Threshold-Based Markov Prefetchers

Carlos Marchani **Tamer Mohamed** **Lerzan Celikkanat** **George AbiNader**

Rice University, Department of Electrical and Computer Engineering
ELEC 525, Spring 2006

Abstract
In this report we present a novel technique for a Markov prefetcher designed to hide memory latency. This prefetcher makes use of the information about previous cache misses in a memory trace and effectively predicts future misses with high accuracy and coverage rate. The analysis of the memory traces of the executed benchmarks reveals that our threshold-based prefetcher poses an optimized memory bandwidth overhead while not hurting the prefetcher coverage and effectively increasing its accuracy. Our data indicate that the average loss in coverage is 1% while the gain in accuracy is 13% and the reduction in memory bandwidth overhead is 39%. The design also achieves more than an order of magnitude reduction in mispredictions as compared to previous results.

Keywords
Hiding Memory Latency, Markov Tables

BACKGROUND
As modern superscalar processors get higher clock speeds and deeper pipelines the cost of memory latency becomes an increasing burden on the system performance. Prefetchers are a mechanism that aims to hide the memory latency in modern processors by fetching data from memory before the processor actually requests it. The data is typically fetched following a model pre-established by the system designer and it is stored in dedicated buffers.

Several researchers have worked on different models for a prefetcher design, including compiler-dependant prefetchers, stride prefetchers, stream buffers and correlation- based prefetchers. Joseph and Grunwald studied Markov prefetchers as a model for correlation prefetchers [1]. In their work, they considered and simulated the above prefetchers in addition to a Markov model. They found that the Markov prefetcher provided the best performance for the SPEC95 applications that were considered. Nonetheless, they recognized the need to reduce the bandwidth usage and the possibility to improve the prefetcher accuracy.

The main idea behind utilizing a Markov table in a prefetcher is to account for the probabilities of transitions from one state to another state. In this context, a state refers to a certain miss address following a given miss address. The frequencies of transition are used to populate a table as shown in table 1, and these frequencies are utilized as a measure of probability for the prediction of future transitions from a reached state. The table is continuously being updated and a stride prefetcher is also used in order to enhance the performance of benchmarks that exhibit a lot of strided accesses.

Table 1: A Markov table populated by the transitions and frequency of transition occurrence for the sequence of states (cache misses): A, B, C, D, C, E, A, B, C

Miss Address (current state)	Next Miss (state)		
A	B [2]		
B	C [1]		
C	D [1]	E [1]	
D	C [1]		
E	A [1]		

HYPOTHESIS
In [1], the authors implemented a Markov design based on the miss occurrences of the L1-Cache; they built a Markov tree based on the next reported misses (states) of a given miss. In fact they would always track and prefetch next four states of a miss. We propose to create and simulate a more dynamic model that adapts to the frequency of occurrence of next states. The prefetcher would only prefetch among the tracked next states theses misses that exceeded in the past a given threshold of appearance. In our opinion, prefetching misses based on past frequency of occurrence using our dynamic threshold would improve prefetcher accuracy and reduce the memory bandwidth.

Our solution assumes that the probability of occurrence of a miss is correlated to the history of occurrence of that miss. In other words, if “miss B” occurred after “miss A” half the time in the past then it has a 50% chance of occurring now and thus is useful to prefetch.

To confirm our assumption we examined the data-L1 miss stream of several SPEC2000 benchmarks and found good evidence to our intuition. Figure 1 shows a sample of the behavior that proves the viability of our hypothesis. In the

VPR benchmark, the frequency of occurrence of misses grows as the program progresses, i.e. the misses that appeared in the past are likely to appear in the future. Their relative frequency of occurrence is more likely to stay constant or change slowly throughout the program's life than to change behavior drastically.

