# Tradeoff between coverage of a Markov prefetcher and memory bandwidth usage

Elec525 Spring 2005

Raj Bandyopadhyay, Mandy Liu, Nico Peña

## Hypothesis

Some modern processors use a prefetching unit at the front-end of the machine to fetch instructions and data into the cache ahead of their use in order to reduce the latency of going to main memory if those accesses were to originally miss in the cache. The prefetcher guesses what the next address of access is going to be. Therefore, a good prefetcher should prefetch only those addresses that are actually used and not waste memory bandwidth by prefetching extraneous addresses.

The paper "Prefetching with Markov Predictors" by D. Joseph and D. Grunwald introduces us to the Markov prefetcher, which is a correlation-based prefetcher that uses the history and probability of previous memory accesses to build a table of future memory fetches[1]. Figure 1 shows a miss reference stream and how that stream can be depicted as a Markov chain. The predictor proposed by Joseph and Grunwald places the information from the chain into a table which can be used to predict the next memory reference.

**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.

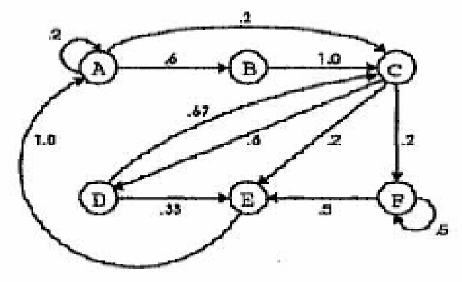


Figure 3: Markov model representing the previous reference string via transition probabilities.

**Figure 1: An address reference string markov model of the reference pattern[1]**

The index to the Markov table is the current address. Each index (or entry) has an associated number of next states, or future addresses, to fetch when the search of the Markov table results in a "hit". These entries also have associated data about their probability of being the correct next state. This data is in the form of hit counters for each next state. In an ideal Markov prefetcher, there are an unlimited number of next states to fetch, and the number of entries in the table is infinite. A real implementation must limit these two variables.

In our study, we investigate the tradeoff between the number of next states and number of entries in the Markov table versus the coverage and accuracy of the prefetches to determine the most efficient configuration for our implementation. We will also look for ways to make the Markov prefetcher use less memory bandwidth by limiting the number of next states prefetched to be less than the number of next states in the table. Similar to Joseph and Grunwald, we explore the addition of a stride prefetcher in front of the Markov prefetcher in an attempt to filter out patterns that are more easily found by the stride prefetcher and thus reduce the mispredictions from the Markov prefecher. Our implementation of the stride prefetcher is simplified from the one proposed by Grunwald and shows a significant gain in the overall coverage and a reduction in the mispredicted prefetches.

## Markov Prefetcher Architecture

The Markov table and prefetch logic are inserted off-chip between L2 cache and main memory. See Figure 2. Upon an L2 cache miss, the Markov table is searched. If the table search results in a "hit", all valid next states (addresses) are loaded into the on-chip Markov prefetch "buffer" (32 entries in our implementation) at the same level as L2 cache.

The on-chip Markov prefetch buffer is searched during an L1 cache miss at the same time as L2 cache is searched. If the current address hits in the prefetch buffer, then the valid next states in the Markov are loaded into the prefetch buffer. An address can only acquire one valid next state for each time it is seen in the address stream. Thus if we have only seen an address once previously, then we do not send out 4 prefetches for the next state when we see that address again, since the table only knows about one possible next state for that address. Regardless of the results of searching the prefetch buffer, if the address misses in the L2 it is forwarded to the Markov prefetcher as well as to main memory. If the address hits in the Markov table, then again the next states in the table are loaded into the prefetch buffer, replacing those entries that are the oldest in the prefetch buffer.

We use a least recently used (LRU) replacement policy for our Markov prefetcher. Each time around, if there is a hit in the prefetch buffer, the LRU status of that entry in the Markov is updated. The entries are not stored in address order in the Markov table. Hence, when an entry is evicted, the new entry merely overwrites the old one, and no reshuffling or shifting of the entries occurs.

