. 13(c) shows data prefetched to a series-stream cache.

The three lines in each graph indicate different methods for inserting prefetch instructions. The hardware SPT is compared to two software-prefetch trials. In one case the useful prefetch data is collected from simulation trace of a parallel-stream cache, and in the other, it is collected from a series-stream cache.

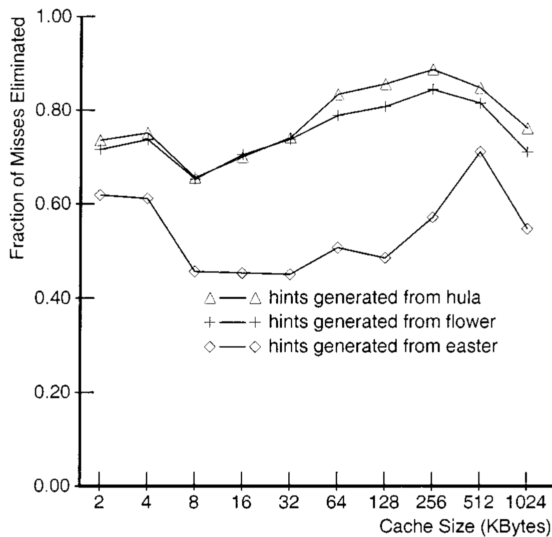


Fig. 14. Effect of varying movie used to make prediction file. Prediction file was generated using a 2-kB direct-mapped main cache and series-stream cache. Only the 200 most effective instructions issue prefetches. Fraction of misses eliminated is reported for execution of hula performed with a direct-mapped main cache and parallel-stream cache.

In general, the shapes of the three curves for no stream cache, series, and parallel-stream caches matches the shape of the same curve for the hardware-based stride prediction. The software prediction results in similar performance to the hardware-directed prefetching without the cost of a hardware SPT. Indeed, the software-directed prefetching does even better for a number of cases. Recall that for the small caches, the hardware will prefetch an excessive amount of data knocking out useful data, and degrading performance. The software-directed prefetching eliminates this problem since only the more effective prefetches are inserted. Software prefetching is an effective replacement for the hardware SPT.

A. *Effect of Frame Size*

Fig. 14 illustrates the effect of varying the movie displayed while holding the cache size, associativity, and configuration of the cache used to collect the prefetch data constant. The image hula was used with a series-stream cache and a direct-mapped main cache of size 2 kB was used to collect the prefetch statistics. The first 200 most useful instructions are used to execute prefetches. Data is shown across a range of cache sizes for a direct-mapped main cache with a parallel-stream cache for the images hula, flower, and easter. Data is presented in terms of fraction of misses eliminated.

The movies hula and flower perform approximately the same, and "easter" appears significantly worse. This is due to the frame size of the movies. Hula and flowers share the same frame size at 352×240, while easter has a frame size of 240×176. This indicates that the frame dimensions are important components of stride information. It is important that the movie displayed when collecting the trace information has the same frame size as the movie displayed.

B. *Effect of Sorting*

We have shown that software prefetching is a generally effective technique to replace the SPT. In this section a methodology is developed for automatic insertion of software-prefetch instructions. We focus on the case with no stream cache. In this way, the prefetching methodology can be applied automatically by a compiler on any machine that supports a software-prefetch instruction.

The task is to determine which software-prefetch instructions to insert. In this section, we look at different possibilities for sorting the useful prefetches. The hint table in this section is generated by sorting only useful prefetches that would have been performed by a static prefetch. Data that was prefetched by the SPT's dynamic striding mechanism is not included.

For a given main cache size, associativity, and movie type, there are two possible orderings of instructions. One is to order by accesses from the parallel-stream cache simulation, and one is to order by accesses to the series-stream cache simulation. The accesses to the series-stream cache record whether the indicated instruction prefetches data was accessed, and does not give any measure of how much the data is used once it has been accessed. The accesses to the parallel-stream cache give an indication of how useful the data is once it has been prefetched.

The relative execution time across a range of cache sizes for a constant 90% of useful prefetches inserted is shown in Fig. 15. The series ordering wins out in all cases. This is because the parallel ordering prioritizes more by the total number of accesses rather than just the first access, as in the series ordering. For the execution time, only the first access is important, since on the first miss, the data is demand fetched into the cache automatically.

