Comparch II - Part 3 Notes (Cache Coherence)

Ethan Graham

June 21st 2023

Introduction

Shared Memory Multiprocessors

In this model, we have several processors on the same bus. A problem arises when multiple of the processors want to access main memory. We need some kind of arbiter to decide who is *master* at a given time.

This shared memory bus becomes a bottleneck.

This is called ***UMA*** (uniform memory access)

Distributed Memory Multiprocessors

In this model, every processor is a complete system with cache, I/O and memory. Everytime that processors want to exchange data, they go through a slow interconnection network.

We can pretend that everyone is using the exact same memory. - "if I want to load something from my own memory, just do it" - "If I can't do that, I request it from the OS who deals with getting the memory associated with that page."

This second case takes a lot longer than just accessing from a processor's own memory. This is refered to as ***NUMA*** (non-uniform memory access)

Distributed memory multiprocessors are much more cost effective and scalable than shared memory multiprocessors.

Intel Nehalem Example

The `L3` cache of this processor is logically shared (*same addressing space enforced by virtual memory system*) for all, but physically distributed across all four cores.

Cache Coherence Problem Example

Assume that processor P_1 and processor P_2 perform the following line of code at the same time:

```
ldw $r2, x
```

Following this, P_1 executes the following:

```
addi $r1, $r2, $r4
stw x, $r1
```

The problem now is that P_2 has a stale version of the data at address `x`.

"Shared memory model - Is this really a new problem?"

Not quite. Recall the DMA (*direct memory access*). It would handle lengthy memory transfers between peripherals and system (a very monotonous task) while the memory continues executing instructions. It is set up with initial parameters such as `src` and `dest`. It takes over the memory bus except for when the CPU needs it.

Coherent Memory System Elements

1. Preservation of Program Order

If P writes to address `x`, and then reads from address `x` ,the returned value is the value that was previously written(*assuming that no one has written there in the meantime*).

2. Coherent View

If P_1 writes to `x` and after a sufficiently long time interval P_2 reads from `x`, then the value read by P_2 is the value written by P_1 (*assuming that one has written there in the meantime*).

3. Write Serialization

If P_1 writes to `x` and P_2 reads from `x`, all the processors in the system see the writes in the same order.