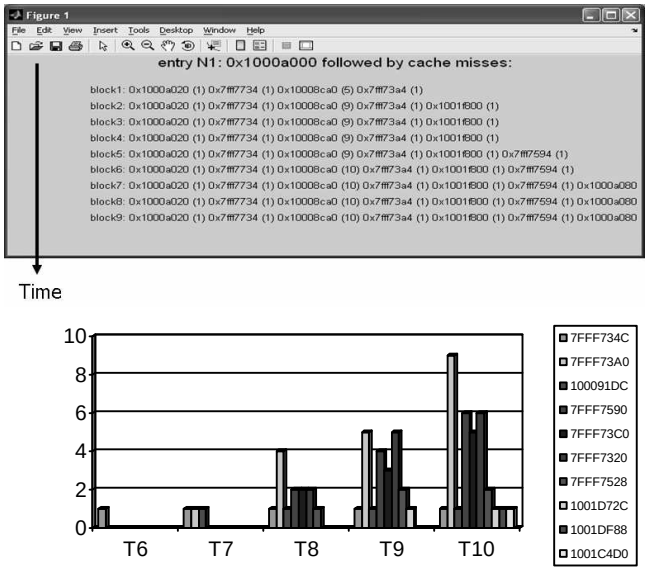

Figure 1: VPR sample next misses from L1 miss stream shows the number of occurrences of "next-miss" at progressive time periods of the program.

Early examination of the miss stream confirmed several observations from previously published results [1]. The most notable is that some benchmarks –like mcf- will benefit little from a Markov prefetcher alone; as most of their cache misses exhibit a strided access behavior and not a correlation between previous history and the current time instances. The abundance of the strided access thus overwhelms the number of misses that can benefit from a correlation prefetcher. For that reason we decided to place a stride prefetcher in series and ahead of the Markov prefetcher.

In this report, we did not consider timeliness and we did not run any timing simulations on the data. On the other hand we gauge memory bandwidth usage by looking at the number of memory references initiated by the Markov prefetcher. In addition, having implemented the Markov prefetcher following a stride prefetcher we will measure coverage and accuracy for the Markov independently.

In this paper we will use the same terminology established in [1] to evaluate our prefetcher, namely we will use the two following metrics:

- The coverage defined as the fraction of memory references that were supplied by the Markov prefetcher and not demand-fetched or supplied by the stride prefetcher.

$$\text{Coverage} = \frac{\text{hits in prefetch buffer}}{\text{total cache misses}}$$

- Accuracy is defined as the fraction of Markov prefetched cache lines that were actually used by the processor.

$$\text{Accuracy} = \frac{\text{unique hits in prefetch buffer}}{\text{total prefetches}}$$

The definition for accuracy has a main deviation from the definition of the coverage in that it uses a different metric for the number of hits in the prefetch buffer. In our experiments we found that a single prefetched entry in the prefetch buffer may get used several times and this caused the calculated accuracy to be above 100% and thus we modified the definition to account for the effect of mispredictions which limits the accuracy to be always less than 100%.

- The definition of misprediction used in [1] and the previous reports is given as the number of useless prefetches to the number of cache misses and this value can be in excess of 100%.

$$\text{Misprediction} = \frac{\text{total prefetches} - \text{unique hits in prefetch buffer}}{\text{cache misses}}$$

In our project we were analyzing the performance of the prefetcher on the memory accesses of the data stream. A similar arrangement can be implemented for the instruction stream as well.

ARCHITECTURE

We assume a conventional modern architecture for the cache structure of the processor. We have separate data and instruction L1 cache; we will vary their size from 4 to 32 KB- and a unified 1MB L2 cache. In addition, we place two buffers at the same level as L1, between L2 and the processor: the stride prefetcher which holds four cache lines of 32 bytes each and the Markov prefetch buffer which holds 32 lines of cache. These two additional buffers will be dedicated to data memory references only.

Feeding the two buffers will be the stride prefetcher and the Markov prefetcher. The stride prefetcher looks for patterns of strided accesses to memory in the miss stream of the L1, when such patterns are detected, a demand is placed to the L2 to pass the data to the prefetcher the data into the stride buffer, such a demand would likely occur later when the memory reference actually misses in the L1 cache. If no strided access is identified for a given miss, it is passed to the Markov prefetcher. The Markov table will use that entry to build the next state tree. If the miss follows a previously known miss it would be recorded as a next state or increment the corresponding next state for that pre-miss. Also if the recent miss has a tree of next states that follow it, a demand for some or all of these next states will be passed to L2.

