Performance Evaluation of Modified MOESI Cache Coherency Protocol for Multiprocessor System

Piyush Kasat, Mandar Raje

Department of Electrical Engineering, San Jose State University, San Jose, California 95192.

Introduction

Incorporating the cache coherency mechanisms through thread level parallelism is immensely important for the performance growth. To improve the efficiency of the processors, we use cache memories to reduce the latency of data access from main memory by using shared memory architecture. However, since multiple processors can modify or read same cache line, it becomes very challenging to keep data consistence and keep Main memory updated for the latest data in case some cache line is evicted[1].

Fast memories are known as caches which can be off chip or on chip. There is good chance of temporal or spatial locality in most programs i.e. first possibility is if processor reads or write a memory address once, they might access it again in near future. So rather than reading it every time from main memory, it is a good idea to store it locally. Or another possibility could be if processor reads or writes from or to a particular location, there is good chance that it accesses the nearby locations in near future. For this purpose caches are very useful as they hold a group of neighboring data known as Cache lines[2].

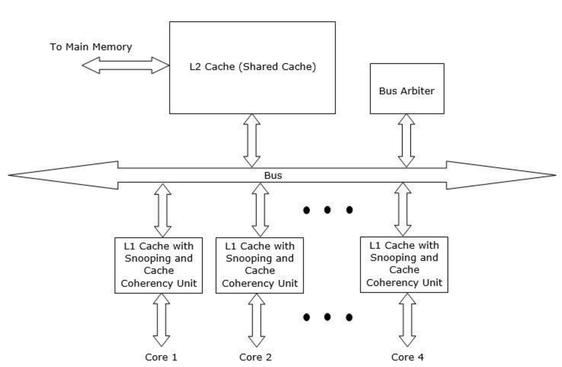
Now when the system has many processors, all these processors have separate cache memories, and thus the same cache line is shared by multiple processors at the same time. So there might be the problem of inconsistency of data for the same cache line owned by multiple processors. So, if particular processor is updating the shared cache line, the copy of the cache line in other processor's cache has to be invalidated to make sure that other processors do not read an outdated value of the modified cache line. This is nothing but the problem of cache coherency.

Methodology

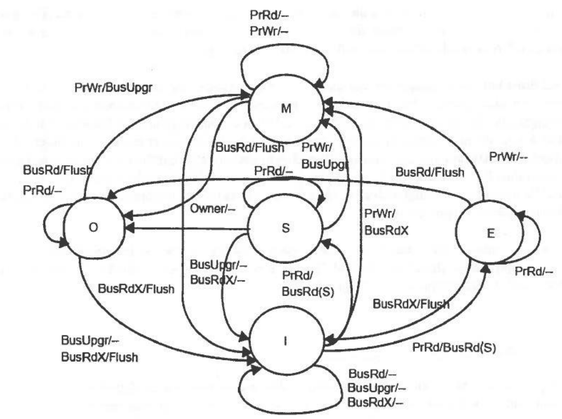
To address the issue of cache coherency there are several protocols proposed and are used in real-world computer architecture. Over the years, cache coherency protocols have evolved for the better, Intel's Pentium IV processors used MESI protocol whereas AMD used MOESI protocol .

Here we are demonstrating modified version of MOESI cache coherency protocol for a quad-core processor architecture. Each core has its own private L1 cache and all the cores have a shared L2 cache. Following Figure depicts the architecture of multi-processor system employed in this project.

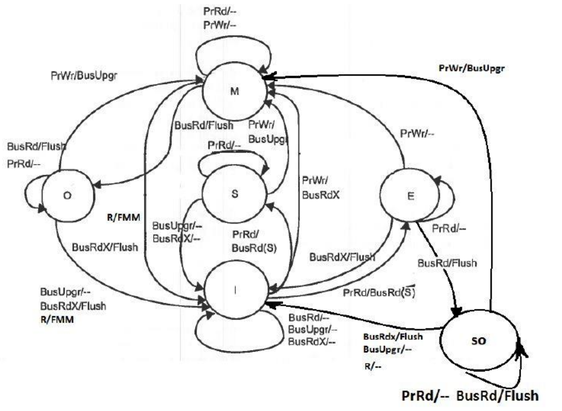
Methodology



MOESI protocol has a state machine representation, as shown below, each state signifies the cache line's status, which is derived from the instructions executed by the processors. The states in MOESI protocol are: M – Modified (Exclusive Dirty), O – Owner (Shared & Clean/Dirty), E – Exclusive (Exclusive Clean), S – Shared (Clean), I – Invalid.



For the MOESI protocol, Owner state has data ambiguity as there are two entry point, one from Modified state (Dirty Data) and other from Exclusive state (Clean Data). To clear this ambiguity, a new state is added to MOESI protocol called SO - Shared Owner (Shared Clean) state and the state machine for the new protocol is represented below.



