CS152 Discussion Section

Cache Coherence

Lecture 21

April 15-19
Spring 2024

Agenda

- Snoopy Cache Coherence Protocol
- Directory-based Protocol

Review: Cache Coherence Basics

Cache Coherence vs Memory Consistency

- Informally, a memory system is coherent if any read of a given memory location returns the most recently written value

- **Coherence**: What values can be returned for a read
  - Ordering of operations to the same memory location
- **Consistency**: When a written value will be returned by a read
  - Ordering of operations to different memory locations

Coherence Invariant #1

Preservation of program order:

> A read by a processor P from location X that follows a write by P to X, with no intervening writes to X occurring between the write and read by P, always return the value written by P

Coherence Invariant #2

Eventuality:

A read by a processor from location X that follows a write by another processor to X returns the written value if

1. The read and write are “sufficiently” separated in time
2. No other writes to X occur between the read and write

Coherence Invariant #3

Write serialization:

> Two writes to the same location by any two processors are seen in the same order by all processors

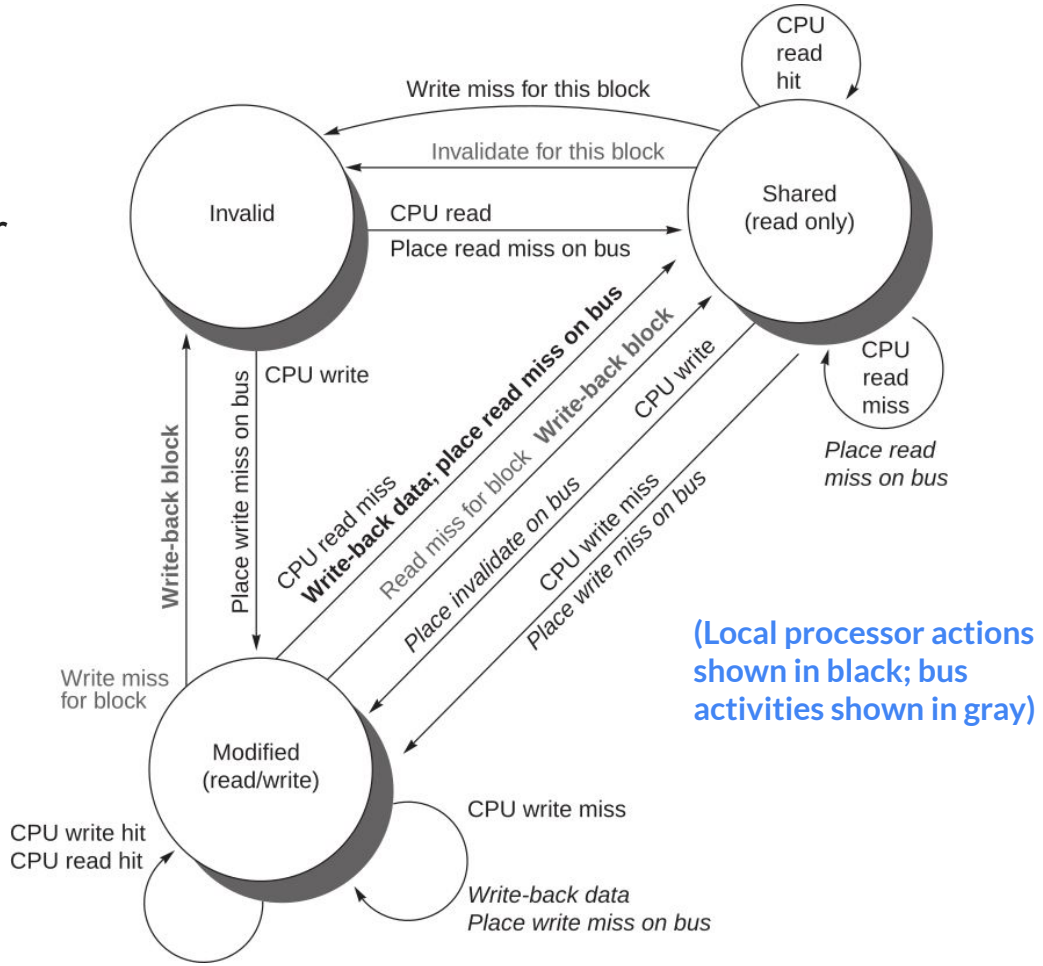
Snoopy vs Directory Protocols
MSI

Cache Coherence Protocols

- **Snooping**: Each cache tracks the status of a cached line by monitoring a broadcast medium (e.g., bus) for transactions
