INTERNATIONAL JOURNAL OF NATURAL AND APPLIED SCIENCES (IJNAS), VOL. 12, Special Edition (2019); P. 9 – 14, 1 TABLE, 4 FIG.

Models for addressing cache coherence problems in multilevel caches: A review paper

I. I. Arikpo* and A. T. Okoro

ABSTRACT

The computational tasks of the 21st century require systems with high processing speed. This can be achieved by having, among other system specifications, a high capacity microprocessor and large cache size in a single processor system. The cache prevents round-trips to RAM for every memory access. Nevertheless, dependence on a single cache for both data and instructions limits the performance of the system regardless of the capacity of the processor and cache size. Multilevel cache is a technological advancement that has made it possible for a multi-core processor system to have more than one cache. The introduction of multilevel cache technology no doubt speeds up system operations and enhances performance, but is also not without limitations. Cache coherence issues are common problems inherent in this design. Different protocols have been designed to ameliorate the issues surrounding cache coherence. The aim of this paper is to discuss the efficiency of these protocols in addressing cache coherence problems in multilevel cache technologies.

INTRODUCTION

Computer hardware consists of many components including the memory unit, which is primarily used for information storage. In considering the hierarchy of computer memory with respect to speed, cost and size, an on-chip memory element called cache acts as a staging device between the main memory and processor to improve memory access times by the microprocessor (Figure 1). It is usually a small-sized high-speed random access memory of the computer that is usually integrated into the central processing unit (CPU) chip or as a separate chip on the motherboard with interconnect bus to the CPU. Its purpose is to store program instructions or data that are repeatedly used by the system, rather than fetching such instructions and/or data from the main memory for next program execution. Once these instructions and/or data are cached, the system limits its call to the main memory, but uses the cached information, thereby speeding up the computing process, while reducing DRAM (dynamic random-access memory) accesses and power. This explains why systems with larger cache sizes tend to perform faster than systems with higher capacity processors but small cache sizes (Rouse, 2019).

The closer a cache is to the processor, the faster the computing power of the processor. The concept is predicted on the premise of locality of reference, which is the tendency of the processor to access same set of memory locations for the reuse of recently used instructions and data over a short period of time (Stallings, 2013). With the ever-increasing rate of processor speed, an equivalent increase in cache memory is necessary to ensure that data is timely synchronized between main memory and processor (Vivek, 2016).

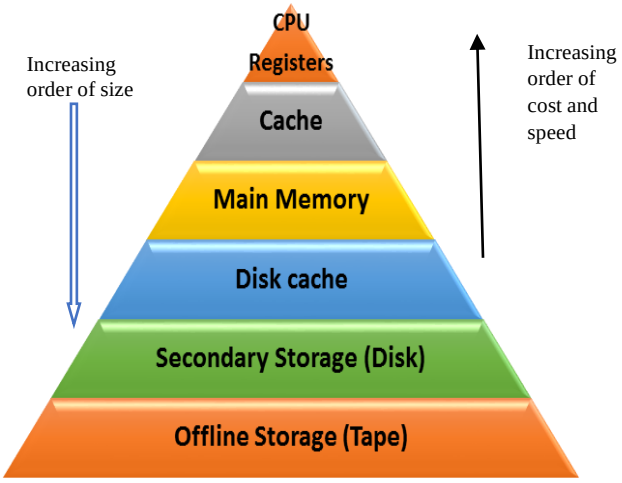

Fig. 1. Computer memory hierarchy

Some terminologies common with cache include: cache HIT, cache MISS, hit ratio and MISS penalty. Cache HIT describes the situation in which a memory access by processor for data or instruction finds datum in cache. It is served by reading data from the cache, which is faster than recomputing a result or reading from

*Corresponding author. Email: iwara.arikpo@unical.edu.ng
Department of Computer Science, University of Calabar, Nigeria

a slower data store; thus, the more requests that can be served from the cache, the faster the system performs. Cache MISS describes the process if what was searched is not found. The total number of hits divided by the sum of total number of hits and miss equals the hit ratio. A hit ratio close to the value of 1 is usually desirable. Miss penalty describes the extra time taking by cache to get data from the memory whenever there is a miss in cache (Vivek, 2016). A reduction of miss rate, miss penalty and time it takes to hit the cache will overly improve cache performance. Multilevel cache architecture design is one of the reliable technologies for enhancing cache performance by reducing the miss penalty and energy consumption.