In Modified MOESI we have tweaked the transition to Owner state. Thereby, now there is a transition from Exclusive state to the Shared Owner state and not to the Owner state. Thus, Shared Owner state is guaranteed to have clean data. The original transition in MOESI from Modified state to Owner state is retained as it is. Thus it is guaranteed that Owner state will have dirty data since it is obtained from Modified state. That means a particular processor will write and modify the data and when other processors' cache ask for that same location then the processor's cache which has the latest updated copy becomes the owner for that block and supplies it to requesting cache.

Thus there is a dedicated state for clean and dirty data. This is achieved by the introduction of Shared Owner state. Thus, in longer runs of instructions there would no need to keep a track of data whether its clean or dirty since they have dedicated states. The protocol state diagram becomes a bit complicated but it is OK to increase the complexity as long as it improves the performance.

All of this can be better handled if there is some instruction prediction mechanism in the system. It will depend on the application for which the system is used. It solely depends on the number of Loads and Stores in the instruction list.

The instruction format defined for this design is given below, a 32 bit address coming from the Processor is broken into 23 bit Tag, 5 bit Set and 2 bit Word & Byte. Cache designed for this project is 2-way set associative cache such that size of local cache for each processor is 1MB.

31 to 9	8 to 4	3 to 2	1 to 0
Tag (23 bits)	Set (5 bits)	Word (2 bits)	Byte (2 bits)

Results

The test stimulus for the design are address traces of a 64 bit Linux machine. These 64 bit address traces were translated in accordance to the instruction set used in this design. Total of 1,898,807 address traces were given to test this design. Out of 1898807 instructions, 999812 were Load Instructions whereas 898995 were Store Instruction.

These address traces were ran on design based on MOESI – SO, MOESI and MESI cache coherency protocols. All the three designs were run for a standard time of 1 Billion block cycles. Table below summarizes the performance result of each cache coherency protocol.

	MOESI-SO	MOESI	MESI
# Write Hit	75964	70686	64418
# Read Hit	555522	563314	571955

From the above table it is observed that MOESI –SO gives maximum write hits whereas MESI leads in reads. MOESI stayed the second best for both the cases.

Summary

From the obtained results it can be concluded that the performance of MOESI-SO cache coherency protocol is dependent on cache size. The MOESI-SO protocol model that was designed for this project had a single level 2 way set associative cache of size 1MB each. With just a single level of cache present in the design the write hit rate trumped MOESI and MESI protocol. The performance of this design shall improve if the cache size is increased. Current design was tested for 32 bit addresses, thus limiting the number of sets and tags. For a 64 bit address instruction, the performance of MOESI-SO protocol should better MOESI and MESI for both read and write hit rates.

The performance of the same design can be improved by adding L2 and L3 cache and moving over to 64 bit address instruction set. Cache designing can also be upgraded by adding more number of sets and ways thus resulting in increasing the overall size of the cache. Because in the end, if there's an improvement of at least 2% of hit rate overall, then by industry standards it will be a great accomplishment.

Key References

1. Effects of Cache Coherency in Multiprocessors, Michel Dubois , And Faye A. Briggs, IEEE Transactions on Computers, Vol. C-3 November 1982
2. Cache Coherence Techniques For Multicore Processors, Michael R. Marty, Doctor of Philosophy (Computer Sciences) University Of Wisconsin – Madison
3. Computer Architecture, A Quantitative Approach" by John L. Hennessy, Stanford University and David A. Patterson, University of California Berkeley.
4. GoeffLowney, "Why Intel is designing multicore processor", Paper Presentation. July 2006.
5. Impact of Cache Coherence Protocols on the Power Consumption of STT-RAM-Based LLC , Mu-Tien Chang , Shih-Lien Lu , and Bruce Jacob , University of Maryland - Research Funded by Intel Labs
6. Comparing Cache Architectures and Coherency Protocols on x86-64 Multicore SMP Systems, Daniel Hackenberg Daniel Molka Wolfgang E. Nagel, MICRO'09, Copyright 2009 ACM 978-1-60558-798-1/09/12
7. Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System - Daniel Molka, Daniel Hackenberg, Robert Schone and Matthias S. Muller , 2009 18th International Conference on Parallel Architectures and Compilation Techniques

Acknowledgements

I am grateful to all those who helped me through this project in terms of academic and moral support. First of all I would like to thank my project advisor, Prof. Morris Jones for his support, guidance and undivided attention towards making this project a success. Without him, it would have not been possible to complete this project in timely manner as required. I am grateful to him for creating an UNIX account for my project on his server so that I wouldn't face any host server performance issues during semester crunch time. I would also like to take this opportunity to thank San Jose State University, Electrical Engineering Department and the EECAD Lab.