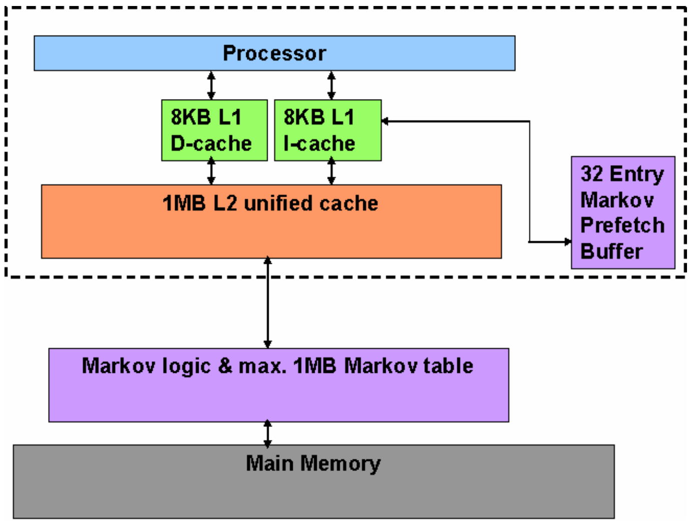


**Figure 2: Diagram of memory hierarchy with Markov prefetcher inserted**

In our implementation, we use a direct mapped, 8KB L1 instruction cache and 8KB L1 data cache. Each L1 cache block is 32-bytes with single cycle latency.

Our L2 cache is a direct mapped, 1MB unified instruction/data cache with a block size of 128-bytes and 12-cycle latency. The Markov paper uses an L2 cache of 4MB in size, but the paper was written in 1997 and used SPEC95 benchmarks. We changed the size of the L2 cache to 1MB because the size of the data sets in the SPEC2000 benchmarks fit too well in 4MB of cache, thus we were getting a sparse number of L2 cache misses to prove our concept with. We think this is a reasonable change because most modern day year 2005 processors don't necessarily have as much as 4MB of L2 cache on-chip.

Grunwald and Joseph implemented their Markov prefetcher for L1 cache misses; we implemented ours for L2 cache misses. See Figure 3 for the Markov paper's implementation. From Figure 3 it appears that L2 cache and main memory are off-chip. Modern day processors have L2 cache on chip, so in our implementation we use L2 cache misses as our miss reference trace. Moving the prefetcher beyond the L2 cache would seem to cause a decrease in accuracy since the address stream would be interleaved with I and Dcache accesses which may not exhibit the patterns that arise looking at a single cache as in [1].

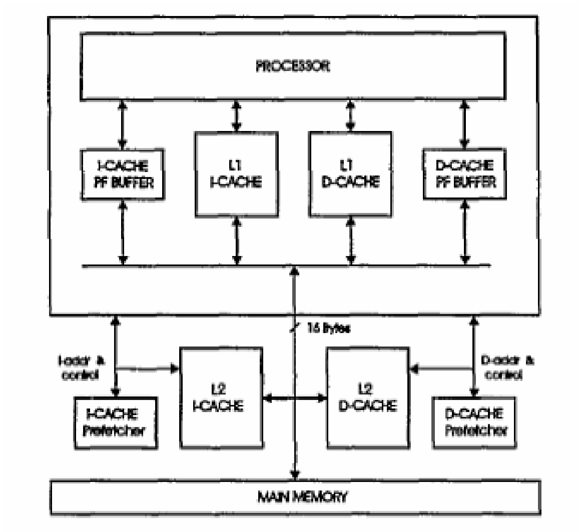


**Figure 3: Memory hierarchy with Markov prefetchers inserted[1]**

The size of the Markov prefetch table in the paper is fixed at 1MB (2^20). The ideal number of next states as determined by the Markov paper is four. If each next state (address) is 4 bytes, then each table entry (index) plus four next states is 2^2*5. Therefore the number of indices (entries) in the paper's Markov table is (2^20) / (2^2*5) = 2^15 ~ 2^16, or roughly 50k entries. The paper also simulates using 1, 2, 4 and 8 next states, where the number of table entries varies dynamically while keeping the table size fixed at 1MB.