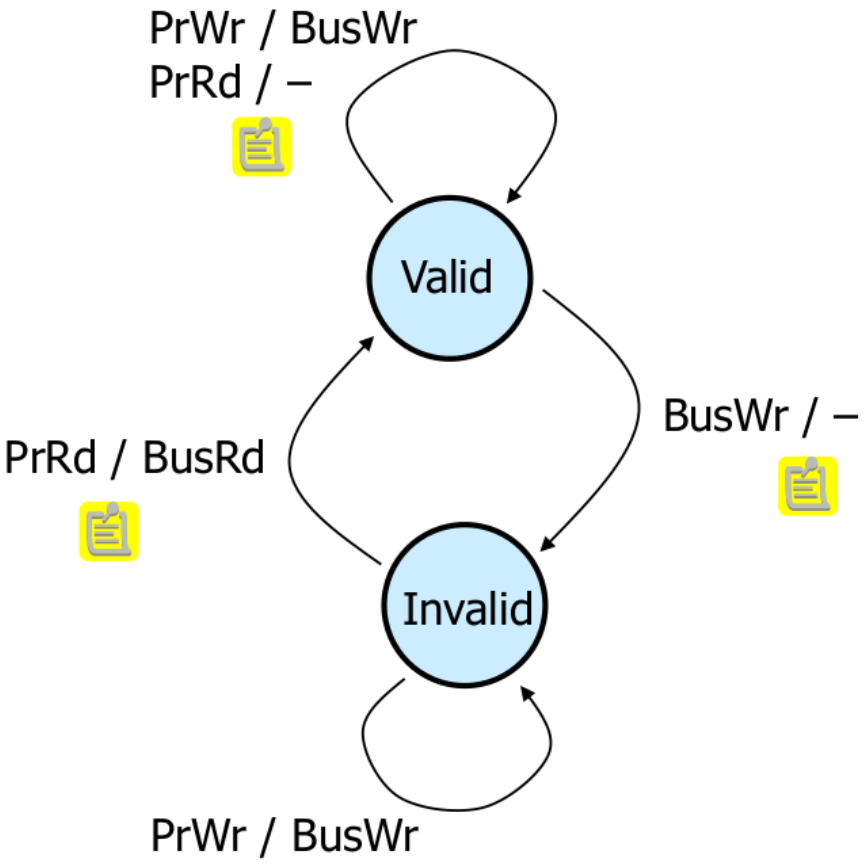
Snoopy Cache-Coherence Protocols

The general idea is that each cache ***snoops*** on all bus transactions, looking for transactions relevent to one of its lines, and then take appropriate action to preserve coherence.

- Invalidate

- Update
- Supply Value

Simple Invalidate Snooping Protocol (2-State)



In this state machine, every cache machine is in one of two possible states: **valid** or **invalid**. We define in the state diagram the actions that cause state changes, and the ones that trigger reactions from the processor.

This protocol assumes a write-through, no-write-allocate cache.

Valid State

If I (*the cache line*) contain valid data and I snoop a `PrRd` request for me (*a read from the processor*) then I do nothing - I remain valid and I return the result. (this `PrRd` is associated to a read from the processor that this cache line is associated with)

If I contain valid data and get a `PrWr` write request, then I will remain valid (because I will have the new data) and I write to memory with a `BusWr` (because this is a write-through cache)

Data of course becomes valid when we have to evict (*no more space in cache*) but this is implicit in this diagram.

The `BusWr/-` from the diagram represents the following: when I see that someone is writing to this data address (the one in this cache line), then I will know that the data contained in it is stale. Because this cache is no-write-allocate, the cache will not be overwritten, and thus we have to invalidate the data (*we move to the invalid state in the diagram*).

Invalid State

If I get a `PrWr` from the processor that I (*the cache line*) am associated to, then I will not write-over the data that I have. I will simply write through to main memory with a `BusWr` (since I am a write-no-allocate, write-through cache)

If I snoop a `BusWr`, I do nothing in this model.

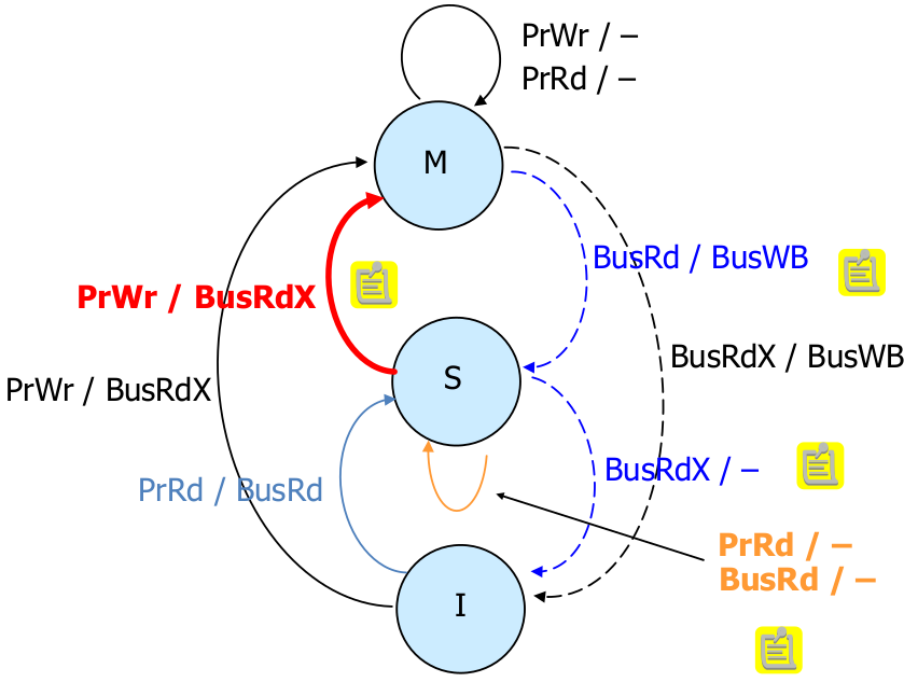
If I get a `PrRd` from the processor that I am associated to, then I perform a `BusRd` to get the valid data, and move to the valid state.

