Prefetching using Markov Predictors

Doug Joseph
IBM T. J. Watson Research Lab
P.O. Box 218
IBM. T. J. Watson Research
Yorktown Heights, NY 10598
djoseph@watson.ibm.com

Dirk Grunwald
Department of Computer Science
University of Colorado
Boulder, Colorado, 80309-0430
grunwald@cs.colorado.edu

Abstract

Prefetching is one approach to reducing the latency of memory operations in modern computer systems. In this paper, we describe the *Markov prefetcher*. This prefetcher acts as an interface between the on-chip and off-chip cache, and can be added to existing computer designs. The Markov prefetcher is distinguished by prefetching *multiple reference predictions* from the memory subsystem, and then prioritizing the delivery of those references to the processor.

This design results in a prefetching system that provides good coverage, is accurate and produces timely results that can be effectively used by the processor. In our cycle-level simulations, the Markov Prefetcher reduces the overall execution stalls due to instruction and data memory operations by an average of 54% for various commercial benchmarks while only using two thirds the memory of a demand-fetch cache organization.

1 Introduction

Processors normally fetch memory using a demand-fetch model: when the processor issues a load instruction, the specified datum is fetched from memory. If the datum is not in the cache, a request is made to the external memory system to fetch the datum. By comparison, a memory prefetching mechanism attempts to provide data *before* the processor requests that data. We assume that the the data is placed in a *prefetch buffer* where it can be accessed by the processor, or uses some other mechanism to perform a *non-binding* prefetch that avoids disturbing the current cache contents.

There are three important metrics used to compare memory prefetchers: coverage, accuracy and timeliness. Coverage indicates the fraction of memory requests that were supplied by the prefetcher rather than being demand-fetched. Accuracy indicates the fraction of the prefetched cache lines offered to the processor that were actually used. Unless prefetched memory references are provided to the processor before they are needed, the processor may still stall during execution. Timeliness indicates if the data offered by the prefetcher arrives before it is needed, but not so early that the data must be discarded before it can be used. The ideal prefetcher has large coverage, large accuracy and produces timely data – the prefetcher offers the processor all the data it needs, only the data it needs and before the processor needs the data.

This paper is to appear in the 1997 Proceedings of the International Symposium on Computer Architecture (ISCA'97), June 1997, Denver Colorado.

We believe prefetching mechanisms are designed in a two-step process: first, the architect envisions a "model" describing the way that programs behave when accessing memory, and then attempts to construct a physical realization of that model that provides suitable prefetch coverage. In this paper, we describe a hardware prefetch mechanism that offers better performance than other prefetch mechanisms. In memory-level simulations, we find that this prefetching mechanism can reduce the memory overhead to the cycles-per-instruction by 54%, greatly reducing the memory stalls encountered by our model processor. Moreover, although we devote one MByte of memory to the prefetcher data structures, our prefetcher reduces the memory stalls while reducing the total amount of memory devoted to the combination of the prefetcher and second level cache – using 3 MBytes of memory, our prefetcher outperforms a demand-fetch memory system that uses 4 MBytes of memory.

We first describe the memory access model assumed by our prefetcher and how the prefetcher is physically implemented. We then briefly survey alternative prefetcher designs. We follow this by with a description of the experimental design and analysis we used to estimate the performance of this prefetcher.

2 Prefetcher Design and Implementation

Hardware prefetching is a prediction process: given some current prediction state, the prefetcher guesses what a future memory reference may be and requests that location from the memory subsystem. There are a number of possible sources of prediction information. One such source is the address reference stream, or the sequence of addresses referenced by the processor. However, using this prediction source requires that the prefetching hardware be on the same chip as the processor. The prefetcher would need to be very efficient, since it may then need to analyze many references per cycle.

The prefetchers we examine use the *miss address stream* as a prediction source. These references are presented to the external memory subsystem and, due to the first-level caches, these miss references occur much less frequently than memory references. Figure 1 shows a schematic design for the prefetcher, which can be built as an external part of the memory subsystem. We'll see that this is important because the prefetcher may need to use considerable state information to be effective. As shown in Figure 1, we assume the processor has on-chip prefetch buffers that are examined concurrently with the first level caches. Thus, prefetched data items do not displace data resident in the cache. The prefetched data only contends with the normal demand-fetched memory references for the processor bandwidth.

Prefetch mechanisms usually assume that programs access

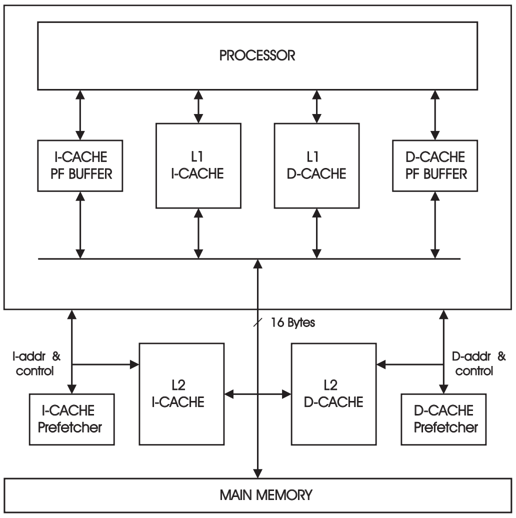


Figure 1: System design

A, B, C, D, C, E, A, C, F, F, E, A, A, B, C, D, E, A, B, C, D, C

Figure 2: Sample miss address reference string. Each letter indicates a cache miss to a different memory location.

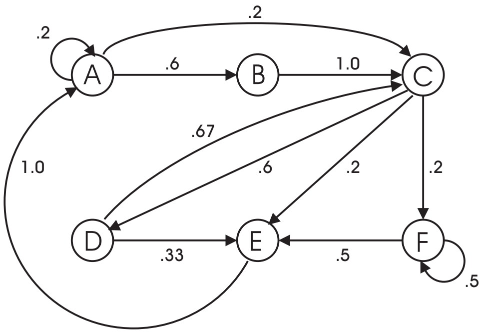


Figure 3: Markov model representing the previous reference string *via* transition probabilities.

memory using a particular pattern or access model. For example, stream buffers [7] assume that memory is accessed as a linear stream, possibly with a non-unit stride. Once an access model has been determined, architects design a hardware mechanism to capture or approximate that reference stream. In the next section, we describe the access model we assume and a hardware implementation that captures that access model.