In our experiments, we vary our table size, with our largest table being 16k (or 2^14) entries with 16 next states. We estimate this size to be 4*(16+1)*2^14 = 2^20, or roughly 1.1MB.

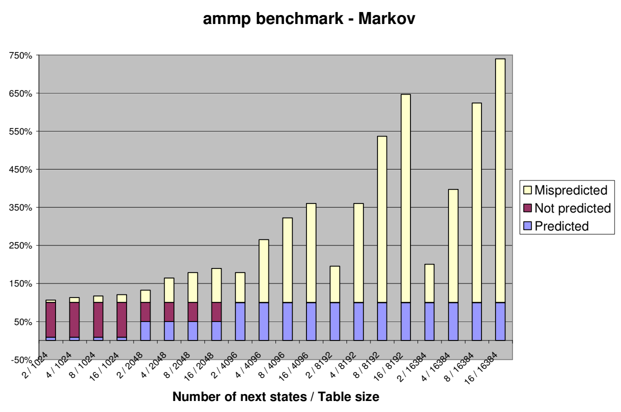
## Experimental methodology and analysis

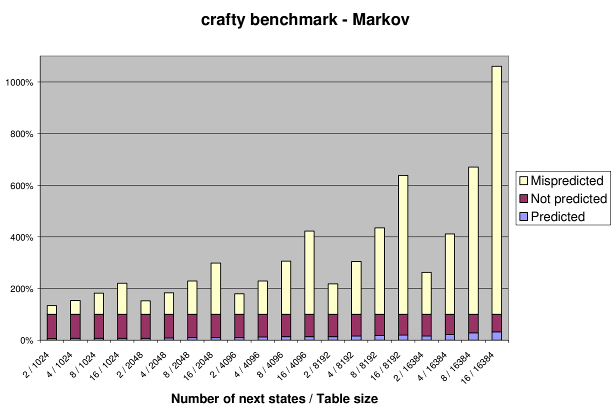
Markov Prefetcher

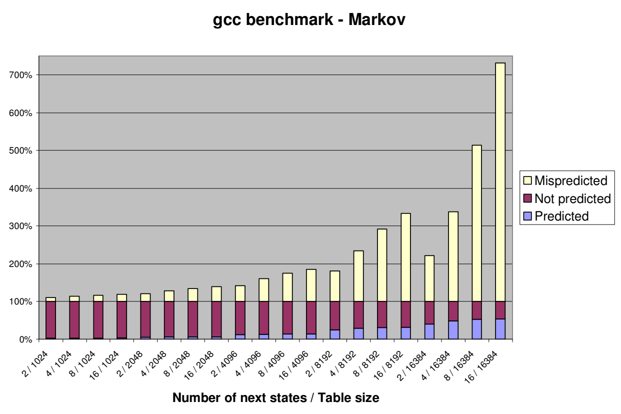
The first step in our experiment was to duplicate the Markov paper's Markov prefetcher model as close as possible, with the added exception of the changes noted in the previous section. As described previously, we decreased the size of the L2 cache as compared to the paper in order to reel in more L2 cache misses from our SPEC2000 benchmarks. We also implemented the Markov prefetcher between L2 cache and main memory, instead of between L1 and L2 cache since modern day processors implement L2 cache on-chip.

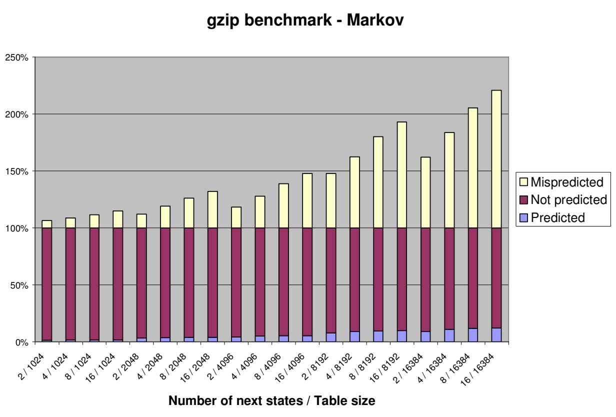
We wrote the Markov prefetcher in C and used input traces generated by the SimpleScalar 3 tool, sim-cache. The inputs to our Markov prefetcher are SPEC2000 benchmark traces which we generated by adding a "print" statement to sim-cache.c's L2 cache miss handler. Upon entering the L2 cache's miss handlers, the memory address is dumped out into a file, which we used as our trace. Due to limited time for the project, we chose only six SPEC2000 benchmarks. The six benchmarks we chose for our experiment show the varying accuracy of the prefetcher for different workloads. They are: ammp, crafty, gcc, gzip, mcf, and vpr.