MSI Snooping Protocol (3-State)

This stands for ***Modified, Shared, Invalid***

We keep the same processor actions `PrRd` and `PrWr`, and define the following bus actions:

- `BusRd`: bus read ***without*** intention to modify. This could come from memory or from another cache.
- `BusRdX`: bus read ***with*** intention to modify. In this case, we must invalidate all other copies of the data.
- `BusWB`: the cache controller puts the content on the bus and memory is updated.



Modified State (*we have the freshest data*)

I stay in this state as long as only the processor is sending reads and writes. In this case all I do it return data for a `PrRd` and modify my data for a `PrWr`.

When I get snoop a `BusRd`, I will share my copy to the bus with a `BusWB` (because I have the most recent version, this is why I am in the *modified* state) and move to the ***shared*** state since now other caches have a copy of my data.

When I snoop a `BusRdX`, when I write to bus with a `BusWB` like before. Now, since I know that the data will be modified, I have to move to the ***invalid*** state.

Shared State (*the data we have is shared with other caches*)

Like in the *modified* state, if I snoop a `PrRd` I will do nothing but return the data that I have. This time, I remain in the same state if I get a `BusRd` since I am already aware that the data is shared with other caches.

When I snoop a `BusRdX`, I again move to the ***invalid*** state since I know that the data that I have will be modified.

When the processor that I am associated with writes over the data with a `PrWr`, I send a `BusRdX` so that other caches invalidate their data. I move to the ***modified*** state because I have the modified my data. Note that the *read* action in the

`BusRdX` here is useless. It's just the action that we have that will allow us to tell other caches to invalidate their data.

Invalid State (*the data we have has been modified by someone else*)

When I get a `PrRd`, i.e. my processor wants the data associated with this cache line, I perform a `BusRd` so that I can get access to a fresh version of the data and I move to the ***shared*** since I have the same data as is contained in other caches.

When I get a `PrWr`, i.e. my processor wants to modify the data, I perform a `BusRdX` and move to the ***modified*** state since I will have the freshest version of the data.

MESI Snooping Protocol (4-state)

Often called the *Illinois* protocol. We won't make a diagram for it. It's good to know what states it contains however.

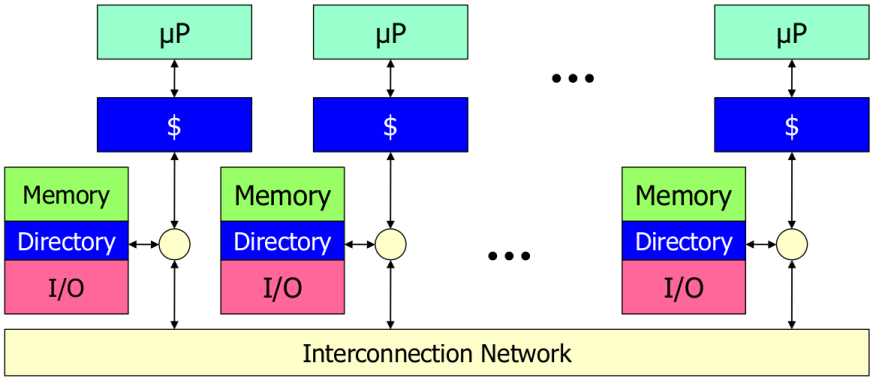
- **Modified** (dirty data)
- **Exclusive** (clean unshared). Only copy of the data, not dirty.
- **Shared**
- **Invalid**

Directory-Based Cache-Coherence Protocols

In this model, every processor represents a full system complete with a cache, a main memory and an `I/O`. We call them *nodes*. Every node contains a directory that keeps track of the state of every cache block (*invalid, valid, etc...*). All nodes communicate over an ***interconnection network***.

This model is difficult to implement in practice, especially if there are many nodes.

- Typically needed in Distributed Memory Architectures



Here μP is a microprocessor, $ is a cache.

