Parallel Computer Architecture
Cache Coherence in Multiprocessor Systems

Prof. Naga Kandasamy
ECE Department, Drexel University

April 15, 2019

The following material has been borrowed/adapted from: Culler, Singh, and Gupta, *Parallel Computer Architecture*, Morgan Kaufmann, 1999.

The cache coherence problem in a multiprocessor system is both pervasive and performance critical. Fig. 1 shows an example of this problem using a system with three processors whose caches are connected via a bus to shared main memory. A sequence of accesses to location u is made by the processors. First, processor P_1, reads u from main memory, bringing a copy into its local cache. Then processor P_3 reads u from main memory, bringing a copy into its cache. A little later, processor P_3 writes location u, changing its value from 5 to 7. With a write-through cache, this will cause the main memory location to be updated; however, when processor P_1, reads location u again (action 4 in the figure), it will unfortunately read the stale value 5 from its own cache instead of the correct value 7 from main memory. This is a cache coherence problem.

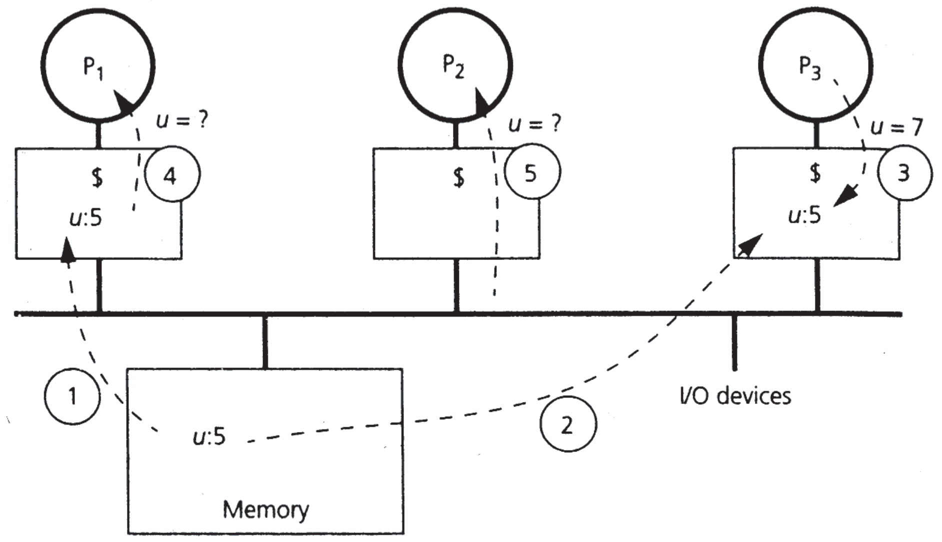

Figure 1: The cache coherence problem.

The above situation is even worse with write-back caches. P_3's write would merely set the dirty bit associated with the cache block holding location u and would not update main memory right away. Only when this cache block is subsequently replaced from P_3's cache would its contents be written back to main memory. Thus, not only will P_1 read the stale value, but when processor P_2 reads location u (action 5), it will miss in its cache and read the stale value of 5 from main memory instead of 7. Finally, if multiple processors write distinct values to location u in their respective write-back caches, the final value that will reach main memory will be determined by the order in which the cache blocks containing u are replaced and will have nothing to do with the order in which the writes to u occur.

Formal Definition of Coherence

A multiprocessor memory system is said to be *coherent* if the following properties are satisfied:

- Operations issued by any processor occur in the order in which they were issued to the memory system by that processor.
- The value returned by each read operation is the value written by the last write to that location in the serial order.

Two properties are implicit in the definition of coherence: *write propagation* means that writes become visible to other processors (or processes); *write serialization* means that an writes to a location (from the same or different processes) are seen in the same order by all processes. For example, write serialization means that if read operations by process P_1 to a location see the value produced by write $w1$ (from P_2, say) before the value produced by write $w2$ (from P_3, say), then reads by another process P_4 (or P_2 or P_3) also should not be able to see $w2$ before $w1$. Note that there is no need for an analogous concept of read serialization since the effects of reads are not visible to any process but the one issuing the read.