It is important to know that multicore processors require each processor to have its own cache (Figure 2). This enables the cores to read or write to their respective caches without interfering with other CPU cores (Houman, 2016). When the processing task is completed, the cores are expected to communicate and synchronize data in the respective caches. Multilevel cache employs direct mapping, fully associative mapping and set associative mapping as configurations in addressing the communication between the processor cores.

Direct mapping requires storing of memory blocks to a pre-determined, specific and unique cache location. This is then accessed using 4-bit field comprised of bit size of main memory, bit size of each word, number of words in each block and number of cache lines. Once a cache line is filled and there is need for new data, the old data is overwritten with the new data (Houman, 2016). This often results in cache incoherence if the replacement is not effected across all the caches. Fully associative maps data to any cache line without constraints, while set associative allocates every block of memory into k cache lines where k is the total number of blocks in the cache.

Despite the possibility of cache coherence problem which are discussed in the next section, multilevel caches are useful for fast responsive and cost-effective systems. The direct mapping and other configurations used in its design enhances memory access and data transfer among the cores and memory, but has the limitation of cache incoherence.

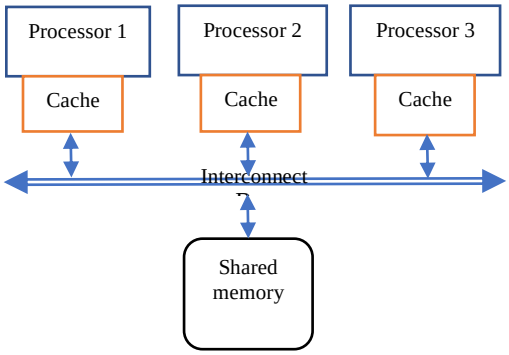

Fig. 2. Multicore system with shared memory

Cache coherence

With the advancement in technology, multiprocessors or distributed shared memory systems (DSM) are now designed to have more than one level of cache embedded in each processor, such that multiple copies of program instruction and operands exist simultaneously in each processor cache and the main memory. Though this arrangement helps to improve operating performance of the system since the system will readily make use of the instruction in the cache of a particular processor that is in active state, it also introduces a bottleneck known as the cache coherence problem which often arises when update of information in a particular cache is not reflected in all other caches holding the same information (Stallings, 2013). This disparity of data across the caches is often as a result of write back policy – an approach used by the memory to perform write operation only on the cache memory (Dandamudi, 2003).

Cache coherence is therefore a design consideration in computer memory architecture that ensures consistency of data stored in cache memory (Techopedi, 2019). It prevents overwriting or loss of data by maintaining a uniform state for each cached block of data. Different software and hardware techniques have been employed to efficiently manage cache coherence problems. The software approach (usually through device drivers) flushes dirty[1] data from caches and invalidates old data to enable sharing with other processors in the system. This has effect on the processor cycles, bus bandwidth and power which often results in high cost of cache maintenance and inefficient cache utilization, especially for high rates of sharing between caches. Software solutions rely on

[1] Data in cache is modified and the main memory has an old and stale copy

Models for addressing cache coherence problems

compiler or operating system dealing with coherence problem. It cannot deal with coherence problem at runtime (Kumar & Arora, 2012). The hardware approach (commonly known as cache coherence protocols and adjudged the most efficient solution), provides dynamic recognition at run time of potential inconsistency conditions (Kumar & Gupta, 2017). Problem of inconsistency is addressed as it occurs, thereby providing more effective use of caches and improved system performance (Stallings, 2013).

Cache coherence protocols

Classified based on their implementation of cache coherence: invalidation-based or update-based protocols; cache coherence protocols are technological innovations that allow recently used local variables get into the appropriate cache and stay there through numerous reads and write, while using the protocol to maintain consistency of shared variables that might be in multiple caches at the same time. Invalidation-based protocol is more popular. It invalidates sharers when a write miss (upgrade) appears on a bus such that sharers will miss next time they try to access the line. It is helpful for write back caches. Update-based protocol updates the sharer caches with new value on a write. There can be multiple readers as well as multiple writers. If a processor updates a shared line, then updated line is distributed to all other processors and caches containing that line can update it (Kumar & Gupta, 2017). It is not very attractive for write back caches.