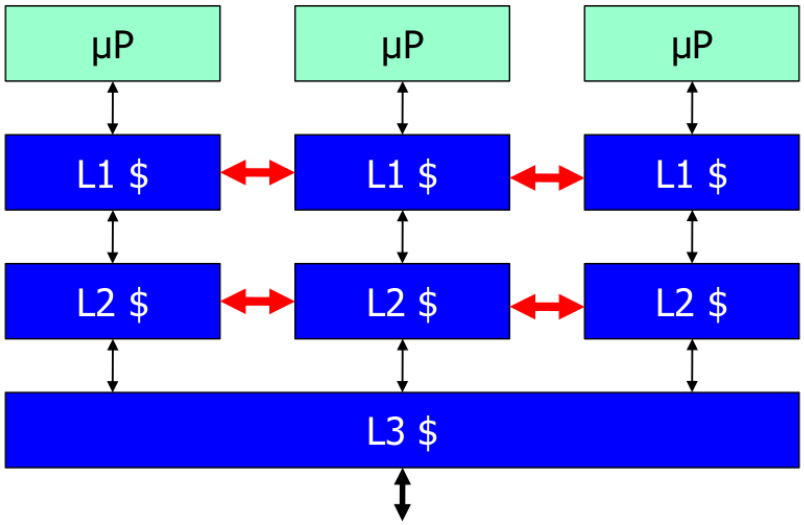
Snoopy Protocols vs. Directory-Based Protocols

Snoopy protocols are distribued coherence protocols at the ***cache-level***. They have scalability issues, and may have unneccessary but unavavoidable coherence traffic. They are however very fast.

Directory-based protocols are centralized protocols at the ***memory-level***. They are scalable, only have as much coherence traffic as necessary. They are however susceptible to latency issues due to centralizaton, and have granularity issues caused by these aforementioned latency issues.

Multilevel Caches

In this model some caches are private and some are shared.



In this diagram, the micro processors μP have private L_1 and L_2 caches, but share a L_3 cache that communicates with main memory.

The only way for a cache to snoop in a system like this is through messages to the cache on the level below it.

Inclusion Property Between Caches at Levels $n-1$ and n

We assert the following:

- The content of the L_{n-1} cache is ***always*** a subset of its corresponding L_n cache. A bus transaction at L_{n-1} is also relevant for L_n, and the snoop at L_n is sufficient.
- If a cache line in L_{n-1} is marked as owned/modified, then it should also be marked as such on L_n.

These inclusion properties aren't naturally maintained. For example if the cache at L_2 decides to evict a line that is potentially also in L_1, then something special must be done to not violate the inclusion principle. (otherwise L_1 will no longer be a subset of L_2).

Essentially, we need to propagate invalidations up the hierarchy to to keep L_1 updated with what is happening at L_2

Deciding what is easier between snooping and maintaining inclusion is outside of the scope of this course.

Note: The intel core series usese a snooping protocol between caches related associated with different **cores** (micro-processors), whereas the intel nehalem architecture uses an inclusion.

Coherence or Consistency

Informally we can define these two terms as follows...

Coherence

What values will be returned by a set of read and write operations to a single address? The issue is data integrity - if I violate coherence, then I will get the wrong data.

Consistency

When will the values be returned on reads to multiple addresses? This is no longer an issue of data integrity - the data will be correct from the perspective of a single processor, but perspective from different processors could have different orderings and contradict the expectations of the programmer.

Example: Consistency

Imagine that P_1 runs:

```
a = 0
...
a = 1
assert(b == 0) ...
```

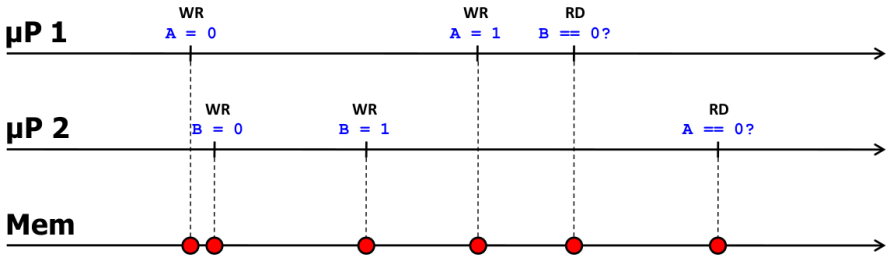
and P_2 runs concurrently:

```
b = 0
...
b = 1
assert(a == 0) ...
```

Can both of these assertions be true? Not really. Here, the problem is ***when*** the values become visible to the other thread.

