ECE 511 Assignment 3
Markov Data Predictor

Vikram Sharma Mailthody

(NetID- vsm2, UIN - 663250535)

Contents

Goals of the Assignment:

1. Understand different types of data predictors– tagged, stride and Markov
2. Analyze the performance difference between tagged, stride and Markov with MI, MESI coherence protocol
3. Build MSI coherence protocol
4. Analyze how coherence protocol MSI effects the performance

More details of the assignment can be found here. [10]

Software used:

Gem5 and its dependencies

Tasks:

The assignment requires to submit 2 results (checkpoint and final). Following are the main steps involved for completing the assignment.

1. Use fs.py as the default script for running the full system emulation
2. Section 2 - run fs.py with default L1,L2 cache sizes and in ruby model.
3. Run tagged, stride model with 4 different benchmarks - bodytrack, blackholes, fludanimate, x264 for each of coherence protocol – MI, MESI.
4. Write code for Markov based data predictors
5. Write code for MSI coherence block
6. Run MSI and Markov modes with 4 bench marks.
7. Analyze the data obtained in these configuration and provide comments

For checkpoint – task 1, 2, 3,4 and 6 are completed.

Introduction:

Gem5 is a computer system simulator platform initially developed at University of Wisconsin-Madison Square (there are several other collaborators like MIT, UMich, ARM, etc). It's a modular simulator allowing the user to parameterize, extend and rearrange as required. The simulator supports both x86 and ARM ISA allowing the user the flexibility of choice. The simulator is primarily developed using C++ language and scripts to run are typically written in python. There are two main modes of operation when using the simulator – system call emulation mode and full system mode (se and fs respectively). The System call emulation mode is mainly used to simulate a binary file that are linked statically or dynamically. The full system mode runs complete system with choice of operating system to boot in the simulation environment. Further gem5 supports configurable CPU models, pluggable memory systems and device models providing flexibility, availably and enhancing collaboration for the computer system researcher.[1]

For this assignment, we will be using x86 ISA and Full System emulation mode. The primarily goal of this assignment is to understand how data prediction works, different types of predictors and understand importance of coherence protocol. 5 benchmarks are used to measure the hardware requirement, displacement of useful data, coverage, timeliness and effect on bandwidth with each of these predictors.

The checkpoint report would discuss about Markov predictor and analyze its results with existing ones. Further optimization would not be taken care in the checkpoint report. Next report would have information on optimization and also MSI coherence implementation.

Report organization:

Next section describes about existing predictor results. Followed by Markov predictor design will be discussed. In the next section, Markov predictor design is discussed. Lastly, analyses of these predictors with different bench marks is studied and compared.

Default Branch Predictors:

Baseline architecture:

In order to measure the conditional branch predictor accuracy, system design shown in figure 1 is used. It houses:

- 8 core out of order x86 ISA based full system with TimingSimpleCPU configuration
- Ruby model memory and cache
- Vmlinux kernel, Ubuntu OS.
- 8 L2 cache and with crossbar topology.
- DDR3 1600 memory

Figure 1: High level baseline architecture for branch prediction

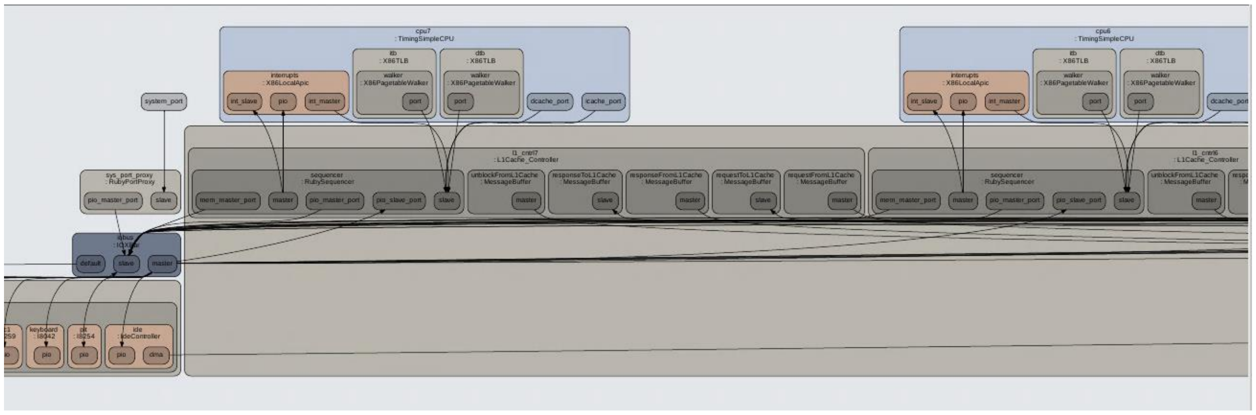

Figure 2: Zoomed version of the same.