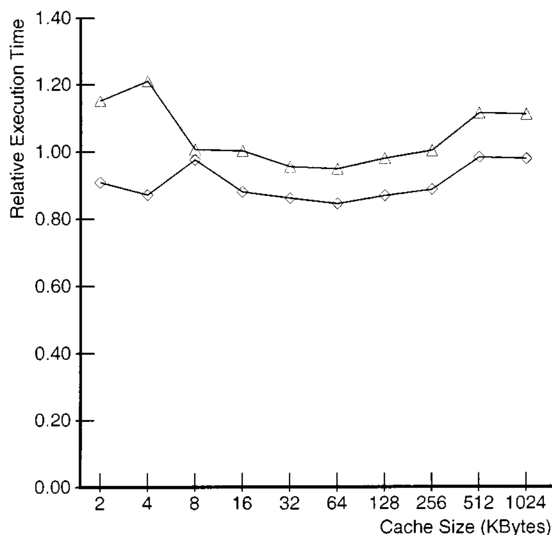
C. *Percent of Prefetch Instructions to Insert*

In this section, we determine the best number of prefetch instructions to insert. In Fig. 16, we show the effect of inserting different percentages of the total possible number of prefetch instructions for a single movie, hula. Fig. 16(a) shows fraction of misses eliminated for a range of direct-mapped caches. Aside from the smaller caches, the fraction of misses eliminated gets continuously better the more prefetch instructions are inserted. For the smaller caches, there is some benefit in inserting fewer instructions so that less useful data is replaced by speculative prefetches. Relative execution time is shown for a direct-mapped 128-kB cache in Fig. 16(b). Data is shown for three movies: hula, easter, and bicycle. The percentage of useful prefetch instructions inserted is varied along the x axis. This illustrates the cost of inserting all prefetch instructions, since the 100% shows the worst relative execution time. There is a pattern of relative execution time decreasing to an optimal point after which the overhead of executing extra instructions begins to offsets the benefit of prefetching. Fraction of misses eliminated tends to continuously improve as more instructions are inserted.

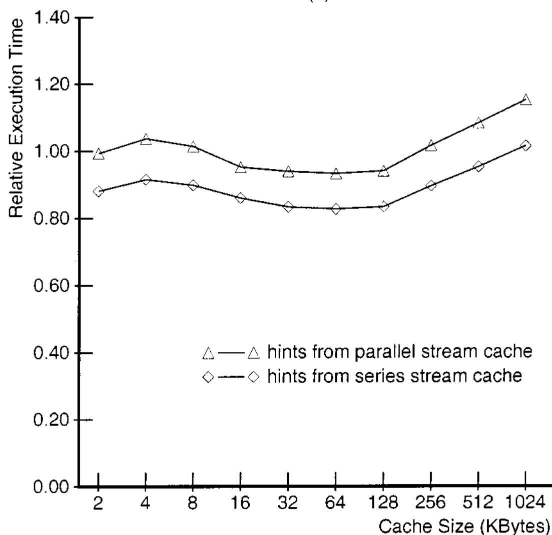
For a given cache size, the tangent to the minimum point of the relative execution time shows the optimal relative execution time achievable. This also indicates the optimal percentage of prefetch instructions to insert. Inserting instructions to capture 90% of available prefetches is the optimal choice for most cache sizes.

IX. Software Prefetching Performance

In Fig. 17, we show relative execution time across a range of cache sizes for the optimal number of prefetches inserted, 90%. For the parameters chosen for this model, we can achieve up

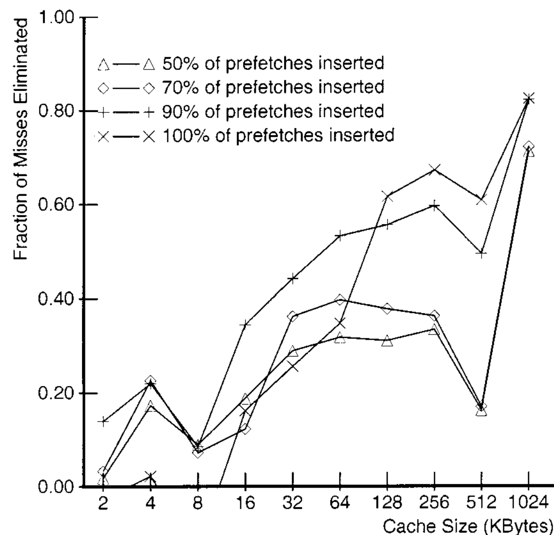


(a)

