Cache Coherence Protocols Simulation and Performance Comparison

Durganila Anandhan, Ian Taylor, Jessica Armstrong (Celebi), Manjari Rajasekharan
Department of Electrical and Computer Engineering, Portland State University

***Abstract*–Through programming a cache snooping performance simulator we were able to compare the performance of a few common snooping protocols, namely: MI, MSI, and MESI. This required the programming a reasonable representation of cores, caches, FSM snooping protocols, as well as a performance abstraction. For the performance abstraction we added a timing taxation system for any core stalling due to FSM state changes, especially for waiting on data to be written from the memory into the requesting cache. Our study of the MI, MSI, and MESI protocols in terms of core performance show that the MI can be up to 2x slower than the MSI or MESI snooping protocols. By using MSI or MESI we can see up to a 2x speedup depending on the data usage of a particular input code.**

INTRODUCTION:

Many compute-systems are hitting the memory wall today. We require caches with improved performance to combat this problem. But with caches comes the cache coherence problem. So, we require utilization of good cache coherency protocol to ensure correctness and improve system performance. Our study compares three common cache coherency protocols to see which is the best in terms of performance. Along the way we will be able to compare the protocols and gain a deeper understanding of each of them. As well, we can understand which types of code may benefit more from MESI/MSI/MI protocols.

CODING IMPLEMENTATION:

In order to define the scope of our project and coding implementation we defined the following specifications.

- Number of cores: 4
- Number of caches: 1 cache per core
- Size of caches: unlimited size
- Direct-mapped caches with 64B cachelines

As well, we made coding decisions to reduce our scope. For example, we assume that if there are requests for multiple cores at one timestamp, that the first one encountered in our input file is the winner of this contention and will be serviced first. It is essentially randomization of who wins during contention of cores over the same memory location at the same time. Another method of simplification is that we coded our simulator sequentially without parallel programming methods, such as the use of pthread. So, there is an assumption that we make in order to make this sequential coding more accurate for a parallel 4 core system.

- We assume that any core stalls for our cores happen simultaneously. Therefore, we just tax our code with the cost of the max stall time, not the summation of all core stall times.

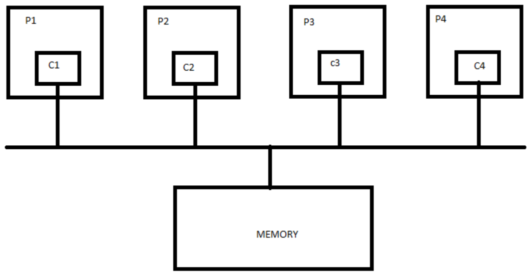


Figure 1. Cache and Core Abstraction Diagram

One complication of this sequential approach is that when we have contention of many core requests at the same timestamp, we are not reducing our code time for that but instead running our code back to back for each core request. Therefore, we are summing the time that would have been happening in parallel. However, if we had solved this by programming with pthread, then that speedup of our simulator performance would have taken place on any of our cache snooping protocols. Therefore, we suspect that our results between MI/MSI/MESI would not have changed drastically had we used pthread parallel methodology.

As another example of scope reduction, we don't send any actual data from our input file to our code, we only code the request information so that we don't need to worry about keeping track of memory data itself. We simply send in the following inputs:

- Core sending the memory request
- Memory location for request
- Timestamp of request
- Access type (read/write)

Example input:

1,	0x1000,	0,	R
2,	0x2004,	10,	W
0,	0x2000,	10,	R
3,	0x2004,	10,	R

We implement the core as a block that includes its own cache. The caches are implemented as separate doubly linked lists so that we can traverse through the cache data for debugging, report out, and for usage by our protocol functions. In a cache we save the following information:

- Memory location in this cache line
- Status of this memory location in this cache (M, E, S, I, etc)

Our code takes the following steps in order to simulate a cache snooping protocol:

1. Pull input data for one request into our code
2. In each cache, check the status of that memory location from previous cache usage
3. For each core, understand if the request is a processor or bus request
4. For each cache, run our cache snooping protocol (MI or MSI or MESI) and from the FSM calculate the new status of that memory location in each cache
5. Calculate the timing tax for each cache status change and delay our code for the maximum timing tax
6. Insert the new status of the memory location into each cache
7. Loop through steps 1-6 until all input requests have been processed

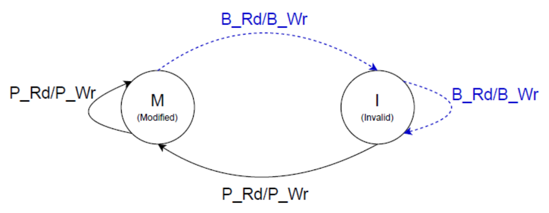


Figure 2. MI protocol FSM

For this study, we implemented the following common cache snooping protocols with the below FSM diagrams in Figure 2,3,4.

