CoFee : Contention-Free Snooping Cache Coherence protocols for Multicore Systems

Gandhar Deshpande, Monil Shah

ABSTRACT

This report summarizes the work done to create a cache coherence simulator supporting 4 different bus snooping based protocols using a single level cache and main memory. The idea is to gain insights into bus contention, scalability of coherence protocol and advantage of one protocol over other. We talk about the various protocols available and supported in our simulator. We present our code development methodology and protocols implementation. We also speak of the possible limitations of our implementation and future works. We also propose our own cache coherence protocol that builds upon MOESI to reduce bus contention

1 Introduction

Memory is one of the fundamental components of any processor. Over time, as Dennard scaling and Moore's Law pushed the processors to be smaller and faster, the memory did not scale in the same manner leading to the famous "Memory Wall". To mitigate the performance degradation due to this gap, computer architects have used various methods. Largely these work around the cache memory, which allows fast access to memory for the processors. As scaling failed, architects have moved from single processor machines to multiprocessor machines which have shared memory.

As memory hierarchy becomes shared across multiple cores, we start getting a new problem of memory consistency and coherence. The data written by one processor and read by another processor should be the same, it should not be stale. To implement this condition, architects came up with various coherence protocols. Some of the most famously used coherence protocols are Modified-Shared-Invalid (MSI),Modified-Shared-Exclusive-Invalid (MESI), Modified-Shared-Owned-Invalid (MOSI) and Modified-Shared-Exclusive-Owned-Invalid (MOESI). To explore these concepts in depth, we implemented a cache coherence simulator which will run a given trace, collect the stats and show the results of the simulation. In this report, we talk about our implementation of a cache coherence simulator.

The report is arranged as follows:

- Section 2 talks about the background work related to the different coherence protocols and their working
- Section 3 expounds on the simulator implementation and development process.
- Section 4 discusses the evaluation methodology

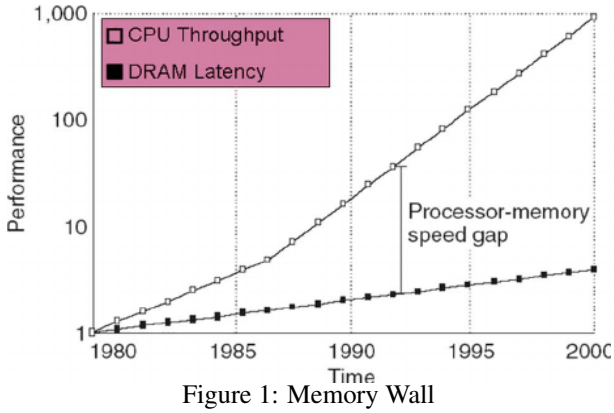

Figure 1: Memory Wall

- Section 5 provides a conclusion to the project.
- Section 6 provides our code for reference.
- Section 7 shows future works for improvements.

2 Background Work

In this section, we talk about the different coherence protocols. We start by talking about the concept of cache coherence, and then we look briefly at some of the examples of coherence protocols.

2.1 Cache Coherence

A coherent cache is a cache which has the information regarding its own memory and strives to use the latest available value instead of using stale values [1]. To explain the concept of coherence, lets take a look at what a incoherent system would look like, to understand why coherence is of paramount importance. Imagine 2 cores working on the same value in memory A. Since both the processors are using this data, both processors would have this value cached in their caches. As long as both of them are just reading the data and not modifying data, we do not see any problem. But since both are using the same values, if one of the processor modifies this value, the second processor's cache copy immediately becomes stale. In this scenario, the second processor needs to be informed that the value of this variable A has been modified and it should use this new value.

This example perfectly shows how important coherence is in a multicore system. If the variable is not updated in the

second core, the processor would produce incorrect results. To overcome this, there are multiple possible protocols. Some of the more famous ones are Valid-Invalid, MSI, MESI, MOSI and MOESI. These coherence protocols each have different features which make them desirable in different use-cases. We will talk about each of them in brief below.