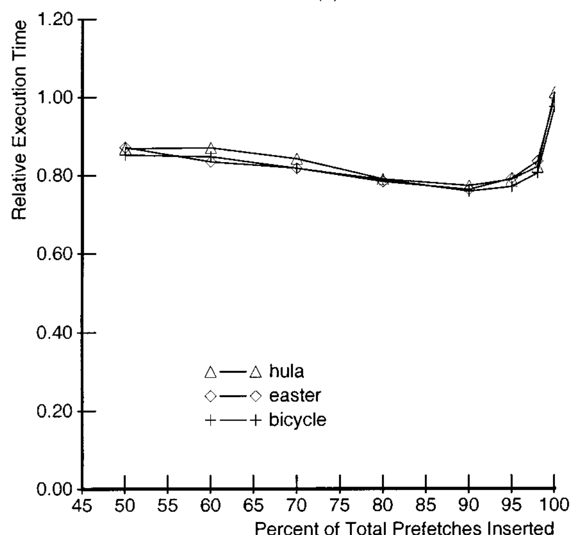


(b)

Fig. 15. Relative execution time for 90% of available prefetches inserted comparing prediction file generated from series and parallel-stream caches. (a) Direct-mapped cache. (b) 2-way associative cache.



(a)



(b)

Fig. 16. Performance data for different numbers of prefetch instructions inserted for movie hula in a direct-mapped cache. (a) Fraction of misses eliminated. (b) Relative execution time for a 128-kB main cache.

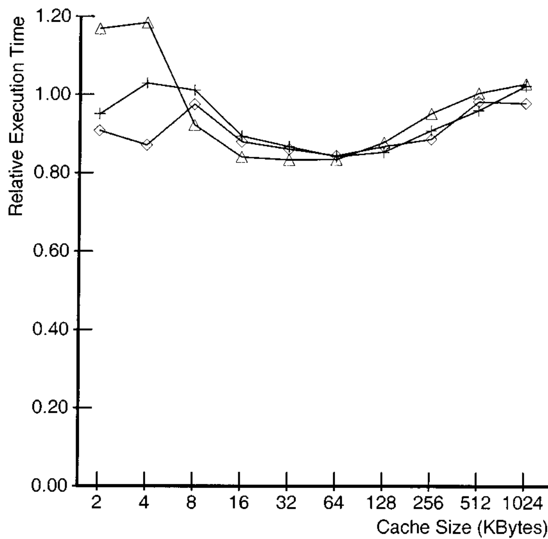
to a 20% improvement in execution time for a range of caches between 16 and 256 kB by adding only software-prefetch instructions. In the next section, we investigate how adjusting our memory access model affects performance.

A. *Effect of Execution Time Parameters*

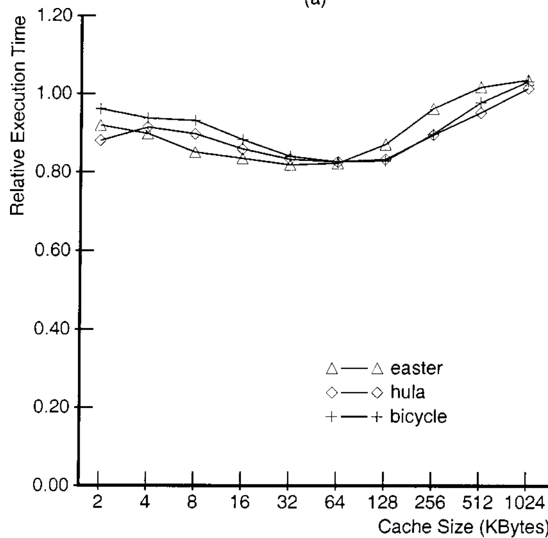
In this section, we investigate how altering certain parameters in our model effects relative execution time. Until this point, we have been assuming that accesses to memory for both demand and prefetch misses cost 25 cycles, and that instructions consist only of load stores. We now consider the impact of altering the memory access cost and instruction mix parameters. A memory access cost of 100 cycles and an instruction mix in which loads and stores comprise only 30% of instructions is investigated. 30% is the measured instruction mix for mpeg_play, as determined in [16]. A full range of graphs with memory access costs of 10, 25, 50, and 100 cycles, for both direct mapped and 2-way associative and a number of instruction mix parameters is available in [15]. We assume that all instructions except cache misses execute in one cycle. We show data for three movies in Fig. 18.