2.1 Modeling Memory References via Markov Processes

We assume that the miss reference stream can be approximated by an observed Markov model. Assume that our miss reference stream is that shown in Figure 2. In this example, different memory locations are identified by different letters. Thus, this reference sequence indicates a missing reference to memory location "A", followed by a miss for "B" and so on. Using this reference string, we can build a Markov model, shown in Figure 3, that approximates the reference string using a transition frequency. Each transition from node X to node Y in the diagram is assigned a

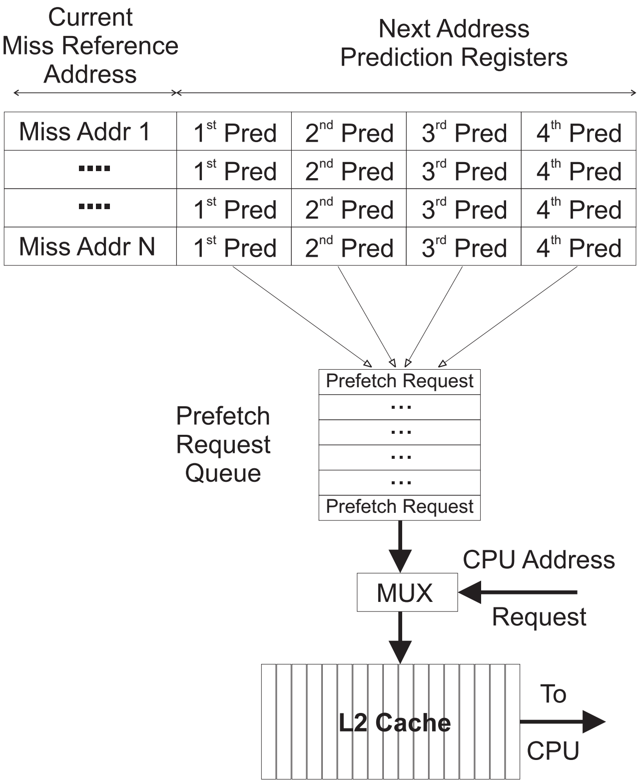


Figure 4: Hardware Used to Approximate Markov Prediction Prefetcher

weight representing the fraction of all references X that are followed by a reference Y. For example, there are five references to node A in the the example miss reference sequence. Of these, we see the pattern "A, A" 20% of the time, the pattern "A, C" 20% of the time and the pattern "A, B" 60% of the time. This example uses one previous reference to predict the next reference. Intuitively, if the program were to execute again and issue the same memory references, the Markov model could be used to predict the miss reference following each missing reference. For example, on re-execution, the appearance of an A may lead the hardware to predict that A, C or B will be the next missing reference and issue prefetch requests for each address. In general, a n-history Markov model can use more history information – for example, given the training sequence A, B, C, the prefetcher would predict C if the miss sequence A, B was seen. We have examined the performance of general n-history models and found little added benefit from the additional information, and thus focus on 1-history models in this paper.

2.2 Realizing the Markov Prefetcher in Hardware

There are several problems encountered in assuming a "pure" Markov model of memory references. In practice programs don't repeat exactly the same reference patterns from one execution to another, and the transition probabilities "learned" in one execution may not benefit another. Furthermore, it is difficult to efficiently represent a pure Markov model in hardware because each node may have an arbitrary degree and the transition probabilities are represented as real values. Lastly, programs reference millions of addresses and it may not be possible to record all references in a single table. Despite these drawbacks, our later data will show that a

"Markov-like" model of memory references is an effective prefetch mechanism. Thus, we need to make a set of design choices that address the problem of representing a Markov transition diagram in hardware. We first describe the design of the Markov predictor and then justify those decisions with simulation studies.

The first decision is to continuously rebuild and use the Markov model as a program is executing. Thus, the approximate Markov model captures the *past activity* of all programs on a system and uses that information to predict the future references. We limit the number of possible nodes in the transition diagram and limit their maximal out-degree. More concretely, we represent the Markov transition diagram using a table such as shown in Figure 4. In this sample configuration, each state in the Markov model occupies a single line in the prediction table, and can have up two to four transitions to other states. The total size of the table is determined by the memory available to the prefetcher. When the current miss address matches the index address in the prefetch table, all of the next address prediction registers are eligible to issue a prefetch, subject to mechanisms described later intended to improve prefetch accuracy. However, not all possible prefetches actually result in a transfer from the L2 cache. Each prefetch request has an associated *priority*. Prefetch addresses are stored in the prefetch request queue, and higher priority requests can dislodge lower-priority requests. The prefetch request queue contends with the processor for the L2 cache, and the demand fetches from the processor have higher priority. Thus, after a series of prefetches, the prefetch request queue may be full, and lower-priority requests will be discarded.

Once a fetch request is satisfied by the L2 cache, it is placed in the on-chip prefetch buffers. Demand-fetch requests are directly stored in the cache. We model the on-chip prefetch buffers as a 16 entry fully associative FIFO buffer. When the processor queries it, all entries are associatively searched in one cycle. If a match is found, it is relocated to the head of the FIFO, and all the entries from the head to the vacated slot are shifted down by one. The FIFO is also searched when updated to avoid duplicate entries. If there are no duplicates when adding an entry, an empty slot is filled, or if there are no empty slots the last slot (the least recently used entry) is replaced. This design is similar to the stream buffer design of Farkas *et.al.* [6]. The primary difference is that in [6], the entire buffer is shifted, discarding all the entries above the matching one.

There are many other parameters that affect the performance of this hardware configuration. We used trace-driven simulation and a memory-level performance model to determine the importance of those parameters and to compare the performance of the Markov prefetcher to previously suggested prefetchers. We next describe prior work on prefetchers and then describe the experimental design to compare the performance of Markov prefetchers to previous designs. We then show the effect of the various parameters in the Markov prefetcher implementation.

3 Prior Work

Hardware and software prefetching schemes have been devised that are effective on structured workloads [14, 4, 12, 8]. However research on prefetching for unstructured workloads is not nearly as common and only recently have results in this area begun to appear [17, 3, 9, 13]. The section on *correlation-based prefetching* is especially relevant since Markov prefetching is an evolution of correlation based prefetching.

Static predictors Almost all static predictors rely on the compiler to determine possible L1 cache misses and embed the information into the code in the form of prefetch instructions. Mowry *et.al.* [12] show that structured scientific codes are very amenable to this approach. However, they also show that their techniques failed to improve performance of the pointer intensive applications used in their study. In terms of hardware resources, compiler based schemes are inexpensive to implement. However, since prediction information is embedded in the program at compile time, compiler based schemes lack the flexibility to account for the dynamic behavior of a workload. Compiler based techniques have been proposed which insert prefetch instructions at sites where pointer dereferences are anticipated. Lipasti *et. al.* [9] developed heuristics that consider pointers passed as arguments on procedure calls and insert prefetches at the call sites for the data referenced by the pointers. Ozawa *et.al.* [13] classify loads whose data address comes from a previous load as *list accesses*, and perform code motions to separate them from the instructions that use the data fetched by list accesses.

Stride Prefetchers Chen and Baer investigate a mechanism for prefetching data references characterized by regular strides [4]. Their scheme is based on a *reference prediction table* (RPT) and *look-ahead program counter* (LPC). The RPT is a cache, tagged with the instruction address of load instructions. The entries in the RPT hold the previous address referenced by the corresponding load instruction, the offset of that address from the previous data address referenced by that instruction, and some flags. When a load instruction is executed that matches an entry in the RPT, the offset of the data address of that load from the previous data address stored in the RPT is calculated. When this matches the offset stored in the table, a prefetch is launched for the data address one offset ahead of the current data address. In [4], the *reference address stream* was used to index the reference prediction table. In practice, we found little performance difference between using the reference addresses or the miss address stream. Our later simulations of stride prefetchers use the miss address stream.

