ECE 511 Assignment 3
Markov Data Prefetcher and MSI Coherence

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

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

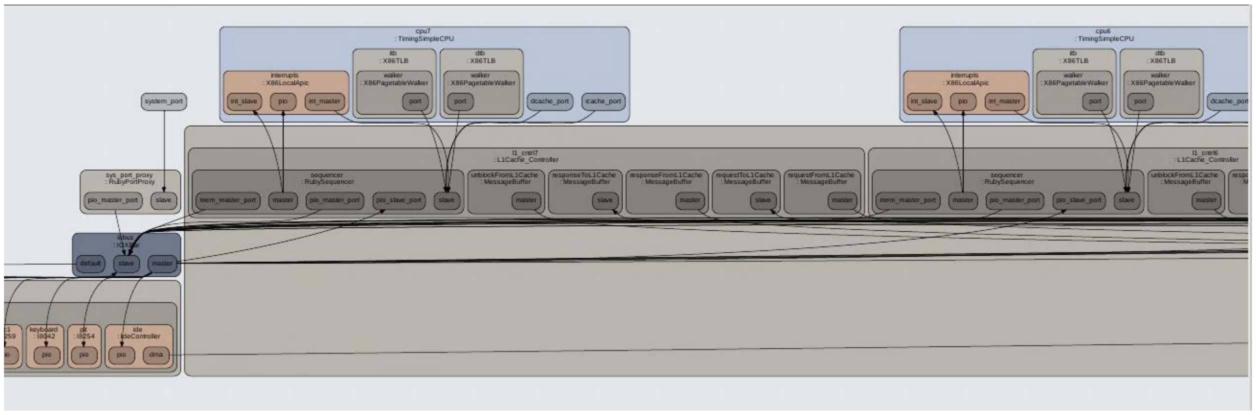


Figure 2: Zoomed version of the same.

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.

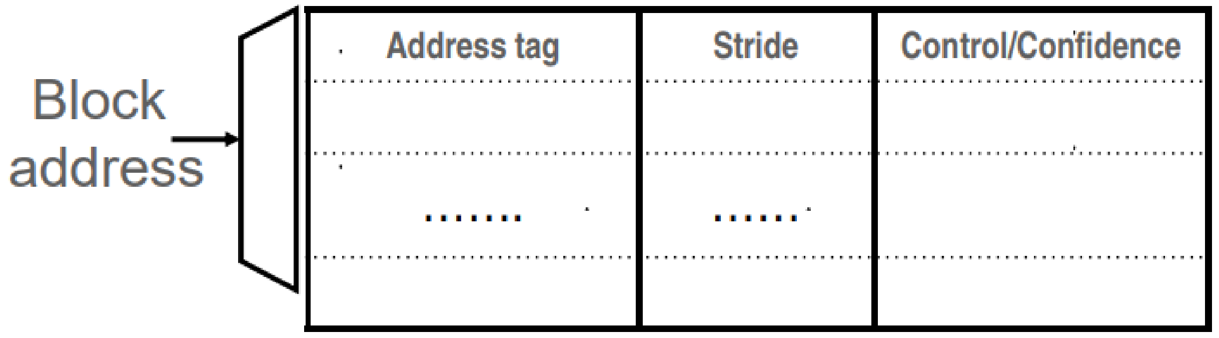


Figure 3:High level view of gem5 stride prefetcher.

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.
    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.

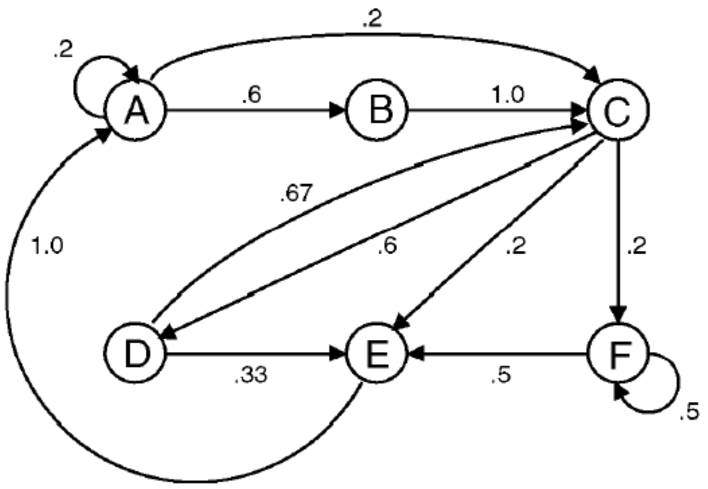


Figure 5: Markov model as described in [4].

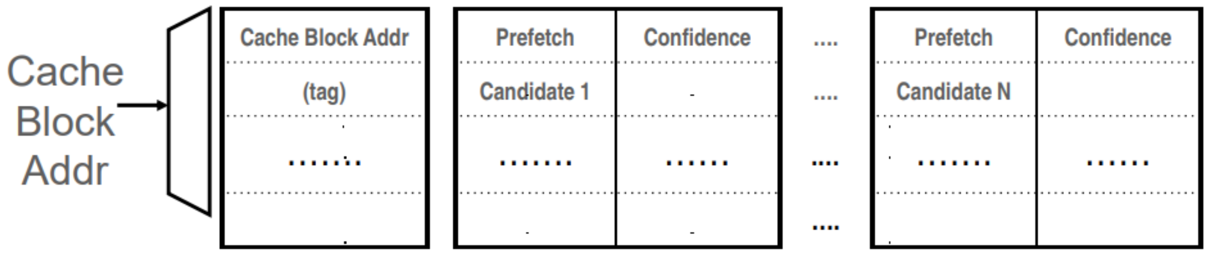


Figure 6: Markov model implemented in gem5.

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. 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.

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:

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.

