Markov Prediction Scheme for Cache Prefetching

Pranav Pathak, Mehedi Sarwar, Sohum Sohoni
ECE Department Oklahoma state University, Stillwater OK 74074

***Abstract*—Cache prefetching improves hit rates in cache memories. In this paper we are testing the effectiveness of Markov prefetching scheme based on Markov models in order to predict memory references that will cause a miss in L1 cache.**

Our experiments show that Markov history table size of 32 is sufficient and a prefetch buffer size of eight can help achieve hit rates of up to twenty percent in L1 miss stream depending on the locality of reference present in instruction and data set of the application.

***Index Terms*—Markov model, Prefetching**

I. INTRODUCTION

Demand fetching is commonly employed to bring the data from main memory to the processor as and when required for the processor. In contrast prefetching tries to get the data from the main memory *before* processor requests for it. Memories are orders of magnitude slower than processors. Predicting and issuing prefetches for future memory references by the processors is one of the many techniques employed to overcome this difference in the speeds.

In this paper we are trying to test the usefulness of a stochastic Markov model to predict memory block addresses that will be prefetched. Andrei Markov proposed Markov model [1]. Markov property states that, if present state is known then we can predict the future states, irrespective of what the past states were. To put it in other words if present state is known then future and past states are independent. A Markov chain is a sequence of random variables with the Markov property.

Markov predictors find diverse applications in computer science. Optimal resource allocation and higher quality of service is much needed requirement in case of wireless networks. In order to improve the above factors, intelligent prediction of network behavior plays a very important role. Gani[Et.al] show the use of Markov tools to predict the number of wireless devices that are connected to a specific instant of time[2]. Bartels uses Markov based of implementation of prediction by partial matching to learn and predict the access patterns of real applications. Further they us this PPM scheme for adaptive memory prefetching from disk to memory.[3]

When it comes to prefetching cache blocks, Jouppi Et.al [4] introduced stream buffers as a significant method for improved direct mapped cache performance [4]. Doug Joseph Et. al [5] use the ground work laid by Jouppi and show that a simple effective and realizable Markov prefetcher can be built as an off-chip component.

[1]miss history table: It stores history of miss addresses. It records which block occurred after a given block in miss stream from L1 cache. It is a LRU structure. A sample is shown in Table 1

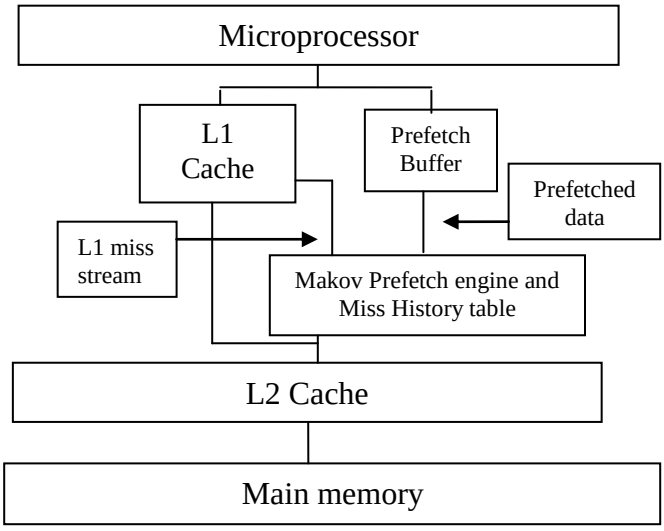

Fig.1 Memory Hierarchy and Markov prefetch engine. This figure shows modified memory hierarchy along with Markov prefetch engine and prefetch buffer.

They compare the performance of Markov predictors with other prefetching schemes like static predictors, indirect stream predictors and correlation based prefetching, stride prefetchers and stream buffers. Comparison of all these models showed that Markov prefetcher is the best choice for prefetching.

We take this idea further by implementing Markov prediction based prefetching scheme and try to find out best possible choices for how large these prefetch buffers should be and how much history should be stored so that Markov prediction model can significantly improve L1 cache miss rates.