Cache Coherence through Bus Snooping

We now discuss several methods of ensuring cache coherence through bus snooping. This solution to cache coherence arises from the very nature of a bus. The bus is a single set of wires connecting several devices, each of which can observe every bus transaction, for example, every read or write on the shared bus. When a processor issues a request to its cache, the cache controller examines the state of the cache and takes suitable action, which may include generating bus transactions to access memory. Coherence is maintained by having all cache controllers "snoop" on the bus and monitor the transactions, as shown in Fig. 2. A snooping cache controller may take action if a bus transaction is *relevant* to it, that is, if it involves a memory block of which it has a copy in its cache. Thus, processor P_1 may take an action, such as invalidating or updating its copy of the location, if it sees the write from P_3 to the same cache block. Since the allocation and replacement of data in caches is managed at the granularity of a cache block (usually several words long) and cache misses fetch a block of data, coherence is maintained at the granularity of a cache block as well. In other words, either an entire cache block is in valid state in the cache or none of it is. Thus, a

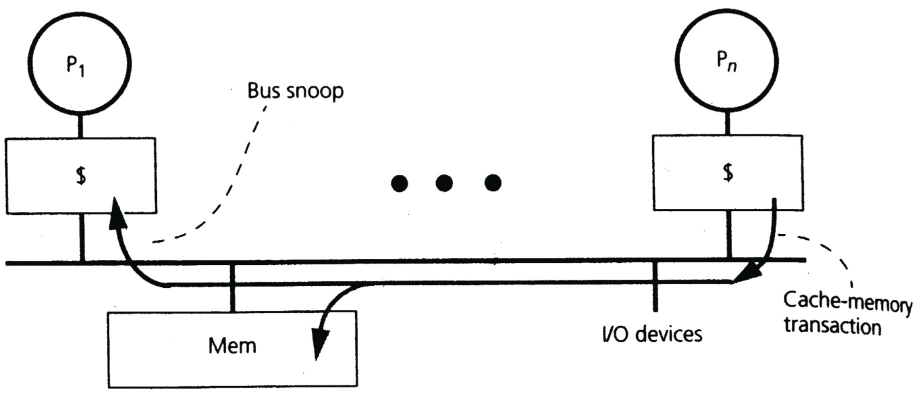

Figure 2: A snooping cache-coherent multiprocessor.

cache block is the granularity of allocation in the cache, of data transfer between caches, and of coherence.

Fig. 2 shows how a snooping protocol maintains cache coherence. Multiple processors with private caches are placed on a shared bus. Each processor's cache controller continuously snoops on the bus watching for relevant transaction and updates its state suitably to keep its local cache coherent. The gray arrows show the transaction being placed on the bus and accepted by main memory, as in a uniprocessor system. The black arrow indicates the snoop.

We will now discuss three snooping-based cache invalidation protocols. A cache-invalidate protocol is one that invalidate cached copies (other than the writer's copy) on a write.

- Two-State Invalidation Protocol for Write-Through Caches.
- Three-state Write-Back Invalidation Protocol.
- Four-State Write-Back Invalidation Protocol.

Fundamentally, a snooping protocol is a distributed algorithm represented by a collection of cooperating finite state machines. It is specified by the following basic components:

- The set of states associated with memory blocks in the local caches.
- The state transition diagram, which takes as inputs the current state and the processor request or observed bus transaction and produces as output the next state for the cache block.
- The actions associated with each state transition, which are determined in part by the set of feasible actions defined by the bus, the cache, and the processor design.

The different state machines for a block are coordinated by bus transactions.