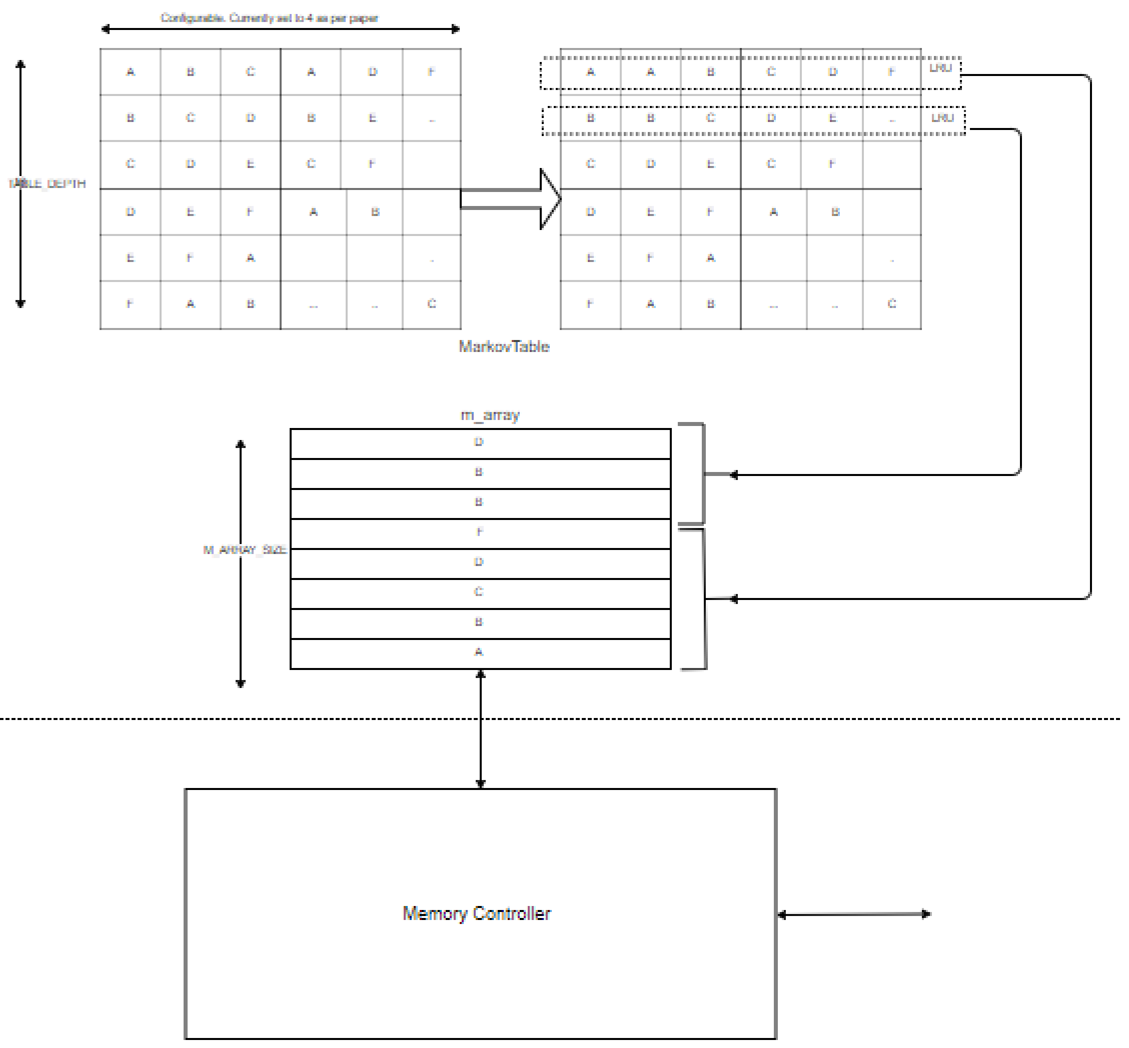


Figure 7: Markov Prefetcher

Coherence Protocol:

1. 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.

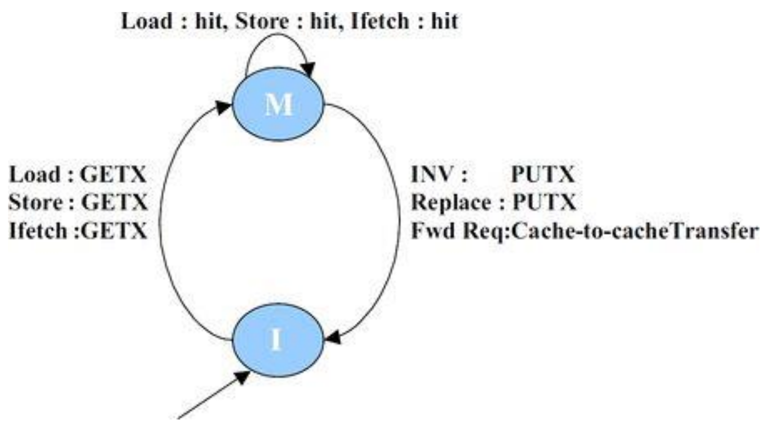
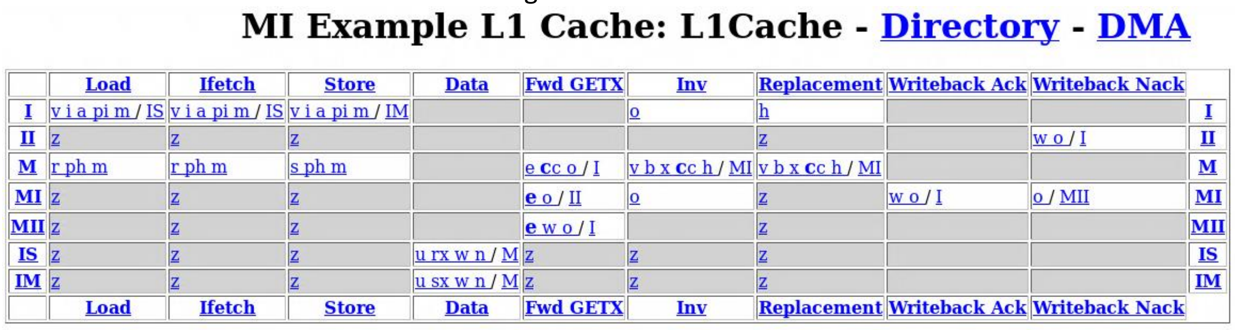

Load : hit, Store : hit, Ifetch : hit
M
Load : GETX
Store : GETX
Ifetch :GETX
INV : PUTX
Replace : PUTX
Fwd Req:Cache-to-cacheTransfer
I

Figure 8: MI State transition.



MI Example L1 Cache: L1Cache - Directory - DMA

	Load	Ifetch	Store	Data	Fwd GETX	Inv	Replacement	Writeback Ack	Writeback Nack	
I	v i a pi m / IS	v i a pi m / IS	v i a pi m / IM			o	h			I
II	z	z	z				z		w o / I	II
M	r ph m	r ph m	s ph m		e cc o / I	v b x cc h / MI	v b x cc h / MI			M
MI	z	z	z		e o / II	o	z	w o / I	o / MII	MI
MII	z	z	z		e w o / I		z			MII
IS	z	z	z	u rx w n / M	z	z	z			IS
IM	z	z	z	u sx w n / M	z	z	z			IM
	Load	Ifetch	Store	Data	Fwd GETX	Inv	Replacement	Writeback Ack	Writeback Nack	

Figure 9: MI Transition table

2. 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.

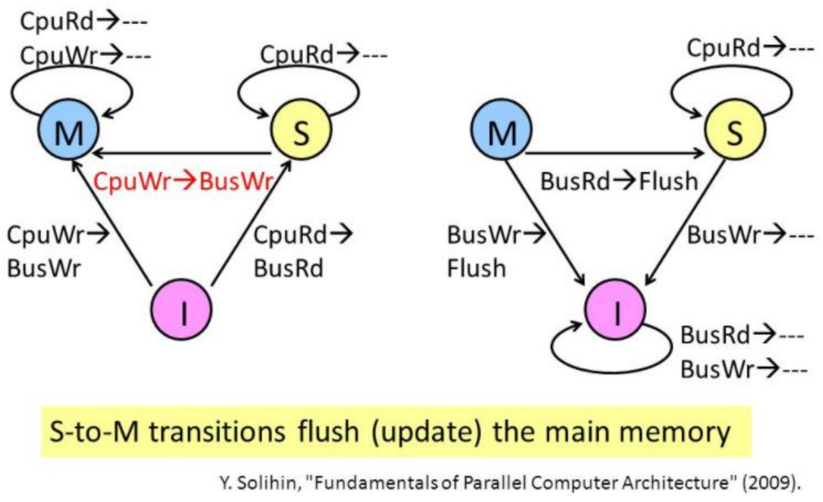
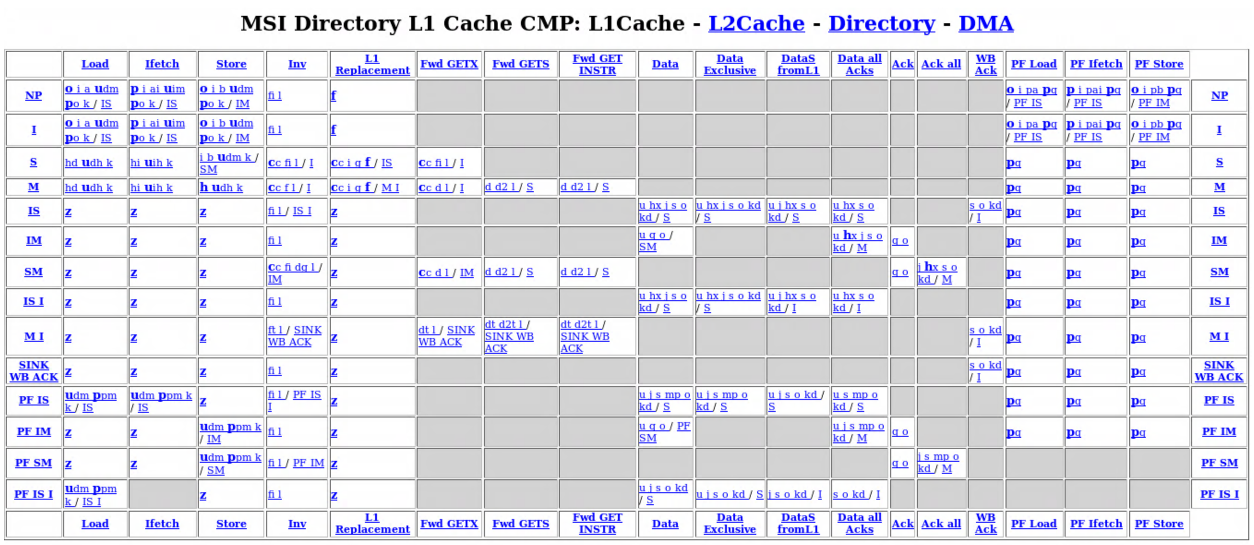