- MI
- MSI
- MESI

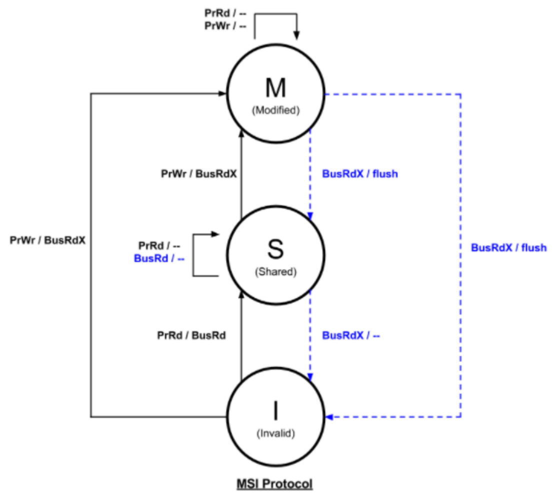


Figure 3. MSI protocol FSM

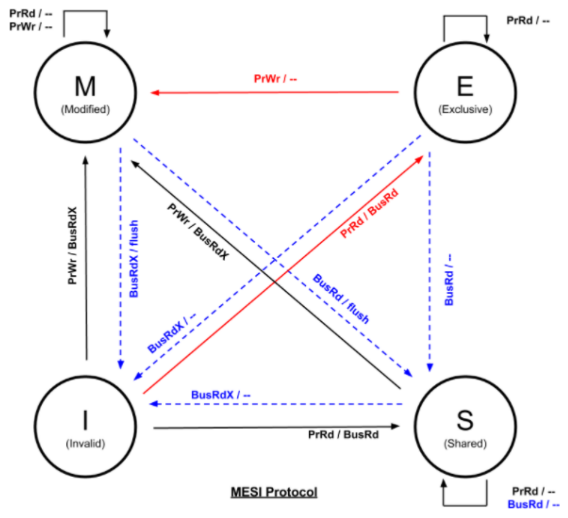


Figure 4. MESI protocol FSM

Along with simulating the different cache snooping protocols we needed to ensure that our simulator accounted for performance differences accurately. This was something that we didn't initially code but upon our testing and consultation with the instructor, we discovered this was an integral part of the simulator. Therefore, we implemented our performance impacts by modeling core stalling timing tasks. For example, any time a compute-system needs to go all the way to memory to get data for a cache we know this takes a very long time. In order to model this type of core stalling time waste we analyzed all paths of our cache snooping protocol FSM paths and based on our computer architectural understanding we taxed each path with appropriate delay times. Tables 1,2,3 show how we taxed the different FSM paths with some explanation.

Cache Protocol	Previous Status	Processor or Bus Signal	New Status	Timing tax	Explanation
MI	I	P_RD or P_WR	M	40ms	We need to go all the way to memory to grab this data
	I	B_RD or B_WR	I	1ms	Default
	M	P_RD or P_WR	M	1ms	Default
	M	B_RD or B_WR	I	1ms	Default

Table 1. MI Protocol

Cache Protocol	Previous Status	Processor or Bus Signal	New Status	Timing tax	Explanation
MESI	I	P_WR	M	40ms	We need to go all the way to memory to grab this data
	I	P_RD	S	2ms	Need to grab the data from another processor
	I	P_RD	E	40ms	We need to go all the way to memory to grab this data
	M	P_RD or P_WR	M	1ms	Default
	M	P_Writeback or B_WR	I	1ms	Default
	M	B_RD	S	3ms	Takes some time to send the data to the other core's cache
	S	P_RD	S	1ms	Default
	S	P_WR	M	1ms	Default
	S	B_WR	I	1ms	Default
	S	B_RD	S	1ms	Default
	E	P_RD	E	1ms	Default
	E	P_WR	M	1ms	Default
	E	B_WR	I	1ms	Default
	E	B_RD	S	1ms	Default

Table 2. MESI Protocol

We only tax the core doing the actual sending of data or requesting data all the way from the memory to its cache. We assume that the cost of getting data all the way from memory takes 20x longer than getting the data from a nearby cache.