- **Directory-based**: Status of cache line is kept at one site (directory); communicate only with nodes that have copies of the line
  - Centralized: Typical for Symmetric Multiprocessors (SMP)
  - Distributed: Common in distributed shared-memory systems (e.g., SGI Origin)
- Snooping and directories can be combined in multi-level memory hierarchies

MSI Protocol

- If line is in M state, no other cache can have the line; multiple differing copies cannot exist
- MESI adds exclusive (E) state: line is resident in one cache still but clean



(Local processor actions shown in black; bus activities shown in gray)

Snooping

Q: A snooping cache coherence protocol requires cores to communicate on a single physical bus. True/false?

A: **False**. Snooping requires a totally ordered broadcast network, but the functionality can be implemented without a single shared-wire bus.

(e.g. having parallel interleaved buses (data, snoopy broadcast) with multiple tag banks)

Sorin et al. A Primer on Memory Consistency and Cache Coherence (2011)

Snooping

Q: In an MSI snooping protocol, a cache line may be in only one of three coherence states. True/false?

A: **True**. But if atomicity of requests and transactions is not guaranteed, transient states are needed.

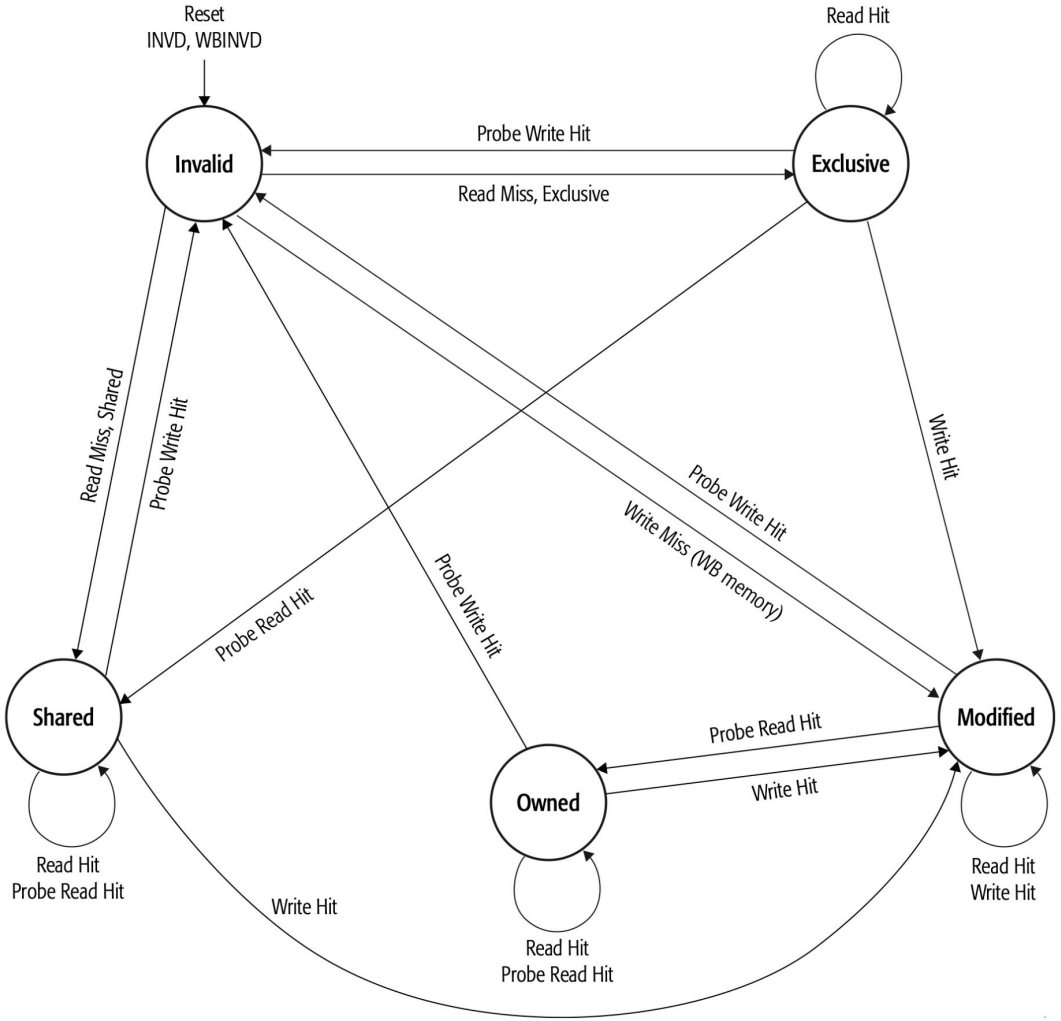
Sorin et al. A Primer on Memory Consistency and Cache Coherence (2011)