Stride Prefetcher:

The stride prefetcher that is available in gem5 is a cache block address based. The other kind of prefetcher is instruction program counter based. Instruction based prefetcher record the distance between the memory addresses references by a load instruction as well as the last address reference by the load. Next time the same load instruction is fetched, prefetch of last address and stride occurs. However in gem5 current implementation, I believe cache block address based stride prefetching is used. The main benefit of cache block address based prefetching is that it can detect A, A+N, A+2N… addresses. This is accomplished by using stream buffers in current stride implementation. Further, the same stream buffer can be used as data storage block which is what is done in the gem5. Figure 3 shows high level data structure of stride prefetcher.

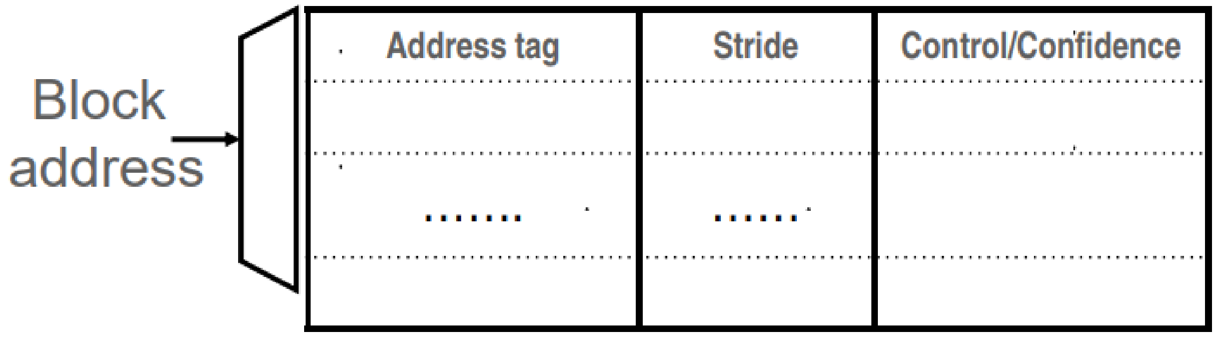

Figure 3:High level view of gem5 stride prefetcher.

Working principle:

1. Each stream buffer can hold one stream of sequentially prefetched cache lines
2. When a load is missed, head of all the stream buffers are checked for address match.
    a. If hit, an enetry from FIFO is popped and cache is updated with data
    b. If miss, allocate a new stream buffer to the miss address. The gem5 implementation does recycle of the stream buffer using LRU policy when full.
    c. Issuance of address will not happen at page boundary
    d. A valid bit is also used to speed up the hit checking.
3. Stream buffer FIFOs are continuously popped with subsequent cache lines whenever there is a room and the bus is not busy.
4. In gem5, prediction mechanism is employed for non-unit strides.

Markov Prefecher:

Markov prefetcher are advance version of correlation based prefetcher. I basic idea is to predict the next address to be fetched based on recent history of previous address misses and markov model of those recent history transitions. Figure 5 shows the Markov model and Figure 6 represent how it will be implemented in the gem5.

Working principle:

1. Track the likely next address after seeing a miss address. If the address is found in markov chain, then issue next set of probable addresses for prefetch. If the address is not found then add a new element to markov chain for tracking. Update the weights or probabilities whenever new miss address is obtained.
2. To Increase coverage prefetch N next addresses.
3. Without LRU accuracy will be dropped. Implement LRU to get better accuracy.
4. Other method to improve prefetch accuracy and coverage is by having longer history. However [4] suggest to use upto 4 address priorities per miss address.

