Asian Journal of Information Technology 15 (24): 5181-5186, 2016
ISSN: 1682-3915

Evaluations of Cache Coherence Protocols in Terms of Power and Latency in Multiprocessors

[1]Babak Aghaei and [2]Negin Zaman-Zadeh
[1]Department of Computer Engineering, Malekan Branch islamic Azad University, Malekan, Iran
[2]Department of Computer Engineering, Tabriz Branch, Islamic Azad University, Tabriz, Iran

Abstract: The shared memory multiprocessors suffer with significant problem of accessing shared resources in a shared memory it will result in longer latencies. Consequently, the performance of the system will get affected. With the object of solving the problem of increased access latency due to large number of processors with shared memory, Cache is being used. Every processor has its own private cache, now they can update or access the data comfortably but again it leads to another serious issue i.e., cache coherency. The magnitude of the potential performance difference between the various cache coherency approaches indicates that the choice of coherence solution is very important in the design of an efficient shared-bus multiprocessor, since it may limit the number of processors in the system. In this paper we evaluate a typical multiprocessor system in terms of power and latency with different cache coherence protocols where GEM5 simulator is used. The traffic is generated with five injection rates (0.1, 0.2, 0.3, 0.4 and 0.5). Power and latency analyzing figures are made up and appeared in experimental result. The result shows MOESI_CMP_token has maximum latency and power consumption.

Key words: Multiprocessor, cache coherency protocols, power, latency, experimental

INTRODUCTION

With increasing core counts, the message-based on-chip network becomes an integral part of future Chip Multi Processor (CMP) systems. Future CMPs with dozens to hundreds of nodes will require a scalable and efficient on-chip communication fabric (Agarwal *et al*., 2009). Figure 1 shows how various components of a CMP system are coupled together.

In multiprocessors system with shared memory, workload can be divided among these processors therefore, they work faster than uniprocessor (Kumar and Arora, 2012). These systems allow the easier development of parallel software and also can increase the system throughput, reliability and they are economical too. The shared memory multiprocessors suffer with significant problem of accessing shared resources in a shared memory it will result in longer latencies. Consequently, the performance of the system will get affected. With the object of solving the problem of increased access latency due to large number of processors with shared memory, Cache is being used. Every processor has its own private cache, now they can update or access the data comfortably but again it leads to another serious issue i.e., cache coherence problem.

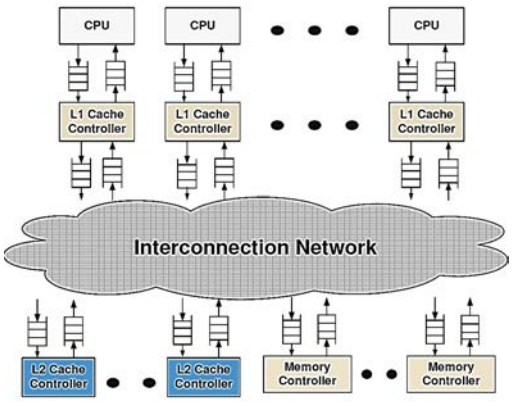

Fig. 1: Communication in the memory system

Cache coherence problem arises when multiple processes are trying to access the same data for updating purpose or one processor is trying to modify the data and rest processors are trying to read simultaneously. It may lead to inconsistent state of data at cache of different processors and the main memory (Lametti, 2010). In this paper we survey the cache coherence protocols and evaluate the cache coherence protocols in terms of power and delay characteristics.

Corresponding Author: Babak Aghaei, Department of Computer Engineering, Malekan Branch, Islamic Azad University, Malekan, Iran

Related work: In literature (Lametti, 2010; Stenstrom, 1990; Protic *et al*., 1998) there are excellent survey and classification of the shared memory and cache coherence protocols. Archibald and Baer (1986) a multiprocessor simulation model is presented and described including a mechanism for simulating explicitly the dynamic reference behavior of shared data while expressing locality of references. The results of this model indicate that the choice of coherence protocol in a shared-bus system is a significant design decision, since the hardware requirements vary and since the performance differences between the protocols can be quite large. Suh in her PhD thesis (Suh, 2006) implemented a coherence cache in an FPGA on the Intel server system and measured the intrinsic delay of coherence traffic and analyzed its efficiency. The authors in (Agarwal *et al*., 1988) claims that directory-based cache consistency protocols are an interesting approach for providing shared memory in a large-scale multiprocessor. They evaluate performance of directory-based protocols in a small-scale multiprocessor environment using trace driven simulation. Survey in (Kumar and Arora, 2012) tries to give a comprehensive overview of hardware and software-based solution to cache coherence problem in shared memory processor. The authors think both approaches perform well but their selection depends on the type of access pattern of shared data block and also number of processors their want to connect. They are used SMP cache simulator and are demonstrated that cache coherence significantly impacts the performance of the processor. The performance includes latency, bandwidth and protocol overhead.

