ECE 511 Assignment 3

Markov Data Prefetcher and MSI Coherence

Vikram Sharma Mailthody

(NetID- vsm2, UIN - 663250535)

Contents

[Goals of the Assignment: 3](#_Toc496984141)

[Software used: 3](#_Toc496984142)

[Tasks: 3](#_Toc496984143)

[Introduction: 3](#_Toc496984144)

[Report organization: 4](#_Toc496984145)

[Default Branch Predictors: 4](#_Toc496984146)

[Baseline architecture: 4](#_Toc496984147)

[Stride Prefetcher: 4](#_Toc496984148)

[Markov Prefecher: 5](#_Toc496984149)

[Working principle: 5](#_Toc496984150)

[Modifications in default repository: 6](#_Toc496984151)

[Implementation details: 7](#_Toc496984152)

[Coherence Protocol: 8](#_Toc496984153)

[1. MI: 8](#_Toc496984154)

[2. MSI: 9](#_Toc496984155)

[3. MESI Coherence protocol: 10](#_Toc496984156)

[MSI Implementation details in gem5: 11](#_Toc496984157)

[Results: 11](#_Toc496984158)

[1. Qualifying MSI coherence implementation: 11](#_Toc496984159)

[2. Discussions: 14](#_Toc496984160)

[Ending thoughts: 16](#_Toc496984161)

[References: 16](#_Toc496984162)

[Appendix A: 17](#_Toc496984163)

# 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](http://courses.engr.illinois.edu/ece511/secure/homework/assignment2.pdf). [10]

# Software used:

[Gem5](file:///C:\Users\LocalAdmin\AppData\Roaming\Microsoft\Word\gem5.org) 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

# 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. 4 benchmarks are used to measure the hardware requirement, displacement of useful data, coverage, timeliness and effect on bandwidth with each of these predictors.

The report discusses about Markov predictor and MSI coherence scheme in detail. High level information of MESI and MI protocol are provided for reference.

# 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

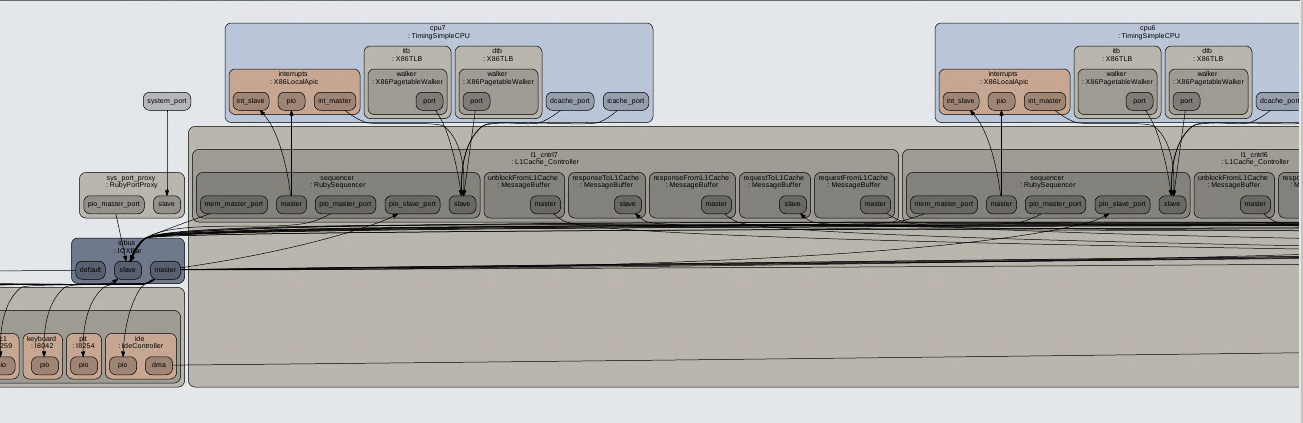


Figure 2: Zoomed version of the same.

## 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.

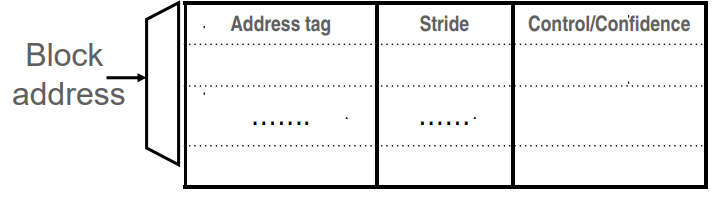


Figure 3:High level view of gem5 stride prefetcher.

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.
   1. If hit, an enetry from FIFO is popped and cache is updated with data
   2. 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.
   3. Issuance of address will not happen at page boundary
   4. 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.

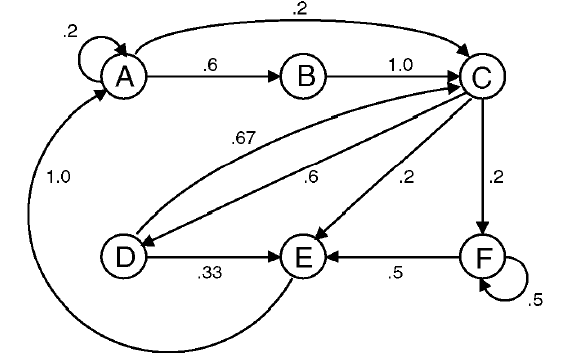


Figure 5: Markov model as described in [4].