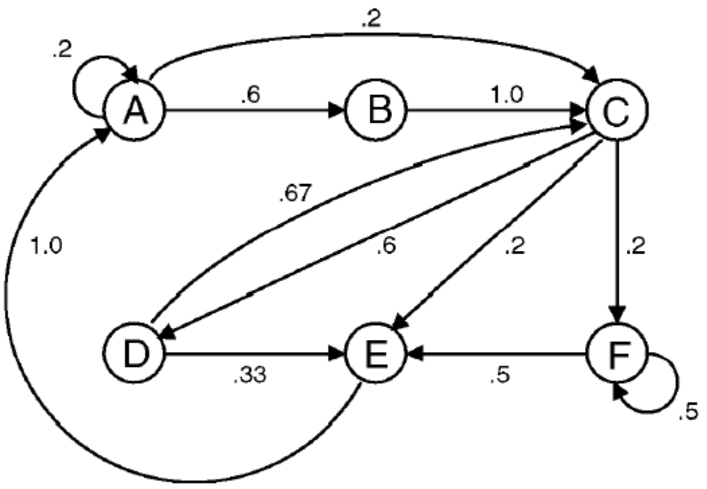

Figure 5: Markov model as described in [4].

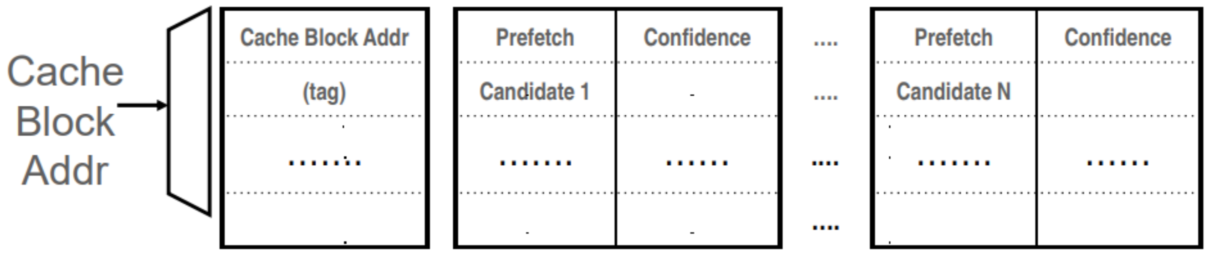

Figure 6: Markov model implemented in gem5.

Modifications in default repository:

The Markov Prefetcher is designed as per figure 7. It consists for three main components, MarkovTable, M_array and LRU module. Specifics and working of these blocks are listed below:

i. Markov Table: This tracks the history of previous miss and associated confidence tag. The table is of 2D array. Each array is called stream. Each stream has list of miss address information. Based on [4], the maximum priority entries per miss address is 4. Beyond which it does not add to accuracy since the probabilities of such transition will be low. Over head of handling the entry outweighs the storing of the misaddress when probability is low.

ii. Least recently used (LRU) algorithm: LRU is used to update the Markov table on every miss address hit or miss. Initially the table is allowed to fill arbitrarily (not exactly arbitrarily, there is some order but I don't have better phrase for the initial fill up mechanism). Once the table becomes full, the prioritization automatically occurs with LRU. Initial latency of fill up is where maximum error of cache miss occurs (also observed in simulation stat which is not discussed in detailed later). If the table depth is huge, then initial latency can induce more cache misses and could cause drop in accuracy. Hence, the current implementation houses only 16 ad depth. One more reason to avoid depth is amount of time spent on simulation which I wanted to avoid. Increasing the depth beyond certain stage does not increases the accuracy or coverage because most recent cache misses are already stored in the markov chain and having long chain will introduce latency which is undesirable. However I personally have not tested this reasoning in the current implementation.

iii. M_ARRAY table: This table has list of addresses that needs to be prefetched whenever the memory controller is free. The table is filled up with a stream from Markov Table where there is miss-Address hit. If the miss address is not found in the markov table, then a new address point is added to markov table and next stride values are also issued from the m_array to improvising the cache miss that can occur. This is done assuming the possibility of latency from converting a stream to m_array buffer entry. Although it may consume more bandwidth, this may improve the overall accuracy. (In current implementation, the stride prefetcher has LRU implementation for m_array. I am retaining it to check how the results changed. In next set of results this will be optimized.)

