MESI protocol

The **MESI protocol** is an Invalidate-based cache coherence protocol, and is one of the most common protocols that support write-back caches. It is also known as the **Illinois protocol** (due to its development at the University of Illinois at Urbana-Champaign[1]). Write back caches can save a lot on bandwidth that is generally wasted on a write through cache. There is always a dirty state present in write back caches that indicates that the data in the cache is different from that in main memory. Illinois Protocol requires cache to cache transfer on a miss if the block resides in another cache. This protocol reduces the number of Main memory transactions with respect to the MSI protocol. This marks a significant improvement in the performance.[2]

Contents

States

The letters in the acronym MESI represent four exclusive states that a cache line can be marked with (encoded using two additional bits):

Modified (M)
: The cache line is present only in the current cache, and is *dirty* - it has been modified (M state) from the value in main memory. The cache is required to write the data back to main memory at some time in the future, before permitting any other read of the (no longer valid) main memory state. The write-back changes the line to the Shared state(S).

Exclusive (E)
: The cache line is present only in the current cache, but is *clean* - it matches main memory. It may be changed to the Shared state at any time, in response to a read request. Alternatively, it may be changed to the Modified state when writing to it.

Shared (S)
: Indicates that this cache line may be stored in other caches of the machine and is *clean* - it matches the main memory. The line may be discarded (changed to the Invalid state) at any time.

Invalid (I)
: Indicates that this cache line is invalid (unused).

For any given pair of caches, the permitted states of a given cache line are as follows:

	M	E	S	I
M	✗	✗	✗	✓
E	✗	✗	✗	✓
S	✗	✗	✓	✓
I	✓	✓	✓	✓

When the block is marked M (modified) or E (exclusive), the copies of the block in other Caches are marked as I(Invalid).

Operation

The state of the FSM transitions from one state to another based on 2 stimuli. The first stimulus is the processor specific Read and Write request. For example: A processor P1 has a Block X in its Cache, and there is a request from the processor to read or write from that block. The second stimulus comes from another processor, which doesn't have the Cache block or the updated data in its Cache, through the bus connecting the processors. The bus requests are monitored with the help of Snoopers,[4] which snoops all the bus transactions.

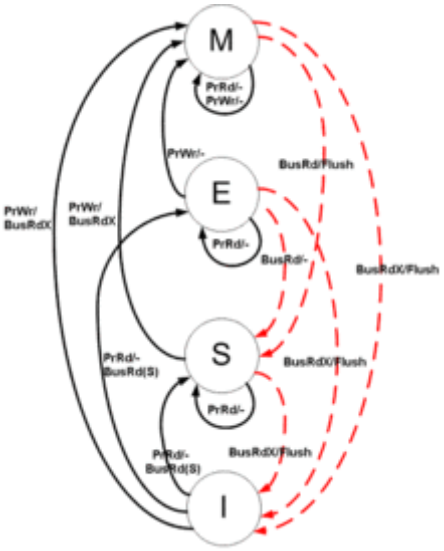

Image 1.1 State diagram for MESI protocol Red: Bus initiated transaction. Black: Processor initiated transactions.[3]

Following are the different type of Processor requests and Bus side requests:

Processor Requests to Cache includes the following operations:

1. PrRd: The processor requests to **read** a Cache block.
2. PrWr: The processor requests to **write** a Cache block

Bus side requests are the following:

1. BusRd: Snooped request that indicates there is a **read** request to a Cache block requested by another processor
2. BusRdX: Snooped request that indicates there is a **write** request to a Cache block requested by another processor that **doesn't already have the block.**
3. BusUpgr: Snooped request that indicates that there is a write request to a Cache block requested by another processor but that processor already has that **Cache block residing in its own Cache**.
4. Flush: Snooped request that indicates that an entire cache block is written back to the main memory by another processor.
5. FlushOpt: Snooped request that indicates that an entire cache block is posted on the bus in order to supply it to another processor(Cache to Cache transfers).

(*Such Cache to Cache transfers can reduce the read miss latency if the latency to bring the block from the main memory is more than from Cache to Cache transfers, which is generally the case in bus based systems. But in multicore architectures, where the coherence is maintained at the level of L2 caches, there is on chip L3 cache, it may be faster to fetch the missed block from the L3 cache rather than from another L2*)

Snooping Operation: In a snooping system, all caches on the bus monitor (or snoop) all the bus transactions. Every cache has a copy of the sharing status of every block of physical memory it has stored. The state of the block is changed according to the State Diagram of the protocol used. (Refer image above for MESI state diagram). The bus has snoopers on both sides:

1. Snooper towards the Processor/Cache side.
2. The snooping function on the memory side is done by the Memory controller.