In this study we survey the cache coherence protocols and evaluate the power and latency of protocols in multiprocessor environment on GEM5 simulator. The main contributions of our work are firstly, the using of high level and accurate (Butko *et al*., 2012) simulator (GEM5) that gives stochastic and near-to hardware result with less hardware-based considerations. Secondly, the evaluation and classification of cache coherence in CMP in terms of latency and power.

Cache coherency: In many systems, the cache coherence techniques are entirely implemented at the firmware level. Two main techniques, called automatic cache coherence techniques are used (Kumar and Arora, 2012):

- Invalidation in which it is assumed that the only valid copy of a block is one of those, usually the last one, that has changed, invalidating all other copies; in a write-through system the copy in M is also valid
- Update, in which each modification of a cache block copy is communicated (multicast) to all other caches

At first sight, the update technique appears simpler, while invalidation is potentially affected by a sort of inefficient "Ping-Pong" effect. However, in the real utilization of cache coherent systems: the update mechanism has a substantially higher overhead, also due to the fact that only a small fraction of nodes contain the same block; on the other hand, processor synchronization reduces the "Ping-Pong" effect substantially so invalidation is adopted by the majority of systems. In all the systems which use the automatic techniques, a proper protocol must exists in order to perform the required actions atomically.

In a generic cache coherence protocol each block in a cache has a state associated with it, along with the tag and data which indicates the disposition of the block. The cache policy is defined by the cache block state transition diagram which is a finite state machine specifying how the disposition of a block changes. while only blocks that are actually in cache lines have sate information, logically, all blocks that are not resident in the cache can be viewed as being in either a special "not present" state or in the "invalid" state.

In a uniprocessor system, for a write-through, write no-allocate cache, only two states are required: valid and invalid. Initially, all the block are invalid; when a processor read causes a fault, the block is transferred from the memory into the cache and it's marked valid. Writes do not change the state of the block, they only update the memory and the cache block if it is present in the valid state. If a block is replaced, it may be marked invalid until the memory provides the new block, whereupon it becomes valid.

A write-back cache requires an additional state per cache line, indicating a "dirty" or modified block. In a multiprocessor system, a block has a state in each cache, and these cache states change according to the state transition diagram. Thus, we can think of a block's cache state as being a set of n states, where n is the number of caches. The cache state is manipulated by a set of n distributed finite state machines, implemented by the units W of each node that act as cache coherence controllers. The state machine that governs the state changes is the same for all blocks and all caches but the current state of a block in different caches is different.

In general, an invalidation-based protocol consists of the following states (Suh, 2006): Modified, Exclusive, Shared, Invalid and Owned. The I state indicates that a data block in the cache is invalid. Multiple processors can share the same data block in their respective caches in the S state but only one processor can have the block exclusively in its own cache in the E state. The E state also indicates that the cached block has not been

modified, since it was brought in from the main memory. A processor can have a modified data block in the M or the O state. The M state indicates that the processor owns the modified data block exclusively in its cache, while the O state specifies that the modified block may be shared with other processors. In continue, we will discuss about all invalidation-based protocols that implemented in GEM5 simulator.

