ECE511 Computer Architecture Nov 17, 2004

Lecture 22

Lecturer: Matt Frank *Scribe: Meeta Sharma Gupta*

1 Modified Exclusive Shared Invalid Consistency Protocol

This protocol provides an optimization on the cache coherence protocol that was discussed in the previous lecture. The optimization is provided by the introduction of an additional state, *Exclusive* state. This state represents *only one* reader of any particular cache line. The optimization provided here is in terms of the write-backs onto the bus. The transition from the Exclusive state to the Modified state can be achieved without going over the bus. This optimizes for the case where the caches are working independently and hence, ideally no data is being shared, and hence, there is no requirement of a write-back onto the bus unlike the case in the previous (MSI) protocol.

Figure 1 and 2 show the state machine for the the CPU and Bus transactions respectively. Just like the earlier protocol, all the transactions that go out on the bus are being snooped, and each cache checks whether there is a hit/miss for the address on the bus in it's cache. An action according to the protocol is taken if there is a hit.

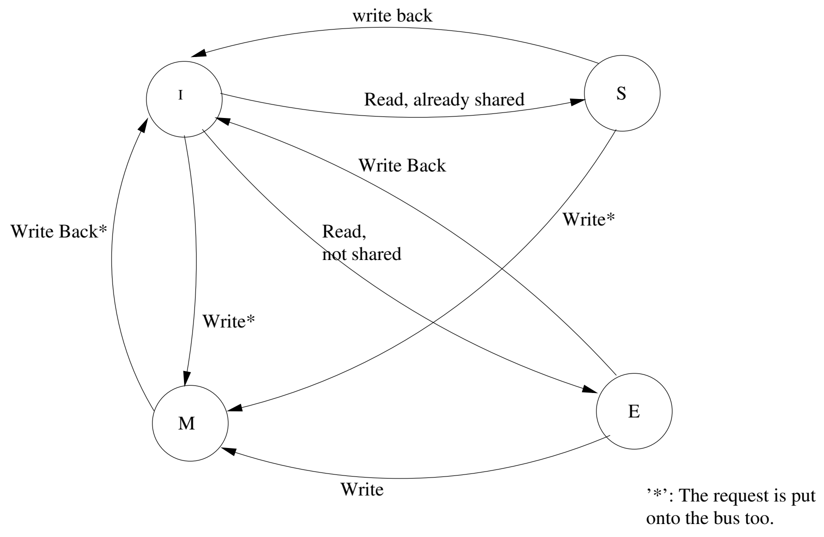

Figure 1: MESI Protocol: CPU Transactions

The memory consistency protocols discussed have worked due to one centralized infrastructure, which is the bus. It is due to this centralizing unit that all the memory transactions are guaranteed to be in-order. Let's consider a scenario,

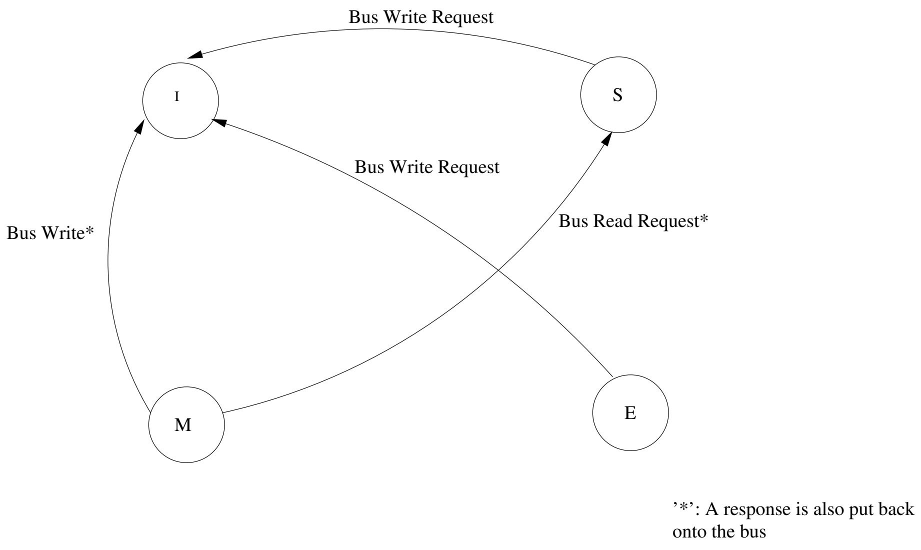

Figure 2: MESI Protocol: Bus Transactions

where we don't have the bus, but instead have an *Internet bubble*, as shown in Figure 3.

2 Out of Order Memory Accesses

Suppose, P1 write *a=1*, it does not have the line containing address *a*, and hence, makes a request onto the Internet. Meanwhile, P1 starts a read for *b* and the data value comes back before the request for *a*. P1 then puts itself into state 'shared'.

In the earlier bus scenario, the guarantee consistency was guaranteed due to the centralizing bus and hence had all the operation in-order of memory transactions.

We would like to do memory request out of order, and take appropriate actions only when there is a memory consistency in question. In our earlier assumptions, we had assumed in-order memory executions, and hence, no store buffers, load queues etc. We would like to relax this assumption and take store buffers, load queues into our model.

Taking the simple example discussed in the previous lectures, of a critical section implementation[1]. Suppose P1 has to do a store on $\mathtt{a} \longrightarrow 1$. Since $\mathtt{loadb}$ is not dependent on $\mathtt{storea}$, it can go ahead and suppose the value for $\mathtt{b}$ hits in the cache. This would lead a shared state for $\mathtt{b}$ with value 0.

Let us assume that on the second processor,P2, the same process happens, it does a $\mathtt{loada} \longrightarrow 0$ before it does a $\mathtt{storeb} \longrightarrow 1$. This would lead to both the processors going into a the critical region speculatively.