Explanation:

Each Cache block has its own 4 state Finite State Machine(refer image 1.1). The State transitions and the responses at a particular state with respect to different inputs are shown in Table1.1 and Table 1.2

Table 1.1 State Transitions and response to various Processor Operations

Initial State	Operation	Response
Invalid(I)	PrRd	▪ Issue BusRd to the bus ▪ other Caches see BusRd and check if they have a valid copy, inform sending cache ▪ State transition to (S)**Shared**, if other Caches have valid copy. ▪ State transition to (E)**Exclusive**, if none (must ensure all others have reported). ▪ If other Caches have copy, one of them sends value, else fetch from Main Memory
	PrWr	▪ Issue BusRdX signal on the bus ▪ State transition to (M)**Modified** in the requestor Cache. ▪ If other Caches have copy, they send value, otherwise fetch from Main Memory ▪ If other Caches have copy, they see BusRdX signal and Invalidate their copies. ▪ Write into Cache block modifies the value.
Exclusive(E)	PrRd	▪ No bus transactions generated ▪ State remains the same. ▪ Read to the block is a Cache Hit
	PrWr	▪ No bus transaction generated ▪ State transition from Exclusive to (M)**Modified** ▪ Write to the block is a Cache Hit
Shared(S)	PrRd	▪ No bus transactions generated ▪ State remains the same. ▪ Read to the block is a Cache Hit.
	PrWr	▪ Issues BusUpgr signal on the bus. ▪ State transition to (M)**Modified**. ▪ other Caches see BusUpgr and mark their copies of the block as (I)Invalid.
Modified(M)	PrRd	▪ No bus transactions generated ▪ State remains the same. ▪ Read to the block is a Cache hit
	PrWr	▪ No bus transactions generated ▪ State remains the same. ▪ Write to the block is a Cache hit.

Table 1.2 State Transitions and response to various Bus Operations

Initial State	Operation	Response
Invalid(I)	BusRd	▪ No State change. Signal Ignored.
	BusRdX/BusUpgr	▪ No State change. Signal Ignored
Exclusive(E)	BusRd	▪ Transition to **Shared** (Since it implies a read taking place in other cache). ▪ Put FlushOpt on bus together with contents of block.
	BusRdX	▪ Transition to **Invalid**. ▪ Put FlushOpt on Bus, together with the data from now-invalidated block.
Shared(S)	BusRd	▪ No State change (other cache performed read on this block, so still shared). ▪ May put FlushOpt on bus together with contents of block (design choice, which cache with Shared state does this).
	BusUpgr	▪ Transition to **Invalid** (cache that sent BusUpgr becomes Modified) ▪ May put FlushOpt on bus together with contents of block (design choice, which cache with Shared state does this)
Modified(M)	BusRd	▪ Transition to **(S)Shared.** ▪ Put FlushOpt on Bus with data. Received by sender of BusRd and Memory Controller, which writes to Main memory.
	BusRdX	▪ Transition to **(I)Invalid**. ▪ Put FlushOpt on Bus with data. Received by sender of BusRdx and Memory Controller, which writes to Main memory.

A write may only be performed freely if the cache line is in the Modified or Exclusive state. If it is in the Shared state, all other cached copies must be invalidated first. This is typically done by a broadcast operation known as *Request For Ownership (RFO)*.

A cache that holds a line in the Modified state must *snoop* (intercept) all attempted reads (from all of the other caches in the system) of the corresponding main memory location and insert the data that it holds. This can be done by forcing the read to *back off* (i.e. retry later), then writing the data to main memory and changing the cache line to the Shared state. It can also be done by sending data from Modified cache to the cache performing the read. Note, snooping only required for read misses (protocol ensures that Modified cannot exist if any other cache can perform a read hit).

A cache that holds a line in the Shared state must listen for invalidate or request-for-ownership broadcasts from other caches, and discard the line (by moving it into Invalid state) on a match.

The Modified and Exclusive states are always precise: i.e. they match the true cache line ownership situation in the system. The Shared state may be imprecise: if another cache discards a Shared line, this cache may become the sole owner of that cache line, but it will not be promoted to Exclusive state. Other caches do not broadcast notices when they discard cache lines, and this cache could not use such notifications without maintaining a count of the number of shared copies.

In that sense the Exclusive state is an opportunistic optimization: If the CPU wants to modify a cache line that is in state S, a bus transaction is necessary to invalidate all other cached copies. State E enables modifying a cache line with no bus transaction.