In [1], the authors decided to always prefetch the next 4 states, in our experiment we will dynamically decide on the number of states to prefetch. In fact we will keep track of the history of up to 16 next states but only fetch those whose frequency of occurrence has exceeded a given threshold. In this architecture, the prefetchers and L1 compete for service from L2. The L1 requests are assumed to always be given priority over the prefetcher requests because they are imminent and certain. The prefetch requests are future and speculative as they are trying to guess an upcoming miss that may not occur. When the miss events are spread out in time, the prefetchers have a better chance of being serviced by the L2 and of being effective at getting data from memory if needed in a timely matter. We have not considered the timeliness aspect in our paper and assume that the prefetchers handle the predicted cache misses sufficiently ahead in time of their actual occurrence.

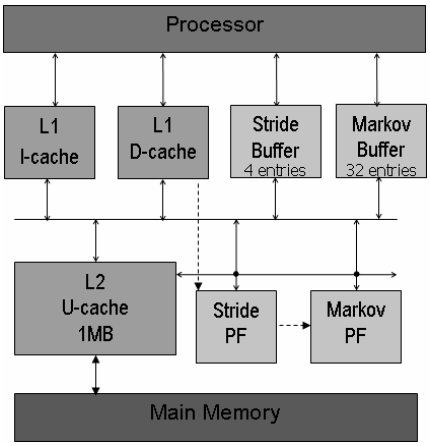

Figure 2: System Design

Table 2: Configuration parameters for the simulator

	Size KB				Rpl	Assoc.
L1-Cache-Data	4	8	16	32	LRU	direct
L1-Cache-Inst	4	8	16	32	LRU	direct
L2-Cache-Uni	1024				LRU	4-way
TLB-Inst	256				LRU	4-way
TLB-Data	512				LRU	4-way

EXPERIMENTAL METHODOLOGY

For our simulations, we used SimpleScalar 3.0 to run the following SPEC2000 benchmarks: mcf, equake, vpr, parser, and gzip. Since we were only interested in the cache miss stream for the L1 data cache, we modified the file cache.c and recompiled simplescalar such that it would dump the data-L1 miss addresses. We then ran sim-cache on the benchmarks and redirected the output to a memory trace file. In running the benchmarks, we used input data sets provided in the ELEC525 class directory on owlnet; the SPEC_2000_REDUCED folder. We stopped the simulations when we have collected information in excess of 6Megabytes which is roughly equivalent to the first 500000 L1 data cache misses that occurred in the benchmark.

We ran simulations for different L1-cache sizes, 4KB, 8KB, 16KB and 32KB. Although we are only interested in the data misses, both the data L1 and the instruction L1 were modified to have the same size. On the other hand we did not modify the size of the unified L2 cache for any of the runs. The complete parameter configuration that we used for simplescalar is listed in Table 2.

We then used the collected miss-stream information to analyze the effectiveness of our dynamic prefetcher. The prefetcher simulation code mimics the behavior of the cascaded Stride/Markov prefetcher described above. Part of the code is responsible for collecting data about the performance of the Stride and Markov prefetchers in order to calculate the accuracy and coverage of each prefetcher and the performance of their combination for a given miss stream. The performance data is reported at the end of the simulation run. The following is an example of the output from analyzing 576853 d-L1 cache misses in the “equake” benchmark.

```
Number of L1 Cache Misses = 576853

STRIDE PREFETCH BUFFER
Number of Prefetches from L2 Cache = 70399
Number of Hits in Buffer = 63443
Number of Unique Hits in Buffer = 62756
Prefetch Coverage = 0.109981
Prefetch Accuracy = 0.891433

MARKOV PREFETCH BUFFER
Number of Prefetches from L2 Cache = 86
Number of Hits in Buffer = 291025
Number of Unique Hits in Buffer = 86
Prefetch Coverage = 0.504505
Prefetch Accuracy = 1.000000
```

As mentioned earlier, the number of hits is used to calculate the coverage and the number of unique hits is used to calculate the accuracy and mispredicitons. In the example output shown above, the accuracy is 100%. This is due to the constraint heuristic used in controlling the prefectching. This did not just impact the accuracy but indirectly greatly impacted the coverage and the mispredictions.