Figure 10: MSI Coherence protocol.



MSI Directory L1 Cache CMP: L1Cache - L2Cache - Directory - DMA

	Load	Ifetch	Store	Inv	L1 Replacement	Fwd GETX	Fwd GETS	Fwd GET INSTR	Data	Data Exclusive	DataS fromL1	Data all Acks	Ack	Ack all	WB Ack	PF Load	PF Ifetch	PF Store	
NP	o i a udm po k / IS	p i ai uim po k / IS	o i b udm po k / IM	fi l	f											o i pa pq / PF IS	p i pai pq / PF IS	o i pb pq / PF IM	NP
I	o i a udm po k / IS	p i ai uim po k / IS	o i b udm po k / IM	fi l	f											o i pa pq / PF IS	p i pai pq / PF IS	o i pb pq / PF IM	I
S	hd udh k	hi uih k	i b udm k / SM	cc fi l / I	cc i q f / IS	cc fi l / I										pq	pq	pq	S
M	hd udh k	hi uih k	h udh k	cc f l / I	cc i q f / M I	cc d l / I	d d2 l / S	d d2 l / S								pq	pq	pq	M
IS	z	z	z	fi l / IS I	z				u hx j s o kd / S	u hx j s o kd / S	u j hx s o kd / S	u hx s o kd / S			s o kd / I	pq	pq	pq	IS
IM	z	z	z	fi l	z				u q o / SM			u hx j s o kd / M	q o			pq	pq	pq	IM
SM	z	z	z	cc fi dq l / IM	z	cc d l / IM	d d2 l / S	d d2 l / S					q o	j hx s o kd / M		pq	pq	pq	SM
IS I	z	z	z	fi l	z				u hx j s o kd / S	u hx j s o kd / S	u j hx s o kd / I	u hx s o kd / I				pq	pq	pq	IS I
M I	z	z	z	ft l / SINK WB ACK	z	dt l / SINK WB ACK	dt d2t l / SINK WB ACK	dt d2t l / SINK WB ACK							s o kd / I	pq	pq	pq	M I
SINK WB ACK	z	z	z	fi l	z										s o kd / I	pq	pq	pq	SINK WB ACK
PF IS	udm ppm k / IS	udm ppm k / IS	z	fi l / PF IS I	z				u j s mp o kd / S	u j s mp o kd / S	u j s o kd / S	u s mp o kd / S				pq	pq	pq	PF IS
PF IM	z	z	udm ppm k / IM	fi l	z				u q o / PF SM			u j s mp o kd / M	q o			pq	pq	pq	PF IM
PF SM	z	z	udm ppm k / SM	fi l / PF IM	z								q o	j s mp o kd / M					PF SM
PF IS I	udm ppm k / IS I		z	fi l	z				u j s o kd / S	u j s o kd / S	j s o kd / I	s o kd / I							PF IS I
	Load	Ifetch	Store	Inv	L1 Replacement	Fwd GETX	Fwd GETS	Fwd GET INSTR	Data	Data Exclusive	DataS fromL1	Data all Acks	Ack	Ack all	WB Ack	PF Load	PF Ifetch	PF Store	

Figure 11: MSI gem5 implementation.

3. 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.

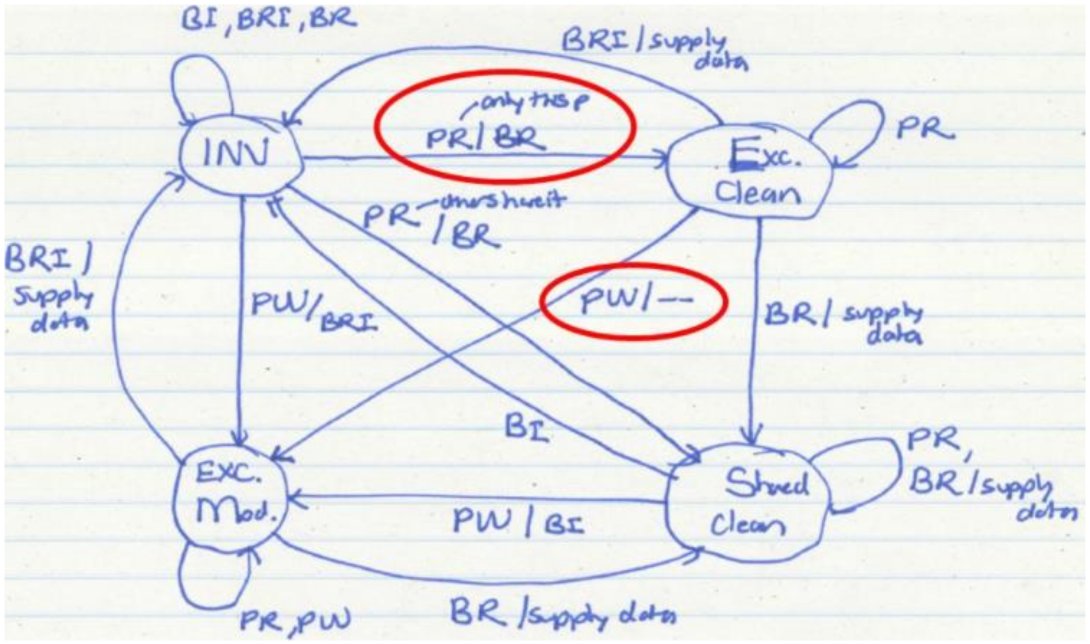
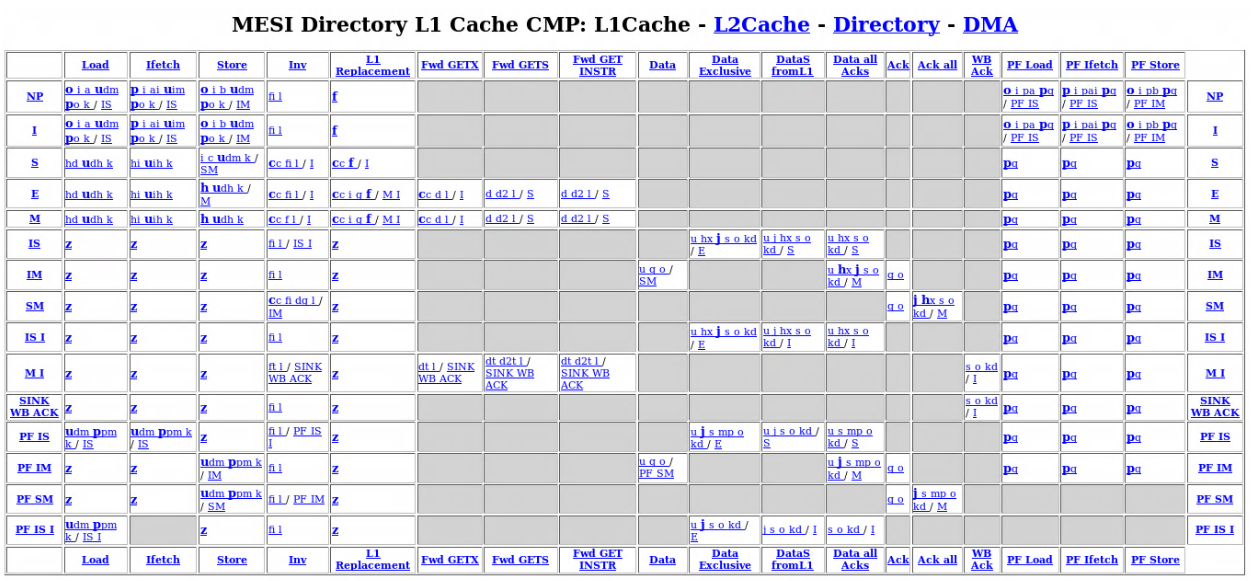