The type of protocol to be used in the memory design of a system depends on the intended program behaviour and hardware cost. There are numerous types of the cache coherence protocols classified into directory-based coherence and snoopy protocols. This paper considers the review of these protocols with interest on their performance and complexity.

Directory-based coherence protocol

This tracks locations of all copies of every block of shared data and store it in a directory (Figure 3). It works as a look-up table used by processors to identify coherence and consistency of data which is currently being updated (Al-Hothali et al., 2010). For each block, a centralized directory maintains information on the state of the block in all other caches. Processor requests for a block is usually channeled to the directory containing block which then forwards the request or serves the request depending on state of block. This means that once a block in memory is updated, the processor consults the directory to either update or invalidate the copies of the data in other caches. It is a kind of point-to-point communication. Though it is complex to implement, it is trackable and optimal for achieving cache coherency in arbitrary interconnection network. It is scalable and suitable for network configuration. Directory schemes bottlenecks include: one-fixed (centralized) location and the overhead of communication between the various cache controllers and the central controller.

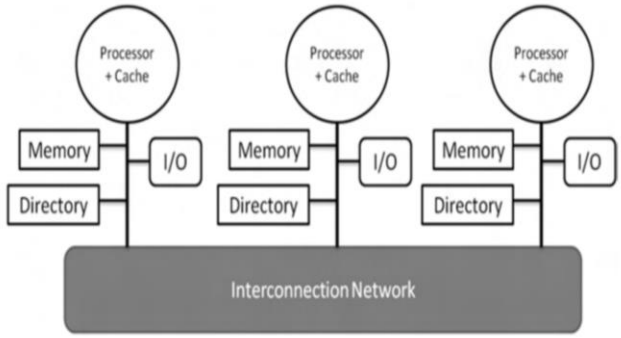

Fig. 3. Directory based protocol

Snoopy Protocol

Bus snooping protocol as shown in figure 4 requires the use of a bus-based communication medium in the machine to broadcast messages to other connected processors (Al-Hothali et al, 2010; Kaushik et al, 2015; Sultan et al, 2015). The concept allows each cache to maintain cache coherence by listening to the interconnect bus for updates or write action on shared data. Each processor monitors the activity on the bus. For read instructions, all caches check to see if they have a copy of the requested block. If yes, they may have to supply the data. Similarly, for write instructions, all caches check to see if they have a copy of the data. If yes, they either invalidate the local copy, or update it with the new value. Snoopy protocol is scalable, inexpensive and efficient. Two primary categories of snoopy protocols are write-through/write-invalidate, which invalidates data in all other caches once data is written in one. The second category is the write-update/write-broadcast which updates all cached copies of the data item that is written. Although this second approach causes no cache miss and makes the new values appear in caches sooner, it consumes considerably much bandwidths due to the need to broadcast (global) all writes to shared cache lines. Pentium IV and PowerPC

are examples of commercial multiprocessors that use snoopy protocols extensively.

Table 1 shows the merits and demerits of directory-based and snoopy protocols respectively.

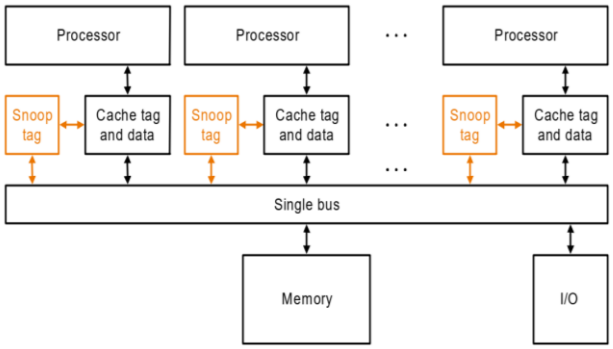

Fig. 4. Snoopy protocol

Table 1. Snoopy vs Directory based protocols

	Snoopy protocols	**Directory based protocols**
Advantages	1. low average miss latency, especially for cache-to-cache misses 2. With enough bandwidths, it is inherently faster.	1.Scale much better than snoopy protocols. 2 Much less bandwidths because messages are point-to-point and not broadcast.
Disadvantages	1. Scalability: Each request must be sent to all nodes in a system, meaning that, if the system gets larger, the size of the (logical or physical) bus and the bandwidth it provides should grow. 2. Not efficient from the point of view of power dissipation.	1 Longer latency. 2.The directory access and the extra interconnect traversal is on the critical path of cache to cache misses. 3. It involves the storage and manipulation of directory state.