Stream Buffers Jouppi introduced *stream buffers* as one of two significant methods to improved direct mapped cache performance [7]. In contrast to stride prefetchers, stream buffers are designed to prefetch sequential streams of cache lines, independent of program context. The design presented by Jouppi is unable to detect streams containing non-unit strides. Palacharla and Kessler [14] extended the stream buffer mechanism to also detect non-unit strides without having direct access to the program context. They also introduced a noise rejection scheme for improving the accuracy of stream buffers. Farkas *et.al.* [6] further enhanced stream buffers by providing them with an associative lookup capability and a mechanism for detecting and eliminating the allocation of stream buffers to duplicate streams. Later, we compare the performance of the design of Farkas *et. al.* to Markov prefetchers.

Stream buffers are used like a prefetch fill buffer for servicing L1 cache misses in this paper. Prefetches are placed in the stream buffer itself rather than a separate prefetch buffer. Stream buffers are *allocated* on L1 cache misses. If any stream buffer contains an entry that matches the current L1 miss reference address, it is taken from the stream buffer, the entries below the one removed are all shifted up to the head of the buffer, and prefetches are launched to sequentially consecutive cache line addresses to fill the vacancies that open up in the bottom part of the stream buffer. If there is no match in any stream buffer, a new stream buffer is allocated to the new stream. In the model employed in this paper, an empty or least recently used buffer is selected for replacement. The noise rejection scheme introduced by Palacharla [14] is also employed in the allocation of stream buffers used in this research. It is a simple filtering mechanism that waits for two consecutive L1 misses to sequential cache line addresses before allocating a stream buffer to the stream.

Stride prefetchers and stream buffers complement one another is various ways. Stream buffers generally exhibit greater prefetch coverage than stride prefetchers, but also are much more inaccurate, even when using allocation filters. However, while detecting non-unit strides is natural for stride prefetchers, providing non-unit stride detection to stream buffers is more difficult [6].

Stream buffers tend to be more efficient in use of resources than stride prefetchers. For example, given the following program fragment:

```
for (i = 0; i < N; ++i)
{
   b = x[i+20];
   c = x[i+22];
}
```

a stride prefetcher will consume two resources in the stride detection table, while only one stream buffer would be allocated. In our studies we have found that using a stride prefetcher in *series* with stream buffers works well. That is, we allow the more accurate stride prefetcher to issue a prefetch first if it is able and otherwise allocate a stream buffer. The combination provides better coverage than either mechanism alone, and is generally more accurate than stream buffers alone (although less accurate than a stride prefetcher alone).

Indirect Stream Detectors Mehrota [11] describes a hardware data prefetching scheme based on the recursion that occurs in linked list traversals. We also simulated this design, but do not present the performance since it was uniformly worse than correlation-based prefetching.

Correlation-Based Prefetching Markov prefetching is a continuing evolution of what has been called *correlation-based prefetching* [3]. The basic concept of correlation-based prefetching was introduced by Baer [2] in the context of paged virtual memory systems. Baer associated a single prefetch address with each memory address referenced and developed algorithms for updating the prefetch address based upon observed reference patterns. When a reference occurs, the associated prefetch address is checked for residence in physical memory. If the prefetch page is not resident, then it is paged in. This pairing of two temporally related related events, such as a current address with a prefetch address, is the essence of correlation-based prefetching. The first address of the pair is referred to as the *parent* or *key* that is used to select a child prefetch address.

The first instance of correlation-based prefetching being applied to data prefetching is presented in a patent application by Pomerene and Puzak [15]. A hardware cache is used to hold the parent-child information. A further innovation they introduce is to incorporate other information into the parent key. They suggest the use of bits from the instruction causing the miss, and also bits from the last data address referenced. They also introduce a confirmation mechanism that only activates new pairs when data that would have been prefetched would also have been used. This mechanism is very much like the *allocation filters* introduced by Palacharla *et.al.* [14] to improve the accuracy of stream buffers and serves a similar purpose here.

Charney and Reeves [3] extend the Pomerene and Puzak mechanism and apply it to the L1 miss reference stream rather than directly to the load/store stream. Besides being the first to publish results on the Pomerene and Puzak scheme, this work improved upon the mechanism in two significant ways. One was to introduce greater lead time into the prefetching with the use of a FIFO history buffer. Instead of entering parent-child pairs into the pair cache, ancestors older than the parent can be paired with the child and entered in the pair cache. Although no results are reported in the impact this had on CPU stalls, it was demonstrated that prefetch lead time could be significantly improved at the expense of lower prefetch accuracy. The other contribution was a study of various alternate structures for the parent key. This study focused primarily on using different combinations of bits from the instruction and data addresses of L1 miss references. In general, there was marginal improvement in prefetch accuracy or coverage in those attempts.

Another important contribution by Charney and Reeves was to show that stride based prefetching could be combined with correlation based prefetching to provide significant improvements in prefetch coverage over using either approach alone, on certain benchmarks. In this scheme, a stride prefetcher is placed at the front end of a correlation-based prefetcher. If the stride prefetcher could make a prediction it would, and the miss references associated with the stride prediction would be filtered out of the miss reference stream presented to the correlation-based prefetcher. Coverage improved for two reasons. One is correlation-prefetchers (and Markov prefetchers) must see a miss reference repeat before it can predict a future miss reference. Stride prefetchers do not have that limitation. The other is that better utilization of the pair cache is achievable when the stride references are filtered out. For the workloads used in this paper, we find that there are insufficient stride references in the applications we examined for this scheme to offer much improvement in prefetch coverage. However, in the case of very "stridy" workloads, it seems clear this approach is advantageous.

Alexander and Kedem [1] proposed a mechanism similar to correlation-based prefetching but used a *distributed* prediction table. In their variation, a correlation-based table was used to predict bit-line accesses in an Enhanced DRAM, and was used to prefetch individual bit lines from the DRAM to the SRAM array.

4 Experimental Design and Simulation Study

There are three important factors that influence the performance of a prefetching scheme: coverage, accuracy and timeliness. Prefetch coverage and accuracy are less dependent on a specific memory system configuration, while timeliness depends greatly on the memory reference penalties of a particular system. In this study, we use a single metric, the fraction of first-level cache misses, to characterize *both* coverage and accuracy. For each application, we record the number of demand-cache misses encountered without any prefetching and normalize these values. Then, we measure and normalize the number of cache misses for different prefetching schemes. We define "coverage" to be the fraction of the miss references satisfied by the prefetch mechanism. A prefetch mechanism will fetch more data from memory than a simple demand-fetch mechanism, and many of those references may be mispredicted fetches. We record the additional references relative to the normalized demand-fetch references, and define this to be a measure of the *inaccuracy* of a prefetcher. We measured timeliness by simulating a non-speculative processor with a detailed memory model and comparing the *memory cycles-per-instruction* (MCPI). This represents the average number of CPU stalls attributed to the memory subsystem. Both simulation studies used the same cache configurations, described below.