iv. Whenever miss is observed in the cache, the address is sent to the M_ARRAY stream buffer to find if it's a hit. This is done primarily to avoid markov table computation overhead and also to avoid pollution of m_array table with duplicates. If the M_ARRAY has a hit, then address request was in pending issue. The algorithm issues next prefetch address of the same block to avoid subsequent missAddress. This is again under the assumption that there can be a second miss. However if it was miss in m_array table, then Markov table is searched for hit. If the hit was found in the Markov table, then the algorithm just updates the LRU so that priority is updated. If miss is reported in the markov table, a new "stream" entry is added to the Markov table. Whenever another miss occurs after that address, the pre-miss address is updated to reflect the connection creating a new node in markov chain. If the chain exceeds the maximum node that the table can accommodate then the unused entry is replaced with new element with LRU implementation.

v. Address going to page boundaries requires to be avoided when using prefetcher. This is done by checking if page boundary conditions and such requests are dropped.

Implementation details/pending items:

1. A new data structure is created for the stream with vector size for Pre-miss entry. During construction, the pre-miss array is resized to 4.
2. The depth of the markov table is considered to be of 16. Based on [4], they suggest they use about 1MB of space for Markov implementation. Which translate to about 32K entries in the table (assuming 8B addressing, 4 width – 1024*1024/ 4*8). However this is not considered in current runs yet. This will be taken in account in next report.
3. LRU implementation is done on the array update itself to reduce the additional time
4. Stream conversion to m_array is not very well coded. Currently it is done with a help of for loop. This should be later converted to shifting mechanism or simpler implementation.
5. LRU in m_array requires to be removed as it's a duplicate.