In our case the prefetcher has to predict a future memory reference and fetch it from the main memory before the processor actually asks for it. We have the current cache state or dataset used by the processor; if we consider a large number of memory references in order to be able predict the next address then size of miss history table[1] becomes a huge overhead. Hence choosing a model that can predict the future only with present state independent of the future is a good idea. Markov model comes close to satisfying these criteria hence it is a good choice for address prediction.

Fig 1 shows our experimental memory hierarchy that includes Markov miss-history table and a prefetch buffer[2].To implement prefetching one way is to consider all the memory references by the processor. But that way the prefetcher will have to handle multiple addresses every cycle and will be inundated by the number of requests it has to handle.

[2]Prefetch Buffer: It is an on chip buffer that holds blocks of data prefetched by Markov prefetch buffer. It is FIFO structure. When a miss occurs in L1 cache this prefetch buffer is checked concurrently with L2 cache.

Miss Address (Current Miss)	Next Miss Address Prediction (Most frequently used)			
A	B[2]	C[1]	D[1]	E[1]
B	D[2]	A[1]		
C	E[1]			
D	A[3]	E[1]		
E	D[1]	B[1]		

Table 1 Miss History table. Miss history table populated for miss sequence of A-B-D-A-C-E-D-A-B-D-E-B-A-D-A-E.

Also all the prefetcher hardware will be on chip making it even more costly. Instead we decided to prefetch based on stream of misses from L1 cache. This will reduce the complexity of the prefetcher. Fig. 1 shows the block diagram of memory hierarchy and where a Markov prefetcher will be placed. As seen from the diagram the Markov prefetch engine and history table are placed between L1 and L2 caches and prefetch buffer is placed alongside the L1 cache.

The Markov prefetch engine monitors the stream of L1 misses and updates miss history table .This history table then helps decide addresses to be prefetched into on chip buffer. Whenever a miss occurs in L1 cache then along with L2 cache prefetch buffer is checked concurrently. If the prediction engine is good enough then we can get more and more hits in the prefetch buffer. This way we reduce effective miss rate of the L1 cache.

Consider following example miss address stream from L1 cache A-B-D-A-C-E-D-A-B-D-E-B-A-D-A-E. In this sequence A occurs four times, B follows A twice and C and D once each. Probability of getting B after you get A is 50% for C it is 25% and D it is 25%. Now whenever the Markov prefetch engine sees miss reference A, It will try to prefetch B which is most likely. Table 1 shows the state of miss-history table for above mentioned stream of misses from L1 cache. The number inside the square bracket indicates the number of occurrences of that particular block after the current miss address.

Prefetched blocks are stored into prefetch buffer. This prefetch buffer is a FIFO structure with finite number of entries. Whenever a miss occurs in L1 cache this prefetch buffer can be checked concurrently with L2 cache and if a hit occurs in this prefetch buffer we are reducing the L1 miss penalty significantly.

Now we extend the same example to see how prediction is performed. We use the L1 miss stream to build a history. Here we basically find out which block is most likely to follow a certain block in the miss stream. State of the Markov table populated with frequently missed addresses for the sequence of cache misses (A,B,D,A,C,E,D,A,B,D,E,B,A,D,A,E). The numbers inside parenthesis denote the count value to keep track of the most frequently missed addresses.

II. EXPERIMENTAL SET UP

In our experiment we did a trace driven simulation to check the effectiveness of the Markov prefetcher. We generated

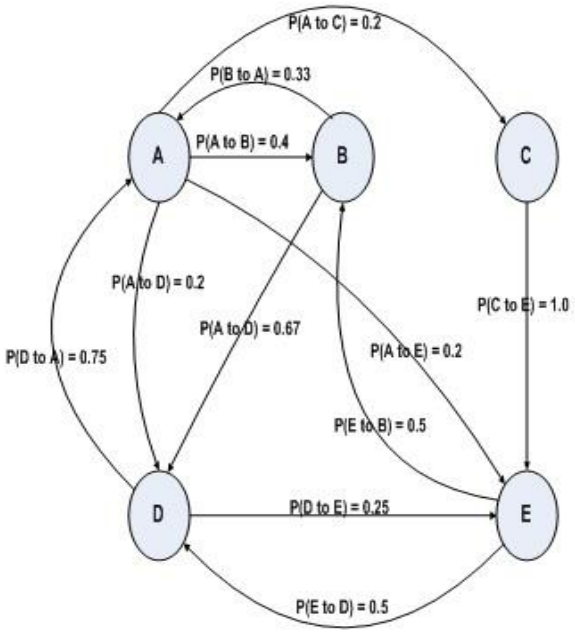

Fig.2 Transition probabilities for example miss sequence. Transitions probabilities are calculated from example miss sequence and shown here.