We assume that all non-memory instructions execute in one cycle. The model processor has a single-cycle on-chip 8KB L1 data cache and an 8KB L1 instruction cache. Each cache has 8-entry single-cycle victim buffers and 32-byte lines. The L1 data cache also has a single-cycle 8-entry write buffer and uses a write-around policy. The second-level (L2) cache were multi-cycle, multi-bank, direct mapped, lockup-free 4MB I and D caches, with 128 byte lines. The L2 data cache uses a write-back with write allocate policy and had one 8-entry address request queue per bank. We model four synchronous SRAM cache banks in the baseline model. The address and data busses have a latency of four cycles. When combined with the four cycle memory latency, and the four cycles to return the data to the processor, the total cache miss penalty is twelve cycles, but new requests can be pipelined every four cycles. Each cache bank has a separate address bus to each L2 cache bank but just one L2 data bus shared by all banks. Thus, there is never address bus contention, but there may be considerable contention for the data bus. The L1-L2 bus bandwidth is 8 bytes/cycle. We used a multi-cycle, multi-banked memory model with one 8 entry address request queue per bank and 4 memory banks. Address and data busses have a latency of 4 cycles. The access latency of a bank is 24 cycles and the L2-L3 bus bandwidth is 4 bytes/cycle.

When simulating stream buffers, we used eight three-entry stream buffers, with associative lookup, non-overlapping stream allocation, and allocation filters. Each stream buffer has single cycle access. When simulating stride prefetchers, we used a stride prefetcher with a 16 entry fully associative *stride detection table*. Access is also single cycle. We also experimented with larger stride tables using four-way set associative tables taking 32-KBytes of storage. There was no advantage to tables larger than 2KBytes for the traces we considered, and there was little difference between a 4-way associative table larger than 2-KBytes or the 16-entry fully-associative table. In earlier work on stream buffers, Farkas *et. al.* [6] used four three-entry stream buffers. We used eight because there was a small, but noticable improvement in coverage up to that point. We also varied the number of entries in the stream buffer and reached the same conclusions stated in [6]: below three entries, prefetches have insufficient lead time and above three entries, accuracy begins to fall off rapidly..

The Correlation and Markov prefetchers are combined with the second-level cache. When modeling the Correlation and Markov prefetchers, we used a bounded or constant amount of storage for the combined prefetch and cache subsystem, but that was not possible while continuing to use direct mapped caches. Thus, we configured the Markov and Correlation prefetchers to use less memory than the corresponding demand-fetch cache or the stride and stream prefetchers. For the Markov and Correlation prefetchers, we used a one MByte data prefetch table and a 2MByte data cache. For the demand-fetch model, we used a 4-MByte data cache. In other words, the prefetch implementations would require two thirds the memory of stride or stream prefetchers when implemented. For the data cache, we also used a 2MB instruction cache and 1MByte prefetch table for the prefetch configurations, and a 4-MBbyte instruction cache for the demand-fetch model.

4.1 Benchmark Applications

There is a deficiency in the most widely used benchmarks for simulation studies (e.g. the SPEC programs suite). Past research has indicated that operating system activity and multi-programming can significantly effect cache performance [10, 5]. However, little has been reported on the impact these factors have on prefetching strategies. At the same time, the cache performance on such workloads tends to be significantly worse than other workloads, making the need for latency reducing methods such as prefetching even more important.

Many important technical and commercial applications give rise to unstructured workloads. Technical applications involving large sparse arrays of data often store such data in a compressed format and access that data via indirection arrays (i.e. $a[b[i]]$). Usually the organization of sparse arrays is not known till run time and may evolve during execution. Another common source of unstructured access patterns in technical and commercial workloads is the use of pointer connected structures. Large graphs or trees of structures are often dynamically generated and may evolve during execution. Algorithms in the application may jump from one part of a tree or graph to another. Consequently, pointers are not always accurate indicators of access patterns. Unstructured technical workloads include such important examples as event driven simulators and wire routing tools for VLSI design, unstructured grid algorithms in computational fluid dynamics, modeling of molecular dynamics, DRAM device level simulation, and structural dynamics analysis. Commercial environments tend to be unstructured because of high process switch rates, high random I/O rates, and they typically involve a large number of user processes [10]. Transaction processing also utilize searching and sorting algorithms that give rise to unstructured access patterns. Examples of commercial workloads include: transaction processing, multi-user software development environments, network and file server kernels, desktop publishing tools, and compilers.

The simulations of this work are based on address traces of technical and commercial industry-standard benchmarks. They were captured on an **IBM RS/6000** running **AIX** using a proprietary tracing tool developed at **IBM**. Cache performance characteristics on most of the traces we used were presented by Maynard *et.al.* [10]. The traces include both instruction and data references obtained throughout the execution of multiple processes containing kernel, user and shared library activity. Four of the traces used were generated from unstructured technical codes, and four were from commercially oriented workloads. Table 1 provides a brief summary of all eight. More information on the benchmarks can be found in [10] and the appendix.

Table 2 shows statistics indicative of the impact of these differences. The first two columns show the percentage of the instructions that are branches and the percentage of these that are taken branches. An analysis of branching behavior helps explain one of the reasons I-cache miss rates tend to be higher for commercial workloads than technical workloads. Typical technical workloads are dominated by short to medium length loops. For such a workload where most branch instructions return control to the head of the loop, the percentage of taken branches is much higher. Also, if the longest instruction loops fit in the I-cache, the I-cache miss rate is very low. In contrast, the percentage of taken branches in commercial workloads is relatively low, indicating that these workloads execute relatively few iterations per loop. The lack of dominant loops is why these commercial workloads have a lower probability of re-executing recent instructions, leading to higher miss rates [10]. We note that Spice is anomalous in this trend, yet still has a very low I-miss rate. As with all the SPEC92 and SPEC95 benchmarks, the instruction working set for Spice is very small and fits comfortably in the I-cache.

The average sequential block size also shows the "branchy" nature of commercial workloads. Block sizes in commercial workloads tend to be much shorter than in technical codes. The last column in Figure 2 shows the fraction of the total number of instructions executed in the operating system. These numbers indicate that much of the work in commercial workloads is actually done by the operating system. One reason for this is the relatively high usage is that there is frequent movement of small amounts of data between the different levels of the system, with few arithmetic operations on the data. In technical workloads, the operating system brings data into application space, and the application performs extensive arithmetic manipulation before handing it back to the operating system to store.

Benchmark	Description
Sdet	Multi-User software development environment from the SPEC SDM benchmark suite.
Laddis	*NFS* file server: basis of the SPEC system-level File Server (SFS) benchmark suite.
Netperf	TCP/IP benchmark for system communication performance.
TPCB	Transaction Processing Performance Council-B benchmark; users connected in client-server configurations, data base server traced.
Abaqus	Structural dynamics analysis tool (**HKS** Inc.)
G92	Computation Chemistry Code (**Gaussian** Inc.)
MM4	Local-Weather model (**Pitt/NCAR**)
Spice	Electronic circuit simulation (from SPEC92, kernel activity not traced).