The coverage is more than 50% and the misprediction is 0%! Such values are in huge contrast with the values obtained in [1] and previous year ELEC525 reports. This is explained by the fact that the entries in the Markov buffer were utilized more than one time and they did not get

evicted by unnecessary prefetches due to the constraint. We will illustrate this in more detail and with more benchmarks in the experimental analysis section of this report.

HW COST OF THE EXPERIMENTAL PREFETCHER

The main blocks in this system are the two prefetch buffers and the two prefetchers. The following is a break down of the estimated hardware cost to implement each one of them.

The Markov prefetcher has the following configuration parameters and associated hardware cost:

- 128 row entries. This is the number of miss addresses analyzed by the prefetcher.
- Up to 16 next states per entry. (We used 4 in our experiments)
- Counters to count the frequency of occurrence of each next state.
- Comparators to detect if a counter has exceeded a preset threshold. The thresholds we used in our experiments are 3, 5 and 10
- Eviction policy for the row and column entries of the table. In our simulation we used the least recently used (LRU) policy for both rows and columns. We intend in future work to consider random eviction and assess the performance vs. the reduced hardware complexity.

The Markov Prefetch Buffer

- A 32 entry fully associative buffer.
- Utilizes the LRU eviction policy

The Stride Prefetcher calculates the step in a strided memory access and thus has the following cost

- Subtractor current miss address from a previous miss address to get current stride
- Comparator to compare currently calculated stride to previously calculated stride and if they match it begins giving predicted requests to the L2 cache.

The Stride Prefetch Buffer

- A four entry fully associative buffer.
- Utilizes the LRU eviction policy.

The estimated storage requirement for the above HW configuration is 128KB. It is also assumed that it takes one cycle to search the combined prefetch buffers on an L1 miss.

EXPERIMENTAL ANALYSIS

In the following, we present our obtained results for running the benchmarks with the different cache size configurations and with different threshold heuristic. The thresholds used in our experiments are 0, 3, 5 and 10. The zero threshold is used to account for aggressive prefetching which does not make use of our heuristic. We use that as a datum for evaluating the effect on performance when the threshold heuristic is used.

Figure 3 illustrates the combined stride/Markov prefetcher performance in terms of accuracy. The point of comparison is the performance in conjunction with the threshold heuristic. The results for “mcf “ are not affected because “mcf” exhibits a large number of strided accesses and since the stride prefetcher supersedes the Markov prefetcher, almost all the predictions by the stride prefetcher were utilized and the Markov prefetcher was prevented from carrying out unnecessary prefetches. This is further illustrated in Figure 4 and Figure 5 which show the prefetcher coverage due to each prefetcher. The Markov prefetcher almost did not operate at all in the case of the “mcf” benchmark. Figure 6 again compares the coverage obtained with an aggressive zero threshold Markov prefetcher and with the threshold set at 5.

The intuitive expectation is that the coverage would go down and accuracy would go up as the threshold is increased. The data in figure 6 indicate that the system with thresholds has max decrease of coverage of 12 % and an average decrease of coverage of 1% with respect to the system without prefetcher threshold.

While this has been the trend in all the cases, it is worthy to note the performance for cache sizes 4k and 8k in the “equake” benchmark. The large increase in accuracy with the ‘5’ threshold Markov prefetcher actually led to a subsequent improvement in coverage. The average increase in accuracy with the threshold Markov is 13%

In figures 7 and 8, we compare the performance of the system with the Markov prefetcher threshold set at 3, 5 and 10. The intuitive expectation that there will be an increasing trend for accuracy with threshold is not satisfied for all cache sizes. This result indicates that it is in general required to decide the parameters of the Markov prefetcher in conjunction with the cache size since there is a performance dependency. This observation requires further investigation in order to determine optimum thresholds vs. cache sizes.