Strict Consistency

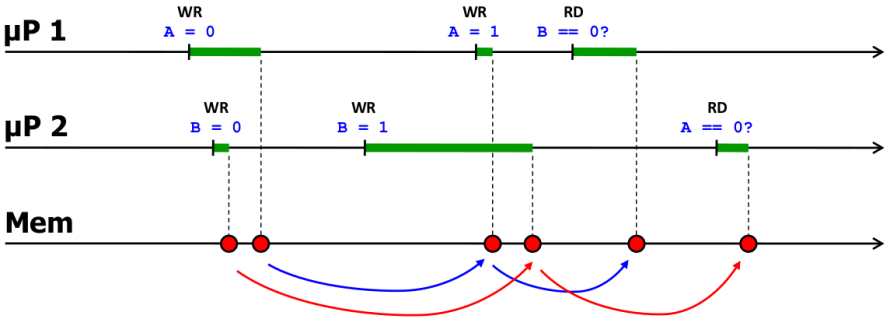
In an ideal world, everything happens in memory in the exact order in which it was issues. In any distributed system this is practically impossible.



Sequential Consistency

This is much more practical. The result of any execution is as if

- The operations of each individual processor were executed in the order specified by the program
- The operations of the different processors were arbitrarily interleaved



Processors load and store in program order. Access to memory is atomic (*no other operation is started until the previous is finished*). We choose the next core to be able to access memory at random.

Cache Coherence Examples Notes

I'll summarize briefly what the Prof. said during the cache coherence example about the ***MESI*** protocol

Firstly, we are not expected to understand the MESI protocol à priori. It suffices to understand how the state diagrams work and we can figure the rest out on the spot, applying what we are meant to know about the ***MSI*** state machine.

Regarding Evictions

Earlier in the document we talked about how eviction is implicit in the state diagram. This is because when a cache line is evicted, it is replaced by a new state machine representing a new cache line. ***No signal is sent out to other caches***, I just cease to exist.

What this implies, notably, is that if I am in the **shared** state of the MESI protocol (*implying that another cache has the same data as I do*) and that data is evicted from other caches, then I should be in the **exclusive** state. This does not happen in practice however since no signal is sent out as I mentioned.

What's the Point of these Protocols?

The idea behind these protocols is to minimize the amount of traffic that we have to send on the bus. If you think about these snoopying protocols, every cache line is sort of self-regulating. We communicate only when necessary with the other processors.

It's important to note that these state machines don't just apply to cache lines, but to cache ways as well. The concepts can be generalized.

What should I do when I see a new diagram during the exam?

Ask a bunch of questions. Study the transitions thoroughly and understand the meaning of the transitions before moving forward. The prof suggests even thinking of potential improvements as this requires a good understanding of the protocol.

what does an invalid state represent concretely?

Even if in the diagrams/exercises we normally have something along the lines of `0x1000 - Invalid`, we should disregard this address completely. It's as if there was nothing in memory associated with this address.

How do I answer the questions during the exam?

There are two primary ways to go about doing this:

- Have a good understanding of the protocol, and reason about it
- Follow the diagram exactly. "I moved from an exclusive state to a shared state. What signals (in the diagram) could have caused this? What states could these signals have come from?"

Often, it is a good idea to do a bit of both and increase confidence in the answer that we arrive at. If they don't correspond then we likely did something wrong.

Don't forget that the questions may try to throw you off with impossible transitions. For example, a cache line moves from **shared** to **exclusive**, which can't happen in practice.

Question: "when is cache-to-cache communication beneficial?"

It could potentially be beneficial when another cache has a piece of data that we are requesting. In the ***MESI*** protocol, when another cache has a piece of data, it simply answers saying *yeah I have it* but doesn't actually send the data. We could save ≈ 100 cycles that would be spent going to main memory if we just asked the neighboring caches.

I emphasize that we don't have to think about the logistics of implementing this in hardware. It suffies to give this example.