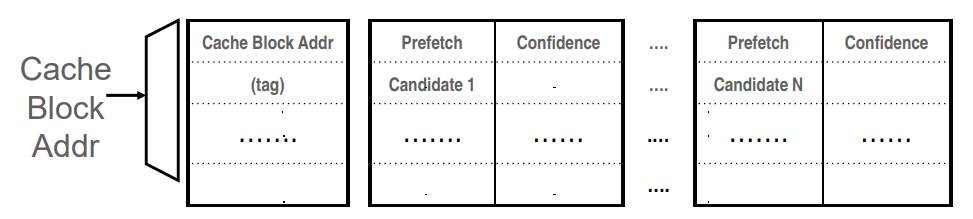


Figure 6: Markov model implemented in gem5.

## 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:

1. 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.
2. 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.
3. M\_ARRAY table: This table has list of addresses that needs to be prefetched. Prefetcher requests for data from memory controller one by one depending on LRU scheme. This is different from checkpoint implementation where data was prefetched in stride manner. 4 requests where generated per address fetch from Markov table. In final report this is optimized by having one most recently used element requested per Markov entry.
4. 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.
5. 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:

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 as this large size is not required to obtained good results.
3. LRU implementation is done on the array update itself to reduce the additional time
4. LRU in m\_array requires to be removed as it’s a duplicate.

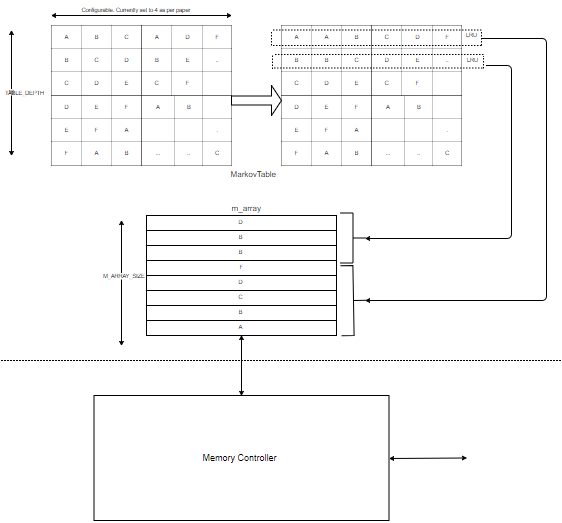


Figure 7: Markov Prefetcher

## Coherence Protocol:

### MI:

Simplest of all the coherence protocol which has only valid and invalid state. The state transition of MI protocol available in gem5 is shown in figure 8. The slicc transition table for the MI protocol is shown in figure 9.

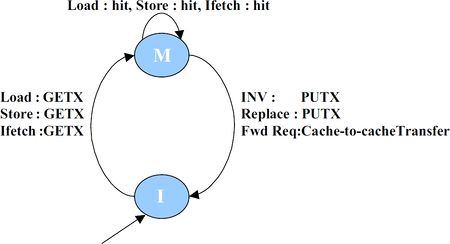


Figure 8: MI State transition.

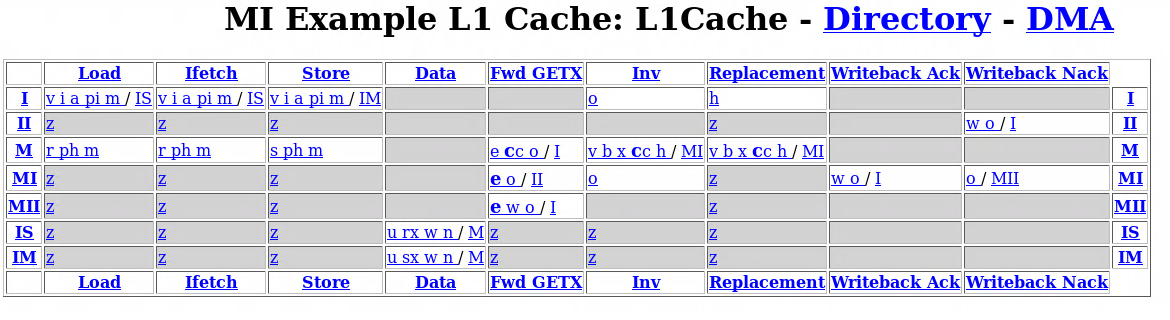


Figure 9: MI Transition table

### MSI:

MSI adds a new state S over MI for reducing memory traffic and sending the data from the cache. The state transition diagram of the MSI as described by Y.Solihin in “Fundamentals of Parallel Computer Architecture (2009)” is shown in figure 10. However when implemented in gem5, the intermediate states have to be taken into account. The gem5 implementation of the MSI protocol with prefetcger is shown in figure 11.

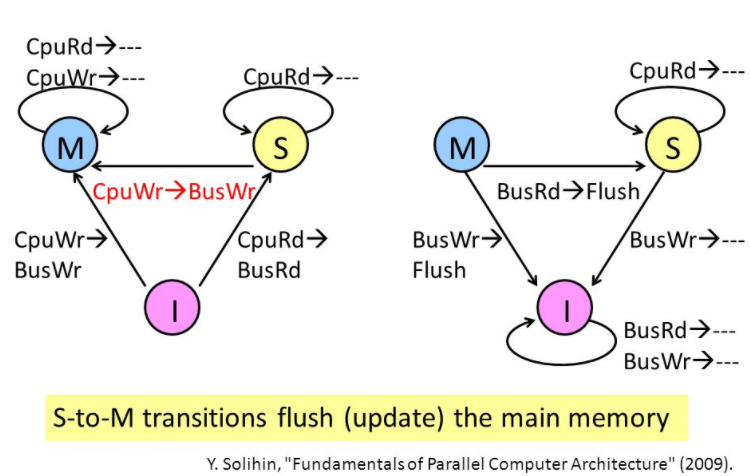


Figure 10: MSI Coherence protocol.