The GEM5 Simulator: The GEM5 simulator (Binkert *et al*., 2011) is a collaborative project based on the M5 simulator (Binkert *et al*., 2006) and the Ruby component of GEMS. The M5 simulator is a full-system simulator designed to model networked systems and Ruby is the memory system component of GEMS. GEM5 provides detailed models for both in-order and Out-of-Order (OoO) CPUs, as well as simple models for fast functional simulation. Multi-threading is supported via both CMP-style systems and SMT-enabled CPUs. It provides two modes: system call emulation mode and full-system mode. In system call emulation mode binaries run directly on the simulator and all operating system functionality is emulated. The memory system allows users to define a variety of cache organizations, e.g., directory-based or bus-based, and coherence protocols (it incorporates SLICC (Atta *et al*., 2012). We choose the GEM5 simulator in this study for a variety of reasons:

- It supports the two most popular ISAs in use today: ARM and x86, among thers
- It is widely-used in academia and industry, and has a BSD-style license; as such it is important to inform the community of its accuracy
- It is able to boot unmodified versions of several relevant operating systems, e.g., Google's Android OS and Ubuntu Linux, and it can run interactive workloads
- It has incorporated most, if not all, of the advanced features of the other full-system simulators: system networking, KVM (Kivity *et al*., 2007) integration, checkpointing support, simpoint (Sherwood, 2002) generation, detailed DRAM models and a configurable cache system

GEM5 simulator provides a flexible, modular simulation system that makes it possible exploring multiprocessor architecture features by offering a diverse set of CPU models, system execution modes, and memory system models. GEM5 is an event-driven simulation framework that has different abstraction levels, balancing simulation speed and accuracy. Furthermore, GEM5 has an open source license, a good object-oriented infrastructure and a very active mailing list. The following cache coherence protocols are supported by GEM5 simulator:

MI example: Example protocol, 1-level cache

MESI_two_level: Single chip, 2-level caches, strictly inclusive hierarchy

MOESI_CMP_directory: Multiple chips, 2-level caches, non-inclusive (neither strictly inclusive nor exclusive) hierarchy
MOESI_CMP_token: 2-level caches

MOESI_hammer: single chip, 2-level private caches, strictly-exclusive hierarchy.

Network test: dummy protocol to operate the network tester, 1-level cache. In addition an overview of each architecture and the current supported features can be found on the official website.

MI example: This is a simple cache coherence protocol that is used to illustrate protocol specification using SLICC. SLICC stands for Specification Language for Implementing Cache Coherence. It is a domain specific language that is used for specifying cache coherence protocols. In essence, a cache coherence protocol behaves like a state machine. SLICC is used for specifying the behavior of the state machine. Since the aim is to model the hardware as close as possible, SLICC imposes constraints on the state machines that can be specified.

This protocol assumes a 1-level cache hierarchy. The cache is private to each node. The caches are kept coherent by a directory controller. Since the hierarchy is only 1-level, there is no inclusion/exclusion requirement. This protocol does not differentiate between loads and stores. This protocol cannot implement the semantics of LL/SC instructions because external GETS requests that hit a block within a LL/SC sequence steal exclusive permissions, thus causing the SC instruction to fail.

Mesi_two_level: This protocol models two-level cache hierarchy. The L1 cache is private to a core while the L2 cache is shared among the cores. L1 Cache is split into Instruction and Data cache. Inclusion is maintained between the L1 and L2 cache. The on-chip cache coherence is maintained through Directory Coherence scheme, where the directory information is co-located with the corresponding cache blocks in the shared L2 cache. The protocol has four types of controllers; L1 cache

controller, L2 cache controller, Directory controller and DMA controller. L1 cache controller is responsible for managing L1 Instruction and L1 Data Cache. Number of instantiation of L1 cache controller is equal to the number of cores in the simulated system. L2 cache controller is responsible for managing the shared L2 cache and for maintaining coherence of on-chip data through directory coherence scheme. The Directory controller act as interface to the Memory Controller/Off-chip main memory and also responsible for coherence across multiple chips/and external coherence request from DMA controller. DMA controller is responsible for satisfying coherent DMA requests.

MOESI_CMP_directory: Cache hierarchy In contrast with the MESI protocol, the MOESI protocol introduces an additional "Owned" state. The MOESI protocol also includes many coalescing optimizations not available in the MESI protocol.

MOESI_CMP_token: This protocol also models a 2-level cache hierarchy. It maintains coherence permission by explicitly exchanging and counting tokens. A fix number of token are assigned to each cache block in the beginning, the number of token remains unchanged. To write a block, the processor must have all the token for that block. For reading at least one token is required. The protocol also has a persistent message support to avoid starvation.

MOESI_hammer: This is an implementation of AMD's Hammer protocol which is used in AMD's Hammer chip (also known as the Opteron or Athlon 64) (Owen, 2006). The protocol implements both the original a Hyper Transport protocol, as well as the more recent Probe Filter protocol.