Two-State Write-Through Invalidation Protocol

A simple invalidation-based protocol for a coherent write-through cache is described by the state transition diagram in Fig. 3. As in the uniprocessor case, each cache block has only two states: invalid (I) which indicates that the cache block is not present in the cache and valid (V) which

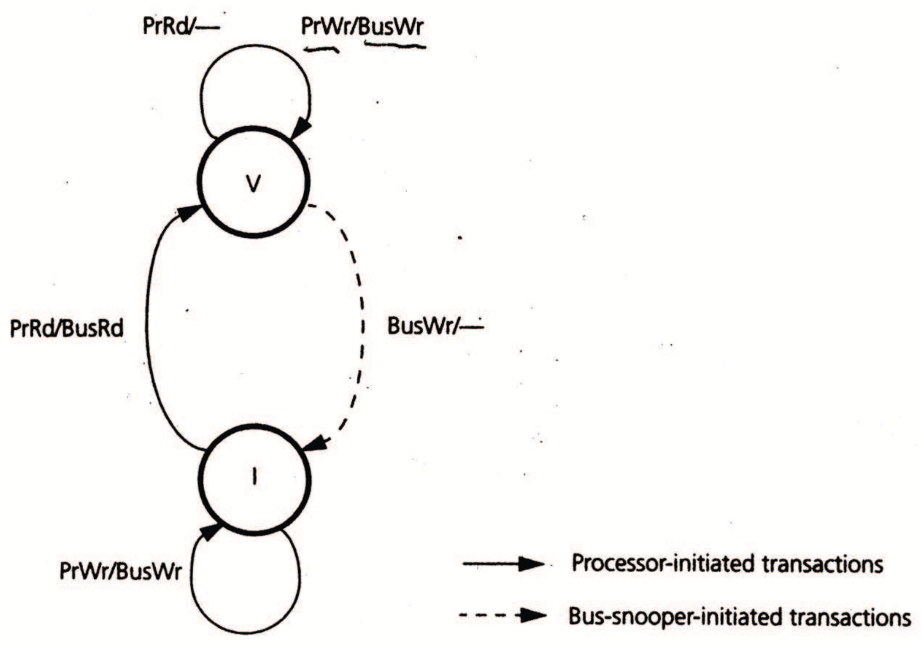
PrRd/—
PrWr/BusWr
V
PrRd/BusRd
BusWr/—
I
PrWr/BusWr
Processor-initiated transactions
Bus-snooper-initiated transactions

Figure 3: Snoopy coherence for a multiprocessor with write-through caches.

indicates that it is. The state transitions are marked with the input that causes the transition and the output that is generated with the transition. The notation A/B (e.g., PrRd/BusRd) means if A is observed, then transaction B is generated. From the processor side. the requests can be read (PrRd) or write (PrWr). From the bus side, the cache controller may observe/generate transactions bus read (BusRd) or bus write (BusWr).

When a controller sees a read from its processor miss in the cache, a BusRd transaction is generated, and upon completion of this transaction, the block transitions up to the valid state. Whenever the controller sees a processor write to a location, a bus transaction is generated that updates that location in main memory with no change of state. When the bus snooper sees a write transaction on the bus for a memory block that is cached locally, the controller sets the cache state for that block to invalid, thereby effectively discarding its copy. Fig. 3 shows this bus-induced transition with a dashed arc. By extension, if any processor generates a write for a block that is cached by any of the others, all of the others will invalidate their copies. Thus, multiple simultaneous readers of a block may coexist without generating bus transactions or invalidations, but a write will eliminate all other cached copies.

A Three-State (MSI) Write-Back Invalidation Protocol

The protocol uses the three states required for any write-back cache in order to distinguish valid blocks that are unmodified (or clean) from those that are modified (or dirty). Specifically, the states are modified (M), shared (S), and invalid (1). Invalid has the obvious meaning. Shared means the block is present in an unmodified state in this cache, main memory is up-to-date, and zero or more other caches may also have an up-to-date (shared) copy. Modified, also called dirty, means that *only* this cache has a valid copy of the block, and the copy in main memory is stale. Before a shared or invalid block can be written and placed in the modified state, all the other potential copies must be invalidated via a read-exclusive bus transaction. This transaction serves to order the write as well as cause the invalidations and hence ensure that the write becomes visible to other processors (write propagation).