Implementation of Cache Coherence

Both Directory-based and snoopy protocols keep track of every block in a cache using three states. These states are generally classified as: *uncached/Invalid* (This means the block is invalid in all caches, no processor cache has it); *Shared/clean* (the block is cached in one or more processor and memory and it is up-to-date) and *Exclusive/dirty* (the block is up-to-date in only one of the processors (owner), memory is out-of-date).

MSI, MESI, MOSI, MOESI and MESIF and MOESIF are different protocols commonly used to further describe these states of blocks in caches and memory.

1. MSI. This is a 3-state write back invalidation technique. It ensures that a dirty block in one cache cannot be valid in any other cache and that a read request gets the most updated data.

The cache block can be in three states: Modified: The cache line is dirty and valid but the cache line is not found in other local caches of other processors. The cache line holds the most recent, correct copy of the data. The copy in main memory is stale (incorrect), and no other processor holds a copy.

Shared/clean: The block is cached in one or more processor(s) and memory and it is up-to-date. That is, the state in which the line is valid, same with main memory. Other processors may have copies of the cache line. Invalid or Uncached: This means the block is invalid in all caches, no processor cache has it. Valid copies of the data can be either in main memory or another processor cache.

The disadvantage of MSI is that each read-write sequence incurs two bus transactions. For instance, a processor that needs to read a block which none of the other processors have and then write to it, will first make a BusRd request to read the block followed by a BusRdX request before writing to the block. This second BusRdX request is not necessary as none of the other caches have the same block, but there is no way for one cache to know about this.

2 MESI protocol: This protocol divides the exclusive state into *Exclusive* and *Modified.* Exclusive: This represents the state in which the line is not dirty but valid. The cache line holds the most recent, correct copy of the data. The copy in main memory is also the most recent, correct copy of the data but the cache line is not found in the local caches of other processors. MESI protocol is implemented in Intel Duo Core, Pentium processors.

3. MOSI protocol: This is an extension of the basic MSI protocol. The *Owned (O) state* holds the most recent correct copy. The copy in main memory can be stale (incorrect). Dirty values can be shared with other sharing caches without even updating the value in the main memory. Only one processor can hold the data in the owned state—all other processors must hold the data in the shared state.

4.MOESI protocol: MOESI represents four states denoting a cache line. The *owned state* is an optimization to the MESI protocol. The advantage of MOESI protocol over others lies in the "exclusive" state which results in saving a bus request especially when a sequential application is running as only one processor will be working on it, all the accesses will be exclusive. MOESI protocol guarantees bandwidth efficiency, Low-Latency Cache-to-Cache Misses and less reliability on bus. It is supported in AMD (Advanced Micro Devices) Dual Core and Opteron processors.

5.MESIF: Developed by Intel for cache coherence non-uniform memory architectures (ccNUMA). The additional state Forward (**F)** a specialized form of the Shared (S) state. It ensures that a cache is assigned as a designated responder for any requests for a given block line. MESIF protocol ensures that, if any cache holds a line in the Shared state, at most one (other) cache holds it in the Forward state (Thomadakis, 2019).

MOESI protocol vs MESIF protocol

Two states –*Owned* in MOESI and *Forward* in MESIF uniquely distinguishe both coherence protocols. Both states generally aid efficient data transfer using direct cache-to-cache transfers instead of depending on data from memory. However, the functions of these states clearly vary from each other. While a cache line in the *forward state* is clean and may be discarded at any time without notice, a cache line in the *owned state* is dirty and must be written back to memory before being discarded.