traces from three applications. The first two traces are from the SPECCPU 2006 benchmark [8]. They are GZIP and GCC. GZIP is a commonly used compression program. Here we have a trace of GZIP compressing a file. The compression algorithm shows temporal as well as spatial locality of reference. Second trace that we have used is that of GC compiler compiling a C program, Compilation trace displays less locality of reference in contrast with GZIP trace But is still uses same set of instructions to compile a piece of code and hence is not completely random in its memory access behavior. The third trace that we used for simulation is of a custom program. This program creates large array of numbers, then calls a function to generate random numbers and uses these random numbers as array indices to access array elements. Mathematical operations are performed on these numbers in the array. Hence we are expecting least locality of reference in addresses accessed by the processor in this trace. Thus we have traces from three different programs that exhibit different amounts of locality of reference.

In our experiment we use separate Instruction and Data caches at level 1. This is usually the case for most of the modern processors. Each cache is 32KB, Block size of 32 and 4 way set associative.

We gauge the effectiveness of the Markov prefetching strategy by monitoring L1 miss stream. For this a Markov history table is maintained that helps predicting the next address. The address is brought into a prefetch buffer and prefetch buffer will be checked for every miss in the L1 cache. Our simulator program records the number of hits in this prefetch buffer. We find out the percentage of hits in the prefetch buffer with respect to number of misses in L1 cache. We use this percentage of hits to quantify the effectiveness of prefetch strategy.

Simulation is performed using a Java program. This program takes the trace file as its input. This trace file contains memory references issued by processor. Each trace file entry has the address of the data used and also has a field to indicate whether it is an Instruction Read, Data Read or Data Write. Depending on this classification simulator will send this address to L1 instruction or L1 Data cache. Simulator program instantiates a java class to simulate the operation of a cache. In our case two instances are created one for L1 instruction cache and another for L1 Data cache. Values for cache size block size and associativity of cache can be set by user. The simulator program reads trace file line by line and depending on whether it is instruction or data, it is sent to the corresponding cache. Whenever a miss occurs in L1 cache that memory reference is sent to Markov history table. Here history table is updated and prefetch buffer is checked. Also counts are maintained for hits and misses in the prefetch buffer. Every simulation terminates by outputting number of hits and misses in prefetch buffer out of total misses in L1 cache.

In our simulation we vary two parameters; first parameter is prefetch buffer size. We simulate for prefetch buffer sizes 1, 2,4,8,16,32 and 64. Second parameter that we can change is the history table size. We simulate for history table size of 4, 32 and 128. Thus overall we perform 21 iterations of the simulation.

III. RESULTS

Figures 3, 4, 5 and 6 show the results obtained from the simulation when trace generated by GZIP application was used as an input to the simulator. We varied prefetch buffer size in powers of two up to 64 and this experiment was repeated for three different history table sizes 4*4, 32*4 and 128*4.

As seen from Figure 3 and 4, for a fixed miss history table size hit rates increases with prefetch buffer size. A larger size allows the prefetch buffer to retain the prefetched blocks longer and hence it avoids any capacity misses.

It is also observed that best results are obtained around the buffer size of eight. Any buffer size below eight produces very low prefetch buffer hit rate on the other hand increasing prefetch buffer size beyond eight produces diminishing returns[3].

Another observation from figures is Data read Accesses produce best hit rates in the prefetch buffer than other types. This means Markov prefetcher captures locality in Data Read type of accesses better for the case of GZIP application.