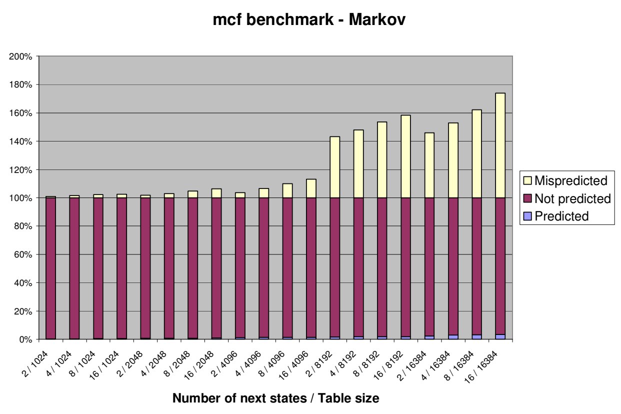
The simulation results for our Markov prefetcher are shown below for all six benchmarks. We varied our table size from 1k entries up to 16k entries, and varied the number of next states at 2, 4, 8 and 16. The same definitions as in [1] are used to produce the diagrams, where the bottom part of the bar is the "predicted" area, or coverage of our Markov prefetcher. This is the percentage of total demand fetches that are caught by our Markov prefetcher. The middle bar is the percentage of demand fetches that were "not predicted", which is (1-coverage). The upper bar area is the number of wasted prefetches that were fetched but unused, or "mispredicted" prefetches.

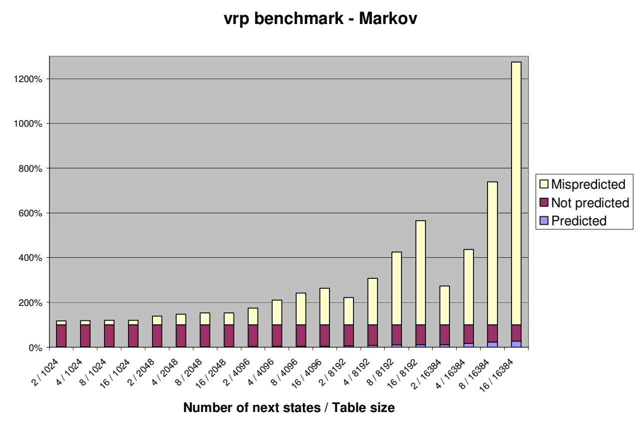












From these results we can make three conclusions. First, increasing the number of next states that are prefetched increases the number of mispredictions (which is a decrease in accuracy). This makes sense since not all of the next states fetched will eventually be used by the program. Second, increasing the number of entries in the table increases the coverage (predicted), but only slightly. This also makes sense because holding more addresses will increase the chances that an address will "hit" in the Markov table. Third, the increase in coverage associated with increasing the number of entries in the Markov table does not offset the cost of such high mispredictions if area or memory bandwidth are considerations for the design.

As can be seen from the results, Markov does very well in only the ammp benchmark, where the coverage (predicted) is as high as 99.4%. The worst is mcf, which has a coverage of only 3.4%. The average coverage over all six benchmarks is 38%. The average wasted prefetches over six benchmarks is 600%! This means that our Markov prefetcher does six times more useless fetches than the demand fetches.

Clearly from the results, the Markov prefetcher does not perform well for all of these benchmarks. Our idea was augment the markov prefetcher by adding another prefetcher to help catch some of the not predicted address and reduce the misprediction rate of the Markov prefetcher similar to the work in [1]. One of these schemes we believe has a high chance of success in decreasing the number of mispredictions and increasing the accuracy of our Markov is adding a stride buffer. The stride buffer is added in front of the Markov in series with it. Our hypothesis is that the stride buffer will catch one-time linear (or strided) memory accesses that Markov would not be able to catch. This is because the Markov prefetcher requires a "training period" and does not catch addresses that have no previous history.