The processor issues two types of requests to its local cache: reads (PrRd) and writes (PrWr). The read or write could be to a memory block that exists in the cache or to one that does not. In the latter case, a block currently in the cache will have to be replaced by the newly requested block, and if the existing block is in the modified state, its contents will have to be written back to main memory.

The bus allows the following transactions:

- Bus Read (BusRd): This transaction is generated by a PrRd that misses in the cache, and the processor expects a data response as a result. The cache controller puts the address on the bus and asks for a copy that it does not intend to modify. The memory system or possibly another cache supplies the data.
- Bus Read Exclusive (BusRdX): This transaction is generated by a PrWr to a block that is either not in the cache or is in the cache but not in the modified state. The cache controller puts the address on the bus and asks for an exclusive copy that it intends to modify. The memory system or possibly another cache supplies the data. All other caches are invalidated. Once the cache obtains the exclusive copy, the write can be performed in the cache.
- Bus Write Back (BusWB): This transaction is generated by a cache controller on a write back; the processor does not know about it and does not expect a response. The cache controller puts the address and the contents for the memory block on the bus: The main memory is updated with the latest contents.

The state transition diagram governing a block in each cache in this snooping protocol is shown in Fig. 4. Here, M, S, and I stand for modified, shared, and invalid states, respectively. The notation A/B means that if the controller observes the event A from the processor side or the bus side, then in addition to the state change, it generates the bus transaction or action B. "-" means no action is taken. Transitions due to observed bus transactions are shown in dashed arcs, while those due to local processor actions are shown in bold arcs. If multiple A/B pairs are associated with an arc, it simply means that multiple inputs can cause the same state transition. Replacements and the write backs they may cause are not shown in the diagram.

In Fig. 4, the states are organized so that the closer the state is to the top, the more tightly the block is bound to that processor. A processor read to a block that is invalid (or not present) causes a BusRd transaction to service the miss. The newly loaded block is promoted, moved up in the state diagram, from invalid to the shared state in the requesting cache, whether or not any other cache holds a copy. Any other caches with the block in the shared state observe the BusRd but take no special action, allowing main memory to respond with the data. However, if a cache has the block in the modified state (there can only be one) and it observes a BusRd transaction on the bus, then it must get involved in the transaction since the copy in main memory is stale. This cache flushes the data onto the bus, in lieu of memory, and demotes its copy of the block to the shared state. The

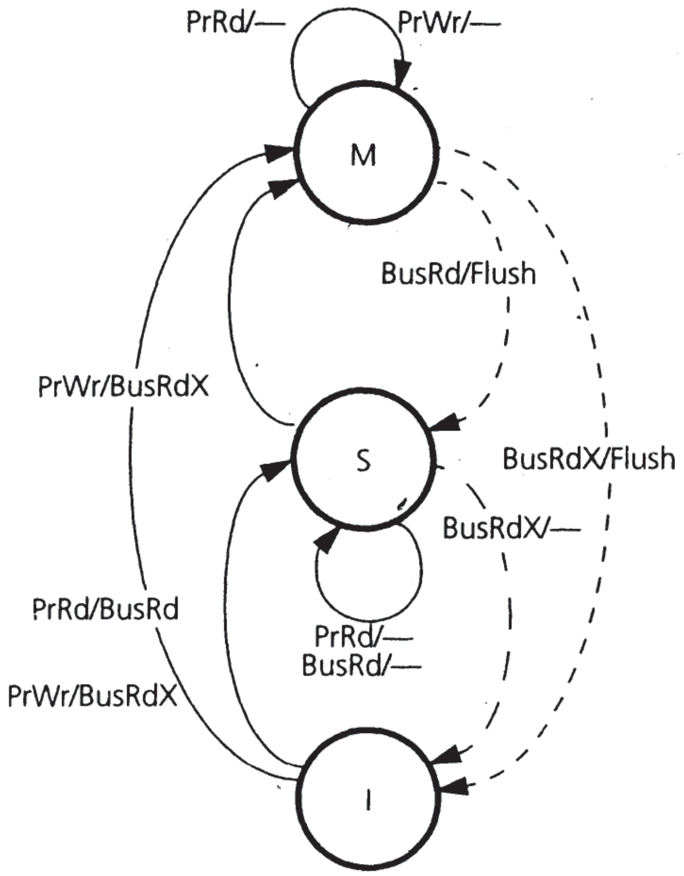

Figure 4: The basic three-state invalidation protocol.