Figure 12: MESI as described by Prof Yale Patt.

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.



MESI Directory L1 Cache CMP: L1Cache - L2Cache - Directory - DMA

	Load	Ifetch	Store	Inv	L1 Replacement	Fwd GETX	Fwd GETS	Fwd GET INSTR	Data	Data Exclusive	DataS fromL1	Data all Acks	Ack	Ack all	WB Ack	PF Load	PF Ifetch	PF Store	
NP	o i a udm po k / IS	p i ai uim po k / IS	o i b udm po k / IM	fi l	f											o i pa pq / PF IS	p i pai pq / PF IS	o i pb pq / PF IM	NP
I	o i a udm po k / IS	p i ai uim po k / IS	o i b udm po k / IM	fi l	f											o i pa pq / PF IS	p i pai pq / PF IS	o i pb pq / PF IM	I
S	hd udh k	hi uih k	i c udm k / SM	cc fi l / I	cc f / I											pq	pq	pq	S
E	hd udh k	hi uih k	h udh k / M	cc fi l / I	cc i g f / M I	cc d l / I	d d2 l / S	d d2 l / S								pq	pq	pq	E
M	hd udh k	hi uih k	h udh k	cc f l / I	cc i g f / M I	cc d l / I	d d2 l / S	d d2 l / S								pq	pq	pq	M
IS	z	z	z	fi l / IS I	z					u hx j s o kd / E	u j hx s o kd / S	u hx s o kd / S				pq	pq	pq	IS
IM	z	z	z	fi l	z				u q o / SM			u hx j s o kd / M	q o			pq	pq	pq	IM
SM	z	z	z	cc fi dg l / IM	z								q o	j hx s o kd / M		pq	pq	pq	SM
IS I	z	z	z	fi l	z					u hx j s o kd / E	u j hx s o kd / I	u hx s o kd / I				pq	pq	pq	IS I
M I	z	z	z	ft l / SINK WB ACK	z	dt l / SINK WB ACK	dt d2t l / SINK WB ACK	dt d2t l / SINK WB ACK							s o kd / I	pq	pq	pq	M I
SINK WB ACK	z	z	z	fi l	z										s o kd / I	pq	pq	pq	SINK WB ACK
PF IS	udm ppm k / IS	udm ppm k / IS	z	fi l / PF IS I	z					u j s mp o kd / E	u j s o kd / S	u s mp o kd / S				pq	pq	pq	PF IS
PF IM	z	z	udm ppm k / IM	fi l	z				u q o / PF SM			u j s mp o kd / M	q o			pq	pq	pq	PF IM
PF SM	z	z	udm ppm k / SM	fi l / PF IM	z								q o	j s mp o kd / M					PF SM
PF IS I	udm ppm k / IS I		z	fi l	z					u j s o kd / E	j s o kd / I	s o kd / I							PF IS I
	Load	Ifetch	Store	Inv	L1 Replacement	Fwd GETX	Fwd GETS	Fwd GET INSTR	Data	Data Exclusive	DataS fromL1	Data all Acks	Ack	Ack all	WB Ack	PF Load	PF Ifetch	PF Store	

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:

1. 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

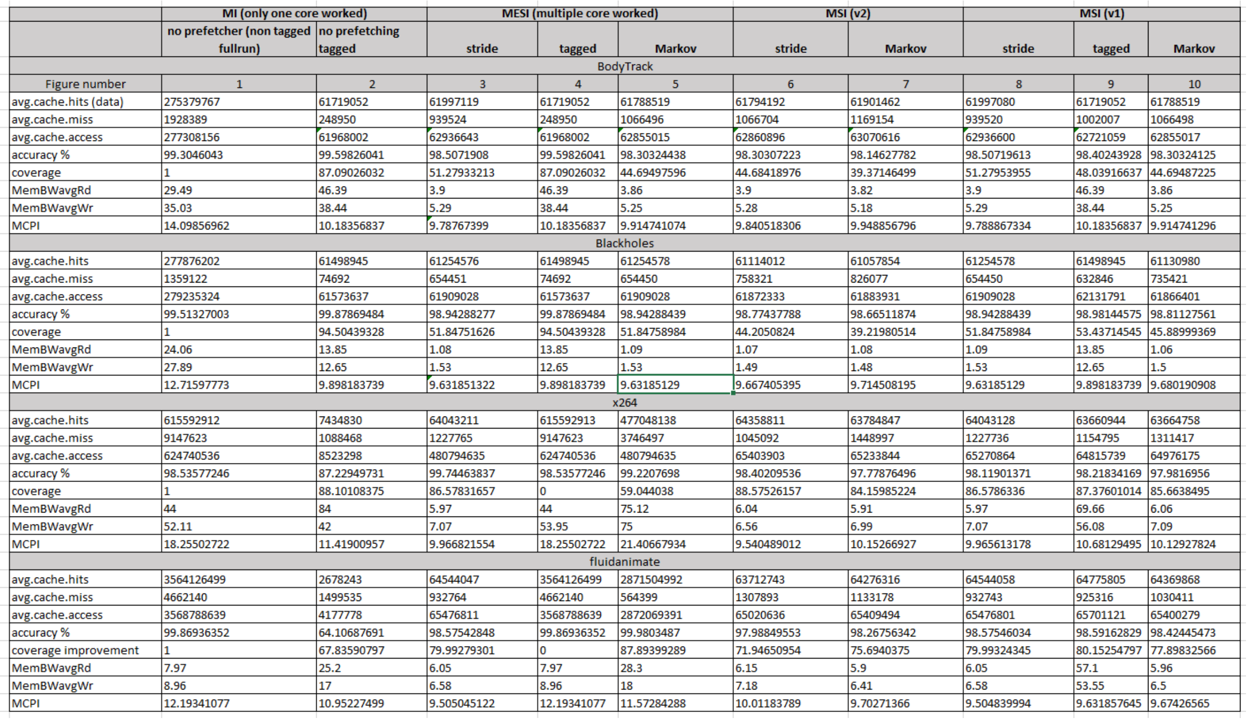
It is quite interesting to note that both versions of MSI implementation satisfied these criteria. However, on further analysis, it turned out that these metrics hardly provided actual implementation data. One right approach to understand which implementation is appropriate is to look in number of state 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



