Cache Coherency in Multiprocessor Systems

The Modified Exclusive Shared Invalid (MESI) algorithm for cache coherency.

<table>
<thead>
<tr>
<th>MESI State</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified (M)</td>
<td>The line is valid in the cache and in only this cache. The line is modified with respect to system memory—that is, the modified data in the line has not been written back to memory.</td>
</tr>
<tr>
<td>Exclusive (E)</td>
<td>The addressed line is in this cache only. The data in this line is consistent with system memory.</td>
</tr>
<tr>
<td>Shared (S)</td>
<td>The addressed line is valid in the cache and in at least one other cache. A shared line is always consistent with system memory. That is, the shared state is shared-unmodified; there is no shared-modified state.</td>
</tr>
<tr>
<td>Invalid (I)</td>
<td>This state indicates that the addressed line is not resident in the cache and/or any data contained is considered not useful.</td>
</tr>
</tbody>
</table>

Note that:

- Exclusive may also be called CleanExclusive
- Modified may also be called DirtyExclusive

Some processors add a fifth state for Shared Modified and call it the MOESI protocol. The caches with the shared modified state update each other’s lines with current data, but do not write it back to main memory.

The five MOESI states are defined as:

- Exclusively Modified (M)
- Shared Modified (O)
- Exclusive Clean (E)
- Shared Clean (S)
- Invalid (I)
Picture the MESI cache states:

**Modified in Cache A**

- Cache A: Valid Data
- Cache B: Invalid Data
- System Memory: Invalid Data

**Shared in Cache A**

- Cache A: Valid Data
- Cache B: Valid Data
- System Memory: Valid Data

**Exclusive in Cache A**

- Cache A: Valid Data
- Cache B: Invalid Data
- System Memory: Valid Data

**Invalid in Cache A**

- Cache A: Invalid Data
- Cache B: Don't Care
- System Memory: Don't Care
**MESI State Diagram**

**Events:**
- RH = Read Hit
- RMS = Read miss, shared
- RME = Read miss, exclusive
- WH = Write hit
- WM = Write miss
- SHR = Snoop hit on read
- SHI = Snoop hit on invalidate
- LRU = LRU replacement

**Bus Transactions:**
- Push = Write cache line back to memory
- Invalidate = Broadcast invalidate
- Read = Read cache line from memory
## MESI State Table

<table>
<thead>
<tr>
<th>State</th>
<th>Event</th>
<th>Action</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid</td>
<td>Read miss, shared (cache copies exist)</td>
<td>Read cache line</td>
<td>Shared</td>
</tr>
<tr>
<td></td>
<td>Read miss, exclusive (no cache copies exist)</td>
<td>Read cache line</td>
<td>Exclusive</td>
</tr>
<tr>
<td></td>
<td>Write miss</td>
<td>• Broadcast invalidate&lt;br&gt;• Read cache line&lt;br&gt;• Modify cache line</td>
<td>Modified</td>
</tr>
<tr>
<td>Shared</td>
<td>Read hit</td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td></td>
<td>Write hit</td>
<td>Broadcast invalidate</td>
<td>Modified</td>
</tr>
<tr>
<td></td>
<td>Snoop hit on read</td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td></td>
<td>Snoop hit on invalidate</td>
<td>Invalidate cache line</td>
<td>Invalid</td>
</tr>
<tr>
<td>Exclusive</td>
<td>Read hit</td>
<td></td>
<td>Exclusive</td>
</tr>
<tr>
<td></td>
<td>Write hit</td>
<td></td>
<td>Modified</td>
</tr>
<tr>
<td></td>
<td>Snoop hit on read</td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td></td>
<td>Snoop hit on invalidate</td>
<td>Invalidate cache line</td>
<td>Invalid</td>
</tr>
<tr>
<td>Modified</td>
<td>Read hit</td>
<td></td>
<td>Modified</td>
</tr>
<tr>
<td></td>
<td>Write hit</td>
<td></td>
<td>Modified</td>
</tr>
<tr>
<td></td>
<td>Snoop hit on read</td>
<td>Write cache line back to memory</td>
<td>Shared</td>
</tr>
<tr>
<td></td>
<td>Snoop hit on invalidate</td>
<td>Write cache line back to memory</td>
<td>Invalid</td>
</tr>
<tr>
<td></td>
<td>LRU Replacement</td>
<td>Write cache line back to memory</td>
<td>Invalid</td>
</tr>
</tbody>
</table>
The synchronization problem, Stone pp 366-369 and section 7.2

A race condition:

Processor A
R3 ← Memory(SharedLoc)
R3 ← R3 + 300
Memory(SharedLoc) ← R3

Processor B
R5 ← Memory(SharedLoc)
R5 ← R5 + 200
Memory(SharedLoc) ← R5

The solution:
Provide a Lock or Semaphore for Memory(SharedLoc). The lock instruction must provide for an atomic read-modify-write in the hardware.