memory and the requesting cache both pick up the block. This. can be accomplished either by a direct cache-to-cache transfer across the bus during this BusRd transaction or by signaling an error on the BusRd transaction and generating a write transaction to update memory. In the latter case, the original cache will eventually retry its request and obtain the block from memory.

Writing into an invalid block is a write miss, which is serviced by first loading the entire block and then modifying the desired bytes within it. The write miss generates a read-exclusive bus transaction, which causes all other cached copies of the block to be invalidated, thereby granting the requesting cache exclusive ownership of the block. The block of data returned by the read exclusive is promoted to the modified state, and the desired bytes are then written into it. If another cache later requests exclusive access, then in response to its BusRdX transaction this block will be invalidated (demoted to the invalid state) after flushing the exclusive copy to the bus.

Writing into a shared block is treated essentially like a write miss, using a read-exclusive bus transaction to acquire exclusive ownership. The data that comes back in the read exclusive can be ignored in this case, unlike when writing to an invalid or not present block, since it is already in the local cache. In fact, a common optimization to reduce data traffic in bus protocols is to introduce a new transaction, called a bus upgrade or BusUpgr, for this situation. A BusUpgr obtains exclusive ownership of the block just like BusRdX by causing other copies to be invalidated but does not cause main memory to respond with data for the block. After the BusUprg/BusRdX transaction is issued on the bus, the block in the requesting cache transitions to the modified state. Additional writes to the block while it is in the modified state generate no additional bus transactions.

Finally, a replacement of a block from a cache logically demotes the block to invalid (or not present) by removing it from the cache. A replacement therefore causes the state machines for two blocks to change states in that cache: the one being replaced changes from its current state to invalid, and the one being brought in changes from invalid to its new state. Replacements are not shown in the state diagram for Simplicity.

The following table shows the state transitions and bus transactions affected by the three-state write-through invalidation protocol for the scenario shown earlier in Fig. 1.

Processor action	State in P_1	State in P_2	State in P_3	Bus action	Data supplied by
1. P_1 reads u	S	-	-	BusRd	Memory
2. P_3 reads u	S	-	S	BusRd	Memory
3. P_3 writes u	I	-	M	BusRdX	Memory
4. P_1 reads u	S	-	S	BusRd	P_3 cache
5. P_2 reads u	S	S	S	BusRd	Memory

A Four-State (MESI) Write-Back Invalidation Protocol

A concern arises with the MSI protocol if we consider a sequential application running on a multiprocessor. 5uch multiprogrammed use in fact constitutes the most common workload on small-scale multiprocessors. When the process reads in and modifies a data item, in the MSI protocol two bus transactions are generated even though there are never any sharers. The first is a BusRd that gets the memory block in the S state, and the second is a BusRdX (or BusUpgr) that converts the block from S to M state. By adding a state that indicates that the block is the only (exclusive) copy but is not modified and by loading the block in this state, we can save the latter transaction since the state indicates that no other processor is caching the block. This new state, called "exclusive" or the E state indicates an intermediate level of binding between shared and modified. It is exclusive, so unlike the shared state, the cache can perform a write and move to the modified state without further bus transactions; but it does not imply ownership (memory has a valid copy), so unlike the modified state, the cache need not reply upon observing a request for the block.

The MESI protocol consists of four states: modified (M), exclusive (E), shared (S), and invalid (I). M and I have the same semantics used in the MSI protocol. We introduce a new state E, the exclusive state, meaning that only one cache (this cache) has a copy of the block and it has not been modified (i.e., the main memory is up-to-date). State S means that potentially two or more processors have this bock in their respective caches in an unmodified state. The bus transactions and actions needed are very similar to those for the MSI protocol, and are shown in Fig. 5.