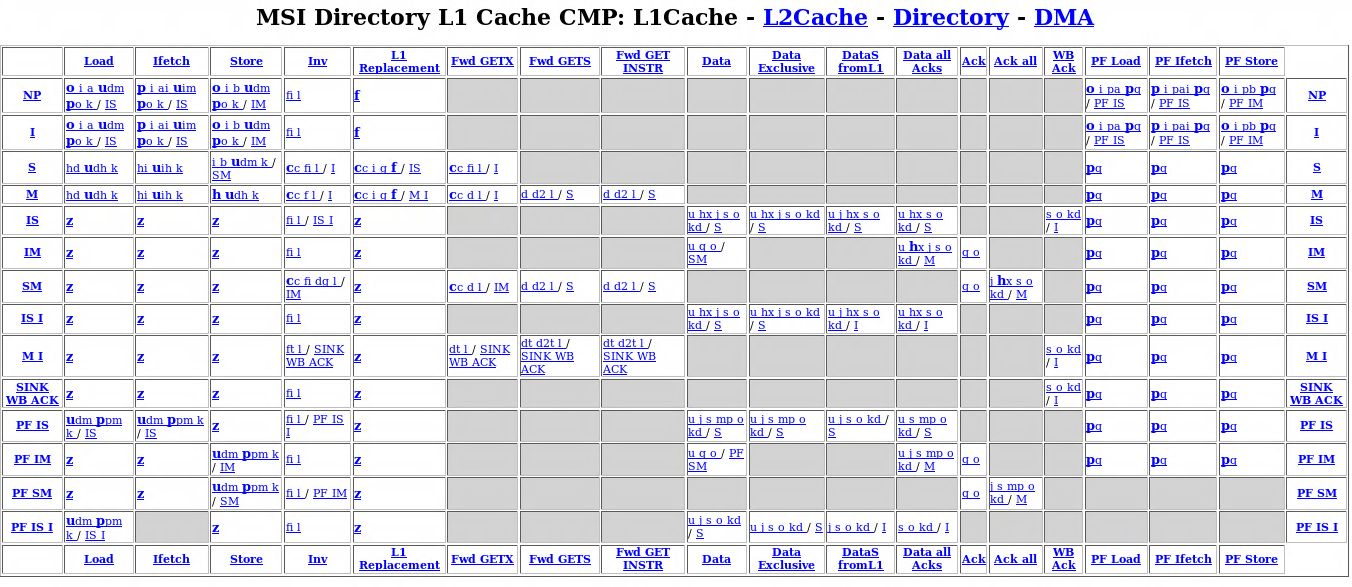


Figure 11: MSI gem5 implementation.

### MESI Coherence protocol:

MESI extends MSI to handle exclusive requests from the processor by adding a new state E. If the read request is hit then the state transitions to shared state otherwise goes to exclusive state. This reduces the overhead of coherence in the bus and improves the performance. The complete state change diagram is well represented by Prof Yale Patt in one of this lecture notes as shown in figure 12.

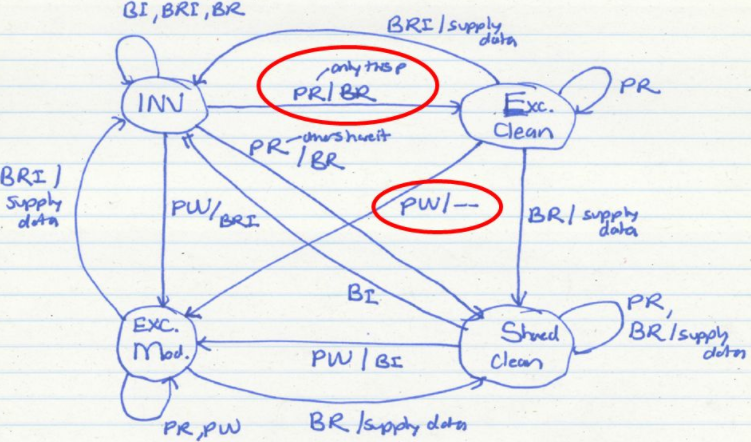


Figure 12: MESI as described by Prof Yale Patt.

For implementing in gem5, state transitions have to be taken care. Hence the state transition table of gem5 MESI protocol with prefetcher is shown in figure 13.