Illustration of MESI protocol operations[5]

Let us assume that the following stream of read/write references. All the references are to the same location and the digit refers to the processor issuing the reference.

The stream is : R1, W1, R3, W3, R1, R3, R2.

Initially it is assumed that all the caches are empty.

Table 1.3 An example of how MESI works All operations to same cache block (Example: "R3" means read block by processor 3)

	Local Request	*P1*	*P2*	*P3*	*Generated Bus Request*	*Data Supplier*
0	**Initially**	**-**	**-**	**-**	**-**	**-**
1	**R1**	**E**	**-**	**-**	**BusRd**	**Mem**
2	**W1**	**M**	**-**	**-**	**-**	**-**
3	**R3**	**S**	**-**	**S**	**BusRd**	**P1's Cache**
4	**W3**	**I**	**-**	**M**	**BusUpgr**	**-**
5	**R1**	**S**	**-**	**S**	**BusRd**	**P3's Cache**
6	**R3**	**S**	**-**	**S**	**-**	**-**
7	**R2**	**S**	**S**	**S**	**BusRd**	**P1/P3's Cache**

Note: *The term snooping referred to below is a protocol for maintaining cache coherency in symmetric multiprocessing environments. All the caches on the bus monitor(snoop) the bus if they have a copy of the block of data that is requested on the bus.*

Step 1: As the cache is initially empty, so the main memory provides P1 with the block and it becomes exclusive state.

Step 2: As the block is already present in the cache and in an exclusive state so it directly modifies that without any bus instruction. The block is now in a modified state.

Step 3: In this step, a BusRd is posted on the bus and the snooper on P1 senses this. It then flushes the data and changes its state to shared. The block on P3 also changes its state to shared as it has received data from another cache. There is no main memory access here.

Step 4: Here a BusUpgr is posted on the bus and the snooper on P1 senses this and invalidates the block as it is going to be modified by another cache. P3 then changes its block state to modified.

Step 5: As the current state is invalid, thus it will post a BusRd on the bus. The snooper at P3 will sense this and so will flush the data out. The state of the both the blocks on P1 and P3 will become shared now. Notice that this is when even the main memory will be updated with the previously modified data.

Step 6: There is a hit in the cache and it is in the shared state so no bus request is made here.

Step 7: There is cache miss on P2 and a BusRd is posted. The snooper on P1 and P3 sense this and both will attempt a flush. Whichever gets access of the bus first will do that operation.

Read For Ownership

A *Read For Ownership (RFO)* is an operation in cache coherency protocols that combines a read and an invalidate broadcast. The operation is issued by a processor trying to write into a cache line that is in the shared (S) or invalid (I) states of the MESI protocol. The operation causes all other caches to set the state of such a line to I. A read for ownership transaction is a read operation with intent to write to that memory address. Therefore, this operation is exclusive. It brings data to the cache and invalidates all other processor caches that hold this memory line. This is termed "BusRdX" in tables above.

Memory Barriers

MESI in its naive, straightforward implementation exhibits two particular performance issues. First, when writing to an invalid cache line, there is a long delay while the line is fetched from another CPU. Second, moving cache lines to the invalid state is time-consuming. To mitigate these delays, CPUs implement store buffers (http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0363e/Chdcahcf.html) and invalidate queues.[6]

Store Buffer

A store buffer is used when writing to an invalid cache line. Since the write will proceed anyway, the CPU issues a read-invalid message (hence the cache line in question and all other CPUs' cache lines that store that memory address are invalidated) and then pushes the write into the store buffer, to be executed when the cache line finally arrives in the cache.

A direct consequence of the store buffer's existence is that when a CPU commits a write, that write is not immediately written in the cache. Therefore, whenever a CPU needs to read a cache line, it first has to scan its own store buffer for the existence of the same line, as there is a possibility that the same line was written by the same CPU before but hasn't yet been written in the cache (the preceding write is still waiting in the store buffer). Note that while a CPU can read its own previous writes in its store buffer, other CPUs *cannot see those writes* before they are flushed from the store buffer to the cache - a CPU cannot scan the store buffer of other CPUs.

Invalidate Queues

With regard to invalidation messages, CPUs implement invalidate queues, whereby incoming invalidate requests are instantly acknowledged but not in fact acted upon. Instead, invalidation messages simply enter an invalidation queue and their processing occurs as soon as possible (but not necessarily instantly). Consequently, a CPU can be oblivious to the fact that a cache line in its cache is actually invalid, as the invalidation queue contains invalidations that have been received but haven't yet been applied. Note that, unlike the store buffer, the CPU can't scan the invalidation queue, as that CPU and the invalidation queue are physically located on opposite sides of the cache.