MOESI Protocol (Q1)

Extends MESI with a fifth “Owned” state

- Allows caches to hold dirty data without invalidating sharers
- Only one cache can be in the “Owned” state while the other caches are in “Shared”
- Owner is responsible for sending data to other caches requesting the line and must write back the data when it downgrades

MOESI Protocol (Q1)



For each of the following new state transitions, indicate whether it is a valid transition. If it is a valid transition, explain what triggers it, what conditions must be true (i.e. do other sharers exist?), and what actions must be taken during the transition.

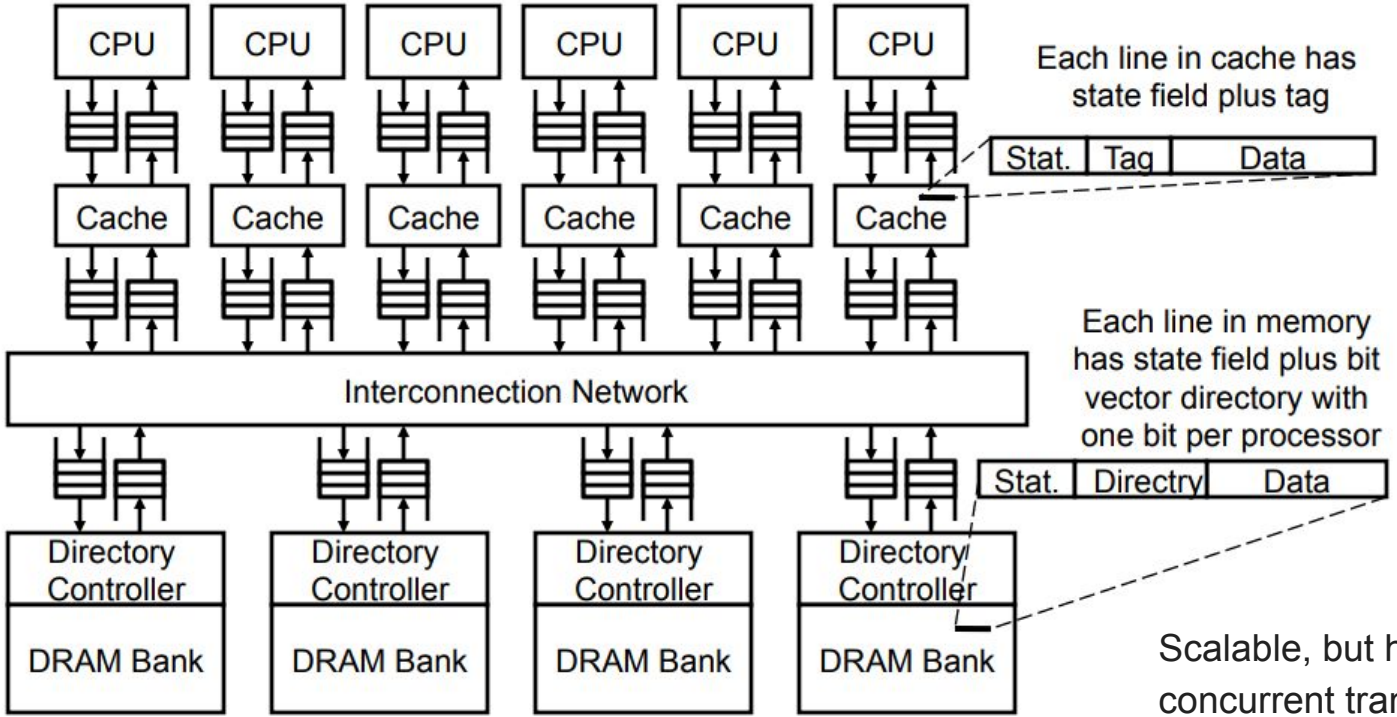
	Trigger	**Condition**	**Action**
I → O			
O → I			
S → O			
O → S			
E → O			
O → E			
M → O			
O → M			