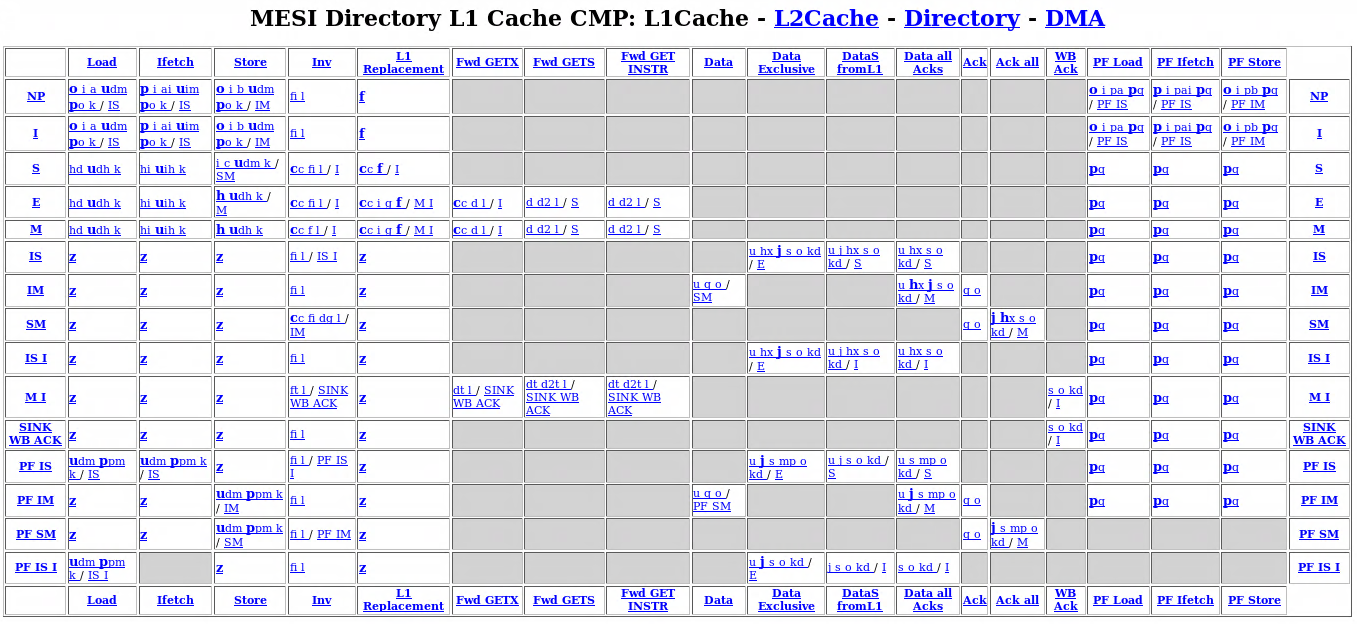


Figure 13: MESI State transition.

## MSI Implementation details in gem5:

I considered baseline as MESI to implement MSI. MI protocol did not support prefetching. Two versions of the MSI protocol was developed.

In version one, I removed all state transitions happening to E. Next I took a hacky approach where I short circuited paths of Data\_exclusive request as data request and converted E->S. I kept all events of Data\_Exclusive. The reason is, If you are Modified, other processors should treat your data as exclusive. MSG file comments indicate Data Exclusive is used for E/M. As long as E is removed properly, M should still be able to use the Data Exclusive, for whatever reason they needed it. Added a transition from S->I upon seeing FWD\_GETX. This scenario can still happen when in shared state and it must handle how E was handling it. Changed all E states to S state in IS transitions.

In version two, I removed all state transitions happening to E. To handle L1-Upgrade in MT (L2 Cache), I added L1\_Upgrade to act as L1\_GETX. This resulted in an error in L1 receiving Fwd\_getx/fwd\_get\_instr/fwd\_gets. I further handled it by transition it to S state. All exclusive unblocks were modified into normal unblocks. To handle appropriate L1\_replacement in S state, I modified S->IS transition with additional actions. Resulting WB\_ack was used to move to invalidate state. I have missed a few minor changes that I had to do in L2 and L1 controller to handle proper data movement. Figure 11 state transition diagram represents all changes done on version two.

# Results:

## Qualifying MSI coherence implementation:

It is very difficult to qualify if the implementation is right when transition state is involved. One can certainly go across different state transitions in MSI state transition table. However, it is time consuming and minor issues still remain. I qualified the appropriateness by looking into statistics obtained by gem5 runs and checked with expectations.

