Lecture 24: Board Notes: Cache Coherency

Part A: What makes a memory system coherent?

Generally, 3 qualities that must be preserved… **(SUGGESTIONS?)**

(1) Preserve program order:
- A read of A by P_1 will reference the value written by the most recent write to A (i.e. by P_1)
- Thus, in the absence of sharing, each processor behaves as a uni-processor would

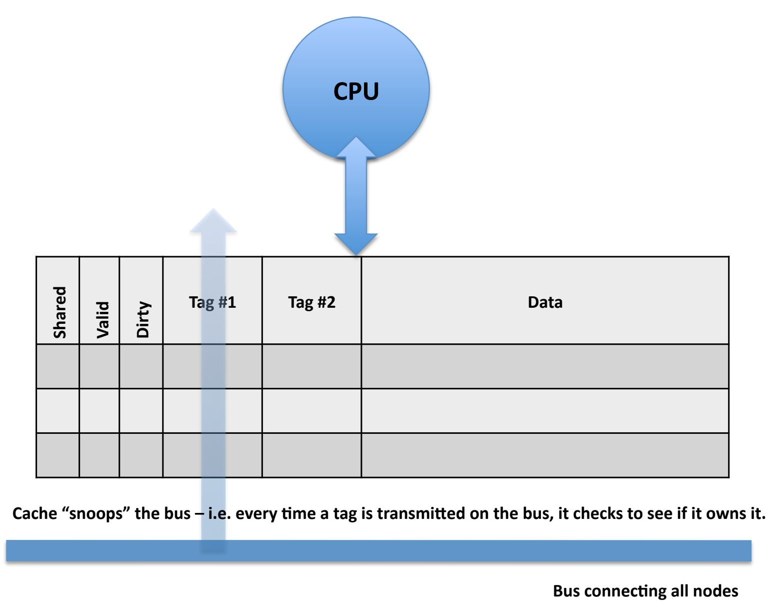
(2) All writes must be seen by all processors:
- If P_1 writes to A, and P_2 reads A after a certain amount of time, and there is no other write to A in between, P_2 reads the value written by P_1.
- Thus, P_2 must eventually see the new value…

(3) Causality must be preserved:
- Writes to the same location are serialized
  - i.e. 2 writes to the same location A are seen in the same order by all processors
- Example:
  - A =0
  - P_1 increments A
  - P_2 waits until A = 1
  - P_2 increments A
  - P_3 sees A = 2
- In other words, different processors should not see these writes in different orders
  - i.e. P_3 should not see the write by P_2 first and then the write by P_1

Hardware must provide this behavior + we would still like to have benefits of caches, etc.

Part B: Snooping

Consider a $, on one node of a multiprocessor (i.e. multi-core chip) with a re-designed block:



- All bus activity must be compared to cache entries
    - i.e. if Node 1 sends out a message saying it just wrote to a block with Tag XYZ, if Node 2 has a valid cached copy of a block with Tag XYZ, then some action will need to be taken
- Why 2 sets of tags?
    - Can use 1 said to do lookups for normal reads and others to do "snoop" checks

MOVE ON TO PART C…

Part C: Snooping – Update vs. Invalidate protocols

When listening on the bus, what to we do if there is a cached copy and a "write" by another node is broadcast?

Answer:
Generally follow 1 of 2 protocols: UPDATE or INVALIDATE

What event?	Update protocol	Invalidate protocol
A burst of writes from 1 processor to 1 address	Each write updates all cached copies (preserves property 2 in Part A)	All cached copies are no longer valid on 1st write; next readgets new copy (preserves property 2 in Part A)
Writes to different words in the same cache block See picture with bus	Update sent for EACH word	No need for subsequent invalidates; first write invalidates other block copies; might still broadcast address depending on coherency protocol
Producer-consumer latency	Producer sends update; consumer reads new value in cache	Producer invalidates consumer's copy; consumer will experience a read miss and must request a new block When writing parallel code, this can degrade performance!

Regarding producer-consumer latency:
- The invalidate protocol ensures that Property 3 above is preserved as writes are ordered by bus invalidates
    - Usually wins…
- The update protocol ensures that Property 3 above is preserved as all nodes see writes in the order in which they obtain access to the bus
    - Means LOTS of bus traffic!

Part D: MSI Cache Coherency Protocol

How do we actually implement snooping?

Can support a protocol called MSI → letters refer to a state the cache block could be in…
- Invalid State:
    - Block B is not in cache C
- Modified State:

    - Block B is in cache C and is dirty
    - Consequences:
        - When this block is kicked out, main memory must be updated
        - We can read or write a block without bus traffic
        - There is no other cached copy of this block
- Shared State:
    - Block B is in multiple caches (C_n's)
    - Consequences and Insight:
        - Multiple copies are being read simultaneously
        - Must send request to "upgrade" to M state before a write

Consider the following state transitions → also, **DRAW PICTURE ON BOARD**:

	State Transition	**Local Request or Bus Message?**	**What's happening?**
1	I → S	Local request	- Cache block currently invalid processor X tries to read - Data not present - Send bus request for data from memory
2	I → I	Bus message	- A cache sees a read or write request for block A … but it doesn't have it so we stay in I - (remember – must always snoop)
3	S → I	Bus message	- Another $ has written to a block that is cached locally - With the invalidate protocol, a locally cached copy must be invalidated
4	S → S	Local request	- We do a local read of data that is already cached locally
5	S → S	Bus message	- Another cache asks for a copy of a block we have in order to do a read - As the request is just for another cached copy for reading, existing copies can stay in the shared state
6	M → S	Bus message	- A block has been modified by node X; node Y wants to read this data - Therefore data must be written back to memory before and/or in addition to going to the cache requesting it - Data is shared again and memory has a copy as well
7	S → M	Local request	- Local process writes to cache - Must broadcast that it is doing a write to invalidate other copies that may be cached - Locally, the block transitions to a modified state
8	M → M	Local request	- If we have a modified copy, and there are no other copies out there, we can read and write as we please
9	I → M	Local request	- Local copy is not in the cache and we want to write - We get it, write to it, and place it in a modified state
10	M → I	Bus request	- Another cache wants to write our modified data - We must invalidate our local copy … as it no longer is the "most recent" and send our data to memory and/or cache (other words in block could be dirty)

Part E: MESI Cache Coherency Protocol

Can the overhead associated with the S → M transition be improved?