In figure 9, we illustrate the percentage of requests carried out by the system which has a thresholded Markov prefetcher normalized with respect to the prefetches of a system with unthresholded Markov prefetchers. In this case 100% represents similar usage to the system without threshold. The merit of evaluating this performance metric is to assess the amelioration in memory bandwidth overhead imposed by the prefetcher. The maximum memory request decrease is 77% and the average request decrease is 39%. The results in general are in line with our proposed hypothesis.

The data we obtained for mispredictions are superior by orders of magnitude to that obtained in [1] and previous ELEC525 reports. This indicates that the inhibiting action of the stride prefetcher over the Markov prefetcher has tremendously improved the Markov prefetcher performance metrics. In other words, the Markov prefetcher does not

operate when it is expected that it will degrade the performance. Also the Markov prefetcher operation is dependent on whether the cache miss is satisfied by either prefetch buffer. This led to having the large favorable discrepancy between the number of hits in the prefetch buffer and the number of unique hits. Although this was not intentionally designed in our code, it came out as a pleasant pug.

CONCLUSIONS AND FUTURE WORK

In this report, we examined the construction of a prefetch buffer that hides memory latency and uses a Markov table for the prediction of the memory addresses for fetching. The prefetcher operation is based on a threshold heuristic in order to increase its accuracy and reduce the memory bandwidth overhead due to unnecessary mispredicted prefetches. While previous work has showed mispredictions in the order of 300%~600%, our threshold-based Markov prefetcher exhibited a misprediction penalty less than 10% in most of our test cases. The superior performance of our architecture is due in part to the accuracy of the stride prefetcher that we implemented. In our design, the Markov prefetcher is cascaded in series with the stride prefetcher and the combined buffers are searched in parallel when an L1 cache miss occurs. The threshold heuristic helped improve the accuracy of the prefetcher with a slight 1% degradation in coverage. The resulting improvement in accuracy in some cases actually led to improvement in coverage. The average increase in accuracy was found to be 13%. The threshold heuristic also led to a decrease of memory bandwidth requirement of 39% on the average.

The hardware cost of the design is relatively high due to the implementation of the LRU policy in evicting row and column entries in the Markov table.

For future work, we would like to experiment with other eviction policies that are less hardware costly and assess the performance. A second suggestion for future work involves the storage requirement of the data structures which is estimated to be 128KB. We think that the storage requirements should be in comparison with the size of the L1 cache and thus more experiments need to be conducted in order to get a better balance in storage requirements.

We are also considering developing a functional hardware model using HDL that can be integrated with a processor core running on a prototyping platform. This model will give an accurate identification of the HW cost and the design tradeoffs.

REFERENCES

[1] Joseph Doug, Grunwald Dirk, “Prefetching using Markov Predictors”, Proceedings of the 24th International Symposium on Computer Architecture, June 1997.