As seen in Figure 5 if we have a large enough Miss history table then for any prefetch buffer size beyond eight the hit rate for Data Read accesses is double than the hit rate for Instruction Read Accesses.

Figure 6 is plot of hit rates for varying miss history table sizes. Miss history table size of four does not produce any usable results with any prefetch buffer size but as we increase this Miss history table size we see more and more hits in the prefetch buffer. This increase in miss history table size means more history is captured and it allows better prediction results from the Markov prefetcher.

[3]Diminishing return: up to buffer size 8 increasing buffer sizes gives better hit rates but beyond 8 the hit rate slows down against increasing hardware costs. The improvement achieved does not justify the hardware cost.

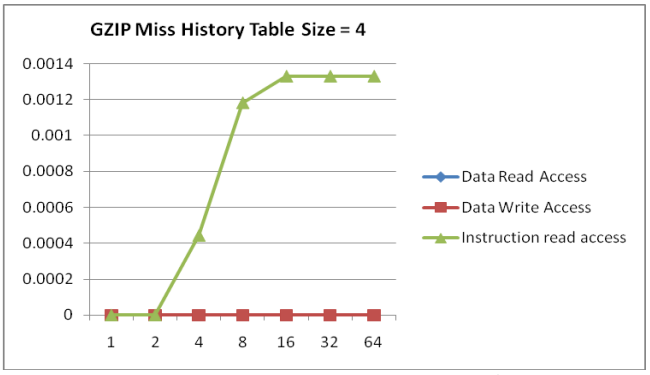

Fig 3: GZIP application with Miss history table size of 4*4. X axis plots prefetch buffer size. Y axis shows % hit rates of prefetch buffer. Figure shows that the History table size of 4 doesn't produce any usable results.

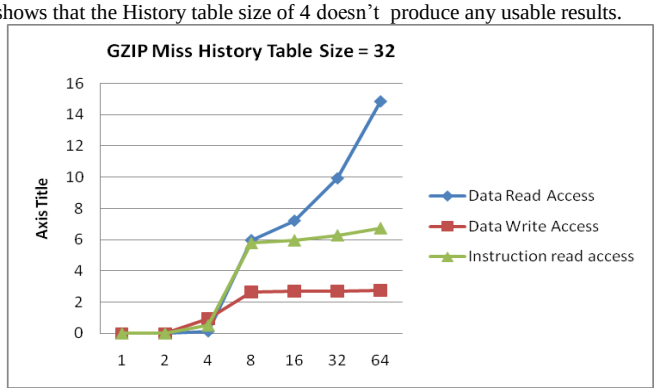

Fig 4: GZIP application with Miss history table size of 32*4. X axis plots prefetch buffer size. Y axis shows % hit rates of prefetch buffer. Upto buffer size of 8 hit rates increase with buffer size after that only Data Read accesses respond increased buffer size.

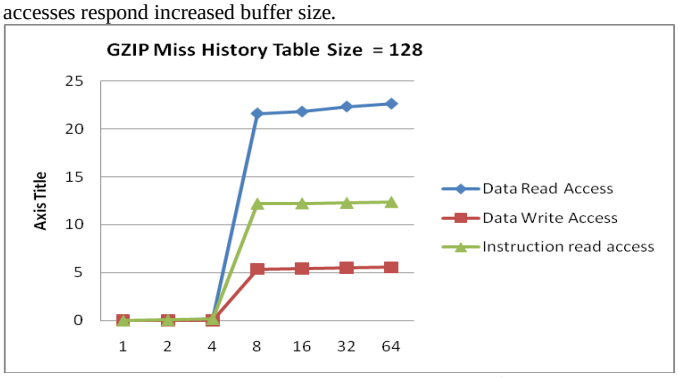

Fig 5: GZIP application with Miss history table size of 128*4 X axis plots prefetch buffer size. X axis plots prefetch buffer size. Y axis shows % hit rates of prefetch buffer. Hit rates increase upto prefetch buffer size of 8 and beyond 8 respond to icnreasing buffer size with very slow increase.