- Yes: If in S state, could be only copy…

- We really just need to invalidate, but instead we send out a write request message that is broadcast to call nodes, memory
- Can cut this overhead by adding an "E" state → which stands for "Exclusive"
    - Eliminates bus operations when node X wants to do a read/write and there are no other cached copies
    - Go from E → M with no bus traffic

Would add 5 states to the MSI state machine

- The first 10 are exactly the same
- There is NO overhead
    - We need 2 bits of information to encode 3 states, we also need 2 bits of information to encode 4 states

Consider the following state transitions → also, **DRAW PICTURE ON BOARD**:

	State Transition	**Local Request or Bus Message?**	**What's happening?**
1	I → E	Local request	- We do a read (when we initially did NOT have the block in our cache AND no other block has the data cached)
2	E → I	Bus request	- Another processor with no cached copy wants to write - Our processor must invalidate its copy - As no modifications have been made (i.e. no dirty bit was set) there is no need to write back to memory too
3	E → E	Local request	- We read our cache copy - No other note has a cached copy so we stay in E
4	E → M	Local request	- We are in E and write our block - Must move to M - Will determine if writeback needed on an invalidate
5	E → S	Bus request	- Another node wants to read data we have cached - No writes were made however so we can stay in S and keep a copy cached

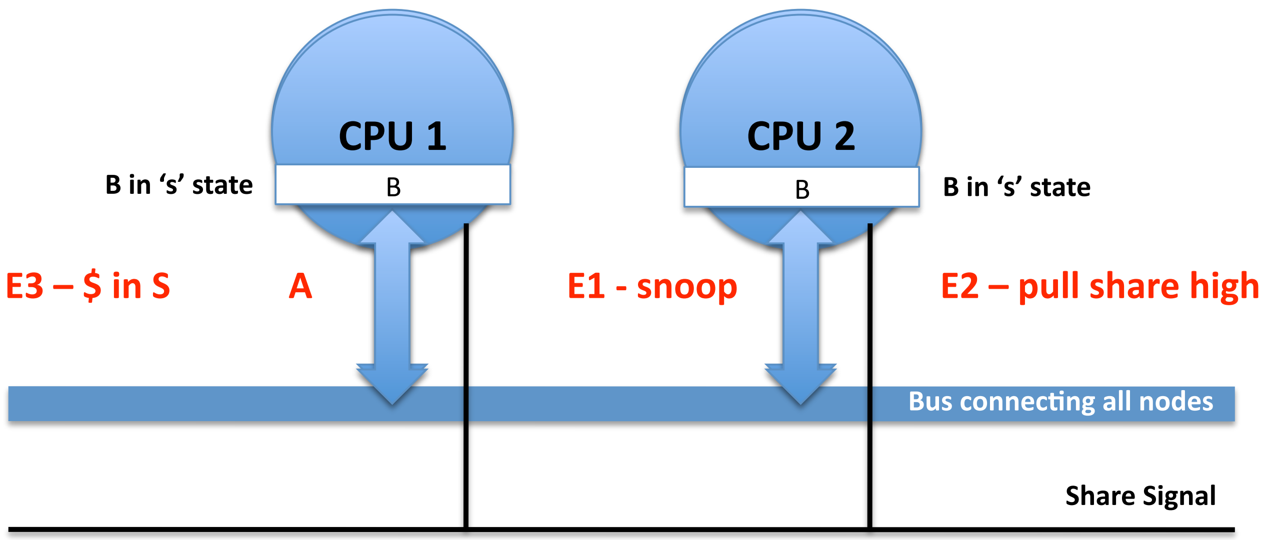
Part F: Support for Intervention + Determining Block State

(i.e. support for intervention + determining block state)

First … how do we know what state to cache block B in?

- If there's an address and data, receiver just sees an address and data.
- Where did it come from?

Realistically, it works like this:



A. CPU1 wants to read B → puts read request on the bus
B. Does CPU1 cache B in 'S' or 'E' state with MESI?
C. Solution → use share signal
D. Share always low until another node pulls it high
E. CPU2 snoops CPU1's requests, pulls share signal high → CPU1 sees share go high and puts B in shared state

Part G: How a Directory Protocol Might Work

Assume the following state:

Directory	Address	Dirty	Presence	
			1 2 3 4 5 6 7 8	
Node #3	5004	0	1 0 0 0 1 0 0 0	# nodes 1,5 have data
	5008	1	1 0 0 0 0 0 0 0	
	5012	0	0 0 0 0 0 1 1 1	

- If request for data at address 5008 from node 2, data should reside on node 3
- Node 2 sends request for data at address 5008 to node 3
- Node 3 checks directory and sees node 1 has a modified copy; requests data for node 2
- Node 3 gets data back, updates directory, sends data to node 2
  - Dirty: 0 Presence: 0 1 0 0 0 0 0 0