	Trigger	Condition	Action
I → O	Not possible under snooping; local cache is not aware of presence of other sharers (write miss would result in I → M instead)		
O → I			
S → O	Local cache writes to line	Other sharers exist	Broadcast modified value if requested
O → S			
E → O	Not allowed. While in E, no other cache has a copy, so transition to M instead.		
O → E			
M → O	Another cache reads the line	None	Broadcast up-to-date value
O → M			

	Trigger	Condition	Action
I → O	Not possible under snooping; local cache is not aware of presence of other sharers (write miss would result in I → M instead)		
O → I	Local cache evicts the line (voluntary writeback or invalidation)	None	Write back to memory
S → O	Local cache writes to line (S -> M -> O)	Other sharers exist	Broadcast modified value
O → S	Another cache writes to line (O -> I -> S)	Other sharers exist; some other sharer is writing	None
E → O	Not allowed. While in E, no other cache has a copy, so transition to M instead.		
O → E	Not allowed. A cache can only be in E if no other cache shares it, but if that is the case, there is no need to write back dirty data.		
M → O	Another cache reads the line	None	Broadcast up-to-date value
O → M	Local cache writes to line (write hit)	None	Invalidate other sharers

Directory Cache Coherence

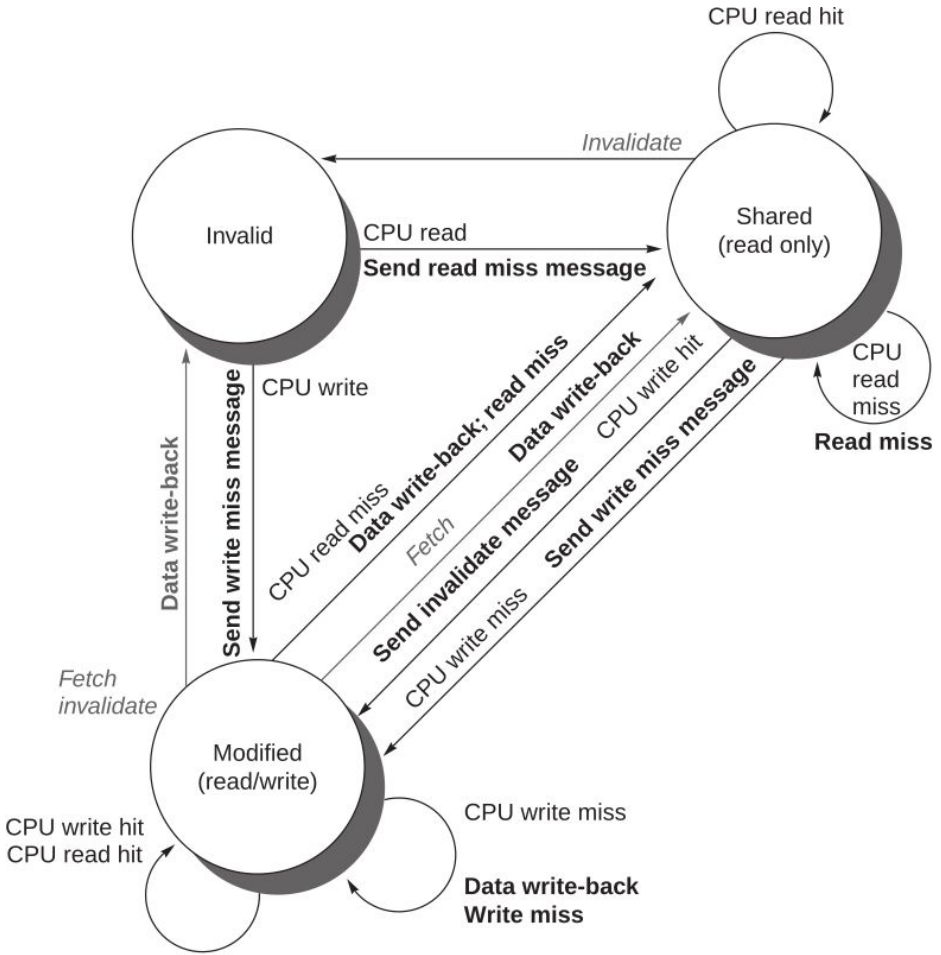
+ Unicast (vs Broadcast)
- Extra bits for each memory line



Scalable, but how to manage concurrent transactions?