The stride buffer works to filter out one-time strided addresses before reaching the Markov. The stride prefetcher also has an on-chip 32-entry prefetch buffer at the same level as L2 cache. L2 cache is searched at the same time the stride prefetch buffer is searched. The stride works by keeping track of the distance between the previous memory address and the current memory address. This distance is saved in a register. If the next time around the new memory address distance is the same as our saved stride value, we prefetch this current memory address plus the stride distance into the prefetch buffer. A stride distance has to be seen twice before prefetches of that distance occur. The stride prefetch buffer replaces the oldest entries in the buffer once all of the entries are full.

## Combined Stride/Markov Prefetcher

Figure 4 shows the stride logic added to our architecture. Here the stride prefetcher is placed in front of the Markov prefetcher in order to filter out access sequences that cannot be predicted by the Markov prefetcher. If an address does not cause a prediction from the stride prefetcher, only then is that address passed to the Markov prefetcher.

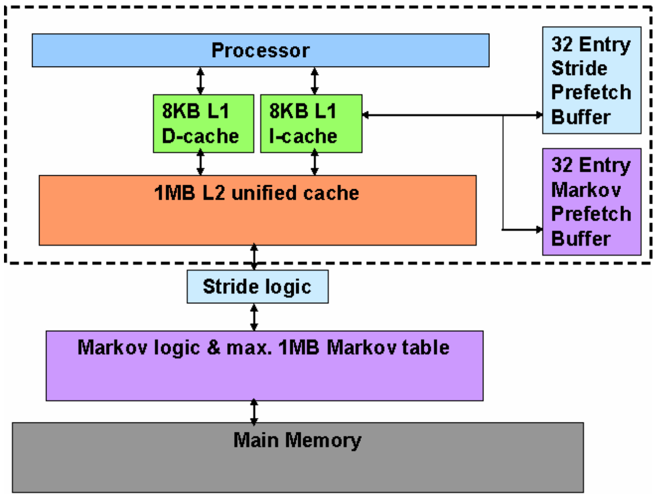
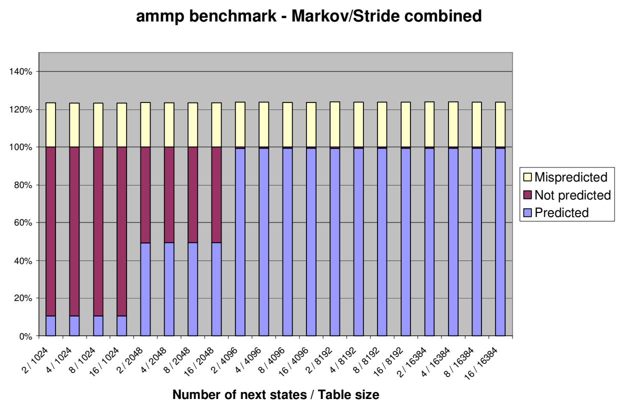
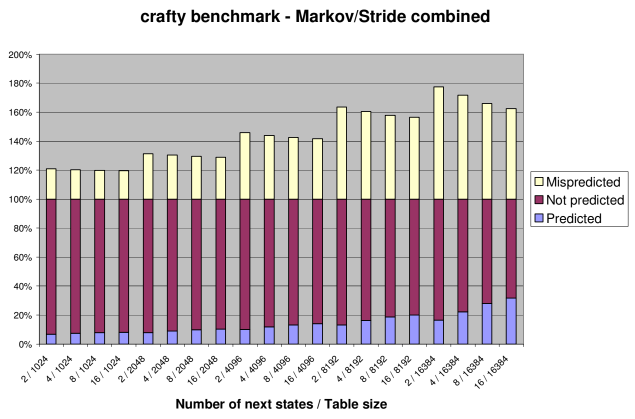
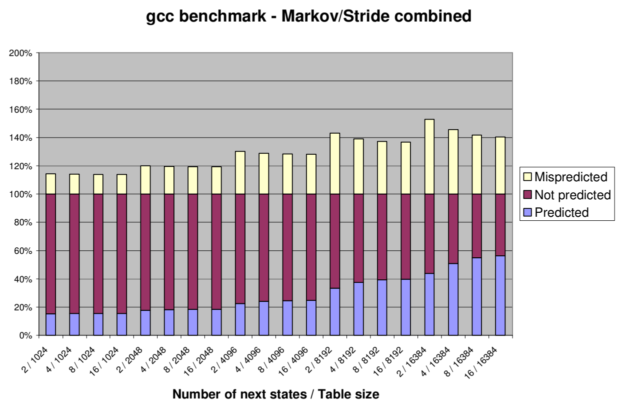