1. MSI has lower demand data hit compared to MESI.
2. Memory Bandwidth consumed should be more than MESI

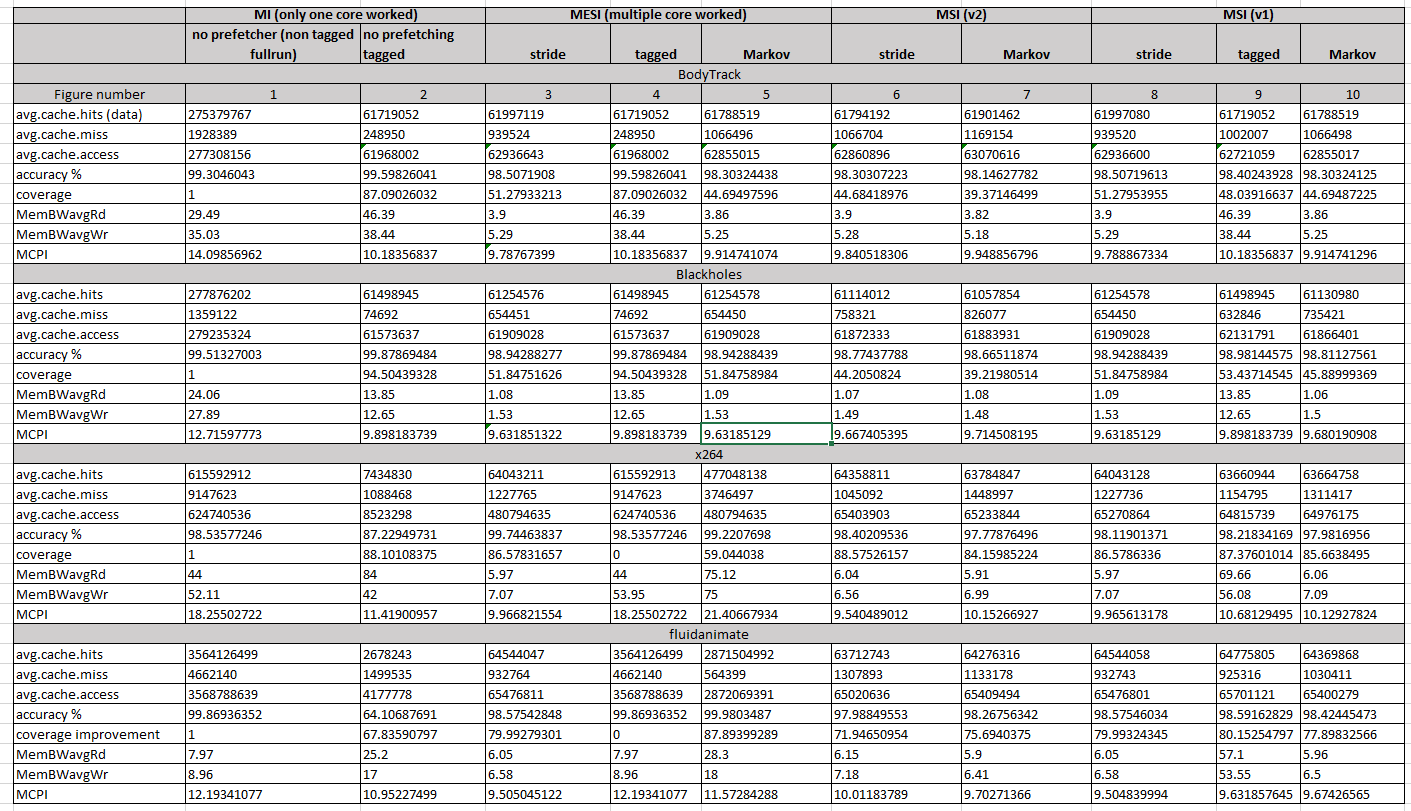
It is quite interesting to note that both versions of MSI implementation satisfied these criteria. However, on further analysis, it turned out that these metrics hardly provided actual implementation data. One right approach to understand which implementation is appropriate is to look in number of state transitions of S in MSI and E/S in MESI. The sum of E/S in MESI and S in MSI should be of similar number (cannot be exact because of prefetcher, etc implementation). I checked this data to qualify which version of coherence protocol is better. V1 and v2 coherence protocol stat screen shot is shown in table 1. (Can check Appendix A. Figure 1 and Figure 2 for exact data)

Table 1: MSI v1 and v2 S transition comparison

|  |  |  |
| --- | --- | --- |
|  | **v1** | **v2** |
| **MESI S** | 555 | 517 |
| **MESI E** | 18306139 | 15085931 |
| **MSI S** | 511 | 15702926 |
| **deviation %** | -99.99721 | 4.086303 |

From the table 1, it is clear that just looking at the cache hit/bandwidth metrics are insufficient. Further, the hacky version provides good data but that data is of no use! The version 2 however is a better implementation and used for studying variations in Markov prefetcher.