The protocol also includes a full-bit directory mode. This protocol implements a 2-level private cache hierarchy. It assigns separate Instruction and Data L1 caches, and a unified L2 cache to each core. These caches are private to each core and are controlled with one shared cache controller. This protocol enforce exclusion between L1 and L2 caches.

Network test: This is a dummy cache coherence protocol that is used to operate the ruby network tester. This protocol assumes a 1-level cache hierarchy. The role of the cache is to simply send messages from the cup to the appropriate directory (based on the address), in the appropriate virtual network (based on the message type). It does not track any state. In fact, no Cache Memory is created unlike other protocols. The directory receives the messages from the caches but does not send any back. The goal of this protocol is to enable simulation/testing of just the interconnection network.

RESULTS AND DISCUSSION

Experimental result: We simulate 16 core CMP architecture with Timing Simple ALPHA cores in GEM5 simulator (Binkert *et al*., 2011). Firstly, we invoke a 4*4 GARNET network (Agarwal *et al*., 2009) on GEM5 with 2D Mesh topology (for more detail see Table 1). The type of all cores are ALPHA 21264. The main purpose is to evaluate the power and latency of network with different cache coherent protocols. So, we configure the GEM5 and choose a cache coherency protocol and for every cache coherence protocol, we set five injection rates (0.1, 0.2, 0.3, 0.4 and 0.5). Since, we can evaluate the cache coherence protocol with different injection rate. For evaluation of total power of network, we sum static and dynamic power of routers. For evaluation of total network latency, we collect all routers delay. Average latacy of network with different cache coherence protocols is depicted in Fig.2. This comparative figure demonstare MOESI_CMP_token protocol has maximum average network lateny. Wherease, MESI_Two_Level and MOESI_CMP_directory protocols have minimum latency.

Total consumed power in routers for every cache coherence protocol with different injection rates is illustrated in Fig. 3. It demonstrate MOESI_CMP_token protocol impose more power consumption than others to CMP architecture. Instead, MOESI_CMP_directory consume power less than others.

Table 1: GEM5 configuration parameters

Component	Configuration
Processor	ALPHA 21264 cores, 500-600 MHz, 15MTr/2.2-V, 0.35 CMOS/six metal layers. up to 2.4 billion instructions per second
L1 Caches	Split I and D, 32 KB 4-way set associative, 2 cycle access time, 64-byt line
L2 Caches	2M private, 15 cycle latency, 64 byte line
Memory	Ruby, 512MB, 16-bytet channel width
Network	Garnet, 4*4 2D Mesh, 10 virtual network, 1cycle on-chip link latency
Router	4 cycle router pipeline, XY routing, 4 VC per virtual network, Round-Robin queuing.
Packet and simulation cycle	Max packet = 1, injection rate: 0.1, 0.2, 0.3, 0.4 and 0.5, fixed packet size, uniform traffic, simulation cycle = 10000
Operating system	Linux Ubuntu LTS 14.4, kernel 3.16.7