	MI (only one core worked)		MESI (multiple core worked)			MSI (v2)		MSI (v1)		
	no prefetcher (non tagged fullrun)	no prefetching tagged	stride	tagged	Markov	stride	Markov	stride	tagged	Markov
BodyTrack										
Figure number	1	2	3	4	5	6	7	8	9	10
avg.cache.hits (data)	275379767	61719052	61997119	61719052	61788519	61794192	61901462	61997080	61719052	61788519
avg.cache.miss	1928389	248950	939524	248950	1066496	1066704	1169154	939520	1002007	1066498
avg.cache.access	277308156	61968002	62936643	61968002	62855015	62860896	63070616	62936600	62721059	62855017
accuracy %	99.3046043	99.59826041	98.5071908	99.59826041	98.30324438	98.30307223	98.14627782	98.50719613	98.40243928	98.30324125
coverage	1	87.09026032	51.27933213	87.09026032	44.69497596	44.68418976	39.37146499	51.27953955	48.03916637	44.69487225
MemBWavgRd	29.49	46.39	3.9	46.39	3.86	3.9	3.82	3.9	46.39	3.86
MemBWavgWr	35.03	38.44	5.29	38.44	5.25	5.28	5.18	5.29	38.44	5.25
MCPI	14.09856962	10.18356837	9.78767399	10.18356837	9.914741074	9.840518306	9.948856796	9.788867334	10.18356837	9.914741296
Blackholes										
avg.cache.hits	277876202	61498945	61254576	61498945	61254578	61114012	61057854	61254578	61498945	61130980
avg.cache.miss	1359122	74692	654451	74692	654450	758321	826077	654450	632846	735421
avg.cache.access	279235324	61573637	61909028	61573637	61909028	61872333	61883931	61909028	62131791	61866401
accuracy %	99.51327003	99.87869484	98.94288277	99.87869484	98.94288439	98.77437788	98.66511874	98.94288439	98.98144575	98.81127561
coverage	1	94.50439328	51.84751626	94.50439328	51.84758984	44.2050824	39.21980514	51.84758984	53.43714545	45.88999369
MemBWavgRd	24.06	13.85	1.08	13.85	1.09	1.07	1.08	1.09	13.85	1.06
MemBWavgWr	27.89	12.65	1.53	12.65	1.53	1.49	1.48	1.53	12.65	1.5
MCPI	12.71597773	9.898183739	9.631851322	9.898183739	9.63185129	9.667405395	9.714508195	9.63185129	9.898183739	9.680190908
x264										
avg.cache.hits	615592912	7434830	64043211	615592913	477048138	64358811	63784847	64043128	63660944	63664758
avg.cache.miss	9147623	1088468	1227765	9147623	3746497	1045092	1448997	1227736	1154795	1311417
avg.cache.access	624740536	8523298	480794635	624740536	480794635	65403903	65233844	65270864	64815739	64976175
accuracy %	98.53577246	87.22949731	99.74463837	98.53577246	99.2207698	98.40209536	97.77876496	98.11901371	98.21834169	97.9816956
coverage	1	88.10108375	86.57831657	0	59.044038	88.57526157	84.15985224	86.5786336	87.37601014	85.6638495
MemBWavgRd	44	84	5.97	44	75.12	6.04	5.91	5.97	69.66	6.06
MemBWavgWr	52.11	42	7.07	53.95	75	6.56	6.99	7.07	56.08	7.09
MCPI	18.25502722	11.41900957	9.966821554	18.25502722	21.40667934	9.540489012	10.15266927	9.965613178	10.68129495	10.12927824
fluidanimate										
avg.cache.hits	3564126499	2678243	64544047	3564126499	2871504992	63712743	64276316	64544058	64775805	64369868
avg.cache.miss	4662140	1499535	932764	4662140	564399	1307893	1133178	932743	925316	1030411
avg.cache.access	3568788639	4177778	65476811	3568788639	2872069391	65020636	65409494	65476801	65701121	65400279
accuracy %	99.86936352	64.10687691	98.57542848	99.86936352	99.9803487	97.98849553	98.26756342	98.57546034	98.59162829	98.42445473
coverage improvement	1	67.83590797	79.99279301	0	87.89399289	71.94650954	75.6940375	79.99324345	80.15254797	77.89832566
MemBWavgRd	7.97	25.2	6.05	7.97	28.3	6.15	5.9	6.05	57.1	5.96
MemBWavgWr	8.96	17	6.58	8.96	18	7.18	6.41	6.58	53.55	6.5
MCPI	12.19341077	10.95227499	9.505045122	12.19341077	11.57284288	10.01183789	9.70271366	9.504839994	9.631857645	9.67426565