2.2 Bus Snooping

Bus snooping is a mechanism where a cache controller monitors transactions on the shared bus and update its cache coherence metadata if required. Based on the bus activity it decides to send data to requester or invalidate its cache block. This is a very simplistic and cost effective cache coherence protocol and doesn't require sophisticated implementation. For less than 8 core system, bus snooping protocols work decently well but face bus contention problems when scaled to large number of cores. That gives rise to directory based protocols.

2.3 Valid-Invalid Protocol

The Valid-Invalid protocol is one of the very simplistic protocols used for coherence. As the name says, it has only 2 states - valid and invalid. At a time only one cache block can be in valid state. If one of the processors modifies/requests this data, it will send a signal to the other processor telling it to invalidate its cached copy of the data. With this the protocol requirement is completed. The second processor will now request the data from the the main memory again and use the new updated copy of the data. Since this is case, the first processor must write back the data to the main memory every time it writes to the cache.

2.4 MSI Protocol

MSI protocol has three different possible states - Modified, Shared and Invalid. This protocol was built to improve over the V-I protocol. The V-I protocol does not allow data to be shared between more than one processors. However, as the number of processors increases, this becomes unwieldy, since a significant chunk of the memory bandwidth is now consumed in just writing back to the memory. MSI allows for this modification. When a processor writes to the cache, it still must send an invalidation request to all the processors. However, the data can be kept in the cache in a modified cache until other core requires ownership. It needs to be only written back when another processor requests the same data in a shared state, at which point the data gets written back and the line is invalidated. This improves the performance by reducing the traffic writing back to the main memory. This protocol is good when data is mostly used in read only mode.

2.5 MESI Protocol

MESI protocol extends the MSI protocol by adding another state called Exclusive. This state optimizes a common case where a processor receives access to read a particular cache and wishes to modify it immediately after like Read-Modify-Write. In MSI protocol, the processor needs to essentially obtain a write permission by broadcasting on the bus that it wishes to move into the Modified stage. If the cache line is not available in any other processor, the processor gets the data in an Exclusive state when it requests the data, instead