In general, the maximum benefit added by prefetching is increased as the memory access time is increased. As the memory access time is increased, the fraction of execution time spent in the memory system is also increased. The more time that is spent in the memory system, the more potential there is for prefetching to improve performance. This is indicated in the graphs in the way the curves tend to dip lower and lower for the higher memory access costs. At a memory access cost of 100 cycles, a 60% improvement in performance is possible.

Changing the instruction mix from 100% to 30% similarly affects the fraction of execution time that is spent in the memory system. With less time spent in the memory system, there is less potential for improvement. This is illustrated in that the downward dip on the graphs becomes smaller as you increase the instruction mix. This has the additional effect of decreasing the

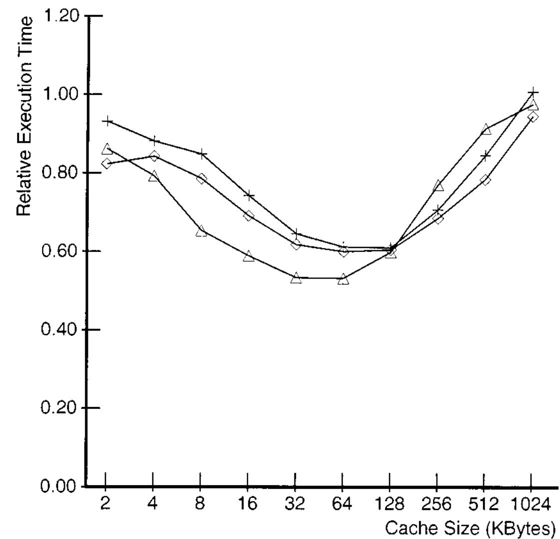


(a)

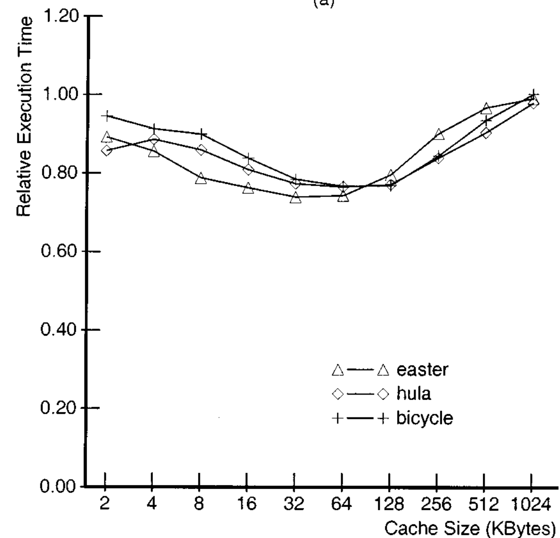


(b)

Fig. 17. Relative execution time for 90% of prefetches inserted.(a) Executed with a direct mapped cache. (b) Executed with a 2–way associative cache.



(a)



(b)

Fig. 18. Relative execution time for a 2-way associative cache with 90% of prefetches inserted. (a) Memory access cost of 100 cycles and an instruction mix of 100% loads and stores. (b) Memory access cost of 100 cycles and 30% loads and stores.

overhead cost of additional prefetch instructions. Since there are additional instructions executed in the 30% mix, execution of added prefetch instructions is less noticeable.

B. Effect of Memory Access Parameters

In this section, we present data taking into account different memory access models. Previously, we simply assumed both full and partial cache hits completed in one cycle. That is, as long as a memory request was issued it was counted as a hit regardless of how many cycles remained for the data to return to memory. To calculate total execution time, we simply multiplied the total number of misses by a constant miss penalty.

In this section, we now fully account for the case of partial hits. If the miss penalty is 25 cycles, and a memory access occurs for that data 10 cycles after the prefetch miss, the balance of 15 cycles is charged to the execution time. Furthermore, we investigate different possibilities for simultaneous number of outstanding prefetches. Previously, we assumed that the memory system was fully pipelined such that there was no limit to how many outstanding prefetches are allowed at one time. We now limit accesses such that only a finite number of accesses are allowed at once. In the limit, where there are as many prefetches allowed as memory access cycles, the result should be the same as previously where a constant memory access penalty was assumed.