Three other possible drawbacks of our implementation is that we don't model 1) the P_Writeback portion of the MESI protocol, 2) the performance core stall cost of waiting on data that is being written back to the memory, and 3) the performance cost of initialing filling the cache with data. For 1 and 2, this was done for simplification of our coding implementation and can be a future addition to the code for further studies. For 3, this is not necessarily a drawback of the code because no matter which protocol we run they would all have the same performance implications of cache data initialization.

Cache Protocol	Previous Status	Processor or Bus Signal	New Status	Timing tax	Explanation
MSI	I	P_WR	M	40ms	We need to go all the way to memory to grab this data
	I	P_RD	S	2ms	Need to grab the data from another processor
	M	P_RD or P_WR	M	1ms	Default
	M	B_WR	I	1ms	Default
	M	B_RD	S	3ms	Takes some time to send the data to the other core's cache
	S	P_RD	S	1ms	Default
	S	P_WR	M	1ms	Default
	S	B_RD	S	1ms	Default
	S	B_WR	I	1ms	Default

Table 3. MSI Protocol

METHOD FOR EVALUATION:

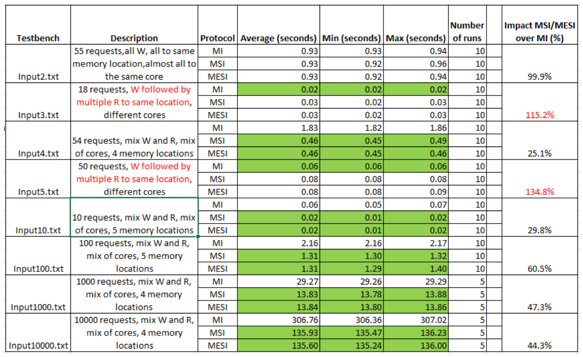
Initially, we coded our caches, cores, and our cache snooping protocol functions but had not yet programmed the timing taxation aspect of our simulator. In order to evaluate the performance aspect of these protocols we simply added a gettime() functionality to the code and measured from the start of the code to the end of the code. Next in the study we created inputs in order to test the run time of these different protocols. However, no matter the input length or the type of inputs, we always saw the same run time for each input running MI or MSI or MESI.

However, we expected to see a performance improvement when using MSI or MESI as compared to running MI. Therefore, we consulted with the instructor and realized that we should be coding the performance simulation aspect of the code. There are a few ways to implement this but our chosen method was to include timing taxation for certain actions, like retrieving data from memory and bringing it all the way back to the cache. As a team we developed the timing taxation system for all three cache snooping protocols as shown in the Code Implementation section of this report.

After we implemented this core stalling performance aspect to the code we finally saw run time differences between our protocols as expected. Then we developed a well rounded testbench which included varied length and activity inputs. We also created a python script which would generate random inputs to use as testbenches. We tested input lengths ranging from 3 requests to 1,000,000 requests.

Results are discussed in the Results section of this report.

RESULTS:



Testbench	Description	Protocol	Average (seconds)	Min (seconds)	Max (seconds)	Number of runs		Impact MSI/MESI over MI (%)
Input2.txt	55 requests,all W, all to same memory location,almost all to the same core	MI	0.93	0.93	0.94	10		
		MSI	0.93	0.92	0.96	10		
		MESI	0.93	0.92	0.94	10		99.9%
Input3.txt	18 requests, W followed by multiple R to same location, different cores	MI	0.02	0.02	0.02	10		
		MSI	0.03	0.02	0.03	10		
		MESI	0.03	0.02	0.03	10		115.2%
Input4.txt	54 requests, mix W and R, mix of cores, 4 memory locations	MI	1.83	1.82	1.86	10		
		MSI	0.46	0.45	0.49	10		
		MESI	0.46	0.45	0.46	10		25.1%
Input5.txt	50 requests, W followed by multiple R to same location, different cores	MI	0.06	0.06	0.06	10		
		MSI	0.08	0.08	0.08	10		
		MESI	0.08	0.08	0.09	10		134.8%
Input10.txt	10 requests, mix W and R, mix of cores, 5 memory locations	MI	0.06	0.05	0.07	10		
		MSI	0.02	0.01	0.02	10		
		MESI	0.02	0.01	0.02	10		29.8%
Input100.txt	100 requests, mix W and R, mix of cores, 5 memory locations	MI	2.16	2.16	2.17	10		
		MSI	1.31	1.30	1.32	10		
		MESI	1.31	1.29	1.40	10		60.5%
Input1000.txt	1000 requests, mix W and R, mix of cores, 4 memory locations	MI	29.27	29.26	29.29	5		
		MSI	13.83	13.78	13.88	5		
		MESI	13.84	13.80	13.86	5		47.3%
Input10000.txt	10000 requests, mix W and R, mix of cores, 4 memory locations	MI	306.76	306.36	307.02	5		
		MSI	135.93	135.47	136.23	5		
		MESI	135.60	135.24	136.00	5		44.3%