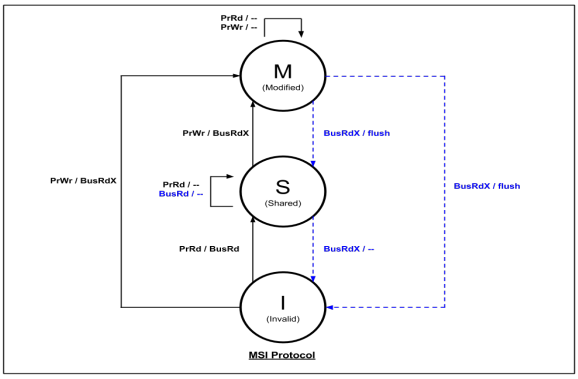

Figure 2: MSI working state machine

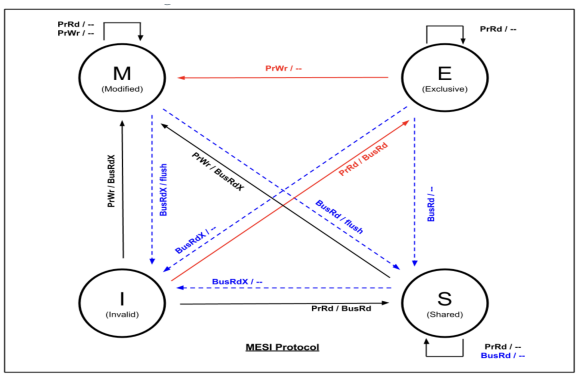

Figure 3: MESI working state machine

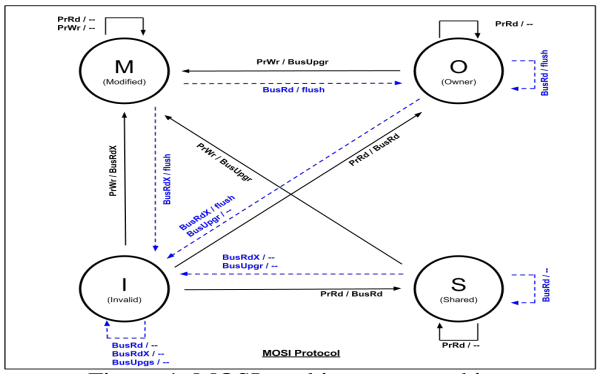

Figure 4: MOSI working state machine

of Shared state. This enables the processor to change the state of the cacheline from Exclusive to Modified silently. This reduces the traffic on the bus significantly, as this use-case is very common even in single-threaded applications. The second benefit is for the data to be returned by the core containing block in exclusive state to the core requesting data instead of being serviced by LLC.

2.6 MOSI Protocol

MOSI protocol extends the MSI protocol in a different way. It adds a different state called Owned. The motivation behind this state is that when a processor holds a particular cache line in a Modified or a Exclusive state, and another processor requests this line, the processor needs to change its state and send the data to the requestor and the main memory controller. The main memory controller is technically the owner of the data. To avoid this, the cacheline is provided to the processor with ownership. In such a case, the processor is responsible for providing the data to any requesting processors in a read mode and still retain the ownership of the block. If any other processor writes to their copy of the data, the current owner must relinquish its ownership. The rquesting core is now required to update main memory with correct data if it gets evicted.

2.7 MOESI Protocol

MOESI protocol is a mix of MESI and MOSI protocol, giving it the best of both worlds. By adding exclusive state, the cache block can move from E to M silently without generating any bus requests. Adding O state allows the cache to remain the owner of the block and broadcast to sharers, reducing multiple requests to the main memory controller.

2.8 CoFee Protocol

CoFee protocol is a protocol we propose as an augmentation of the existing protocol to optimize a common use-case. The main drawback with exclusive state is that while it allows E -> M silent upgrade, in case data is required to be shared across more than 3 cores it needs to come from LLC. The problem with owned state is it requires the data to be modified by a process before some other process tries to ask the read only copy of it. This is where our protocol helps. It extends the exclusive state by modifying it to C state. C state is the exclusive ownership state where the data can remain in the first requesting core's cache block, share it with other cores before writing into it and then move to modified state by invalidating other's. The motivation behind this new state is, it allows shared copy to be supplied by a private L1 cache instead of main memory or the LLC, thereby keeping LLC busy with other important demand requests. This will reduce the traffic of requests to the LLC and allow for faster sharing of data across caches. This is especially useful as we parallelize code over multiple cores that access shared libraries in their application.

3 Development Flow

For our code development, we started out with a starter code which had an associative cache implementation and a structure to configure the cache using command-line options. The cache implementation was a simple class that had basic

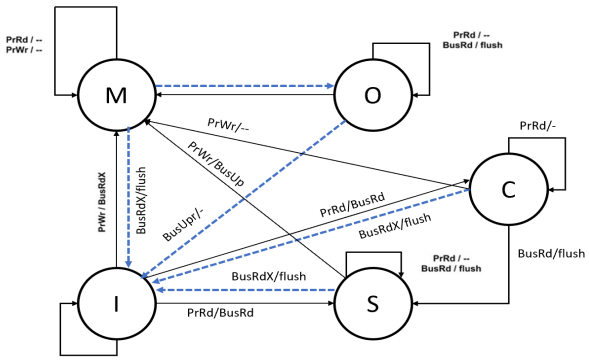

Figure 5: CoFee working state machine

functions to access the cache, and work on a hit or a miss. This code was not built with anything related to coherence.