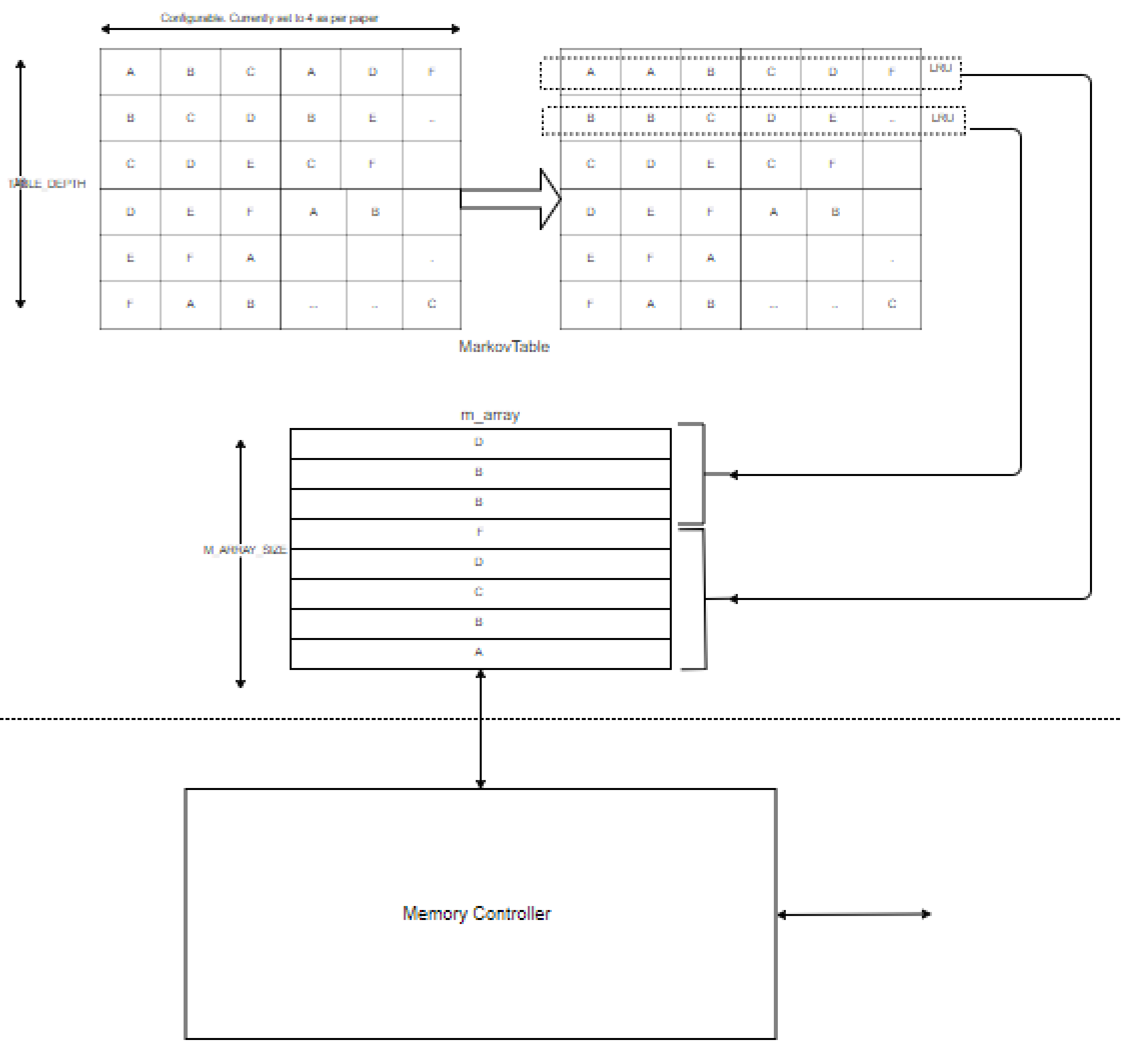

Figure 7: Markov Prefetcher

Results:

Table 1: Results of benchmarks and markov prefetcher

	MI (only one core worked)	no prefetching tagged	MESI (multiple core worked)		
	stride		stride	tagged	Markov
BodyTrack					
avg.cache.hits	275379767	61719052	478505318	61719052	Could not run - gem5.debug works, not gem5.opt. Time not sufficient
avg.cache.miss	1928389	248950	9297677	248950	
avg.cache.access	277308156	61968002	487802995	61968002	
accuracy %	99.3046043	99.59826041	98.09396886	99.59826041	
coverage	78.91923399	97.27852798	-1.640360561	93.47773233	

MemBWavgRd	29.49	46.39	0.59	46.39	
MemBWavgWr	35.03	38.44	0.73	38.44	
MCPI	14.09856962	10.18356837	12.22811404	10.18356837	
Blackholes					
avg.cache.hits	277876202	61498945	117160015	61498945	Could not run - gem5.debug works, not gem5.opt. Time not sufficient
avg.cache.miss	1359122	74692	853468	74692	
avg.cache.access	279235324	61573637	118013483	61573637	
accuracy %	99.51327003	99.87869484	99.27680467	99.87869484	
coverage	70.84767939	99.18348187	90.67005713	98.04313631	
MemBWavgRd	24.06	13.85	0.57	13.85	
MemBWavgWr	27.89	12.65	0.8	12.65	
MCPI	12.71597773	9.898183739	10.07219082	9.898183739	
x264					
avg.cache.hits	615592912	7434830	143087823	615592913	477048138
avg.cache.miss	9147623	1088468	3816924	9147623	3746497
avg.cache.access	624740536	8523298	146904747	624740536	480794635
accuracy %	98.53577246	87.22949731	97.40176946	98.53577246	99.2207698
coverage	1	88.10108375	58.274144	0	59.044038
MemBWavgRd	44	84	5.15	44	75.12
MemBWavgWr	52.11	42	5.07	53.95	75
MCPI	18.25502722	11.41900957	10.43062574	18.25502722	21.40667934
fluidanimate					
avg.cache.hits	3564126499	2678243	693710902	3564126499	2871504992
avg.cache.miss	4662140	1499535	3402812	4662140	564399
avg.cache.access	3568788639	4177778	697113714	3568788639	2872069391
accuracy %	99.86936352	64.10687691	99.51187132	99.86936352	99.9803487
coverage improvement	49.03441036	83.60738085	62.80113424	49.03441036	87.89399289
MemBWavgRd	7.97	25.2	1.02	7.97	28.3
MemBWavgWr	8.96	17	0.99	8.96	18
MCPI	12.19341077	10.95227499	10.58924689	12.19341077	11.57284288