We also look at different techniques for prioritizing memory accesses and prefetches and different possibilities for simultaneous numbers of outstanding prefetches. The configurations simulated are summarized in Table II. In configuration A, we assume demand loads and stores have no priority over prefetches. In other words, the memory system services all requests in the order received. This has an advantage in implementation simplicity. Since all memory accesses are equivalent, no special considerations must be made for prefetches. This has a significant disadvantage in performance in that prefetches will stall the processor even if the data requested is not necessary. We further assume that a maximum of 32 pending prefetches can be queued. When the queue becomes full, additional prefetches

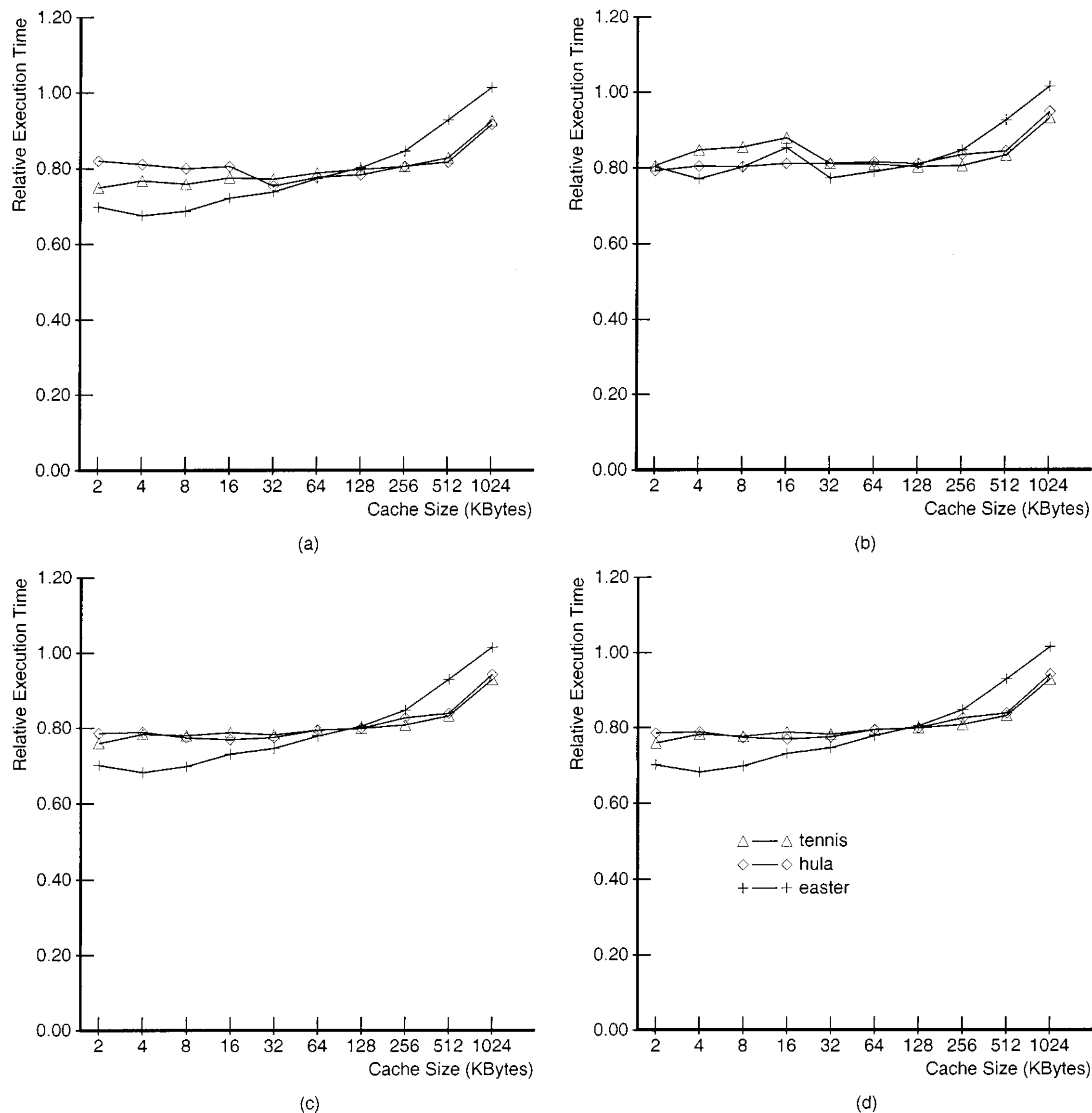


Fig. 19. Relative execution time for different memory models in a 2-way associative cache. (a) Configuration A. (b) Configuration B. (c) Configuration C. (d) Configuration D.