[2] Simplescalar Tool, http://www.simplescalar.com/

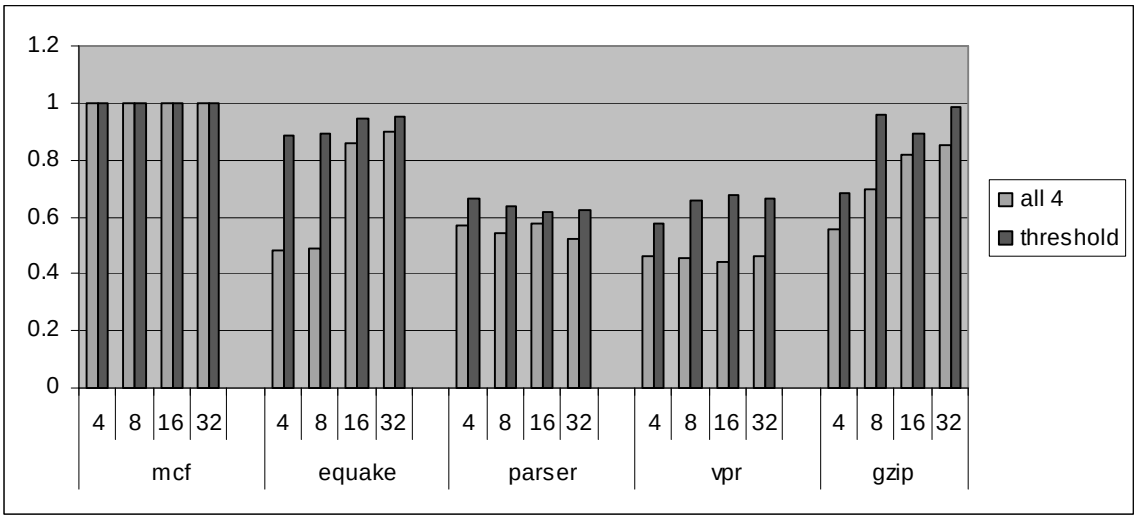

Figure 3: Combined Prefetcher Accuracy with ‘0’ threshold in the Markov prefetcher (getting all expected 4 next entries) vs. getting only entries whose counter crosses the threshold set at 5

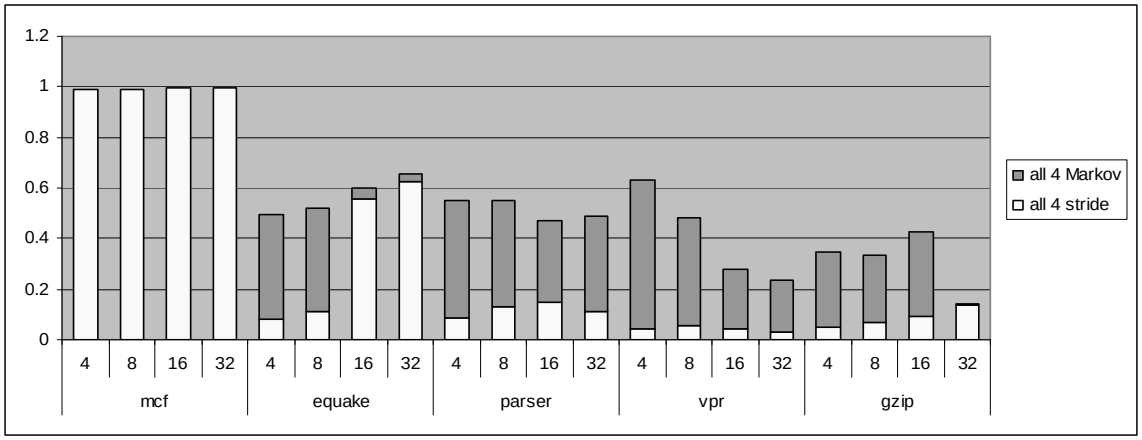

Figure 4: Percentage of coverage due to the stride prefetcher and the Markov prefetcher with the threshold set at '0'

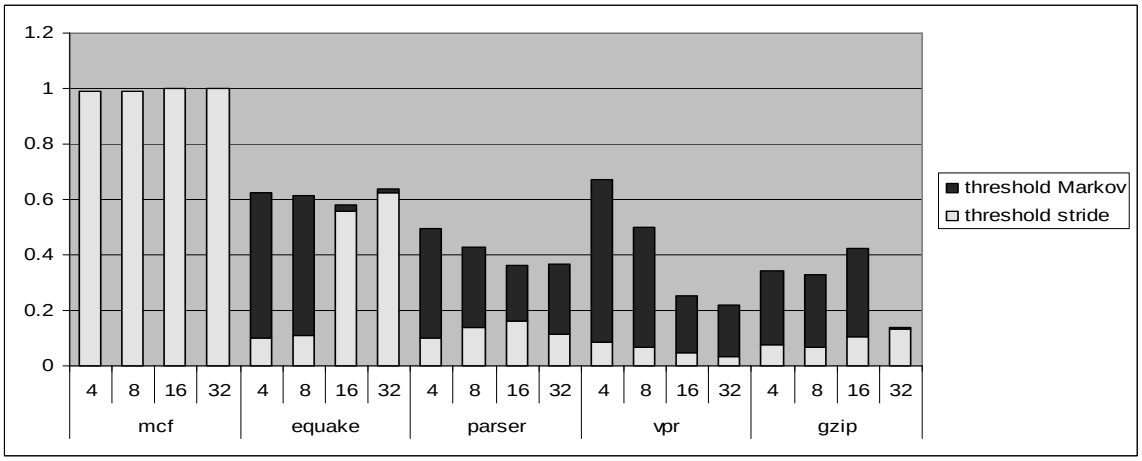

Figure 5: Percentage of coverage due to the stride prefetcher and the Markov prefetcher with the threshold set at '5'