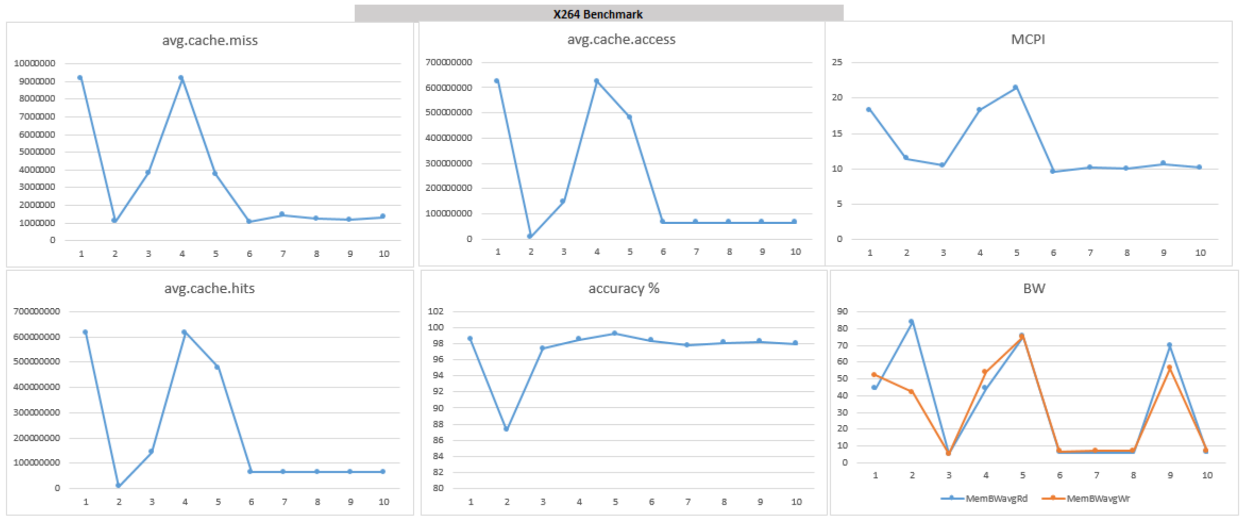


Figure 8: X264 benchmark results

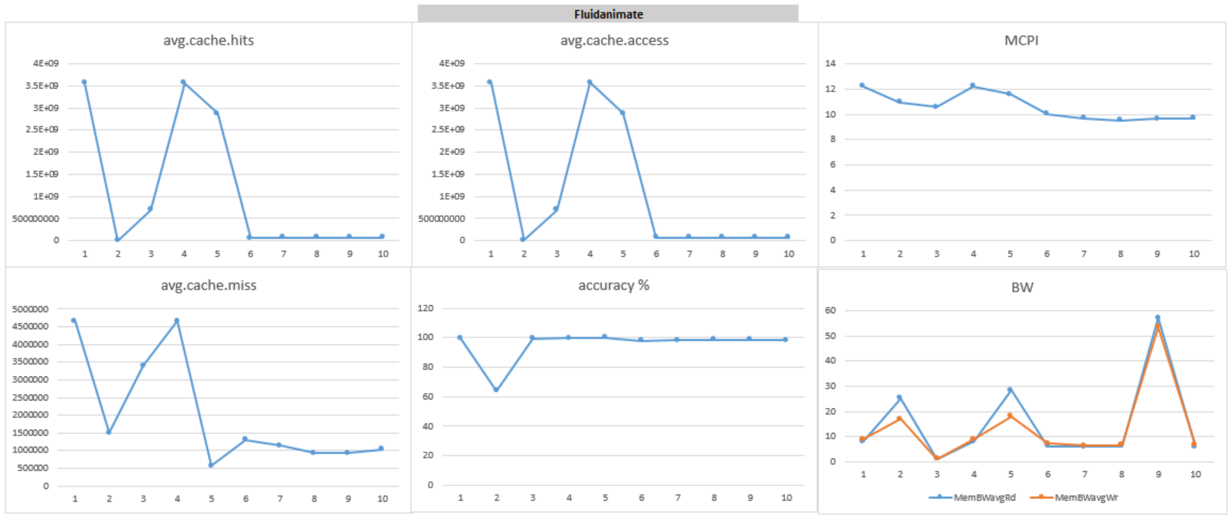


Figure 9: Fluidanimate benchmark results

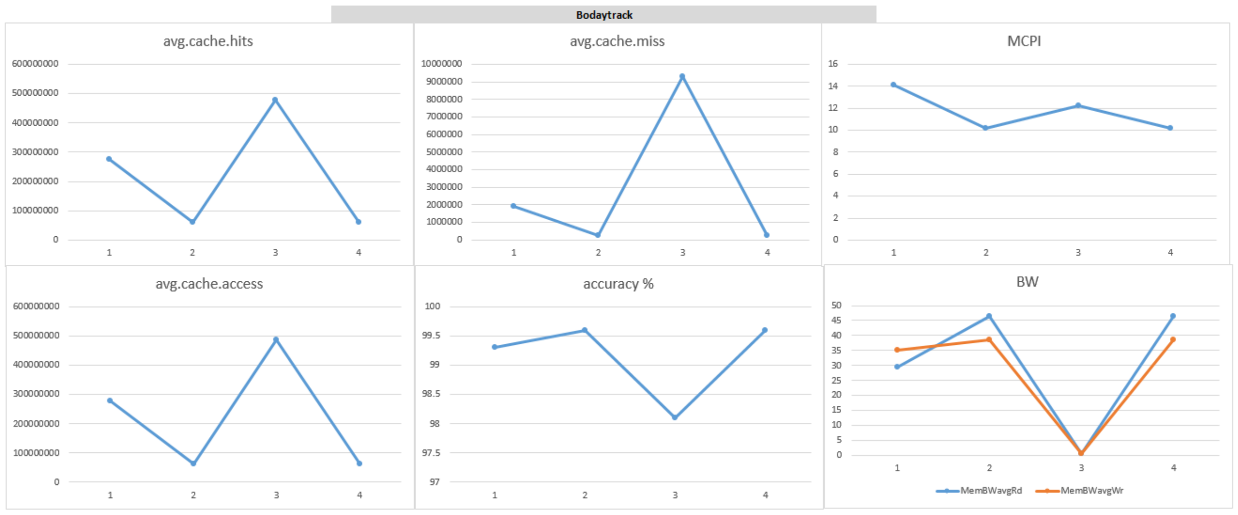


Figure 10: Bodytrack beanchmark results

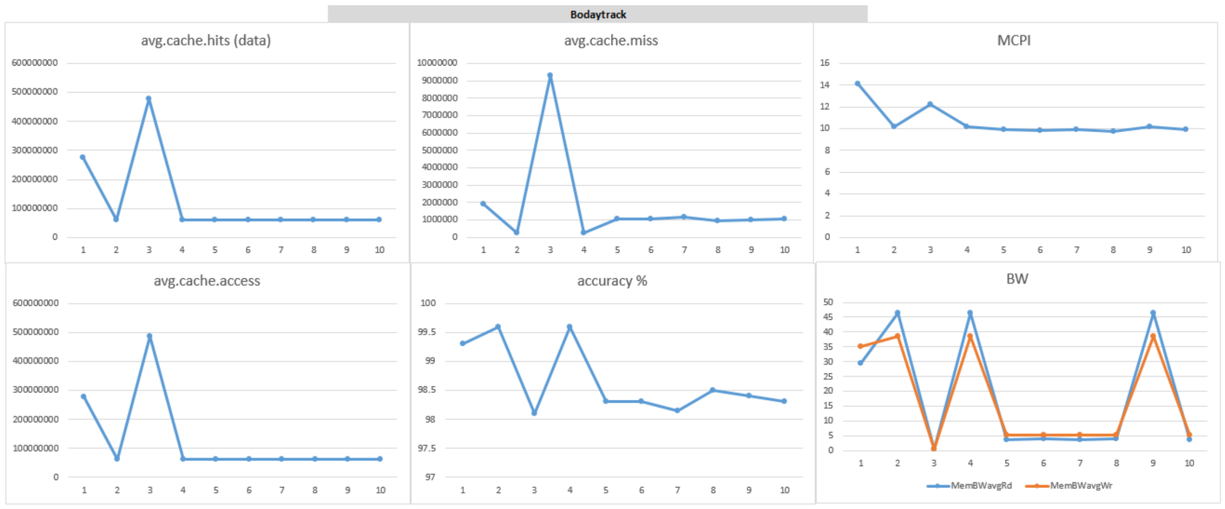


Figure 11: blackscholes benchmark results

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:
    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.
- 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

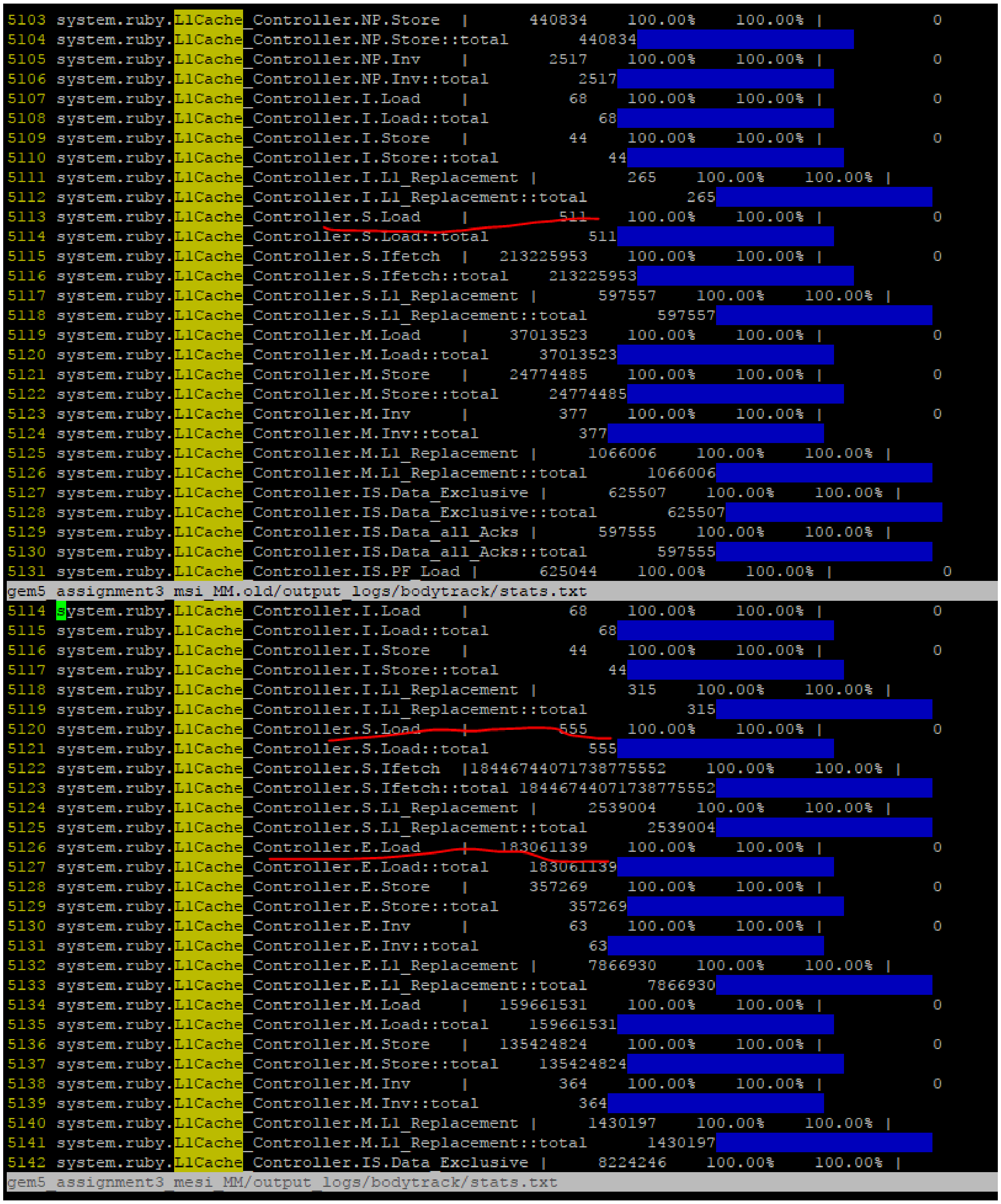
Ending thoughts:

- 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:

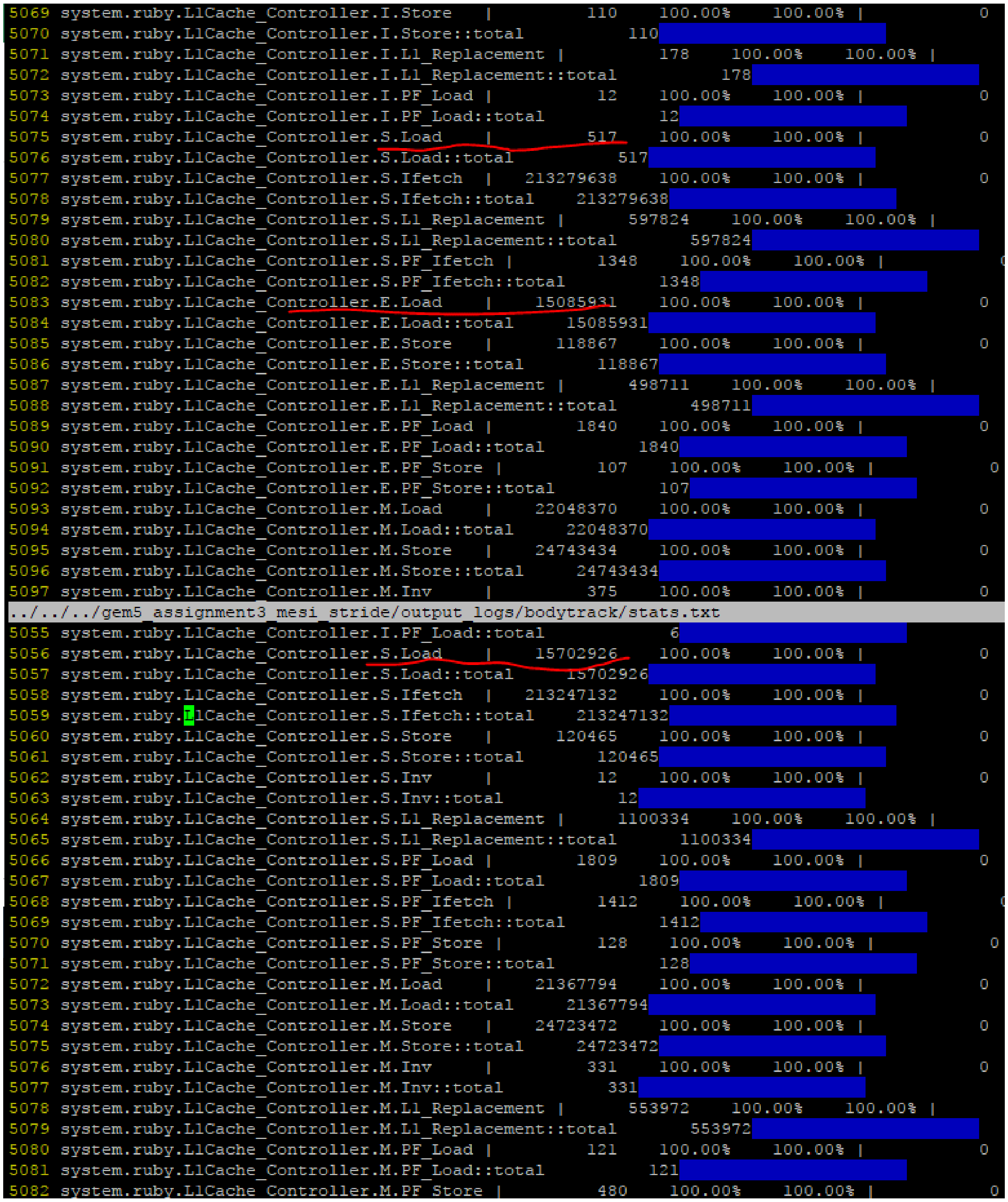
1. Gem5.org
2. www.learning.gem5.org/book/index.html
3. http://courses.engr.illinois.edu/ece511/secure/homework/assignment3.pdf
4. D. Joseph and Dirk Grunwald, “Prefetching using Markov Predictor”
5. learning.gem5.org/book/part3/MSI/cache-transitions.html
6. learning.gem5.org/book/part3/MSI/debugging.html