Table 5. Cache Performance Analysis

The above results table summarizes the simulation run time for multiple input testbench files for each of our three cache snooping protocols.The green rows highlight which protocol performed the fastest for a given input file. An impact percentage of MSI/MESI over MI of greater than 100% means that the MI performed the fastest. An impact percentage less than 100% indicates that MSI/MESI performed the fastest. One interesting finding from the simulation data is that as you increase the number of requests for a few memory locations (temporal locality) then the speedup trends to 2x speedup of MSI/MESI over MI. Another interesting finding is that with only processing write requests in input2 we see no speedup. This is because upon writing we have to invalidate the data in another cache and modify the requesting caches line. Therefore, we simply move between M and I no matter what protocol we implement. Therefore, MESI and MSI collapse to act just like MI.

Input3 and Input5 were special cases which highlight one limitation to our simulator. With those inputs we see that MI performs fastest. However, this is not expected and upon further analysis this is actually flawed results. This flawed result could be solved in further advancements of this study by properly accounting for the performance tax of writing modified cache data to the memory and then a different core waiting to read that data which has to come all the way from the cache in the MI protocol. With that proper implementation we expect to again see MESI/MSI outperform MI.

As well, we expect that if the above cache writeback followed by read performance impact is properly implemented into the simulator, we would see more of a difference between the MESI and MSI protocol. With the current simulator implementation however, we do not see much of a difference between MESI and MSI.

USER CODE EXPLORATION:

For readers who want to access, run, or evaluate this simulator, you can find the code and documentation at the following location: https://github.com/ArmJess/ECE588_FinalProject

Note that the run time measurements are done with Linux OS calls so please run the code on Linux to see accurate output run times. The code can be compiled by running the following line:

```
>>gcc main.c -o main -std=c99
```

Run format:

```
>>./main <input file> <protocol (MI, MSI, or MESI)>
```

Example run:

```
>>./main input4.txt MI
```

CONCLUSION:

As a result of this effort and study we have coded a cache snooping performance simulator for evaluation of MI, MSI, and MESI cache coherence protocols. Further additions to this study can include moving our code to a parallel programming implementation, coding a performance impact for R after W when we cannot share the data between caches (which would affect MI), and adding new cache coherency protocol capabilities to our simulator (such as MESIF or MOESI). We expect the results of these previous changes to clearly still show that MSI/MESI outperform MI protocols. As well, further studies can add directory based protocols instead of snooping protocols to make this performance simulator more robust.

ACKNOWLEDGEMENT:

We thank our Instructor for the ECE:588 class of Advanced Computer Architecture II, Prof. Yuchen Huang for her guidance and the resources she made us available. We also express our gratitude towards the Department of Electrical and Computer Engineering of Maseeh College of Engineering and Computer Science.

REFERENCES:

[1] Q. Yang, L. N. Bhuyan and B. . -C. Liu, "Analysis and comparison of cache coherence protocols for a packet-switched multiprocessor," in IEEE

Transactions on Computers, vol. 38, no. 8, pp. 1143-1153, Aug. 1989, doi: 10.1109/12.30868.

[2] F. Verbeek, P. M. Yaghini, A. Eghbal and N. Bagherzadeh, "A Compositional Approach for Verifying Protocols Running on On-Chip Networks," in *IEEE Transactions on Computers*, vol. 67, no. 7, pp. 905-919, 1 July 2018, doi: 10.1109/TC.2017.2786723.

[3] A. Tiwari, "Performance Comparison of Cache Coherency Protocol on Multi-Core Architecture," *M.S. thesis, NIT Rourkela, Rourkela, Odisha, India, 2014*.

[4] James Archibald and Jean-Loup Baer. 1986. Cache coherence protocols: evaluation using a multiprocessor simulation model. *ACM Trans. Comput. Syst. 4, 4 (Nov. 1986), 273–298.*