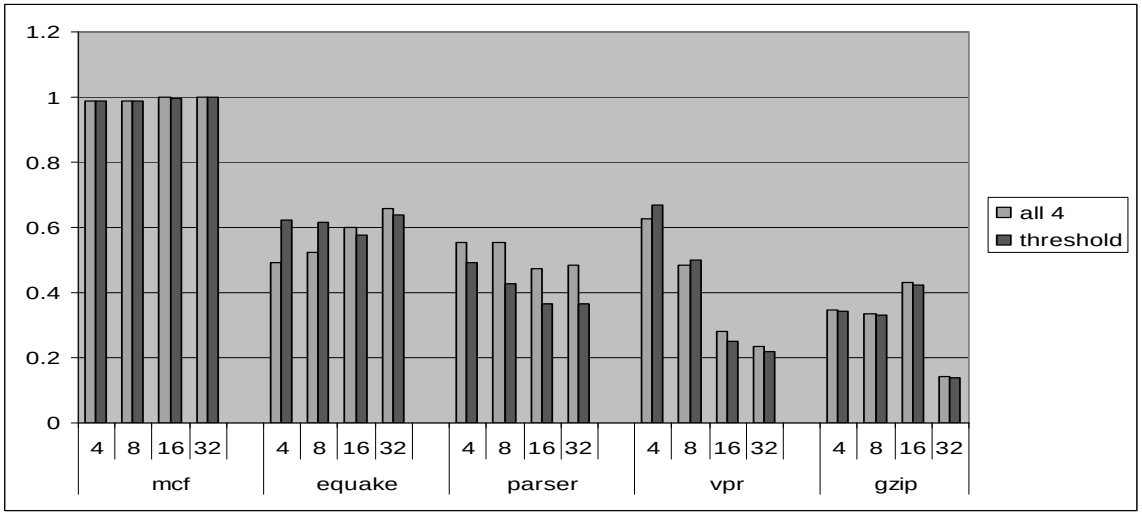

Figure 6: Combined Prefetcher Coverage with the Markov prefetcher thresholds set at '0' and '5'

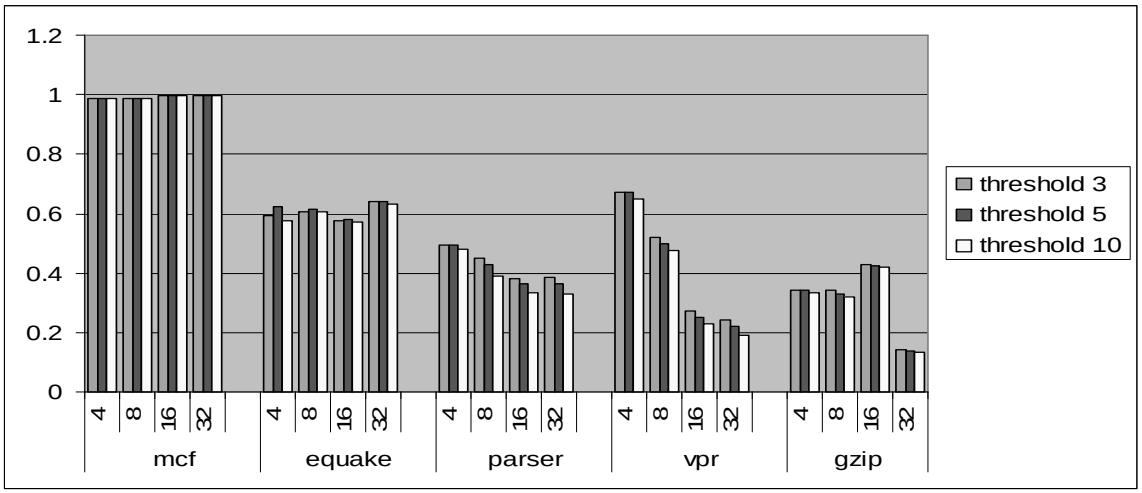

Figure 7: Combined Prefetcher Coverage for different Markov prefetcher threshold levels