Eventually, the stores would enter the pipeline. Suppose P1 makes a write request for $\mathtt{a}$ on the bus. P2 would see the request and change the state for variable $\mathtt{a}$ from *shared* to *invalid*. It would also peek into its load queue and turn the $\mathtt{loada}$

[1]The pseudo-code has been omitted here

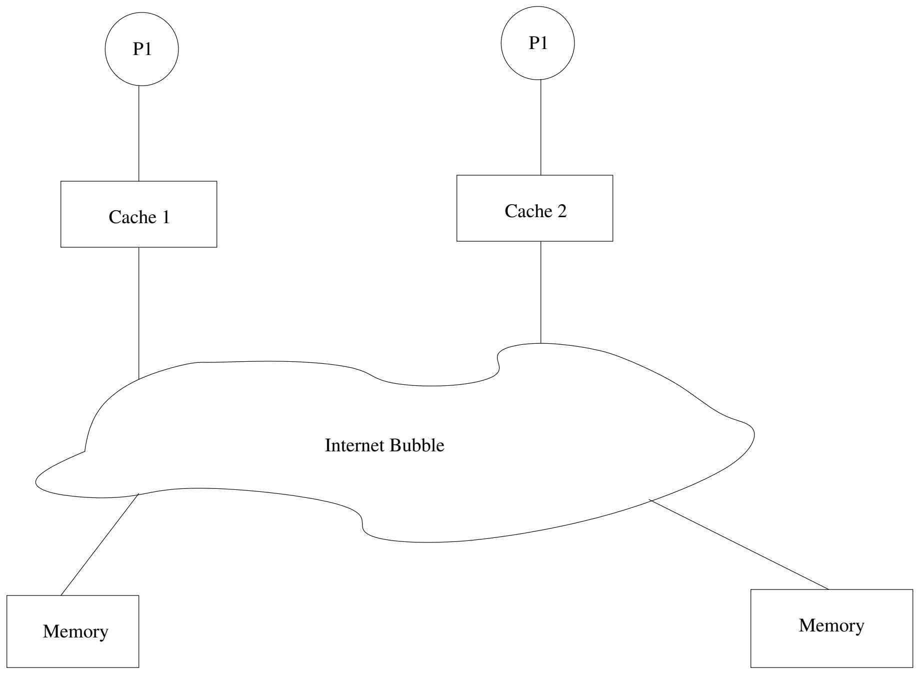

Figure 3: Internet like Interconnect

into a branch mispredict.

3 Directory Based Protocol

In this protocol, the caches communicate over some interconnect, not the centralized bus (As shown in Figure 3). Here, the memory needs to decide which caches the replies for a particular request needs to be sent to. We want to maintain the same invariant, that either no cache has a line (Invalid State), or 1 has it either in *Exclusive* or *Modified* state and multiple have the line in *Shared* state.

The different state transitions are shown in Figure 4. The Memory bank needs to send appropriate messages when a write request for a line comes to it. It needs to take care that the line should not be in modified state in one cache and shared in another. When the memory gets a request for write, it looks up its counter bits and finds the appropriate caches having the line in *shared* state. These caches need to be aware of the write request made on that line. So the memory sends an *invalidation* message to the caches having the line in the *shared* state. It then waits for an acknowledgment from the caches of the invalidation of that particular line. Once it has received acknowledgments from all the caches it would finally give a write permission to the cache that made the write request.

So, here the centralized unit is the memory controller. Likewise, if the state is in *modified* state and a read request is made to that line, the requesting cache needs to wait till the cache having the line in *modified* state sends the data back to the memory and changes to state *shared*. This would ensure that the memory sends

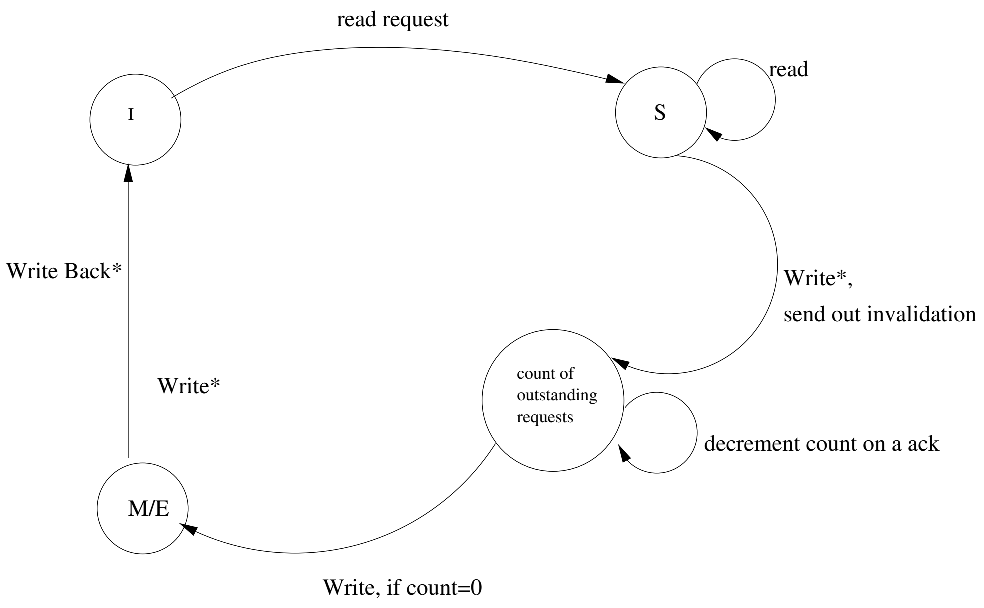

Figure 4: Directory Based Protocol

the consistent data back to the requesting cache. Now, both the caches would be in a *shared* state for that particular line.