To design this bus actions and responses, we first analyzed each protocol. We wanted our simulator to be compatible with most of the coherence protocols used widely in industry. Our simulator supports MSI, MESI, MOSI and MOESI protocols. We started our development process by first looking at what sort of communication is required between the cores. Once we found that, we looked at the various possible states that each processor could go in. We created a table of the various possibilities and made sure that our code covers each of the possible states in the processor.

Once we had conceptualized the various states in which the processor can go and eliminated the states which are not possible, we decided to implement it like a state machine. We used a Bus-based snooping protocol in our simulator, making it configurable to run with any supported coherence protocol. We will now take a deep-dive into the nitty-gritty details of our implementation.

```
//Cache req_cache .. cache_n
req_cache::Access();
BusAction = req_cache::Response();
if(cache != req_cache) {
    this::ReadBus();
    BusReaction = this::BusReaction();
}
req_cache::Change_State(BusReaction);
req_cache::updateStats();
```

Listing 1: Workflow of the code

```
req_cache::Access();
{
    if(Hit){
        setFlags;
        return hitBusAction;
    }
    else{
        fillLine;
        return missBusAction;
    }
}
```

Listing 2: Access Functions

```
cache[1..n]::BusReaction();
{
    case <State> : CacheAction();
                   BusResponse();

}
```

Listing 3: BusReaction Function

On every cache access, the initial code was just accessing the cache, checking if the line was present. If so, it would return the hit. If not, it would replace the oldest line based on LRU and add the cache line into the cache. The cache line simulation is done with just tags and no data. We modified this access function so that the function returns a bus action. In the main function of the code, we read the trace file and run the access based on the trace file. Once the processor has performed the access function, the processor decides on a bus action based on the protocol. If the access is a write to the cache, it will send an action on the bus which will say that the processor is modifying the cache line with the address. This bus action is read by all the other processors and based on this they will take some action and writeback a response on the bus. Based on this response, the requestor processor will then finish the transaction.

4 Evaluation

For evaluating our code, we used 2 traces present with the starter code to understand that our implementation matches with what is expected. This gave us a proof of concept of our implementation to try further experiments. We implemented the code and tested it by making it run for 4 cores. We recorded a variety of statistics from the code. We also created a couple of microbenchmarks as a sanity test. These benchmarks test the functionality of our code by checking if the processor is indeed accessing the data, receiving it from the core it is supposed to receive it from. It also checks the basics of whether a hit/miss is being recorded.

The statistics that are of interest to us to understand bus contention are : INV (invalidations), WB (writeBacks to Memory), C2C (transfer from other Cache), getM message, SU (Silent Upgrade). Benchmark 1 : Read one cache block in all four caches. Write cache block in each cache one after another. Benchmark 2 : Write in one cache block followed by read in next 3 cache blocks. Benchmark 3 : Read in one cache block, followed by reads by other cores to the same cache block.

The above benchmarks are very limited in terms of the number of operations it performs, but they are sufficient in number to identify the importance , advantages, disadvantages of one protocol over another.

4.1 4 core Analysis

As can be seen from the table 1 the main advantage is in terms of Cache to Cache transfers by adding specific states. From 2 it can be seen that invalidations decrease on R/W and from 3 silent upgrades can be seen. The number of invalidations changes across the protocols with MSI taking highest number of invalidations. The number of Writebacks decrease by addition of Exclusive / Owned States depending on the type of workload. There are also some silent upgrades which reduce the traffic at LLC as can be seen by less number of getM messages which are favourable for workloads that do read modify write by the reduction in getM messages seen.