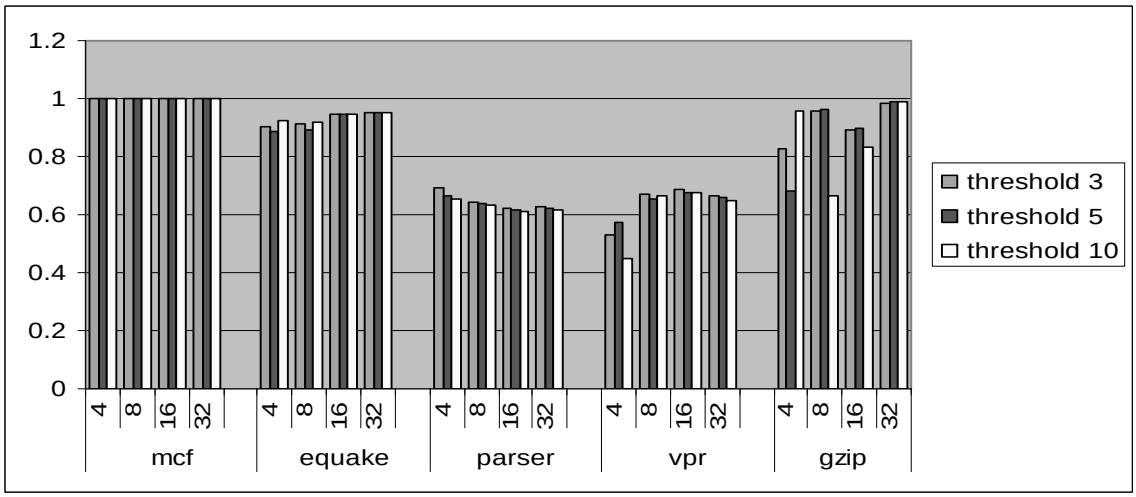

Figure 8: Combined prefetcher Accuracy for different Markov prefetcher threshold levels

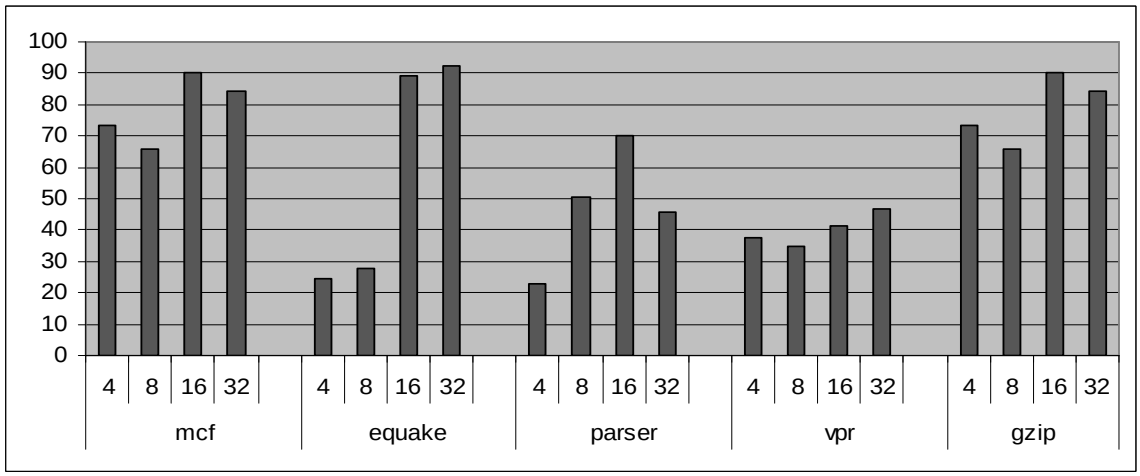

Figure 9: Percentage of L2 requests with Markov threshold set at '5' as compared to 100% being the memory requests when the Markov prefetcher threshold is '0'. A larger L1 cache size makes the number of L2 requests less and a consequently a larger percentage of those is attributed to the prefetcher.