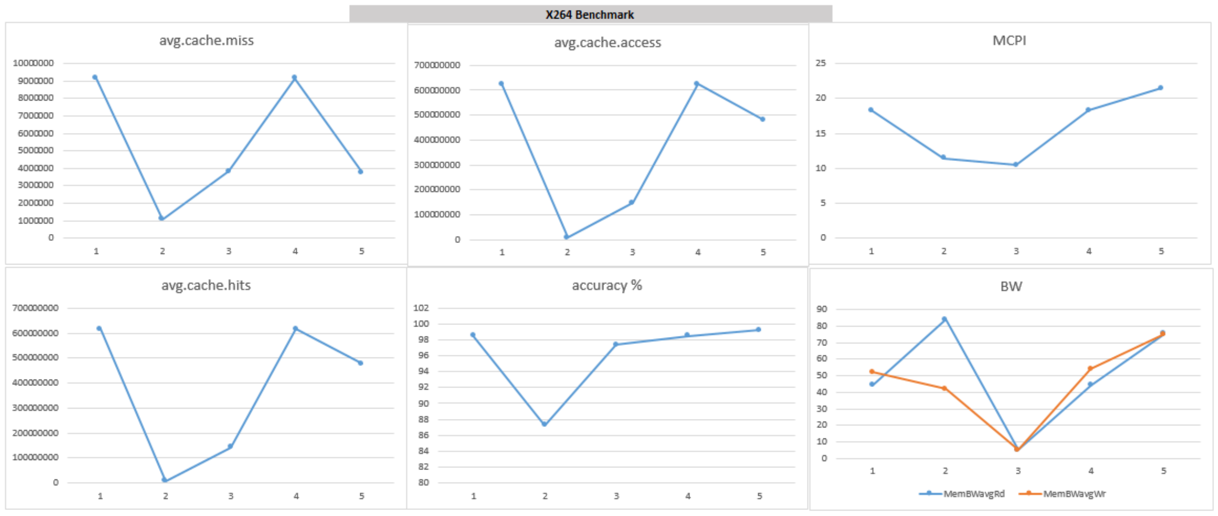

Figure 8: X264 benchmark results

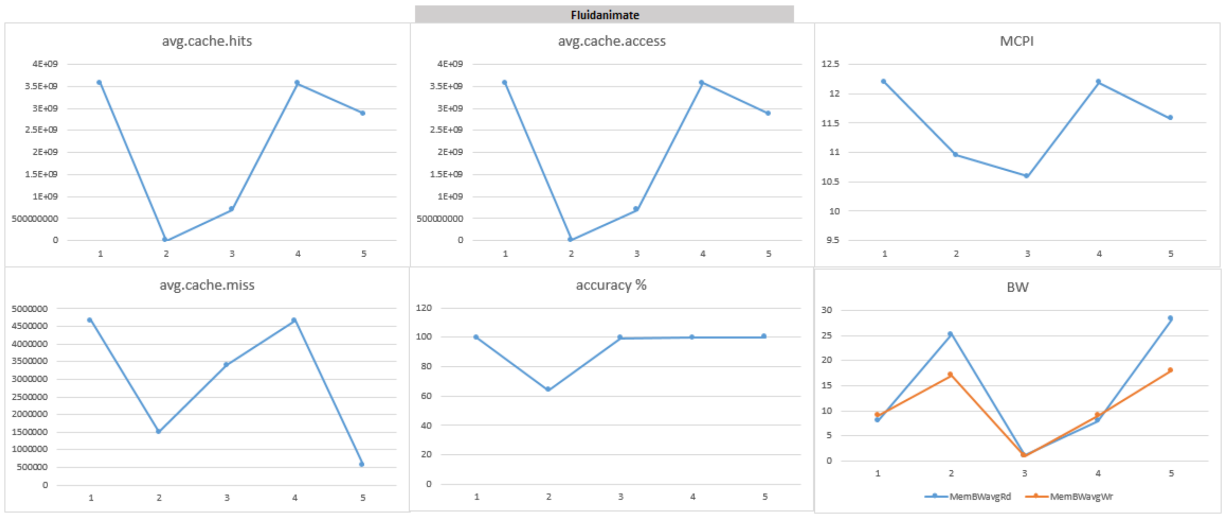

Figure 9: Fluidanimate benchmark results

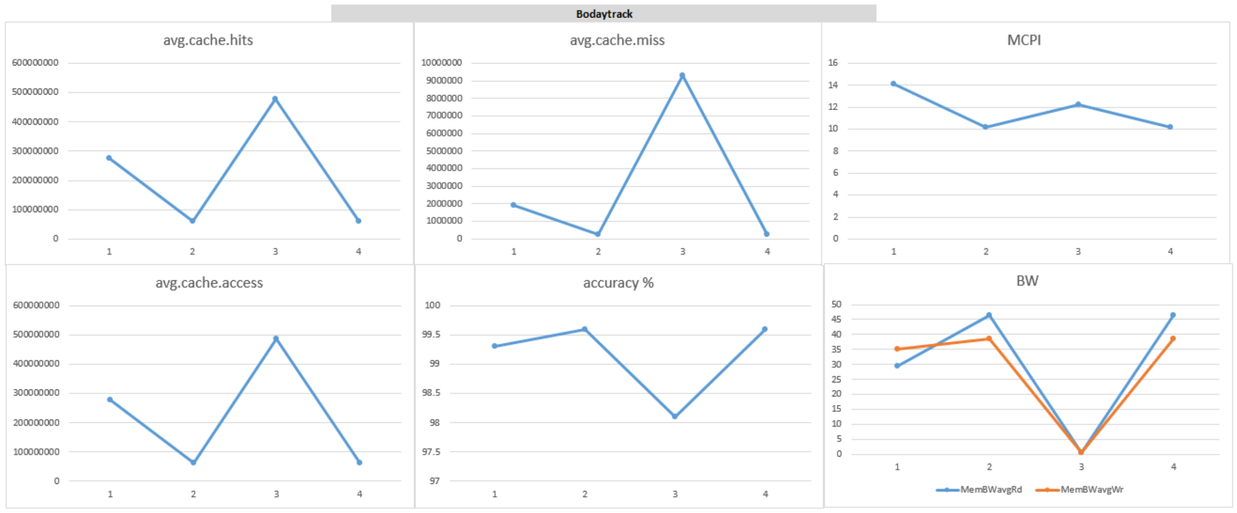

Figure 10: Bodytrack beanchmark results

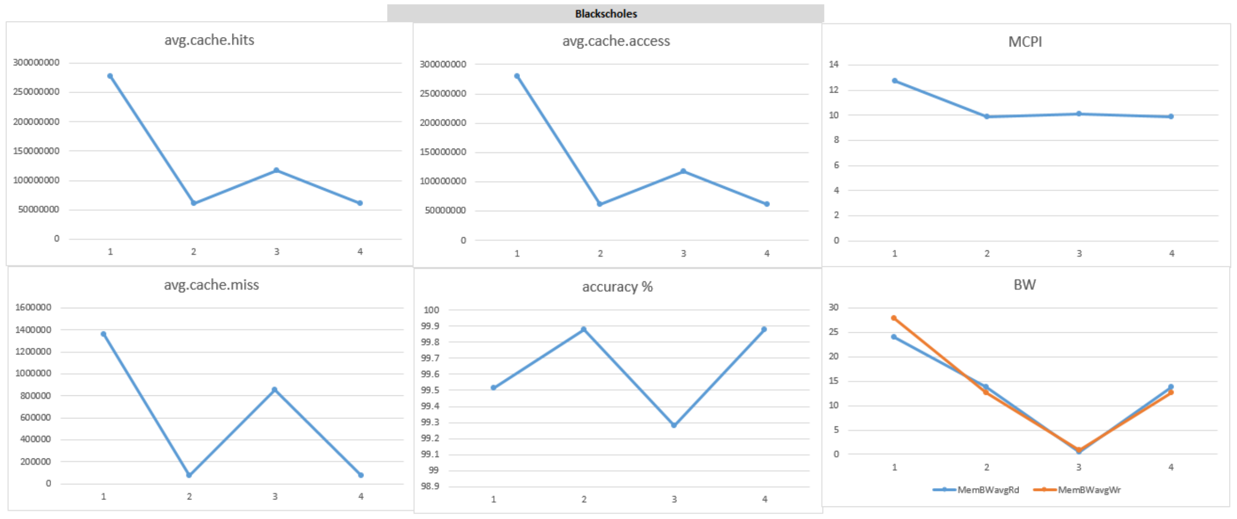

Figure 11: blackscholes benchmark results

Discussions:

Based on the Table 1 and figures 7-11 following points can be derived:

- Current implementation of Markov prefetcher induces lot more requests than the other prefetcher which is clearly visible in the graph 3,4,5 in x264. (similar for fluidanimate). This is occurring primarily due to more addresses getting enqeueued in prefetch stage. Whenever the bus is free, the memory controller schedules memory request. Current implementation should have added a minimum of 4X more memory requests[4]. However it is introducing more than 7x memory requests. The additional bandwidth I suspect is due to current implementation of m_array table where LRU is not required. Ideally the highest priority in the markov table should be sent as memory request. But current implementation does not guarantee that due to the presence of LRU.
- Displacement of useful data is definitely increased in Markov implementation. Increased bandwidth is not translating to average cache access which means that there are lot of

unnecessary data just being requested for no purposes. This can introduce latency for required data for the processor and slow the process down. MCPI reflects this observation clearly for markov when compared to stride. Considering total simulation time, then markov based took about 10% more simulation time over stride implementation. (Note this is not gem5 time, it is compute time inside gem5 that is been discussed).