Table 1: Benchmark 1 4-Core

MSI	MESI	MOSI	MOESI	COFEE	Type
6	6	6	6	6	INV
0	0	0	0	0	WB
3	4	3	4	6	C2C
4	3	4	3	3	GETM
0	1	0	1	1	SU

Table 2: Benchmark 2 4-core

MSI	MESI	MOSI	MOESI	COFEE	Type
17	13	9	13	9	INV
4	4	0	0	0	WB
4	13	13	13	13	C2C
4	4	4	4	4	GETM
0	0	0	0	0	SU

4.2 8 core Analysis

The behavior of protocol changes a lot for 8 core system. As can be seen from the table above, the number of invalidations increases by a big margin across the protocols. As can be seen from the table 4 the main advantage is in terms of Cache to Cache transfers by adding specific states. From 5 it can be seen that invalidations decrease on R/W and from 6 silent upgrades can be seen. The number of invalidations changes across the protocols with MSI taking highest number of invalidations. The number of Writebacks decrease by addition of Exclusive / Owned States depending on the type of workload. There are also some silent upgrades which reduce the traffic at LLC as can be seen by less number of getM messages which are favourable for workloads that do read modify write by the reduction in getM messages seen.

4.3 Cofee Protocol Analysis

As can be seen from the table 1, 2, 3, 4, 5 6 the main advantage is in terms of Cache to Cache transfers by adding specific states which proves our efficacy in reducing bus traffic from LLC and main memory. The other advantage that Cofee sees is in terms of silent upgrades which is also a property of MESI and MOESI.

Table 3: Benchmark 3 4-core

MSI	MESI	MOSI	MOESI	COFEE	Type
6	6	6	6	6	INV
0	0	0	0	0	WB
3	5	3	4	9	C2C
4	3	4	3	3	GETM
0	1	0	1	1	SU

Table 4: Benchmark 1 8-Core

MSI	MESI	MOSI	MOESI	COFEE	Type
14	14	14	14	14	INV
0	0	0	0	0	WB
7	8	7	8	14	C2C
8	8	8	8	7	GETM
0	0	0	0	1	SU

Table 5: Benchmark 2 8-Core

MSI	MESI	MOSI	MOESI	COFEE	Type
49	57	49	49	49	INV
8	8	0	0	0	WB
8	8	57	57	57	C2C
8	8	8	8	8	GETM
0	0	0	0	0	SU

5 Conclusion

Through this project we learned about the various intricacies involved with implementing a cache coherence. We looked at various details to understand how the cache interacts with other caches as well as how implementation varies with snooping and buses. We learned enough details that we started wondering about a condition and proposed a new flow of data to allow for a greater efficiency. Our simulator currently supports MSI, MESI, MOSI, MOESI and COFEE protocols with a single level of cache.

6 Future Works

Our simulator currently implements only single level cache without TLBs. The next extension to our work would be to implement directory based protocol and see how bus contention improves when the cores are scaled to higher numbers. Implementing multi level cache is another task since it involves invalidating L2 and higher caches based on physical address whereas L1 caches will be invalidated on Virtual address thereby requiring TLBs as well.

7 Code

Source code for our simulator is available at `https://github.com/gdpande97/cache-coherence-simulator.git`

8 Contributions

This project was done by both Monil and Gandhar in equal terms of contribution. Gandhar wrote the baseline code for development with the MSI protocol. Monil augmented the code with the MESI, MOSI and MOESI protocols. Both of us worked together to design and conceptualize the working of our simulator and understood the way the different protocols work. Both of us also worked through and debugged the idea of a new state in the coherence protocol- CoFee, which would allow sharing of read-only data through the cache. Monil worked to add the microbenchmarks and Gandhar worked for data collection. Both of us contributed equally to the report.

Table 6: Benchmark 3 8-Core

MSI	MESI	MOSI	MOESI	COFEE	Type
14	14	14	14	14	INV
0	0	0	0	0	WB
7	9	7	8	21	C2C
8	8	8	8	7	GETM
0	0	0	0	1	SU

9 Acknowledgement

We would like to thank Dr Jishen Zhao, Associate professor, University of California San Diego to give us this opportunity to implement a cache coherence simulator for CSE 240B final project for Spring 2022. We would also like to thank TA Vaibhav Tiwari for his continued support. We would also like to thank NCSU for making their cache starter code public.

10 References

[1] D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence*. Morgan amp; Claypool Publishers, 1st ed., 2011.