TABLE II
SUMMARY OF MEMORY ACCESS MODELS SIMULATED

label	simultaneous number of outstanding prefetches	demand references given priority	pf queue size
A	1	no	32
B	1	yes	infinite
C	3	yes	infinite
D	25	yes	infinite

are discarded. If we allowed an infinite number of prefetches to queue up, then the performance with prefetching is always greatly degraded from the base case, since large numbers of waiting prefetches hopelessly stall the servicing of demand misses. Only one outstanding memory access can occur at a time. Performance for configuration A is shown in Fig. 19(a).

In configuration B, we continue to assume that only one memory access can occur at a time, but we now assume loads and stores have priority over prefetches. Now even if there is a queue of prefetches waiting to be executed, a load or store will jump to the front of the queue to be executed. If there is an outstanding memory operation, the balance of the memory penalty must still be incurred as the prefetch returns, but the load or store will be executed immediately afterwards. Because demand loads and stores have priority over prefetches, there is no need to limit the pending queue to 32 entries. Performance data for this case is shown in Fig. 19(b).

Configuration C makes the same assumption, but now assumes the memory is 3-way interleaved so that three memory accesses can be outstanding at once. Finally, configuration D allows for a maximum interleaving so that all memory access return in a constant time of 50 cycles. Performance for these configurations is shown in Figs. 19(c) and (d), respectively.

These figures indicate that as long as demand loads and stores are given priority over prefetches, the number of simultaneous outstanding prefetches does not significantly effect performance. Furthermore, the figures confirm that perfor-

mance benefit from prefetching is enhanced when using 2-way associativity.

X. Summary

In this paper, we investigated a number of prefetching techniques for MPEG benchmarks. The regular memory access pattern of these applications makes some form of data prefetching an attractive strategy for improving memory performance.

Stream buffers can eliminate up to about 50% of data misses for small and moderately sized caches. It is the small cache sizes, where the large number of misses contribute significantly to total execution time, where a large reduction in misses is desirable. The series-stream cache added improvement over the SPT for smaller sized caches, and left the performance improvements intact for large caches. Finally, the parallel-stream cache resulted in extremely good performance enhancements for small cache sizes with a small amount of additional hardware, but in some cases did slightly worse than the SPT for large cache sizes.

The SPT was shown to perform extremely well for large caches, and the series-stream cache and parallel-stream cache perform very well for small cache sizes. In these cases, performance improvements will result from a small increase in hardware. Extremely cost- or area-sensitive applications, where a small cache is required, can benefit significantly from employing such a technique.

Based on performance data analysis from hardware prefetching, a software-prefetching scheme was proposed to replace the hardware SPT. Performance was similar with a reduced cost in hardware. Finally, a methodology was developed to add software-prefetch instructions to existing compiled code. Only a small number of prefetch instructions generate 90% of useful prefetches. On a machine with no special-purpose hardware, other than a prefetch instruction and a 2-way associative cache, a 60% simulated improvement in execution time is observed.

References

[1] A. Peleg and U. Weiser, "MMX technology extension to the intel architecture," *IEEE Micro*, vol. 16, pp. 51–59, Aug. 1996.

[2] A. J. Smith, "Cache memories," *ACM Comput. Surveys*, vol. 14, pp. 473–530, Sept. 1982.

[3] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Proc. 17th Annu. Int. Sympo. Computer Architecture*, May 1990, pp. 364–373.

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

[5] J. W. C. Fu and J. H. Patel, "Data prefetching in multiprocessor vector cache memories," in *Proc. 18th Annu. Int. Symp. Computer Architecture*, May 1991, pp. 54–63.

[6] J. Fu, J. Patel, and B. Janssens, "Stride directed prefetching in scalar processors," in *Proc. 25th Int. Symp. Microarchitecture*, Dec. 1992, pp. 102–110.

[7] J.-L. Baer and T.-F. Chen, "An effective on-chip preloading scheme to reduce data access penalty," in *Proc. Supercomputing'91*, Nov. 1991, pp. 176–186.

[8] T.-F. Chen and J.-L. Baer, "Effective hardware-based data prefetching for high-performance processors," *IEEE Trans. Comput.*, vol. 44, pp. 318–328, May 1995.