Table 1: Results of benchmarks and markov prefetcher



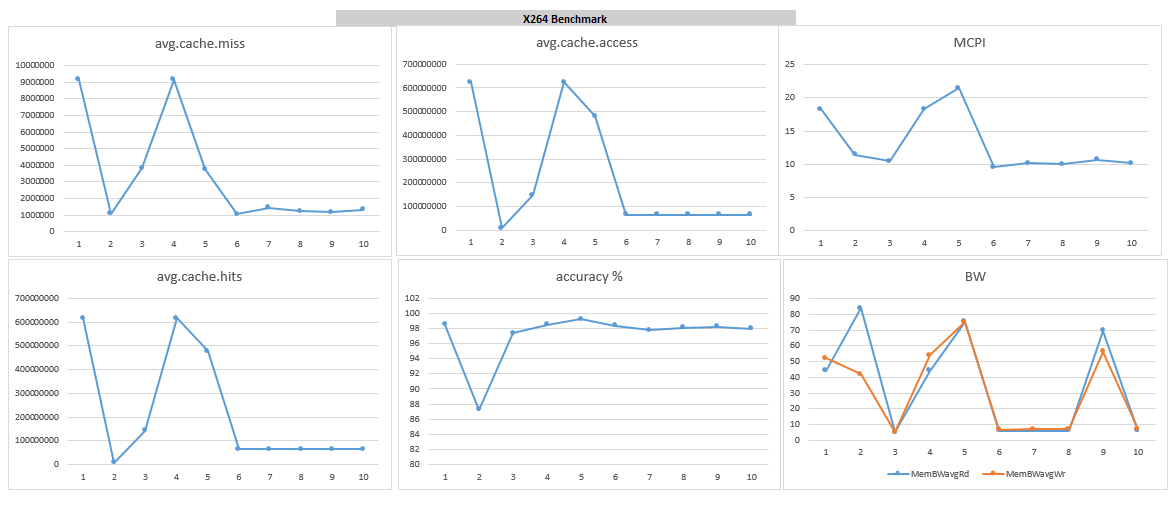


Figure 8: X264 benchmark results

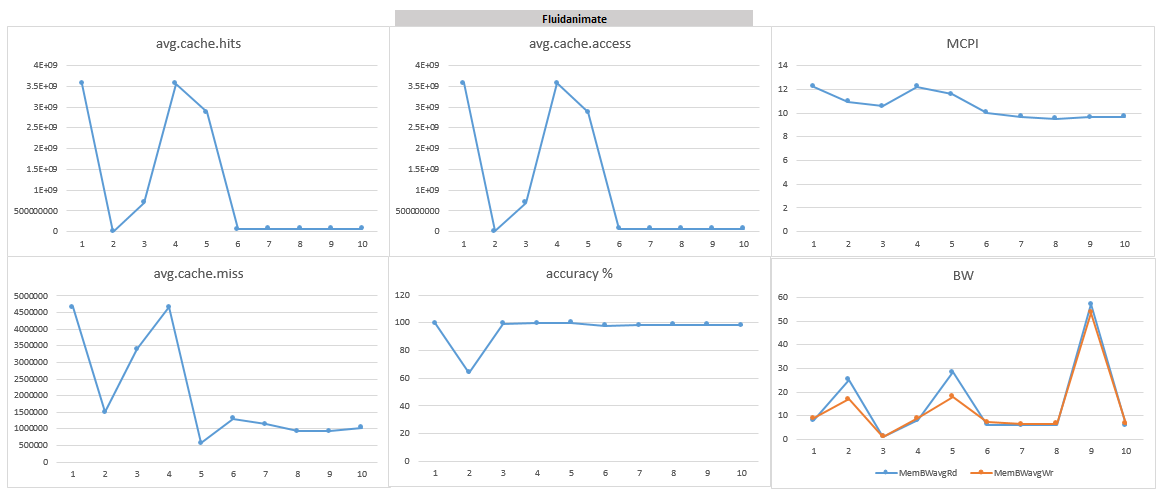


Figure 9: Fluidanimate benchmark results

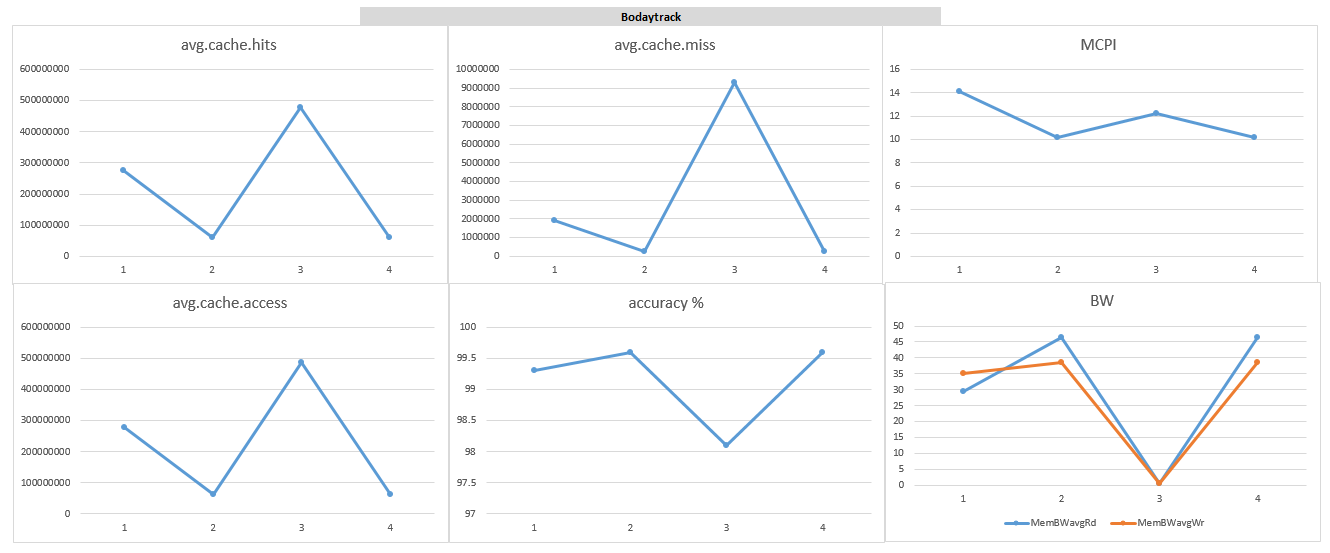


Figure 10: Bodytrack beanchmark results

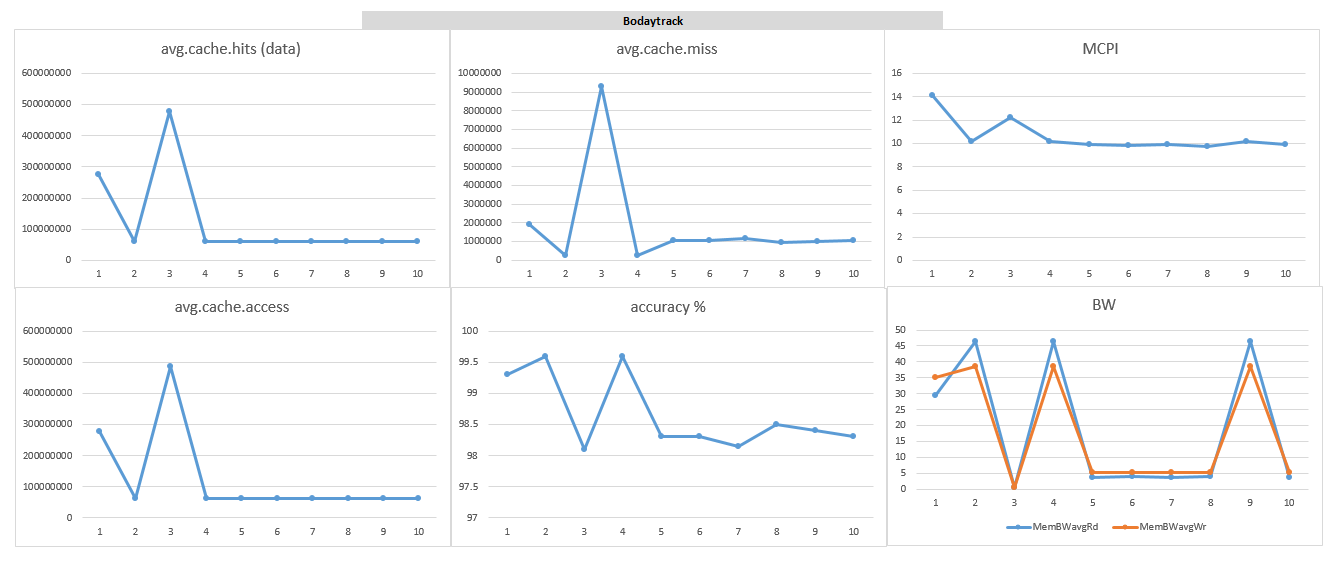


Figure 11: blackscholes benchmark results

## 2. Discussions:

For calculating the cache hits, I have consider only data demand cache hits, misses. All numbers represent for data and excludes the instructions for fair comparison. Coverage is defined average cache miss relativistic to baseline (MI non tagged/no prefetcher) of respective benchmark.

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

* Current implementation of Markov prefetcher induces lot less requests than the checkpoint implementation of the prefetcher. Checkpoint report had higher number primarily due to more addresses getting enqeueued in prefetch stage. Whenever the bus is free, the memory controller schedules memory request. Current implementation schedules the highest priority in the markov table.
* It is quiet surprising to see tagged prefetcher requests for more data and has higher bandwidth when compared to stride and markov. I am not sure why this is caused.
* Between the stride and markov prefetcher, the markov has higher bandwidth and demand hit rates. However, it is clear that the bandwidth was primarily dependent on the kind of coherence scheme used. MESI has lower bandwidth compared to MSI. There is slight reducing in bandwidth in markov over stride when comparing in MSI.
* Data Demand cache misses are more frequent in Markov prefetcher with MSI. However comparing with baseline of MI implementation, the coverage of markov is lower. This is in contrast with what was proposed in paper. I believe this is because of optimization I performed in implementation specifically handling LRU scheme on stream buffer and requesting of only the most frequently used data from the markov table entry.
* 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).
* Next 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:

1. Most the addresses were repeated or were not dependent addresses.
2. 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.

* Timeliness is subjective to discussion in current implementation for two reasons. Timeliness depends on cache coherence scheme and on displacement of useful data. Hence, I disagree with [4] definition. For sake of discussion, I have measured timeliness as per [4] definition. MCPI of markov is very similar to rest of prefetcher scheme. It is slightly higher in Markov and with MESI, it lowers by negligible numbers.
* To check if the cache coherence is performing right, one should look into the number of state transitions or check the state transition table as discussed in section 1 of this chapter.