Table 1: Workloads Focus

Benchmark	% Branches	%Branches Taken	Avg Seq Block Size	% Instrs in OS
Sdet	17.8	66.5	8.4	50
Laddis	18.9	68.7	7.7	100
Netperf	18.6	66.2	8.1	97
TPCB	16.7	68.1	8.9	42
Abaqus	23.9	97.3	4.3	7
G92	39.6	97.6	2.6	0
MM4	8.5	74.4	15.8	4
Spice	16.8	39.7	15.0	0

Table 2: Workload Characteristics

Program	I-Refs (Mil.)	D-Refs (Mil)	L1-I MR	L1-D MR	L2-I MR	L2-D MR
Sdet	32.1	11.1	.067	.102	.001	.010
Laddis	32.5	11.2	.101	.203	.001	.009
Netperf	32.9	11.0	.145	.153	.001	.017
TPCB	31.5	11.4	.130	.198	.002	.027
Abaqus	77.4	26.3	.054	.188	.001	.028
G92	58.8	19.2	.041	.170	.003	.013
MM4	31.8	9.7	.026	.412	.004	.054
Spice	37.1	11.9	.0001	.144	.000	.001

Table 3: 8KB Cache Reference Counts and Miss Rates

Table 3 provides a summary of the number of instruction and data references in each trace and the miss rates obtained with the memory subsystem described above.

5 Performance Comparison

Most of the physical parameters influencing the performance of the Markov predictor have been specified in the simulation environment. However, there are two important parameters that simplify the implementation of a "pure Markov" model. A node or state in a pure Markov model can have an arbitrary fan-out, and each outgoing edge has a transition probability that indicates the likelihood of moving to a particular next state. This transition probability is used to prioritize memory prefetches.

We choose to limit the fanout for each state in the prediction table and to approximate the transition probabilities using an LRU mechanism. Figure 5 shows the effect of varying the maximal fanout for one, two, four and eight prefetch address predictors. This graph shows both the prediction accuracy and coverage on a single axis. The vertical axis is the percentage of cache misses normalized to the same application and cache organization using a demand-fetch organization. Four bars, indicating the different configurations being considered, are shown for each application. Each bar has three components. The lower component represents the *coverage*, or the fraction of miss references that were prefetched and then used by the processor. Larger values are better, but never exceed 100% of the normalized miss references. The middle component represents the fraction of miss references that were not satisfied by prefetch references and had to be demand-fetched. The upper component represents the fraction of miss references that were incorrectly predicted and result in wasted bandwidth. Smaller values are better for this component. The upper component indicates the *accuracy* of the prefetcher, because a more accurate prefetcher would fetch fewer references that were not used.

Clearly, the accuracy decreases and the coverage increases as more prefetch address predictors are added, because *every* predicted address can be fetched fetched when a matching parent key is found in the prefetch table. A larger number of prefetch addresses results in a considerably decreased accuracy with little improvement in coverage. For example, when the TCPB benchmark is simulated, the prefetcher fetches twice as many wasted cache lines when using eight rather than four prefetch predictors, but only increases the coverage by $\approx$10%. In most applications that we examined, four prefetch predictors provided a reasonable balance between coverage and accuracy for the data cache, and we use that configuration for the remainder of the paper. Only two prefetchers were needed for the instruction cache, because there are fewer successors for a given instruction reference.

Since the performance of the Markov prefetcher is dependent on the size of the prefetcher tables, we also investigated a two-tiered allocation strategy, where records are initially allocated to a table with two prefetch address predictors. If more than two addresses are needed, the entry is moved to a separate four-entry table. Although we do not report the results here, this design afforded better performance for a given table size at the expense of a slightly more complex design.

As mentioned, *every* predicted prefetch address is fetched when the parent key is found in the prefetcher. To improve performance, individual references are prioritized based on the likelihood of satisfying the request. We considered two methods to prioritize the references. In the first, we used the true transition probability and in the second, we used a simple LRU algorithm. The LRU algorithm out-performed the true transition probability in every application, and is much easier to implement. Table 4 shows the reference fre-

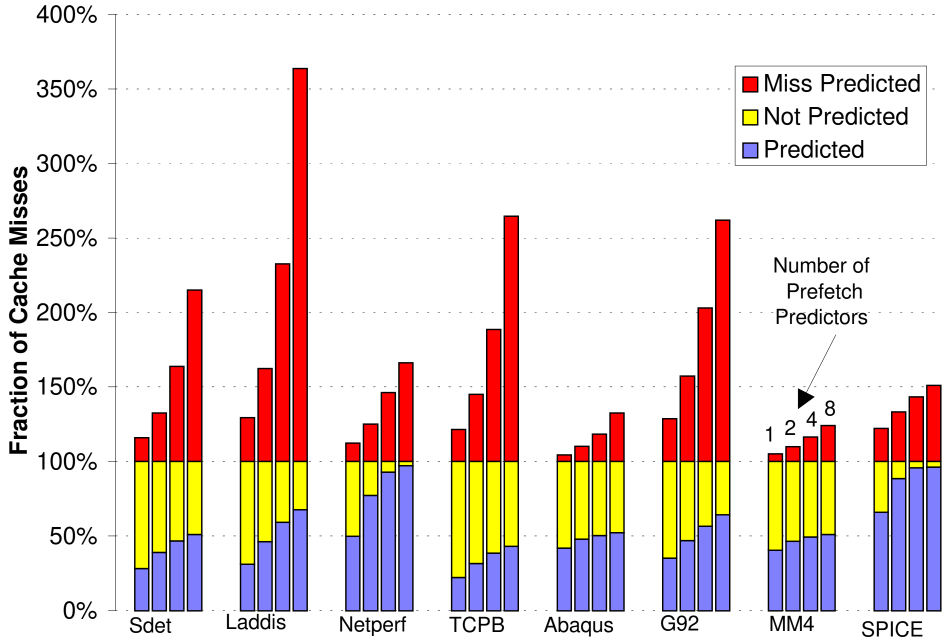


Figure 5: Changes in prefetch accuracy and coverage for the Markov prefetcher as the number of prefetch addresses is increased. Each bar shows the normalized percentage of L1 data cache misses when using one, two, four or eight prefetch address predictors.

Benchmark	Address Predictor			
	0	1	2	3
Sdet	0.342	0.200	0.13	0.097
Laddis	0.344	0.239	0.172	0.123
Netperf	0.754	0.335	0.157	0.107
TPCB	0.511	0.397	0.238	0.205
Abaqus	0.756	0.227	0.103	0.066
G92	0.501	0.256	0.153	0.116
MM4	0.691	0.202	0.093	0.062
Spice	0.606	0.262	0.082	0.034

Table 4: Reference frequency for each data cache prediction address entry for different applications

Benchmark	Address Predictor			
	0	1	2	3
Sdet	152	313	439	581
Laddis	33	56	90	123
Netperf	46	129	320	560
TPCB	34	72	140	204
Abaqus	54	272	486	939
G92	226	431	584	1,264
MM4	97	824	991	1,546
Spice	41	76	230	418

Table 5: Reference lead time for each data cache prediction address entry for different applications.

quency for each predictor address in a Markov prefetcher with four prediction addresses. The columns represent the relative age of the predictor reference and predictor "0" is always the most recently used address predictor. For example, 34.2% of the references correctly prefetched for the Sdet application were predicted by the first predictor address. The large difference in reference frequency for the first and second predictors indicates that an LRU policy is very effective.

The LRU prioritization also orders the data to improve the *lead time*, or the time until the data is needed in our model memory simulator. Table 5 shows the average number of cycles between the time a prefetch is issued and when it is used, for those prefetch requests that are actually used. Larger values indicate a longer "lead time", meaning there is more time to actually provide the data to the processor. In each case, the lead time increases from the most recently used reference (the first column) to the least recently used (the last column). This indicates that the LRU prioritization method not only requests the most frequently needed items, but that it also requests those that are needed soonest.