As a result, memory barriers are required. A store barrier will flush the store buffer, ensuring all writes have been applied to that CPU's cache. A read barrier will flush the invalidation queue, thus ensuring that all writes by other CPUs become visible to the flushing CPU. Furthermore, memory management units do not scan the store buffer, causing similar problems. This effect is visible even in single threaded processors.[7]

Advantages of MESI over MSI

The most striking difference between the two protocols is the extra "exclusive" state present in the MESI protocol. This extra state was added as it has many advantages. When a processor needs to read a block that **none of the other processors have** and then write to it, two bus transactions will take place in the case of MSI. First, a BusRd request is issued to read the block followed by a BusRdX request before writing to the block. The BusRdX request in this scenario is useless as none of the other caches have the same block, but there is no way for one cache to know about this. Thus, MESI protocol overcomes this limitation by adding an Exclusive state, which results in saving a bus request. This makes a huge difference when a sequential application is running. As only one processor will be working on it, all the accesses will be exclusive. MSI performs very badly in this case. Even in the case of a highly parallel application where there is minimal sharing of data, MESI is far faster. Adding the Exclusive state also comes at no cost as 3 states and 4 states are both encoded with 2 bits.

Disadvantage of MESI

In case continuous read and write operations are performed by various caches on a particular block, the data has to be flushed to the bus every time. Thus the main memory will pull this on every flush and remain in a clean state. But this is not a requirement and is just an additional overhead caused because of the implementation of MESI. This challenge was overcome by MOESI protocol.[8] In case of S (Shared State), multiple snoopers may response with FlushOpt with the same data (see the example above). F state in MESIF addresses to eliminate this redundancy.

See also

- Coherence protocol
- MSI protocol, the basic protocol from which the MESI protocol is derived.
- Write-once (cache coherency), an early form of the MESI protocol.
- MOSI protocol
- MOESI protocol
- MESIF protocol
- MERSI protocol
- Dragon protocol
- Firefly protocol

References

1. Papamarcos, M. S.; Patel, J. H. (1984). "A low-overhead coherence solution for multiprocessors with private cache memories" (http://www.csl.cornell.edu/courses/ece5720/papamarcos.isca84.pdf) (PDF). *Proceedings of the 11th annual international symposium on Computer architecture - ISCA '84*. p. 348. doi:10.1145/800015.808204 (https://doi.org/10.1145%2F800015.808204). ISBN 0818605383. Retrieved March 19, 2013.
2. Gómez-Luna, J.; Herruzo, E.; Benavides, J.I. "MESI Cache Coherence Simulator for Teaching Purposes". *Clei Electronic Journal*. **12** (1, PAPER 5, APRIL 2009). CiteSeerX 10.1.1.590.6891 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.590.6891).
3. Culler, David (1997). *Parallel Computer Architecture*. Morgan Kaufmann Publishers. pp. Figure 5–15 State transition diagram for the Illinois MESI protocol. Pg 286.
4. Bigelow, Narasiman, Suleman. "An evaluation of Snoopy Based Cache Coherence protocols" (http://hps.ece.utexas.edu/people/suleman/class_projects/pca_report.pdf) (PDF). ECE Department, University of Texas at Austin.

5. Solihin, Yan (2015-10-09). *Fundamentals of Parallel Multicore Architecture*. Raleigh, North Carolina: Solihin Publishing and Consulting, LLC. ISBN 978-1-4822-1118-4.
6. Handy, Jim (1998). *The Cache Memory Book*. Morgan Kaufmann. ISBN 9780123229809.
7. Chen, G.; Cohen, E.; Kovalev, M. (2014). "Store Buffer Reduction with MMUs". *Verified Software: Theories, Tools and Experiments*. Lecture Notes in Computer Science. **8471**. p. 117. doi:10.1007/978-3-319-12154-3_8 (https://doi.org/10.1007%2F978-3-319-12154-3_8). ISBN 978-3-319-12153-6.
8. "Memory System (Memory Coherency and Protocol)" (http://www.cs.wustl.edu/~roger/569M/24593.pdf) (PDF). AMD64 Technology. September 2006.

External links

- An interactive MESI simulation (https://www.cs.tcd.ie/Jeremy.Jones/vivio/caches/MESIHelp.htm)
- An open source MESI controller (Verilog) (http://opencores.org/project,mesi_isc)