* When compared to MESI, MSI has lower cache hits, higher cache miss and lower accuracy. This is mainly because of multiple senders scenario that can occur in MSI and also due to unnecessarily congestion of bus due to exclusive requests. For body track MESI implementation, had 99.99% of data as exclusive data request. Markov prefetcher although did improve the cache hit rates, it increased the miss rates too. I believe this the memory controller was ignoring the prefetcher requests by prioritizing the exclusive memory requests that was occurring. Hence the memory bandwidth was similar with Markov prefetcher implementation.
* From the table, benchmarks has determination of performance. This is because, not all benchmarks are primarily requesting for exclusive data requests.
* For MSI, transitions on IS state should be larger when compared to MESI. This can be observed in the stats provided. MSI has 14.68% higher IS state transition over MESI.

|  |  |  |
| --- | --- | --- |
|  | MESI | MSI |
| IS | 460721 | 569645 |

# Ending thoughts:

* It turns out that the mis-address can be zero. This scenarios requires special handling in the code.
* The strange issue that I spoke about in checkpoint report, (debug working and opt not?). I found the issue. It was the way in which I was implementing my markov prefetcher. I created a race code which is timing dependent! I know how to create race in gem5 now! ☺
* 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 not to accept it. (Personally!) However, I did not want the extension flexibility to go waste. ☺ So I took it for final for sake of taking it!
* It turns out ruby based model is very easy if you understand what you are doing. However debug options are hard to get. I checked Jason Power book and there was a ruby debugger methodology. I shared it with full class (through Andrew) and I think that helped. ☺ [5][6]
* If this assignment is planned again for next year, then I suggest to split it into 2 assignment. One for Markov and one more for MSI. Combing creates lot of complexity due to dependency.
* I disagree with others claim that it is impossible to qualify if the implementation is right. There are ways to see at high level if it is appropriate. Further, for class assignment high level accuracy is sufficient. Graduate courses should provide open ended questions like this to enrich student thinking ability and I appreciate this question now! (Yes, in checkpoint I complained about complexity! And I am wrong. I stand corrected now!).
* I am sad that discussion in piazza did not happen. However, I must thank Umur and Andrew (my team mates) who discussed different issues they were running out. When some of those issues I faced, I was able to correct them faster. Assignment goals should encourage discussion. When I was working (previous company), we used to have hours of discussion to come up with innovative designs and ideas. Ideas accumulated. However a large group is a mess. I think group (of 2 or 3) assignment are better than individual one for graduate class. I did do a small study across other universities. Stanford and CMU encourages to have group assignment for advance computer architecture course.