Processor
32 Entry Stride Prefetch Buffer
8KB L1 D-cache
8KB L1 I-cache
32 Entry Markov Prefetch Buffer
1MB L2 unified cache
Stride logic
Markov logic & max. 1MB Markov table
Main Memory

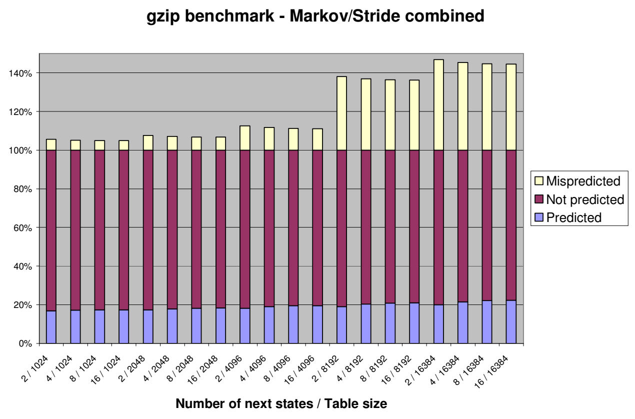
**Figure 4**

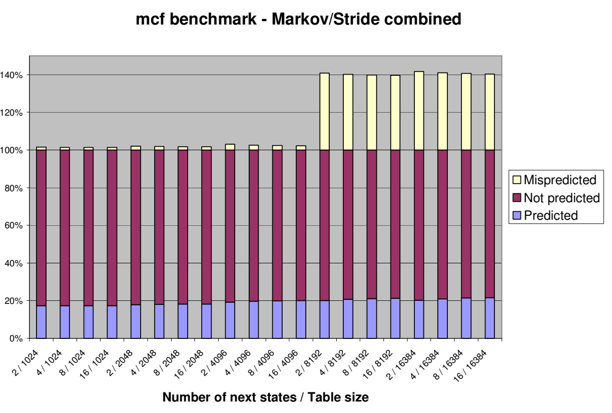
The figures below show the results of adding the stride prefetcher in front of our Markov prefetcher.

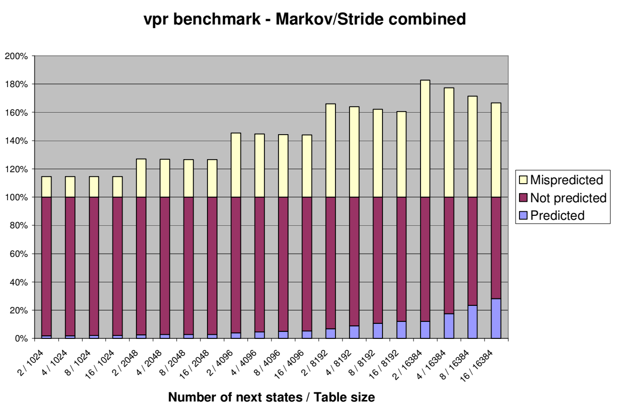












The most notable result from adding the stride prefetcher in front of the Markov is the number of mispredictions decreases dramatically, with only a slight increase in coverage. The average misprediction goes from 600% down to 46% using the combined Stride/Markov architecture. The average prediction rate goes up slightly from 38% to 43%. The table below summarizes the results comparing the percentage of the total L2 miss addresses (demand fetches) that are predicted and mispredicted for the combined Stride/Markov prefetcher to our baseline Markov prefetcher with our largest implementation of 16k entries and 16 next states.