We compared the performance of the Markov prefetcher using four prefetch addresses and an LRU prioritization policy against other prefetch mechanisms. Figure 6 shows the normalized memory transfers for the different methods. From left to right, each column shows the performance for: stream prefetching, stride prefetching correlation prefetching, Markov prefetching, stride, stream and Markov in parallel and stride, stream and Markov in series. The stride, stream and correlation prefetchers are simulated as described in §3.

Figure 6 shows that stream prefetchers provide better coverage than stride prefetchers, but do so by using considerably more bandwidth. The correlation prefetcher (column 3) provides still better coverage and the Markov prefetcher (column 4) provides the best coverage over the set of applications. However, as with stream buffers, this increase in coverage comes at the price of in-

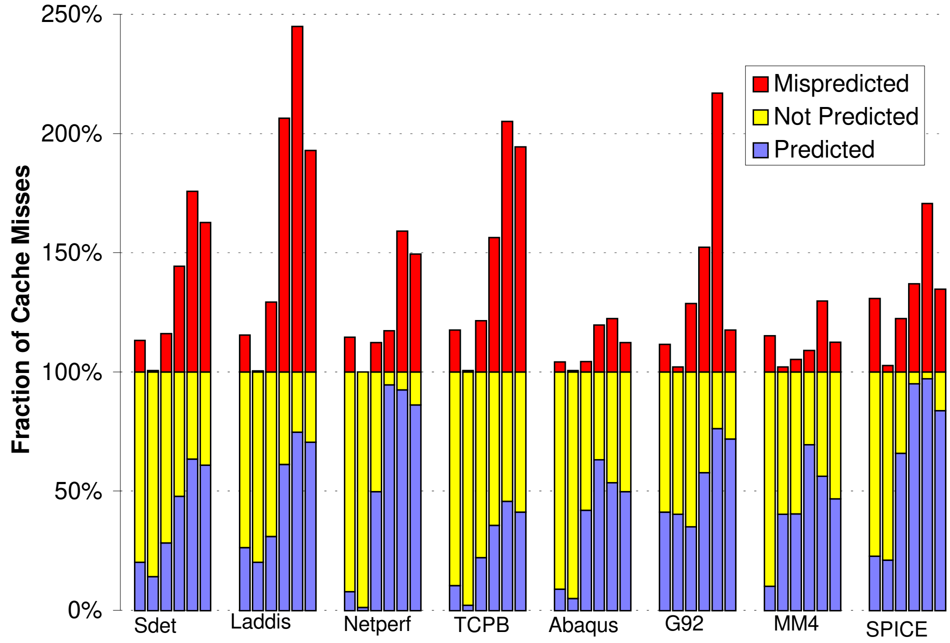


Figure 6: Simulation-based study comparing prefetcher accuracy. From left to right, each column shows the performance for: stream prefetching, stride prefetching correlation prefetching, Markov prefetching, stride, stream and Markov in parallel and stride, stream and Markov in series.

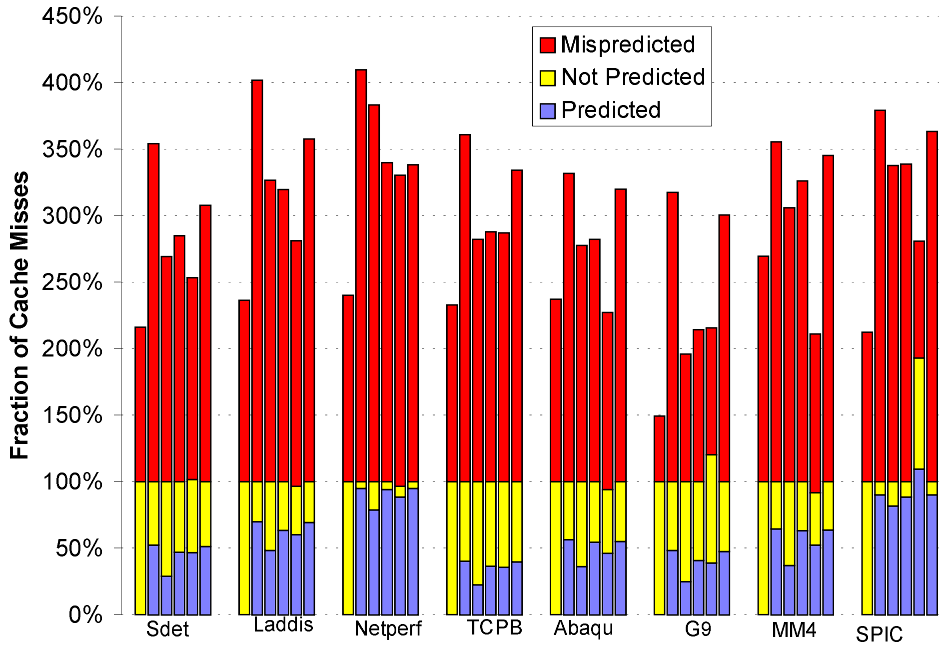


Figure 7: Memory reference overhead for different bandwidth reduction techniques. In this figure, the vertical axis shows the the fraction of *words* transfered, rather than cache lines. In the group for each program, the first bar is for demand fetching (no prefetched), followed by Markov prefetching, Markov prefetching with noise rejection, accuracy-based adaptivity and CNA-based adaptivity. The last bar shows the memory references if the L1-cache is queried prior to prefetching.

creased memory bandwidth demands, often five-fold that of the stream buffers. The last two columns indicate alternative designs that attempt to either improve coverage or improve accuracy. In the fifth column, we use a combination of stride, stream and Markov prefetchers in parallel. This typically results in a large increase in mispredicted references that waste bandwidth, but with a modest improvement in coverage. At times, the coverage actually *decreases*. When the predictors are used in series, the coverage improves again, but by a smaller amount. Likewise, the mispredictions increase, wasting more bandwidth. In part, this occurs because references handled by the stream and stride mechanisms are not used to "train" the Markov predictor, since they are not misses. A more important factor in the "parallel" configuration is interference and contention for the prefetch buffers. For example, both the stream buffer and the Markov prefetcher may predict a specific, yet different address. These differing predicitons must contend for a small number of prefetch buffers and the limited memory bandwidth.