[9] I. Sklenar, "Prefetch unit for vector operations on scalar computers," *ACM Comput. Architec. News*, vol. 20, pp. 31–37, Sept. 1992.

[10] A. K. Porterfield, "Software methods for improvement of cache performance on supercomputer applications," Rice Univ., Houston, TX, Tech. Rep. COMP TR 89-93, May 1989.

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

[12] T.-F. Chen and J.-L. Baer, "A performance study of software and hardware data prefetching schemes," in *Proc. 21st Int. Symp. Computer Architecture*, Apr. 1994, pp. 223–232.

[13] V. Santhanam, E. H. Gornish, and W.-C. Hsu, "Data prefetching on the HP PA-8000," presented at the 24th Annu. Int. Symp. Computer Architecture, June 1997.

[14] D. F. Zucker and A. H. Karp, "RYO: A versatile instruction instrumentation tool for PA-RISC," Comput. Syst. Lab., Stanford Univ., Stanford, CA, Tech. Rep. CSL-TR-95-658, Jan. 1995.

[15] D. F. Zucker, "Architecture and arithmetic for multimedia enhanced processors," Ph.D. dissertation, Stanford Univ., Stanford, CA, June 1997.

[16] S. T. Fu, D. F. Zucker, and M. J. Flynn, "Memory hierarchy synthesis of a multimedia embedded processor," in *Proc. Int. Conf. Computer Design*, Oct. 1996 , pp. 176–184.

Daniel F. Zucker (S'92–M'97) received the B.S. degree with distinction, and the M.S. and Ph.D. degrees, all in electrical engineering, from Stanford University, Stanford, CA, where he was a National Science Fellow and Hitachi Fellow.

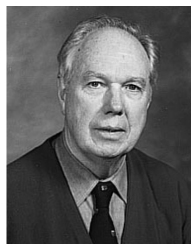
He is currently Chief Technical Officer of ePocrates, Inc., San Carlos, CA, a provider of networked software for handheld computers. He is also founder of ZookWare, LLC, for which he developed "WhatsOn," a popular palm computing consumer application. He has worked for Advanced Micro Devices, Sun Microsystems, Fujitsu Research Laboratories, and most recently led the secure network applications group at TriStrata, Inc. He has published several academic and popular articles on multimedia computer architecture and handheld computing.

Dr. Zucker was the recipient of the Frederick E. Terman Award from Stanford University. He is a member of Phi Beta Kappa, Tau Beta Pi, and ACM.

Ruby B. Lee (M'99) received the A.B. degree with distinction from Cornell University, the M.S. degree in computer science and computer engineering, and the Ph.D. degree in electrical engineering, both from Stanford University, Stanford, CA.

Since September 1998, she has been the Forrest G. Hamrick Professor in Engineering at Princeton University, Princeton, NJ, with an affiliated appointment in Computer Science. During 1989–1998, she was a Consulting Professor of Electrical Engineering at Stanford University, and also a Chief Architect with Hewlett-Packard (HP) Company, leading the interdisciplinary security architecture team for enterprise and e-commerce systems. Prior key contributions at HP include the PA-RISC architecture from initial design through several generations of server and workstation products, the first PA-RISC CMOS microprocessor, MAX and MAX-2, the first multimedia acceleration instructions for microprocessors, and the Intel-HP IA-64 EPIC architecture for emerging 64-bit Intel microprocessors. She has been granted 82 U.S. and foreign patents. Her research interests are in computer architecture, multimedia architecture, and security architecture.

Dr. Lee is a member of Phi Beta Kappa, Alpha Lambda Delta, and ACM.



Michael J. Flynn (S'74–M'75) received the B.S. degree from Manhattan College, Bronx, NY, the M.S. degree from Syracuse University, Syracuse, NY, and the Ph.D. degree from Purdue University, West Lafayette, IN.

He was a designer of mainframe computers at IBM before becoming a Professor of Electrical Engineering at Stanford University, Stanford, CA, in 1975, where he formed the Stanford Architecture and Arithmetic Group. He has authored or co-authored four books and over 250 professional papers.

Dr. Flynn founded both of the specialist organizations on Computer Architecture, the IEEE Computer Society's Technical Committee on Computer Architecture and ACM's SIGARCH. He was the 1992 recipient of the ACM/IEEE Eckert–Mauchly Award and of the 1995 IEEE Computer Society's Harry Goode Memorial Award.