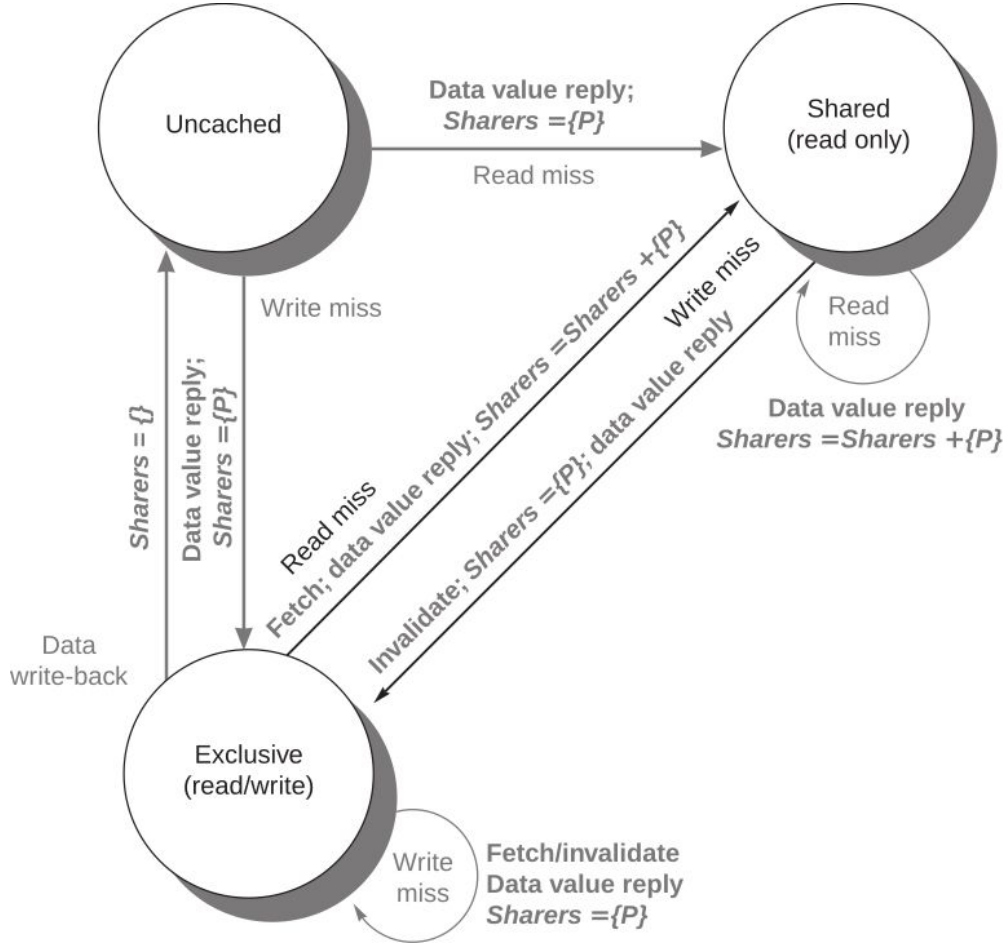
Example Directory Protocol

- Cache side
- Requests from local processor shown in black
- Requests from home directory shown in grey



Example Directory Protocol

- Directory side
- Actions taken by directory shown in bold



Full Bit Vector Scheme (Q2)

In the simplest design, each directory entry contains the state of the cache line and a bit vector with one sharer bit per processor. Assume the directory lines have four states.

How many directory bits are needed per cache line?

1. For 128 cores with private caches:

2. For 1024 cores with private caches:

Full Bit Vector Scheme (Q2)

In the simplest design, each directory entry contains the state of the cache line and a bit vector with one sharer bit per processor. Assume the directory lines have four states.

How many directory bits are needed per cache line?

1. For 128 cores with private caches:

   log(4) = 2 bits for state; 128 bits in bit vector; 2 + 128 = 130

2. For 1024 cores with private caches:
   2 + 1024 = 1026

Hierarchical Bit Vector Scheme (Q2)

- Storing one bit for each sharer per line not be practical for massively multicore systems
  - For a 1024-core system with 64B lines, twice as much memory is used for the sharer bits alone than for actual data storage
- Aggregate cores into groups and represent each group as a single bit in the bit vector
  - Invalidations must now be sent to all cores in a group if that group's bit is set