- Third main observation (not captured in table) goes over non-LRU implementation of the Markov table. In this configuration, the performance of Markov prefetcher was low with accuracy of 95%. This can be attributed to wrong priorities of miss addresses getting accessed. With LRU on markov Table, accuracy improved by about 4%. The LRU implementation was done as per suggestion in [4] and non-LRU implementation statistical data is not provided in the report. It is interesting to note that without the LRU the performance did not drop a lot. This can be due to two main reasons:
    a. Most the addresses were repeated or were not dependent addresses.
    b. There were more CPU requests which had higher priority over prefetcher. The controller can always drop the prefetcher request if CPU is requesting for memory.
- For calculation of coverage, I considered MI_stride x264 benchmark as baseline. (Although I agree that is not correct since each benchmark has different compute and one needs to check with respect to each other, however I wanted to understand how benchmark data effect the prefetcher accuracy and hence I took this approach). Coverage is average cache miss relativistic to baseline. It is quite interesting to see that coverage significantly varies across different cache coherence and prefetcher implementation but is similar across all benchmarks. Secondly, coherence has a say on coverage. This can be attributed to how different individual coherence is achieved. The MESI introduce two new state – E and S. However, S state is the reason why the coverage is improved. In the given coherence scheme, prefetching scheme has variations. It is interesting to note that the Markov prefetcher has highest coverage even with LRU implementation. In [4], they did mention that the coverage would be increased for markov and is similar to current observation.
- Markov Prefetcher does provide better cache hits over stride and is very similar to observation in [4]. This can be attributed to two main points – Markov table and more requests for different address.
- Timeliness is subjective to discuss in current implementation for two possible reasons. Timeliness depends on cache coherence and also depends on how much useful data have been displaced. Hence, I disagree with [4] definition. For sake of discussion, I have measured timeliness as per [4] definition. For x264, Markov model does take more MCPI when compared to stride which contradicts [4]. This is because prefetcher is introducing lot of unnecessary data request and polluting the required memory request. Although LRU should have fixed this issue but I am unsure why still MCPI is large.
- Among all the prefetchers, the tagged prefetcher does the best overall. I am not sure why this happens. Perhaps, I did not code well ☺

Ending thoughts:

- The first part of the assignment was way more complicated than previous assignments. Further, there were so many directionless points. I felt that the complication was not worth the 3% of grade associated with it as it consumed lot of quality time of other projects, coursework and research. Personally, I felt the timeline and quantity of work are not defined well for checkpoint.
- It turns out that the misaddress can be zero. This scenarios requires special handling in the code.

- I found a strange issue and have not figure out a fix yet. Gem5.opt crashes whenever I run normally. However, gem5.debug works smoothly for the given benchmark and code. I am yet to trace the real issue and hence I could not report the statistics for bodytrack and blackscholes. The benchmark runs are hardly any relevant since the observation across benchmarks is consistent based. The only difference that can be seen is the coverage across different benchmarks however, it does have a consistent behavior as discussed in previous section.
- Initially I thought of taking the extension but I decided to do it if required for final submission. Taking extension is just avoiding a problem or pushing it forward. I prefer to accept it. (Personally!)
- I found another interesting point – repeatability and reproducibility. The bug was reproducible in other machines. However, the stats that I got are not same in other machine!!! I have not figure out why this happens but yes, if you run – you may not get the same data but will get similar data. (this does not make sense logically but I don't have explanation for this!)

Next Steps:

1. Implement MSI
2. Fix "mystery bug issues" in Markov prefetcher
3. Optimize markov prefetcher.

References:

1. Gem5.org
2. www.learning.gem5.org/book/index.html
3. http://courses.engr.illinois.edu/ece511/secure/homework/assignment3.pdf
4. D. Joseph and Dirk Grunwald, "Prefetching using Markov Predictor"