5.1 Limiting the Prefetch Bandwidth

In general, the increased need for bandwidth provides the greatest limitation to effective prefetcher implementations. The additional demand increases contention for the processor bus, possibly blocking demand fetches or later prefetched requests. We examined four techniques to improve the accuracy of the Markov prefetcher and thus reduce the bandwidth consumed by prefetching. Figure 7 shows the performance of these techniques using a variant of the previous presentations. In this figure, we show that fraction of sub-block cache misses, where each sub-block is a four-byte word. In the group for each program, the first bar is for demand fetching, where no prefetching is used. Figure 7 indicates that $\approx$50% of the bandwidth for demand-fetched references is wasted, because the part of the data stored in the 32-byte cache line is not references. The second bar shows the same information for Markov prefetching, using four addresses predictors and an LRU prioritization policy. The third bar shows the bandwidth reduction that occurs when a simple "noise rejection" filter is used. This filter was proposed by Pomerene and Puzak [15] and also examined by Charney and Reeves [3]. It is similar to the filter used by Kessler [14] to improve the accuracy of stream buffers – a prefetch request is not dispatched until the prefetch pattern has been seen twice. The fourth column shows the performance for *accuracy based adaptivity*. In this scheme, two bit saturation counters are added to each prediction address, and a link back to the prediction address that was used to make a prefetch is added to each prefetch buffer entry. When a prefetch is discarded from the prefetch buffer without ever being used, the corresponding counter in the prediction address is incremented. When a prefetch in the prefetch buffer is used, the corresponding counter is decremented. When the sign bit of the counter is set, the associated prefetch address is disabled. Prefetch requests from disabled prefetches are placed in a "pseudo-prefetch buffer" so the predictions can be compared against actual prefetch requests, allowing a "disabled" prefetcher to become enabled once again. The fifth column shows the effect of using an automatic *cache-no-allocate* (CNA) policy to reduce the portion of the cache line that is actually placed in the prefetch buffer. This design is based on the CNA mechanism of Tyson *et. al.* [16]. As in [16], we use a table of two bit counters to determine if a specific memory instruction should use a CNA policy. The prefetcher records information both about the predicted address and the instruction predicted to issue the memory reference. If that instruction is marked CNA, only the requested word, rather than the full cache line, is prefetched. The last bar in each group shows the effect of querying the L1-cache prior to prefetching. Some number of prefetch requests are already valid in the cache, but the external prefetch mechanism may not know that unless an external shadow tag table is used.

Of these techniques, only querying the L1 cache can be implemented as a completely external subsystem. The others require modifications to the prefetch mechanisms or interaction between the prefetcher and the processor. In general the CNA-based adaptivity provides the greatest performance improvement, but the CNA mechanism isn't directly applicable to the instruction cache. Instruction references tend to use most of a cache line, and techniques that either accept or reject and entire cache lines (such as the accuracy-based adaptivity) work better than the CNA method, which prefetches only a portion of the cache line. Figure 7 shows that the CNA mechanism changes the number of cache misses used as a "baseline" for some programs (G92, MM4 and Spice). The CNA mechanism improves the basic cache performance for MM4, but decreases the performance for G92 and SPICE. This is an artifact of the CNA bandwidth reduction technique [16].

5.2 Comparison Using a Memory-system Simulator

We have seen that the Markov predictor provides better prefetch coverage than the other prefetchers we examined. We used a memory-level simulator to determine if that improved coverage resulted in better performance for the memory subsystem. Our memory simulator models the contention for the system resources shown in Figure 1. In particular, we wanted to insure that the additional bandwidth consumed by the Markov prefetcher did not reduce the overall memory performance.

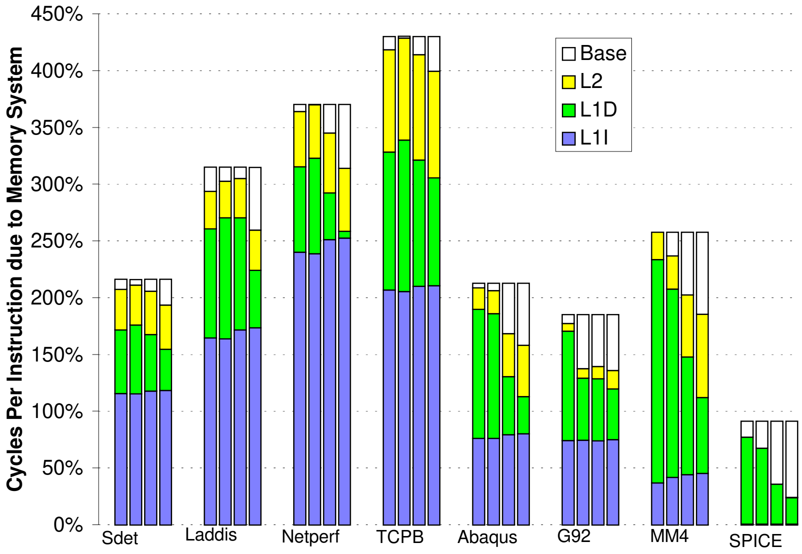
Figure 8(a) shows the memory CPI when different prefetchers are used for only the data cache and Figure 8(b) shows the memory CPI when prefetchers are used for the instruction and data caches. In each case, we used the accuracy-based adaptivity bandwidth filter for each Markov prefetcher. The correlation prefetcher uses the filter suggested by Charney and Reeves, and the stream buffers use the filter proposed by Kessler. From left-to-right, the individual bars indicate the performance of stream, stride, correlation and Markov prefetchers. The segments of each bar indicate the fraction of the overall MCPI attributed to individual parts of the memory system. From bottom to top, the segments indicates the delay due to fetching instructions into the L1 cache, fetching data into the L1 cache, fetching data into the L2 cache and the additional delay incurred by a pure demand-fetch model.

The Markov prefetcher provides the best performance across all the applications, particularly when applied to both the instruction and data caches. Recall that the correlation and Markov prefetchers require *fewer* resources than the stream and stride buffers – the Markov prefetcher used a 2MByte cache and a 1MByte prefetch table, *vs.* the 4MByte cache used in the demand-fetch model.

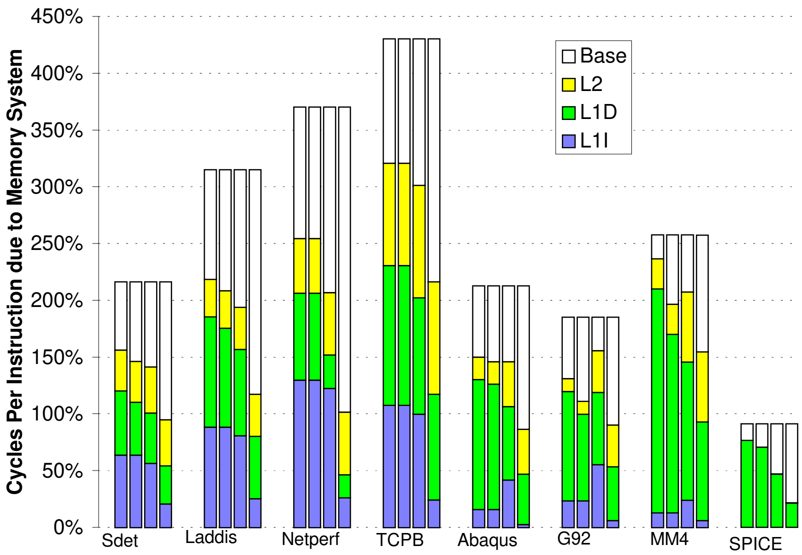
6 Conclusion

We have presented the Markov prefetcher, and simulated its performance over a number of significant applications. Although the Markov prefetcher can be characterized as an extension to the correlation prefetcher previously described by Pomerene and Puzak [15] and Charney and Reeves [3], there are a number of design issues presented by the Markov prefetcher. The prefetcher must be able to launch multiple prefetch requests and prioritize them, and the prefetcher must consider some mechanism to limit the bandwidth devoted to prefetching.