Appendix A:



```
system.ruby.L1Cache_Controller.NP.Store  |      440834     100.00%     100.00% |           0
system.ruby.L1Cache_Controller.NP.Store::total      440834
system.ruby.L1Cache_Controller.NP.Inv    |        2517     100.00%     100.00% |           0
system.ruby.L1Cache_Controller.NP.Inv::total        2517
system.ruby.L1Cache_Controller.I.Load    |          68     100.00%     100.00% |           0
system.ruby.L1Cache_Controller.I.Load::total          68
system.ruby.L1Cache_Controller.I.Store   |          44     100.00%     100.00% |           0
system.ruby.L1Cache_Controller.I.Store::total          44
system.ruby.L1Cache_Controller.I.L1_Replacement |         265     100.00%     100.00% |
system.ruby.L1Cache_Controller.I.L1_Replacement::total         265
system.ruby.L1Cache_Controller.S.Load    |         511     100.00%     100.00% |           0
system.ruby.L1Cache_Controller.S.Load::total         511
system.ruby.L1Cache_Controller.S.Ifetch  |   213225953     100.00%     100.00% |           0
system.ruby.L1Cache_Controller.S.Ifetch::total   213225953
system.ruby.L1Cache_Controller.S.L1_Replacement |      597557     100.00%     100.00% |
system.ruby.L1Cache_Controller.S.L1_Replacement::total      597557
system.ruby.L1Cache_Controller.M.Load    |    37013523     100.00%     100.00% |           0
system.ruby.L1Cache_Controller.M.Load::total    37013523
system.ruby.L1Cache_Controller.M.Store   |    24774485     100.00%     100.00% |           0
system.ruby.L1Cache_Controller.M.Store::total    24774485
system.ruby.L1Cache_Controller.M.Inv     |         377     100.00%     100.00% |           0
system.ruby.L1Cache_Controller.M.Inv::total         377
system.ruby.L1Cache_Controller.M.L1_Replacement |     1066006     100.00%     100.00% |
system.ruby.L1Cache_Controller.M.L1_Replacement::total     1066006
system.ruby.L1Cache_Controller.IS.Data_Exclusive |      625507     100.00%     100.00% |
system.ruby.L1Cache_Controller.IS.Data_Exclusive::total      625507
system.ruby.L1Cache_Controller.IS.Data_all_Acks |      597555     100.00%     100.00% |
system.ruby.L1Cache_Controller.IS.Data_all_Acks::total      597555
system.ruby.L1Cache_Controller.IS.PF_Load |      625044     100.00%     100.00% |           0
gem5_assignment3_msi_MM.old/output_logs/bodytrack/stats.txt
system.ruby.L1Cache_Controller.I.Load    |          68     100.00%     100.00% |           0
system.ruby.L1Cache_Controller.I.Load::total          68
system.ruby.L1Cache_Controller.I.Store   |          44     100.00%     100.00% |           0
system.ruby.L1Cache_Controller.I.Store::total          44
system.ruby.L1Cache_Controller.I.L1_Replacement |         315     100.00%     100.00% |
system.ruby.L1Cache_Controller.I.L1_Replacement::total         315
system.ruby.L1Cache_Controller.S.Load    |         555     100.00%     100.00% |           0
system.ruby.L1Cache_Controller.S.Load::total         555
system.ruby.L1Cache_Controller.S.Ifetch  |18446744071738775552     100.00%     100.00% |
system.ruby.L1Cache_Controller.S.Ifetch::total 18446744071738775552
system.ruby.L1Cache_Controller.S.L1_Replacement |     2539004     100.00%     100.00% |
system.ruby.L1Cache_Controller.S.L1_Replacement::total     2539004
system.ruby.L1Cache_Controller.E.Load    |   183061139     100.00%     100.00% |           0
system.ruby.L1Cache_Controller.E.Load::total   183061139
system.ruby.L1Cache_Controller.E.Store   |      357269     100.00%     100.00% |           0
system.ruby.L1Cache_Controller.E.Store::total      357269
system.ruby.L1Cache_Controller.E.Inv     |          63     100.00%     100.00% |           0
system.ruby.L1Cache_Controller.E.Inv::total          63
system.ruby.L1Cache_Controller.E.L1_Replacement |     7866930     100.00%     100.00% |
system.ruby.L1Cache_Controller.E.L1_Replacement::total     7866930
system.ruby.L1Cache_Controller.M.Load    |   159661531     100.00%     100.00% |           0
system.ruby.L1Cache_Controller.M.Load::total   159661531
system.ruby.L1Cache_Controller.M.Store   |   135424824     100.00%     100.00% |           0
system.ruby.L1Cache_Controller.M.Store::total   135424824
system.ruby.L1Cache_Controller.M.Inv     |         364     100.00%     100.00% |           0
system.ruby.L1Cache_Controller.M.Inv::total         364
system.ruby.L1Cache_Controller.M.L1_Replacement |     1430197     100.00%     100.00% |
system.ruby.L1Cache_Controller.M.L1_Replacement::total     1430197
system.ruby.L1Cache_Controller.IS.Data_Exclusive |     8224246     100.00%     100.00% |
gem5_assignment3_mesi_MM/output_logs/bodytrack/stats.txt
```

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



```
system.ruby.L1Cache_Controller.I.Store      |         110     100.00%     100.00% |           0
system.ruby.L1Cache_Controller.I.Store::total            110
system.ruby.L1Cache_Controller.I.L1_Replacement |         178     100.00%     100.00% |
system.ruby.L1Cache_Controller.I.L1_Replacement::total            178
system.ruby.L1Cache_Controller.I.PF_Load |          12     100.00%     100.00% |           0
system.ruby.L1Cache_Controller.I.PF_Load::total             12
system.ruby.L1Cache_Controller.S.Load       |         517     100.00%     100.00% |           0
system.ruby.L1Cache_Controller.S.Load::total            517
system.ruby.L1Cache_Controller.S.Ifetch     |   213279638     100.00%     100.00% |           0
system.ruby.L1Cache_Controller.S.Ifetch::total      213279638
system.ruby.L1Cache_Controller.S.L1_Replacement |      597824     100.00%     100.00% |
system.ruby.L1Cache_Controller.S.L1_Replacement::total         597824
system.ruby.L1Cache_Controller.S.PF_Ifetch |        1348     100.00%     100.00% |
system.ruby.L1Cache_Controller.S.PF_Ifetch::total           1348
system.ruby.L1Cache_Controller.E.Load       |    15085931     100.00%     100.00% |           0
system.ruby.L1Cache_Controller.E.Load::total       15085931
system.ruby.L1Cache_Controller.E.Store      |      118867     100.00%     100.00% |           0
system.ruby.L1Cache_Controller.E.Store::total         118867
system.ruby.L1Cache_Controller.E.L1_Replacement |      498711     100.00%     100.00% |
system.ruby.L1Cache_Controller.E.L1_Replacement::total         498711
system.ruby.L1Cache_Controller.E.PF_Load |        1840     100.00%     100.00% |           0
system.ruby.L1Cache_Controller.E.PF_Load::total           1840
system.ruby.L1Cache_Controller.E.PF_Store |         107     100.00%     100.00% |           0
system.ruby.L1Cache_Controller.E.PF_Store::total            107
system.ruby.L1Cache_Controller.M.Load       |    22048370     100.00%     100.00% |           0
system.ruby.L1Cache_Controller.M.Load::total       22048370
system.ruby.L1Cache_Controller.M.Store      |    24743434     100.00%     100.00% |           0
system.ruby.L1Cache_Controller.M.Store::total       24743434
system.ruby.L1Cache_Controller.M.Inv        |         375     100.00%     100.00% |           0
../../../gem5_assignment3_mesi_stride/output_logs/bodytrack/stats.txt
system.ruby.L1Cache_Controller.I.PF_Load::total              6
system.ruby.L1Cache_Controller.S.Load       |    15702926     100.00%     100.00% |           0
system.ruby.L1Cache_Controller.S.Load::total       15702926
system.ruby.L1Cache_Controller.S.Ifetch     |   213247132     100.00%     100.00% |           0
system.ruby.L1Cache_Controller.S.Ifetch::total      213247132
system.ruby.L1Cache_Controller.S.Store      |      120465     100.00%     100.00% |           0
system.ruby.L1Cache_Controller.S.Store::total         120465
system.ruby.L1Cache_Controller.S.Inv        |          12     100.00%     100.00% |           0
system.ruby.L1Cache_Controller.S.Inv::total             12
system.ruby.L1Cache_Controller.S.L1_Replacement |     1100334     100.00%     100.00% |
system.ruby.L1Cache_Controller.S.L1_Replacement::total        1100334
system.ruby.L1Cache_Controller.S.PF_Load |        1809     100.00%     100.00% |           0
system.ruby.L1Cache_Controller.S.PF_Load::total           1809
system.ruby.L1Cache_Controller.S.PF_Ifetch |        1412     100.00%     100.00% |
system.ruby.L1Cache_Controller.S.PF_Ifetch::total           1412
system.ruby.L1Cache_Controller.S.PF_Store |         128     100.00%     100.00% |           0
system.ruby.L1Cache_Controller.S.PF_Store::total            128
system.ruby.L1Cache_Controller.M.Load       |    21367794     100.00%     100.00% |           0
system.ruby.L1Cache_Controller.M.Load::total       21367794
system.ruby.L1Cache_Controller.M.Store      |    24723472     100.00%     100.00% |           0
system.ruby.L1Cache_Controller.M.Store::total       24723472
system.ruby.L1Cache_Controller.M.Inv        |         331     100.00%     100.00% |           0
system.ruby.L1Cache_Controller.M.Inv::total            331
system.ruby.L1Cache_Controller.M.L1_Replacement |      553972     100.00%     100.00% |
system.ruby.L1Cache_Controller.M.L1_Replacement::total         553972
system.ruby.L1Cache_Controller.M.PF_Load |         121     100.00%     100.00% |           0
system.ruby.L1Cache_Controller.M.PF_Load::total            121
system.ruby.L1Cache_Controller.M.PF_Store |         480     100.00%     100.00% |           0
```

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