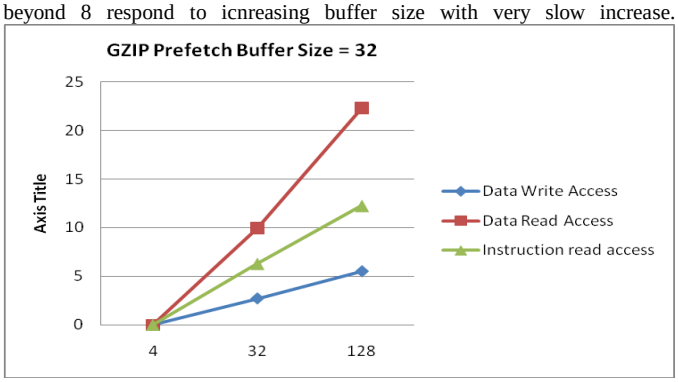

Fig 6: GZIP application with varying Miss history table size and prefetch buffer size of 32. X axis plots prefetch buffer size. Y axis shows % hit rates of prefetch buffer. Increasing History Table size produces higher hit rates for given prefetch buffer size.

Figures 7, 8, 9 and 10 show the results obtained from the simulation when trace generated by GCC application was used as an input to the simulator.

As seen from figure 7 and figure 8 as we increase the prefetch buffer size for a fixed miss history table size we get more hits in the prefetch buffer.A larger size allows the prefetch buffer to retain the prefetched blocks longer and hence it reduces any misses that occurred due to smaller capacity in the prefetch buffer.

Another observation from the figures is that Instruction Read Accesses produce best hit rates in the prefetch buffer as compared to other types. This means Markov prefetcher captures the locality of reference in Instruction read type of accesses better for the case of GCC application. In fact as seen from Fig. 7 and fig 8 only Instruction read type of accesses responds to increased prefetch buffer size with increasing hit rates.

Fig. 10 is plot of hit rates for varying miss history table sizes. Miss history table size of four does not produce any usable results with any prefetch buffer size but as we increase this Miss history table size we see more and more hits in the prefetch buffer. This increase in miss history table size means more history is captured and it allows better prediction results from the Markov prefetcher.

It is seen from the Fig. 10 that only Instruction Read Accesses respond to increasing history table size thus we can see that Markov prefetcher captures locality of reference in insturction read access stream from the GCC application. It can be also stated that GCC application does not exhibit a locality of reference in Data Read and Write Accesses. This fact seems logical when we consider that GCC trace has the memory references of a complier which runs a set of instructions again and again to compile a program. Hence instructions executed by the processor will be same but they will operate on different datasets. That is the reason why hit rate for data accesses does not increase with prefetch buffer size. From this observation we can also say that GCC's L1 miss stream does not exhibit much locality that can be captured by our Markov modeled prefetching.

Apart from Instruction Read accesses other accesses do not respond to increase in buffer size. Thus increasing buffer size does not give us sufficient improvement in hit rate and returns on hardware investment are diminishing. Looking at Fig. 10 for varying table sizes we observe that increasing history table size from 4 to 32 leads to better hit rates by around three times for Instruction Read Accesses but at table size of 128 results do not get any better so we can say that table size of 32*4 is a modest choice for GCC application.

Figures 11, 12, 13 and 14 show results obtained for simulation when trace generated by Random program application was used as an input to the simulator. As seen from Fig. 11, 12, 13 and 14, as we increase the prefetch buffer size for a fixed miss history table size we get more hits in the prefetch buffer. A larger size allows the prefetch buffer to retain the prefetched blocks longer and hence it reduces any misses that may have occured due to smaller capacity in the prefetch buffer. This is specifically true for Data Read and Write Accesses.