When the block is first read by a processor, if a valid copy exists in another cache, then it enters the processor's cache in the S state, as usual. However, if no other cache has a copy at the time (for example, in a sequential program) it enters the cache in the E state. When that block is written by the same processor, it can directly transition from E to M state without generating another bus

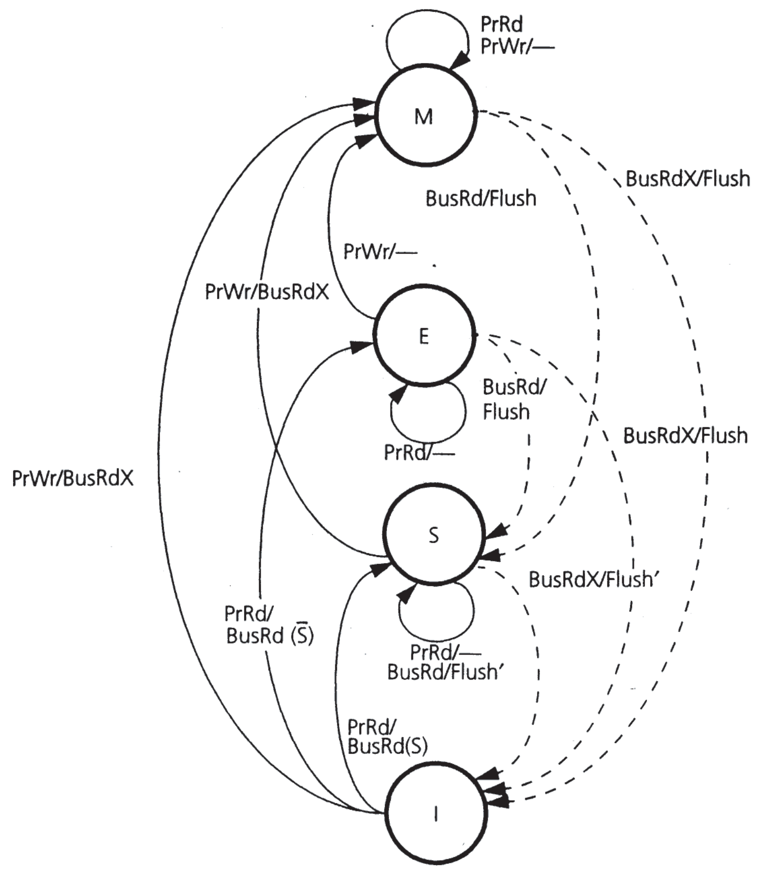

Figure 5: State transition diagram for the MESI protocol.

transaction since no other cache has a copy. If another cache had obtained a copy in the meantime, the state of the block would have been demoted from E to S by the snooping protocol.

The MESI protocol requires an additional signal, called the shared signal (S) which must be available to the controllers in order to determine on a BusRd if any other cache currently holds the data. During the address phase of the bus transaction, all caches determine if they contain the requested block and, if so, assert the shared signal. This signal is a wired-OR line, so the controller making the request can observe whether any other processors are caching the referenced memory block and can thereby decide whether to load a requested block in the E state or the S state.

In Fig. 5, the notation BusRd(S) means that the bus readd transaction caused the shared signal to be asserted; BusRd($\bar{S}$) means unasserted. A plain BusRd means that we don't care about the value of S for that transition. A write to a block in any state will promote the block to the M state, but if it was in the E state, then no bus transaction is required. Observing a BusRd will demote a block from E to S since now another cached copy exists. As usual, observing a BusRd will demote a block from M to S state and will also cause the block to be flushed onto the bus; here too, the block may be picked up by both the requesting cache and by main memory. Also, note that it is possible for a block to be in the S state even if no other copies exist since copies may be replaced (from S to I) without notifying other caches.