WIKIPEDIA

MSI protocol

In computing, the **MSI protocol** - a basic cache-coherence protocol - operates in multiprocessor systems. As with other cache coherency protocols, the letters of the protocol name identify the possible states in which a cache line can be.

Contents

- Overview
- State Machine
- Usage
- Variants
- See also
- References

Overview

In MSI, each block contained inside a cache can have one of three possible states:

- **M**odified: The block has been modified in the cache. The data in the cache is then inconsistent with the backing store (e.g. memory). A cache with a block in the "M" state has the responsibility to write the block to the backing store when it is evicted.
- **S**hared: This block is unmodified and exists in read-only state in at least one cache. The cache can evict the data without writing it to the backing store.
- **I**nvalid: This block is either not present in the current cache or has been invalidated by a bus request, and must be fetched from memory or another cache if the block is to be stored in this cache.[1]

These coherency states are maintained through communication between the caches and the backing store. The caches have different responsibilities when blocks are read or written, or when they learn of other caches issuing reads or writes for a block.

When a read request arrives at a cache for a block in the "M" or "S" states, the cache supplies the data. If the block is not in the cache (in the "I" state), it must verify that the block is not in the "M" state in any other cache. Different caching architectures handle this differently. For example, bus architectures often perform snooping, where the read request is broadcast to all of the caches. Other architectures include cache directories which have agents (directories) that know which caches last had copies of a particular cache block. If another cache has the block in the "M" state, it must write back the data to the backing store and go to the "S" or "I" states. Once any "M" line is written back, the cache obtains the block from either the backing store, or another cache with the data in the "S" state. The cache can then supply the data to the requester. After supplying the data, the cache block is in the "S" state.

When a write request arrives at a cache for a block in the "M" state, the cache modifies the data locally. If the block is in the "S" state, the cache must notify any other caches that might contain the block in the "S" state that they must evict the block. This notification may be via bus snooping or a directory, as described above. Then the data may be locally modified. If the block is in the "I" state, the cache must notify any other caches that might contain the block in the "S" or "M" states that they must evict the block. If the block is in another cache in the "M" state, that cache must either write the data to the backing store or supply it to the requesting cache. If at this point the cache does not yet have the block locally, the block is read from the backing store before being modified in the cache. After the data is modified, the cache block is in the "M" state.

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

	M	S	I
M	✗	✗	✓
S	✗	✓	✓
I	✓	✓	✓

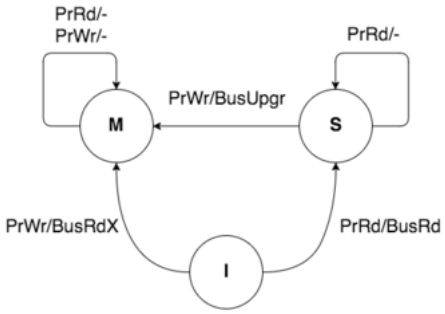
State Machine

Processor requests to the cache include:

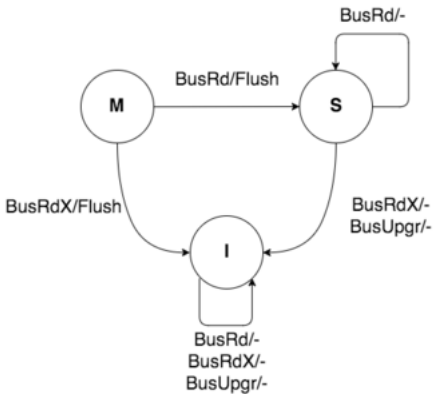
- PrRd: Processor request to read a cache block.
- PrWr: Processor request to write a cache block.

In addition, there are bus side requests. These include:

- BusRd: When a read miss occurs in a processor's cache, it sends a BusRd request on the bus and expects to receive the cache block in return.
- BusRdX: When a write miss occurs in a processor's cache, it sends a BusRdX request on the bus which returns the cache block and invalidates the block in the caches of other processors.
- BusUpgr: When there's a write hit in a processor's cache, it sends a BusUpgr request on the bus to invalidate the block in the caches of other processors.
- Flush: Request that indicates that a whole cache block is being written back to the memory.[2]



State diagram of processor requests for the MSI protocol.



State diagram of bus transactions for the MSI protocol.

State Transitions:

- **Invalid**:
  - On a PrRd, BusRd is issued and state changes to **Shared**.
  - On a PrWr, BusRdX is issued and state changes to **Modified**.
  - On a BusRd, BusRdX or a BusUpgr an invalid block remains **Invalid**.
- **Shared**:

- On a PrRd, the block remains in the **Shared** state.
- On a PrWr, BusUpgr is issued and state changes to **Modified**.
- On a BusRd, the block remains in the **Shared** state.
- On a BusRdX or BusUpgr, the block transitions to **Invalid**.

- **Modified**:
  - On a PrRd or PrWr, the block remains in the **Modified** state.
  - On a BusRd, the cache block is flushed onto the bus and state changes to **Shared**.
  - On a BusRdX, the cache block is flushed onto the bus and state changes to **Invalid**.[2]
  - A BusUpgr is not possible. Note that by being in the **Modified** state in one particular processor, a cache block has to be in the **Invalid** state in all other processor(s), as the **Modified** state is allowed in either none or only one processor. This effectively negates the possibility of a BusUpgr on the bus, which would require this block to be in the **Shared** state in one of the processor(s) that, as seen above, issues a PrWr.

Usage

This protocol is similar to the one used in the SGI 4D machine.[3]

Variants

Modern systems use variants of the MSI protocol to reduce the amount of traffic in the coherency interconnect. The MESI protocol adds an "Exclusive" state to reduce the traffic caused by writes of blocks that only exist in one cache. The MOSI protocol adds an "Owned" state to reduce the traffic caused by write-backs of blocks that are read by other caches. The MOESI protocol does both of these things.

See also

- Coherence protocol
- MESI protocol
- MOSI protocol
- MOESI protocol
- MESIF protocol

References

1. Fuchsen, R. (2010-10-01). "How to address certification for multi-core based IMA platforms: Current status and potential solutions". *Digital Avionics Systems Conference (DASC), 2010 IEEE/AIAA 29th*: 5.E.3–1-5.E.3-11. doi:10.1109/DASC.2010.5655461 (https://doi.org/10.1109%2FDASC.2010.5655461). ISBN 978-1-4244-6616-0.
2. Solihin, Yan (2016). *Fundamentals of Parallel Multicore Architecture*. Chapman & Hall/CRC Computational Science Series.
3. Suh, Taeweon (December 2006). "INTEGRATION AND EVALUATION OF CACHE COHERENCE PROTOCOLS FOR MULTIPROCESSOR SOCS" (https://smartech.gatech.edu/bitstream/handle/1853/14065/suh_taeweon_200612_phd.pdf) (PDF).