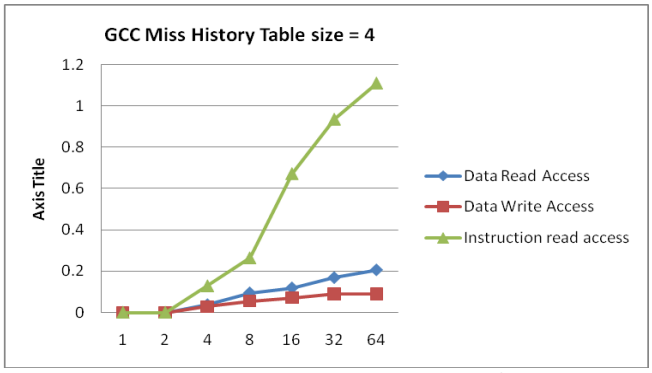

Fig. 7: GCC application with Miss history table size of 4*4 and varying buffer sizes. X axis plots prefetch buffer size. Y axis shows % hit rates of prefetch buffer. Instruction Read Data Accesses produce better hit rates with increasing buffer size. Hit rates are very small for small history table size.

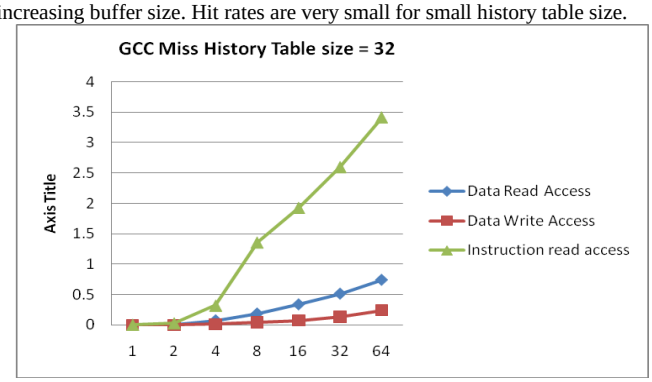

Fig. 8: GCC application with Miss history table size of 32*4 and varying buffer sizes. X axis plots prefetch buffer size. Y axis shows % hit rates of prefetch buffer. Hit rates for Instruction Read Accesses increases with increaseing buffer size. Hit rates for data are very small for all buffer sizes**.**

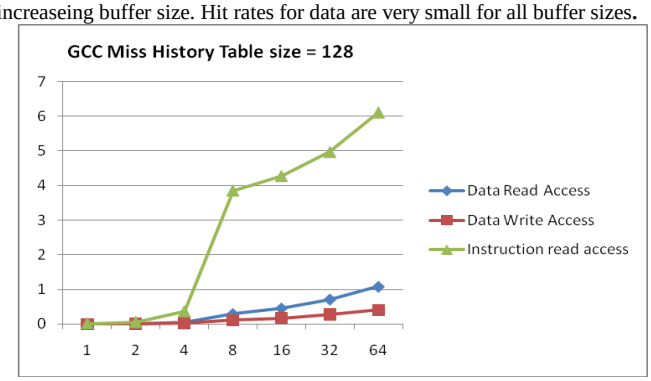

Fig. 9: GCC application with Miss history table size of 128*4 and varying buffer sizes. . X axis plots prefetch buffer size. Y axis shows % hit rates of prefetch buffer. Hit rates for Instruction Read Accesses increase rapidly upto buffer size of 8, beyond that increase slows down.

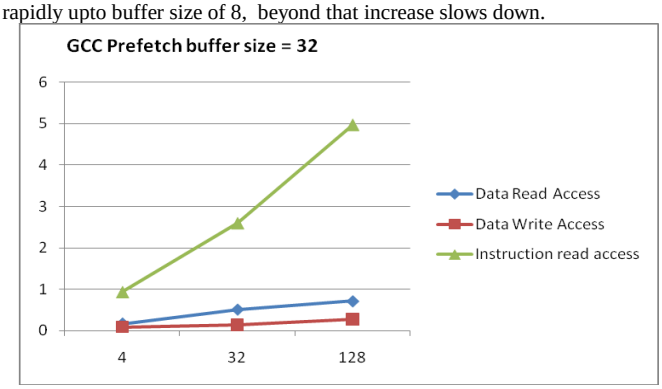

Fig. 10: GCC application with varying Miss history table size and prefetch buffer size of 32. X axis plots prefetch buffer size. Y axis shows % hit rates of prefetch buffer. Instruction Read Accesses respond with higher hit rates to increasing History Table size.