Processor A
Lock SpinLock1 (success)
R3 ← Memory(SharedLoc)
R3 ← R3 + 300
Memory(SharedLoc) ← R3
Unlock SpinLock1

Processor B
Lock SpinLock1 (fail)
Lock SpinLock1 (fail)
Lock SpinLock1 (fail)
Lock SpinLock1 (success)
R5 ← Memory(SharedLoc)
R5 ← R5 + 200
Memory(SharedLoc) ← R5
Unlock SpinLock1

Actually, the Lock instruction would be implemented as (for example) Stone’s Test-and-set (TS) instruction (p.424).

Test-and-set MemoryLoc – in one memory cycle:
• read out the contents of the addressed word & set CC
• write back all 1’s

Define lock = any memory location (e.g. SpinLock1)
if the memory location = 0, the lock is unlocked
if the memory location = 1, the lock is locked

Lock: TEST-AND-SET SpinLock1
Unlock: STORE SpinLock1, 0
BRANCH not-zero Lock
Continue with locked code
What happens to the caches of processors A and B, and to the memory bus during this sequence?

**Assume both coming from Invalid state:**

**Processor A**
- Write miss on TS LockLoc
- Broadcast Invalidate
- Read cache line
- Modify cache line with TS
- Marks cache line modified
- Recognizes snoop invalidate
- Writes back cache line
- Marks cache line invalid
- Send OK to proceed
  - continues with locked code

**Processor B**
- Write miss on TS LockLoc
- Broadcast invalidate
- --- wait for OK from owner

**Sometime later**
- Write miss on store 0 into LockLoc
- Broadcast invalidate
- --- wait on OK from owner
- Receives OK to proceed
- Read cache line
- Modify cache line with TS
- Marks cache line modified
- Spins on TS instruction in cache
- . . .
- Recognizes snoop invalidate
- Writes back cache line
- Marks cache line invalid
- Send OK to proceed
- Write miss on TS LockLoc
- Broadcast invalidate
- --- wait for OK from owner
- Receives OK to proceed
- Read cache line
- Modify cache line with TS
- Marks cache line modified
- continues with locked code
Can you imagine what happens if there are three or more processors contending for the lock?

**Solutions to cache thrashing on synchronization locks:**

- Use normal non-cached memory locations for locks  
  ⇒ Still saturates the memory bus

- Provide a special, high-speed non-cached memory for locks  
  ⇒ Also need a special bus to access it

- Provide a special synchronization processor to handle locks  
  ⇒ Best solution for high-performance

**The memory consistency problem** – Stone section 6.5 (p.392)

- We optimize execution of a processor by using several execution units (superscalar)

- Some instructions are executed out of order, but the *processor ensures correct result-ordering* where explicit dependencies exist.

- The LOCK and UNLOCK instructions must be defined to have explicit dependencies on all other instructions:
  
  ⇒ The LOCK instruction must ensure that NO instructions following it have started prior to its completion (Stone’s ACQUIRE).
  
  ⇒ The UNLOCK instruction must ensure that ALL instructions preceding it have completed prior to its beginning execution (Stone’s RELEASE).

- These orderings must be maintained across all processors and memories. See Stone page 396-401 for the discussion on weak consistency and release consistency.
**Strong Consistency**

- *All* instructions must have globally consistent ordering.

**Weak Consistency**

- *Synchronizing* instructions must have globally consistent ordering with all instructions.

**Acquire Consistency**

- The *acquire* (lock) instruction must be globally complete before any following instruction begins.

**Release Consistency**

- The *release* (lock) instruction must not begin before all preceding instructions are globally complete.
From the MIPS R10000 User Manual

The R10000 processor *behaves* as if strong ordering is implemented, although it does not actually execute all memory operations in strict program order.

In the R10000 processor, store operations remain *pending* until the store instruction is ready to graduate. Thus, stores are executed in program order, and memory values are precise following any exception.

For improved performance however, cached load operations may occur in any order, subject to memory dependencies on pending store instructions. To maintain the appearance of strong ordering, the processor detects whenever the reordering of a cached load might alter the operation of the program, backs up, and then re-executes the affected load instructions.

Specifically:

♦ *Whenever a primary data cache block is invalidated* due to an external coherency request, its index is compared with all outstanding load instructions.

♦ If there is a match and the load has been completed, the load is prevented from graduating.

♦ When it is ready to graduate, *the entire pipeline is flushed*, and the processor is restored to the state it had before the load was decoded.

Since the R10000 processor *behaves* as if it implemented strong ordering, a suitable system design allows the processor to be used to create a shared-memory multiprocessor system with strong ordering.