Snooping coherence protocols (cont.)

A four-state update protocol

[§5.3.3] When there is a high degree of sharing, invalidation-based protocols perform poorly. Blocks are often invalidated, and then have to be re-fetched from memory.

Wouldn’t it be better to send new values out rather than invalidation signals? This is the motivation behind update-based protocols.

We will look at the “Dragon” protocol, initially proposed for Xerox’s Dragon multiprocessor, and more recently used in sun SparcServer multiprocessors.

This is a four-state protocol, with two of the states identical to those in the four-state invalidation protocol:

• The E (exclusive) state indicates that a block is in use by a single processor, but has not been modified.
• The M (modified) state indicates that a block is present in only this cache, and main memory is not up to date.

There are also two new states.

• The Sc (shared-clean) state indicates that potentially two or more caches hold this block, and main memory may or may not be up to date.
• The Sm (shared-modified) state indicates that potentially two or more caches hold this block, main memory is not up to date, and it is this cache’s responsibility to update main memory when the block is purged (i.e., ____________ ____________).

A block can be in Sm state in only one cache at a time.

However, while a block is in Sm state in one cache, it can be in Sc state in others.
It is possible for a block to be in Sc state in some caches without being in Sm state in any cache. In this case, main memory is up to date.

Why is there no I (invalid) state?

Here is a state-transition diagram for this protocol.

In diagrams for previous protocols, if a block not in the cache was referenced, we showed the transition as coming out of the I (invalid) state.

In this protocol, we don’t have an invalid state. So, looking at the diagram above, can you see what is supposed to happen when a referenced block is not in the cache?

What happens if there is a read-miss and—
- the shared line is asserted?
- the shared line is not asserted?

What happens if there is a write-miss and—
- the shared line is asserted?
- the shared line is not asserted?

If there’s a write-miss and the shared line is asserted, what else happens?

Why is only a single word broadcast?

♦ Let us first consider the transitions out of the Exclusive state.
  - What happens if this processor reads a word?
  - What happens if this processor writes a word?

There is one more transition out of this state. What causes it, and what happens?

♦ Now let us consider the transitions out of the Shared-Clean state.
  - What happens if this processor reads a word?
  - What happens if this processor writes a word?
There is one more transition out of this state. What causes it, and what happens?

♦ Next, let’s look at the transitions out of the Shared-Modified state.
  • What happens if this processor reads a word?
  • What happens if this processor writes a word?

How many more transitions are there out of this state?
What causes the first one, and what happens?

What causes the second one, and what happens?

♦ Finally, let’s look at the transitions out of the Modified state.
  • What happens if this processor reads a word?
  • What happens if this processor writes a word?
  • What happens if another processor reads a word?

Let’s go through the same example as we did for the 3-state invalidation protocol.
A three-state update protocol

Whenever a bus update is generated, suppose that main memory—as well as the caches—updates its contents.

Then which state don’t we need?

What’s the advantage, then, of having the fourth state?

The Firefly protocol, named after a multiprocessor workstation developed by DEC, is an example of such a protocol.
Here is a state diagram for the Firefly protocol:

![State Diagram](image)

**Key:**
- CRM — CPU read miss
- CWM — CPU write miss
- CWH — CPU write hit
- BR — bus read
- BW — bus write

A “!” following a transition means SharedLine was asserted. An “x” means it was not.

- Processor-induced transitions
- Bus-induced transitions

Read hits do not cause state transitions and are not shown.

What do you think the states are, and how do they correspond to the states in

- 
- 
- 

The scheme works as follows:

- On a read hit, the data is returned immediately to the processor, and no caches change state.

- On a read miss,

  - If another cache (other caches) had a copy of the block, it supplies (one supplies) it directly to the requesting cache and raises the SharedLine. The bus timing is fixed so all caches respond in the same cycle.

All caches, including the requestor, set the state to *shared*.

If the owning cache had the block in state *dirty*, the block is written to main memory at the same time.
• If no other cache had a copy of the block, it is read from main memory and assigned state \textit{valid-exclusive}.

• On a write hit,
  • If the block is already \textit{dirty}, the write proceeds to the cache without delay.
  • If the block is \textit{valid-exclusive}, the write proceeds without delay and the state is changed to \textit{dirty}.
  • If the block is in state \textit{shared}, the write is delayed until the bus is acquired and a write-word to main memory initiated.

Other caches pick the data off the bus and update their copies (if any). They also raise the SharedLine. The writing cache can determine whether the block is still being shared by testing this line.

  ◊ If the SharedLine is not asserted, no other cache has a copy of the block. The requesting cache changes to state \textit{valid-exclusive}.

  ◊ If the SharedLine is asserted, the block remains in state \textit{shared}.

• On a write miss,
  • If any other caches have a copy of the block, they supply it. By inspecting the SharedLine, the requesting processor determines that the block has been supplied by another cache, and sets its state to \textit{shared}.

  The block is also written to memory, and other caches pick the data off the bus and update their copies (if any).

  ◊ If no other cache has a copy of the block, the block is loaded from memory in state \textit{dirty}.

In update protocols in general, since all writes appear on the bus, write serialization, write-completion detection, and write atomicity are simple.