Review Findings

1. The essence for the modifications of these invalidation-based protocols from MSI to MOESIF is to reduce the number of write-backs to-and-from the memory; thereby improving bandwidth usage of the systems.
2. Although the MOESI protocol has fewer write-backs because of its feature for dirty sharing, it lost out on cache-to-cache transfers since the shared state is not allowed to flush.
3. Exchange or addition of extra state to the MSI protocol reduces the possibility of coherence misses, while improving the efficiency of cache-to-cache transfer.
4. The performance output of the protocols is found to be in increasing order of the modifications – **MSI→**(*exclusive state overcomes the drawback of MSI that each read-write sequence incurs 2 bus transaction*) **MESI→**(*memory-based transfers is greatly reduced since the new state 'owned' can provide modified data to other processors without even writing it to main memory*) **MOESI→** (*F is elected to take charge of forwarding data when there is multiple clean copies of data*) **MOESIF** (Gupta, 2016).
5. In summary, a system with MOESIF protocol will perform far better in management of cache coherency compared to a system implemented with MSI protocol (Gupta, 2016; Patil et al., 2019).

CONCLUSION

Multiprocessor systems require each processor to have its own cache. It is ideal for caches in such designs to maintain coherency using different protocols. The implementation of these protocols is

usually attributed to a specific machine, thereby leaving the question on which protocol can be adjudged the best. Comparing cache protocols would require comparing the performance on different machine architectures. This approach is not famous since differences in machine architecture or other implementation designs inevitably complicate the protocol comparison.

REFERENCES

Al-Hothali, S., Soomro, S., Tanvir, K., & Tuli, R. (2010). Snoopy and Directory Based Cache Coherence Protocols: A Critical Analysis. *Journal of Information & Communication Technology, 4*(1):1-10.

Dandamudi, S. (2003). Cache Memory. In S. Dandamudi, *Fundamentals of Computer Organization and Design.* Springer.

Gupta, S. (2016). *Cache Coherence Protocols*. Retrieved from Github:https://github.com/sahilgupta5/CacheCoherenceProtocols/blob/695a75b56c4c1bcd65074152dbd29d78ccf0b79c/Final%20Report.pdf

Houman, P. (2016). *cache memory 2016 IEEE PAPER.* Retrieved fromhttp://cal.ucf.edu/3801 reports_summer_2016/effect_multilevel.pdf

Kaushik, R., Pavan, K. S., & Meenatchi, S. (2015). Comparative study on Cache Coherence Protocols. *IOSR Journal of Computer Engineering, 17*(3), 71-75.

Kumar, M., & Arora, P. (2012). A Survey of Cache Coherence Protocols in Multiprocessors with Shared Memory. *UACEE International Journal of Computer Science and its Applications, 2*(1), 148-152.

Kumar, S., & Gupta, K. (2017). Comparative Study and Performance Analysis of Cache Coherence Protocols. *International Journal of Computer Sciences and Engineering, 5*(5), 213-216.

Patil, G., Mallya, N., & Raveendran, B. K. (2019). MOESIF: a MC/MP cache coherence protocol with improved bandwidth utilisation. *International Journal of Embedded Systems, 11*(4). doi:10.1504/IJES.2019.100866

Rouse, M. (2019). *What is cache memory?* Retrieved from TechTarget: https://searchstorage.techtarget.com/definition/cache-memory

Stallings, W. (2013). *Computer Organization and Architecture: Designing for performance.* Boston: Pearson.

Sultan, A., Abdulwahab, A., & Mohammed, A. (2015). Cache Coherence Mechanisms. *International Journal of Engineering and Innovative Technology, 4*(7), 158-167.

Techopedi. (2019). *Cache coherence.* Retrieved from Technopedia: https://www.techopedia.com/definition/300/cache-coherence

Thomadakis, M. (2019). *The Architecture of the Nehalem Processor and Nehalem-EP SMP Platforms.* Retrieved from Wayback Machine: https://web.archive.org/web/20140811023120/http://sc.tamu.edu/systems/eos/nehalem.pdf

Vivek, C. (2016). Cache Memory: An Analysis on Performance Issues. *International Journal of Advance Research in Computer Science and Management Studies, 4*(7), 26-28.