Our study has shown that a simple, effective and realizable Markov prefetcher can be built as an off-chip component of the memory subsystem. The design we present takes fewer resources than a normal demand-fetch cache, yet reduced the memory CPI



(b) Data and Instruction References



(b) Data and Instruction References

Figure 8: The average number of CPU stalls attributed to the memory subsystem (MPCI) when using different prefetchers for just the data references and both data and instructions. Across each application, the vertical bars indicate the MCPI for a demand-fetch model. From left-to-right, the individual bars indicate the performance of stream, stride, correlation and Markov prefetchers. The segments of each bar indicate the fraction of the overall MCPI attributed to individual parts of the memory system.

by an average of 54% over the applications we examined. The Markov prefetchers we examined provide better prefetch coverage and more timely prefetches than other prefetchers at the expense of reduce accuracy. However, the combination of bandwidth reduction filters and prioritization of the prefetch requests makes the Markov prefetcher the most effective prefetcher we examined.

There are a number of ways the Markov prefetcher can be improved. In particular, both the Markov and correlation prefetchers must observe a reference sequence prior to predicting references. It would be desirable to abstract a *reference pattern* from several instances of reference sequences. Likewise, it would be useful to have some mechanism to issue prefetch requests for patterns or sequences that have not yet occurred, eliminating the training period needed by the Markov predictor.

References

[1] T. Alexander and G. Kedem. Distributed predictive cache design for high performance memory system. In *Second International Symposium on High-Performance Computer Architecture*, Feb 1996.

[2] J.L. Baer. Dynamic improvements of locality in virtual memory systems. *IEEE Transactions on Software Engineering*, 2, March 1976.

[3] M.J. Charney and A.P. Reeves. Generalized correlation based hardware prefetching. Technical Report EE-CEG-95-1, Cornell University, Feb 1995.

[4] T.F. Chen and J.L. Baer. Reducing memory latency via non-blocking and prefetching caches. In *ASPLOS-V*, pages 51–61, Oct 1992.

[5] Z. Cvetanovic and D. Bhandakar. Characterization of alpha axp using tp and spec workloads. In *21th Annual International Symposium on Computer Architecture*, pages 60–70, April 1994.

[6] K.I. Farkas and N.P. Jouppi. Complexity/performance tradeoffs with non-blocking loads. In *21th Annual International Symposium on Computer Architecture*, pages 211–222, April 1994.

[7] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully associative cache and prefetch buffers. In *17th International Symposium on Computer Architecture*, May 1990.

[8] A.C. Klaiber and H.M. Levy. An architecture for software controlled data prefetching. In *18th International Symposium on Computer Architecture*, May 1991.

[9] M.H. Lipasti and et.al. Spaid: Software prefetching in pointer and call intensive enviornments. In *Proceedings of 28th Annual International Symposium on Microarchitecture*, pages 231–236, Nov 1995.

[10] A.M. Maynard, C.M. Donnelly, and B.R. Olszewski. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. In *ASPOS-VI*, Apr 1994.

[11] S. Mehrota and H. Luddy. Examination of a memory access classification scheme for pointer-intensive and numeric programs. Technical Report CRSD tech report 1351, CRSD University of Illinois, Dec 1995.

[12] T.C. Mowry, M.S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *ASPLOS-V*, pages 62–73, Oct 1992.

[13] T. Ozawa and et.al. Cache miss heuristics an preloading techniques for general-purpose programs. In *Proceedings of 28th Annual International Symposium on Microarchitecture*, pages 243–248, Nov 1995.

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

[15] J. Pomerene and et.al. Prefetching system for a cache having a second directory for sequentially accessed blocks. Technical Report 4807110, U.S. Patent Office, Feb 1989.

[16] Gary Tyson, Matt Farrens, and Andrew Pleszkun. A modified approach to data cache management. In *Proceedings of 28th Annual International Symposium on Microarchitecture*, pages 93–105, Nov 1995.

[17] Z. Zhang and T. Torrellas. Speeding up irregular applications in shared memory multiprocessors: Memory binding and group prefetching. In *22th Annual International Symposium on Computer Architecture*, pages 1–19, June 1995.

A.Benchmark Descriptions

The System Evaluation Corporation (SPEC SDM) suite includes two benchmarks: Sdet and Kenbus. Sdet is intended to represent a general purpose computing environment. A large number of users run concurrently with commands separated by little or no think time. Commands executed by the users include: edit, compile, bind, ksh, grep, ls, cp, rm and mkdir. The benchmark is run with multiple user levels to show throughput as a function of number of users. The benchmark is traced consists of 64 users and includes kernel, user and shared library activity.

The Laddis benchmark measures the capacity of a file server running NFS. It is essentially a re-packaging of the NHFSSTONES benchmark created by Legato Systems. It runs by having one or more client workstations submit NFS requests to the server. The requests must be completed with an average response time of 50 milliseconds. The benchmark was run using a single client with a load of 20 transactions per second, and six processes submitting requests.

The Netperf benchmark measures the performance of system communications. It important because it mimics interactive network environments, such as telnet, where response time is critical. As currently implemented Netperf provides a simple representation of interactive workloads and data xfers. The benchmark traced consisted of a single machine sending packets to it self. In this scenario a single one byte packet was "ping-ponged" between two processes 10,000 times. The metric was the latency of the transfers in elapsed time.

The Transaction Processing Performance Council (TPC) benchmarks are the best known and most frequently cited measures of commercial performance. The TPC is a collection of representatives from many system and database vendors. At the time of this writing the TPC has released three benchmarks called TPC-A, TPC-B and TPC-C, along with rules for running them and for reporting results. Although some simulations in this thesis research were executed with all three benchmarks, only one was selected for every test case: TPC-B. TPC-B simulates a banking application where users submit many simple debit or credit transactions from automated teller machines. The system is expected to respond within two seconds. The number of users and size of the database are scaled with the reported performance metric: transactions per second (TPS). For example, a system which reports a TPS rating of 80 would have a database roughly eight times as large as a system with a rating of 10. The benchmark used in this thesis was traced with 60 users in a client-server configuration.

Abaqus is a structural dynamics analysis product, like NASTRAN and ANSYS. This particular application is from the set of timing runs provided by HKS, Inc.. It is described as: "This is a large linear analysis consisting of a 40x40 mesh of shell elements of type S8R5. The total number of degrees of freedom is 39366; the maximum wavefront is 522 and the RMS wavefront is 503. Most of the time is spent in the inner loop in the solver, so that computers with fast vector processors should run this problem quickly."

G92 is a computational chemistry application from the Gaussian '92 Application Test Suite from Gaussian,Inc. Since the runtime of this test is about 8 minutes and consists of many processes (none running concurrently with the other), the tracing effort concentrated on user mode only of the top two processes: l703.exe and l1002.exe. These processes accounted for over 70% of the total runtime and almost 77% of the total user-mode runtime.

MM4 is a local-weather model owned by NOAA (National Oceanic and Atmospheric Administration). It uses Navier-Stokes equations to solve a finite element grid.

Spice is a floating point intensive electronic circuit simulation benchmark from the SPEC92 suite. Only the user activity was traced for this benchmark.