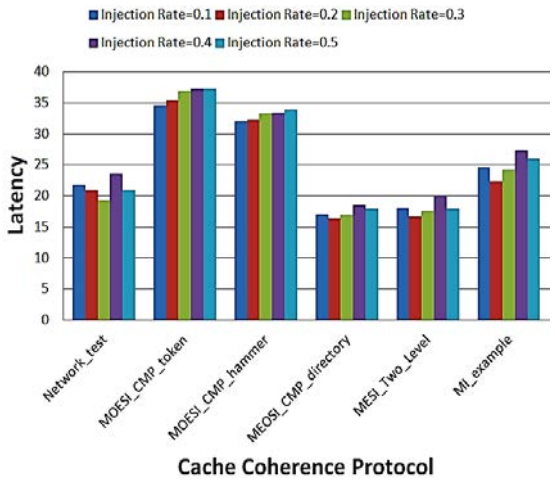

Fig. 2: Average network latency

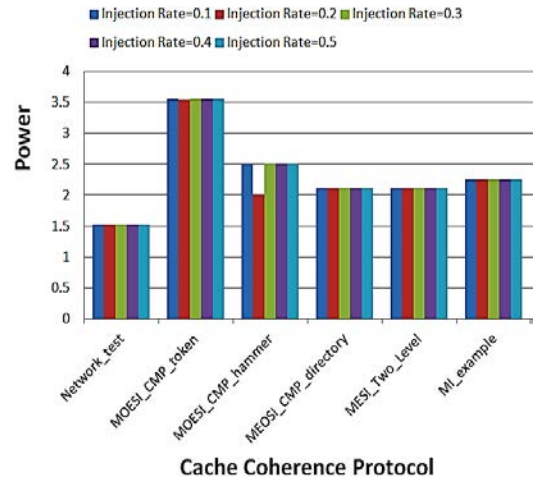

Fig. 3: Total Router Power consumption

CONCLUSION

In this study, we investigated the impact of cache coherence protocols to CMP architectures. For this purpose we used the GEM5 simulator and invoked Garnet network on it. According result figures, MOESI_CMP_token protocol has maximum average network lateny and power consumption. Wherease, MESI_Two_Level and MOESI_CMP_directory protocols have minimum latency and Network_test has minimum power consumption.

ACKNOWLEDGEMENTS

This research project with title "Evaluation of Cache Coherence Protocols in terms of Power and Latency in Multiprocessors" was supported by Islamic Azad University, Malekan Branch. We thank Islamic Azad University, Malekan Branch for financial support.

REFERENCES

Agarwal, A., R. Simoni, J. Hennessy and M. Horowitz, 1988. An evaluation of directory schemes for cache coherence. Proceedings of the Conference on ACM SIGARCH Computer Architecture News, May 30-June 2, 1988, IEEE, New York, USA., ISBN:0-8186-0861-7, pp: 280-298.

Agarwal, N., T. Krishna, L.S. Peh and N.K. Jha, 2009. GARNET: A detailed on-chip network model inside a full-system simulator. Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, April 26-28, 2009, IEEE, New York, USA., ISBN: 978-1-4244-4184-6, pp: 33-42.

Archibald, J. and J.L. Baer, 1986. Cache coherence protocols: Evaluation using a multiprocessor simulation model. ACM. Trans. Comput. Syst., (TOCS), 4: 273-298.

Atta, I., P. Tozun, A. Ailamaki and A. Moshovos, 2012. Slicc: Self-assembly of instruction cache collectives for oltp workloads. Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture, December 1-5, 2012, IEEE, Washington, DC, USA., ISBN:978-0-7695-4924-8, pp: 188-198.

Binkert, N., B. Beckmann, G. Black, S.K. Reinhardt and A. Saidi *et al*., 2011. The gem5 simulator. ACM. SIGARCH. Comput. Archit. News, 39: 1-7.

Binkert, N.L., R.G. Dreslinski, L.R. Hsu, K.T. Lim and A.G. Saidi *et al*., 2006. The M5 simulator: Modeling networked systems. IEEE. Micro, 26: 52-60.

Butko, A., R. Garibotti, L. Ost and G. Sassatelli, 2012. Accuracy evaluation of gem5 simulator system. Proceedings of the 7th International Conference on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), July 9-12, 2012, IEEE, Montpellier, France, ISBN:978-1-4673-2572-1, pp: 1-7.

Kivity, A., Y. Kamay, D. Laor, U. Lublin and A. Liguori, 2007. Kvm: The Linux virtual machine monitor. Proc. Linux Symp., 1: 225-230.

Kumar, M. and P. Arora, 2012. A survey of cache coherence protocols in multiprocessors with shared memory. Proc. Intl. Conf. Adv. Comput. Sci. Electron. Eng., 2: 148-152.

Lametti, S., 2010. Cache coherence techniques. Master's Thesis, School of Computer Science, University of Pisa, Pisa, Italy.

Protic, J., M. Tomasevic and V. Milutinovic, 1998. Distributed Shared Memory: Concepts and Systems. Vol. 21, John Wiley & Sons, Hoboken, New Jersey, USA.,.

Sherwood, T., E. Perelman, G. Hamerly and B. Calder, 2002. Automatically characterizing large scale program behavior. ACM. SIGOPS. Operating Syst. Rev., 36: 45-57.

Stenstrom, P., 1990. A survey of cache coherence schemes for multiprocessors. Comput., 23: 12-24.

Suh, T., 2006. Integration and evaluation of cache coherence protocols for multiprocessor socs. Ph.D Thesis, School of Electrical and Computer Engineering, Georgia Institute of Technology, Georgia, Georgia Tech.