For a 1024-core system with 64-byte cache lines, how many cores must be in each group to reduce the amount of directory state to 10% the amount of physical memory?

Hierarchical Bit Vector Scheme (Q2)

- Storing one bit for each sharer per line not be practical for massively multicore systems
  - For a 1024-core system with 64B lines, twice as much memory is used for the sharer bits alone than for actual data storage
- Aggregate cores into groups and represent each group as a single bit in the bit vector
  - Invalidations must now be sent to all cores in a group if that group's bit is set

For a 1024-core system with 64-byte cache lines, how many cores must be in each group to reduce the amount of directory state to 10% the amount of total physical memory?

$2 + 1024 / N <= 64*8 / 10$

$2 + 1024 / N <= 512 / 10$

$N \geq 21$

So there must be at least 21 cores per directory group.

Reducing Storage Overhead (Q2)

One inefficiency of this system is that you must store directory bits for every line in memory, no matter if it is cached or not. How could you reduce this inefficiency?

Reducing Storage Overhead (Q2)

One inefficiency of this system is that you must store directory bits for every line in memory, no matter if it is cached or not. How could you reduce this inefficiency?

- Add a shared last-level cache inclusive of all other private caches; then directory only needs to contain directory entries for lines held in the last-level cache, not all lines in memory
- Hierarchical directory structure