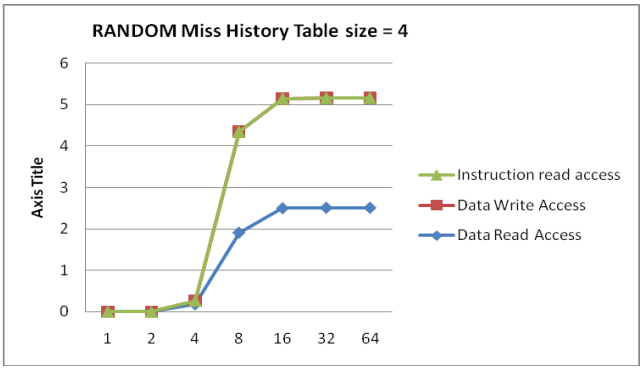

Fig. 11: Random application with Miss history table size of 4*4 and varying buffer sizes. X axis plots prefetch buffer size. Y axis shows % hit rates of prefetch buffer.

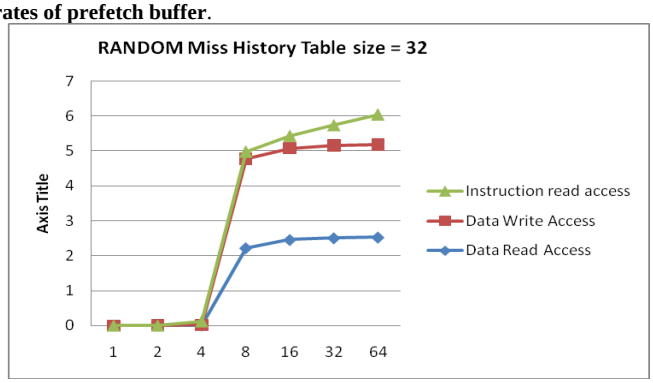

Fig. 12: Random application with Miss history table size of 32*4 and varying buffer sizes. . X axis plots prefetch buffer size. Y axis shows % hit rates of prefetch buffer.

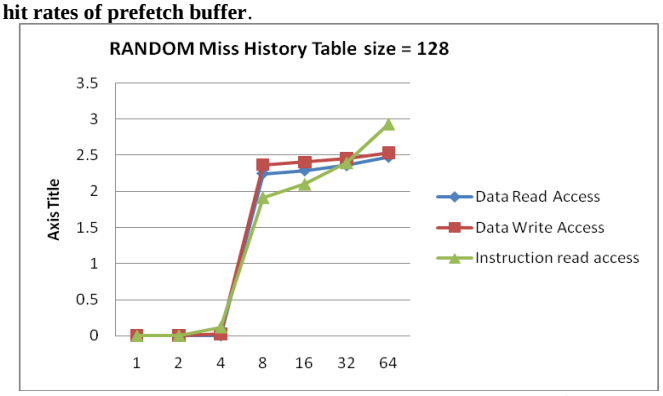

Fig. 13: Random application with Miss history table size of 128*4 and varying buffer sizes. . X axis plots prefetch buffer size. Y axis shows % hit rates of prefetch buffer.

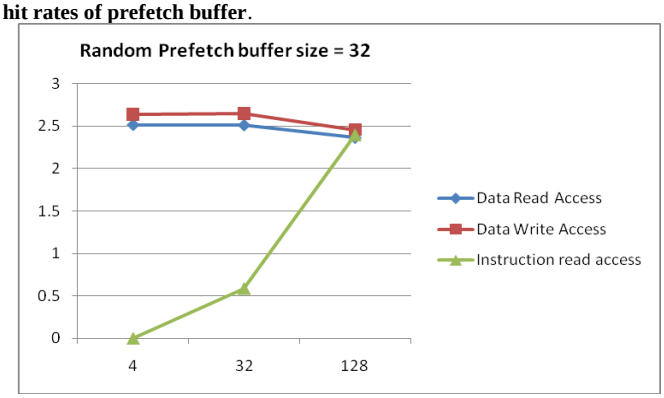

Fig. 14: Random application with varying Miss history table size and prefetch buffer size of 32. . X axis plots prefetch buffer size. Y axis shows % hit rates of prefetch buffer.