# References:

1. Gem5.org
2. [www.learning.gem5.org/book/index.html](http://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”
5. learning.gem5.org/book/part3/MSI/cache-transitions.html
6. learning.gem5.org/book/part3/MSI/debugging.html

# Appendix A:

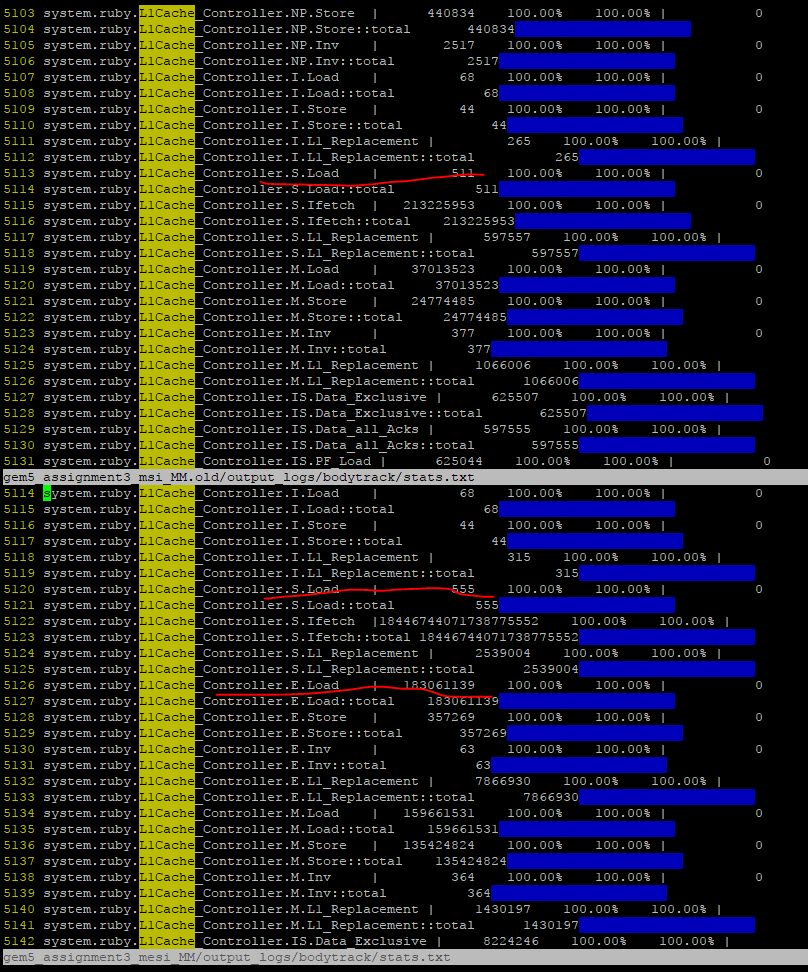


Figure 1: MSI v1 coherence protocol stats showing S/E transition values.

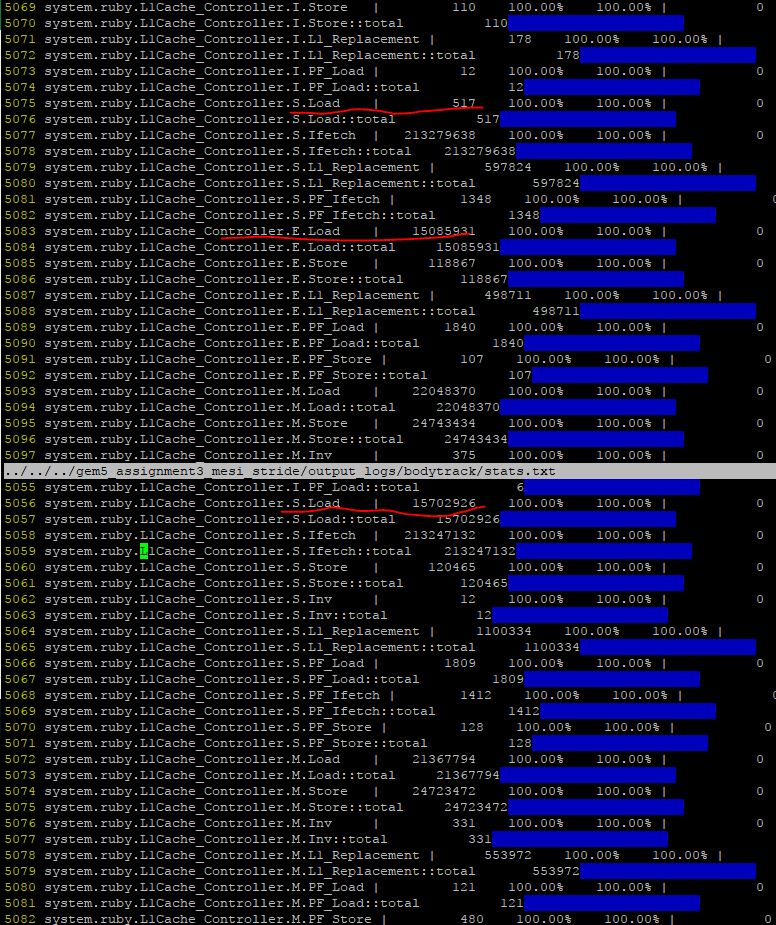


Figure 2: MSI v2 coherence protocol stats showing S/E transition values.