| **Benchmark** | **Markov %Predicted** | **Markov %Mispredicted** | **Combined %Predicted** | **Combined %Mispredicted** |
|---|---:|---:|---:|---:|
| ammp | 99.4 | 640.6 | 99.4 | 23.9 |
| crafty | 31.5 | 959.9 | 31.6 | 62.5 |
| gcc | 54.0 | 631.8 | 56.2 | 40.4 |
| gzip | 12.0 | 120.8 | 22.2 | 44.6 |
| mcf | 3.4 | 73.9 | 21.7 | 40.4 |
| vpr | 27.5 | 1173.8 | 28.2 | 66.6 |
| **Average** | **38.0** | **600.1** | **43.2** | **46.4** |

The advantage of combining stride and Markov together is that the stride prefetcher can filter out many one-shot strided accesses to the Markov. As we can see from the results, the Markov catches most cases in the ammp benchmark; whereas the stride catches a lot more cases in the mcf benchmark.

We did an additional experiment to see if limiting the number of next states that are prefetched would decrease the misprediction further. For instance, instead of prefetching all next states, we would only prefetch half of them. From our results, we concluded that there was no benefit in doing so. In fact, the misprediction was slightly higher and the prediction slightly lower.

| **Benchmark** | **Combined w/ fetching only half next states - % Predicted** | **Combined w/ fetching only half next states - % Mispredicted** |
|---|---|---|
| Ammp | 98.8 | 24.5 |
| Crafty | 21.9 | 72.2 |
| gcc | 49.0 | 47.5 |
| gzip | 21.5 | 45.3 |
| mcf | 20.8 | 41.3 |
| vpr | 19.3 | 75.6 |
| **Average** | **38.5** | **51.1** |

## Performance Analysis

For our performance analysis, we compared our best case Markov to our best case combined Stride/Markov to see how their CPU performances stack up. The best case Markov is 2 next states with 8k entries. The best case combined Stride/Markov is 16 next states with 4k entries.

We used the SimpleScalar 4.0 alpha simulator for the performance analysis. We modified the cache_timing.c file to replace the default prefetcher with our Markov/stride prefetcher, which was now connected to the L2 instead of the L1 cache as in the default. We intended to perform 2 sets of performance experiments: with the Markov/stride prefetcher combined, and with the Markov prefetcher alone.

However, we were unable to obtain any results at the time of writing this report, since the simulation has been running for several days over a single benchmark and has not completed or produced any output. As a sanity check, we are also running a simulation with the original simplescalar 4.0 code, which is also currently running.

## Cost of Implementation

The largest cost associated with the Markov prefetcher implementation is the area for the Markov table. In our best case of 4k entries and 16 next states the table size is roughly equal to a 256 kilobyte cache. This is nearly a quarter of the size of the Markov table proposed in [1] and we have a single table, whereas they implemented two separate tables.

Given more time, would we have liked to find a tool that could convert our Markov and combined Stride/Markov C models into VHDL or Verilog, and then synthesize them using a standard library to get an estimate of the transistor gate count.

## Conclusion

In this project we discovered that Markov prefetching is still useful beyond the L1 cache for certain applications. However, the Markov prefetcher alone does not improve the bandwidth requirements. Augmenting Markov prefetcher in series with a simple stride prefetcher can dramatically reduce mispredicted prefetches.

Our modification using a limited number of next states that are prefetched also showed that the implementation requirements for prefetcher can be smaller than previously proposed with similar performance.

Unfortunately, we could not obtain results for the performance of the prefetcher in a more realistic processor, inspite of integrating with SimpleScalar 4.0 alpha processor simulator, because of the time taken to run the simulations.

One drawback of our prefetching scheme is that it might not be suitable for SMT applications. In that case, an equivalent system may be implemented with separate tables with for different threads.

## References

[1] D. Joseph and D. Grunwald. Prefetching using Markov predictors. In 24th Annual International Symposium on Computer Architecture, pages 252-263, 1997.