For all figures of random application, Hit rates increase with prefetch buffer size upto buffer size of 8, beyond that hit rates stabilize. Instruction Accesses respond better to increasing buffer size.

Another observation from the figures is that Instruction Read Accesses produce better hit rates for larger size of Miss history table size. This fact is very evident from figure 14.

Figure 14 is plot of hit rates for varying miss history table sizes. Miss history table size does not affect the hit rates in prefetch buffer. This fact is evident in Data Read and Data Write Access streams. This phenomenon can be explained if look at type of program this random application is. This random application creates a huge array of numbers and performs mathematical operation on values at random array indices. This means instructions for generating random numbers and performing mathematical operations will be repeated and will have more locality of reference and data access stream will not exhibit such trend. Thus increased miss history table size does not help data access stream.

It is seen from the figure 14 that only Instruction Read Accesses responds to increasing history table size thus we can see that Markov prefetcher captures locality of reference in Instruction read access stream from the random application. It can be also stated that random application does not exhibit a locality of reference in Data Read and Write accesses.

IV. Conclusions

From all the above results we conclude that for a fixed Miss History table size increasing prefetch buffer size produces larger hit rates. This is because with increasing prefetch buffer size we reduce the number of capacity misses occurring in prefetch buffer. We allow the prefetch buffer to retain the prefetched data longer before it gets evicted by new prefetches thereby increasing the chances of getting a hit.

It can also be observed that larger Miss History table sizes produce better hit rates in the prefetch buffer. This is due to the fact the larger miss history table means more history will be captured this means we have more accurate transition probabilities for addresses and hence predictions coming from Markov prefetcher will be more accurate.

GZIP shows good locality of reference in accessing data as well as instructions hence prefetcher produced best results for GZIP. Random program showed least locality of reference in data accesses hence prefetcher results are poor. Results for GCC show more locality in accessing instructions and hence more and more instructions accesses are captured by the prefetch buffer for increasing prediction resources. Overall the percentages of hits for GC compiler trace lie between those for GZIP and random program.

Looking at the results from all three applications we can conclude that a Miss History table size of 32 * 4 and prefetch buffer size of 8 is a good choice to improve the effective hit rate in L1 cache for all three applications. This way we limit the hardware cost and still reap the benefits of prefetching to improve the effective L1 miss rate.

V. Future Work

We would like to extend our experiment to a more diverse and complete set of applications. We are looking at SPEC CPU 2006 benchmark for our further experiments. This will cover a diverse set of applications and allow us test the performance of Markov prefetching for different datasets and

gauge its effectiveness. Also, we will be able to come up with a prefetch buffer size and a miss history table size that can perform well across a diverse set of applications and hence can be used on a general purpose computer.

Testing the timeliness of algorithm is the next logical step. For that we have to use an execution driven simulator and find the number of prefetches that were available on time for the processor to use.

VI. REFERENCES

[1] AA Markov *Extension of the law of large numbers to dependent events Bull. Soc. Phys. Math. Kazan, 1906*

[2] MD. Osman Gani, H. Sarwar, C. M. Rahman *Prediction of State of Wireless Network Using Markov and Hidden Markov Model. Journal of Networks, 2009*

[3] G.E. Bartels *Markov Prediction for adaptive network memory prefetching GE Bartels - 1998*

[4] D. Joseph and D. Grunwald, *Prefetching using Markov predictors IEEE TRANSACTIONS ON COMPUTERS, VOL. 48, NO. 2, FEBRUARY 1999*

[5] N.P. Jouppi *Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers Computer Architecture, 1990. Proceedings., 17th ..., 2002 - ieeexplore.ieee.org*

[6] A. Agarwal, M. Horowitz, J. Hennessey *An Analytical cache model, ACM Transactions on Computer Systems, Vol. 7, No. 2, May 1989*

[7] University of Central Florida Mathematics Department, presentation about Markov Model. *Conditional Probability More on Markov Model*

[8] JL Henning *SPEC CPU2006 benchmark descriptions ACM SIGARCH Computer Architecture News, 2006 - portal.acm.org*