EECS 570
Lecture 11
Directory-based Coherence
Winter 2020
Prof. Satish Narayanasamy

http://www.eecs.umich.edu/courses/eecs570/

Slides developed in part by Profs. Adve, Falsafi, Hill, Lebeck, Martin, Narayanasamy, Nowatzyk, Reinhardt, Roth, Smith, Singh, and Wenisch.
Announcements

- Midterm, Wednesday 2/26

- **NO** cheat (reference) sheet
  - We initially had announced that we will allow this. But, given the volume of information across the papers, we think it won’t be productive.
Readings

For today:
- Daniel J. Sorin, Mark D. Hill, and David A. Wood, A Primer on Memory Consistency and Cache Coherence, Chapter 8

For Monday 2/24:
Directory-Based Coherence
Scalable Cache Coherence

- **Scalable cache coherence**: two part solution

- **Part I**: bus bandwidth
  - Replace non-scalable bandwidth substrate (bus)...
  - ...with scalable bandwidth one (point-to-point network, e.g., mesh)

- **Part II**: processor snooping bandwidth
  - Interesting: most snoops result in no action
  - Replace non-scalable broadcast protocol (spam everyone)...
  - ...with scalable directory protocol (only spam processors that care)
Directory Coherence Protocols

• Observe: physical address space statically partitioned
  + Can easily determine which memory module holds a given line
    ❍ That memory module sometimes called “home”
  – Can’t easily determine which processors have line in their caches
     ❍ Bus-based protocol: broadcast events to all processors/caches
       ± Simple and fast, but non-scalable

• **Directories**: non-broadcast coherence protocol
  ❍ Extend memory to track caching information
  ❍ For each physical cache line whose home this is, track:
    ❍ **Owner**: which processor has a dirty copy (I.e., M state)
    ❍ **Sharers**: which processors have clean copies (I.e., S state)
  ❍ Processor sends coherence event to home directory
    ❍ Home directory only sends events to processors that care
Basic Operation: Read

Node #1: Load A (miss)

Directory: Get-S A

Node #2: Data A

A: Shared, #1
Basic Operation: Write

Node #1

Read A (miss)

Node #2

Directory

A: Shared, #1

A: Mod., #2

Get-M A

Data A

Invalid B A

Inv-Ack A

Fill A

Read A

AEC 570
Centralized Directory

- **Single directory** contains a copy of cache tags from all nodes

- **Advantages:**
  - Central serialization point: easy to get memory consistency (just like a bus...)

- **Problems:**
  - Not scalable (imagine traffic from 1000’s of nodes...)
  - Directory size/organization changes with number of nodes
Distributed Directory

- Distribute directory among memory modules
  - Memory block = coherence block (usually = cache line)
  - “Home node” → node with directory entry
    - Usually also dedicated main memory storage for cache line
  - Scalable – directory grows with memory capacity
    - Common trick: steal bits from ECC for directory state
  - Directory can no longer serialize accesses across all addresses
    - Memory consistency becomes responsibility of CPU interface
What is in the directory?

- Directory State
  - Invalid, Exclusive, Shared, ...
  - # outstanding invalidation messages, ...

- Pointer to exclusive owner

- Sharer list
  - List of caches that may have a copy
  - May include local node
  - Not necessarily precise, but always conservative
Directory State

- Few stable states – 2-3 bits usually enough
- Transient states
  - Often 10’s of states (+ need to remember node ids, …)
  - Transient state changes frequently, need fast RMW access
- Design options:
  - Keep in directory: scalable (high concurrency), but slow
  - Keep in separate memory
  - Keep in directory, use cache to accelerate access
  - Keep in protocol controller
    - Transaction State Register File – like MSHRs
Pointer to Exclusive Owner

• Simple node id – $\log_2$ nodes
• Can share storage with sharer list (don’t need both...)
• May point to a group of caches that internally maintain coherence (e.g., via snooping)
• May treat local node differently
Sharer List Representation

- Key to scalability – must efficiently represent node subsets
- Observation: most blocks cached by only 1 or 2 nodes
  - But, there are important exceptions (synchronization vars.)

OLTP workload
[Data from Nowatzyk]
Idea #1: Sharer Bit Vectors

- One bit per processor / node / cache
  - Storage requirement grows with system size

0 1 1 0 0 0 0 1
Idea #2: Limited Pointers

- Fixed number (e.g., 4) of pointers to node ids
- If more than $n$ sharers:
  - Recycle one pointer (force invalidation)
  - Revert to broadcast
  - Handle in software (maintain longer list elsewhere)
Idea #3: Linked Lists

- Each node has fixed storage for next (prev) sharer
- Doubly-linked (Scalable Coherent Interconnect)
- Singly-linked (S3.mp)
- Poor performance:
  - Long invalidation latency
  - Replacements – difficult to get out of sharer list
    - Especially with singly-linked list... – how to do it?
Directory representation optimizations

- Coarse Vectors (CV)
- Cruise Missile Invalidations (CMI)
- Tree Extensions (TE)
- List-based Overflow (LO)
Clean Eviction Notification

• Should directory learn when clean blocks are evicted?

• Advantages:
  - Avoids broadcast, frees pointers in limited pointer schemes
  - Avoids unnecessary invalidate messages

• Disadvantages:
  - Read-only data never invalidated (extra evict messages)
  - Notification traffic is unnecessary
  - New protocol races
Sparse Directories

- Most of memory is invalid; why waste directory storage?
- Instead, use a directory cache
  - Any address w/o an entry is invalid
  - If full, need to evict & invalidate a victim entry
  - Generally needs to be highly associative
Cache Invalidation Patterns

- Hypothesis: On a write to a shared location, # of caches to be invalidated is typically small
- If this isn’t true, directory is no better than broadcast/snoop
- Experience tends to validate this hypothesis
Common Sharing Patterns

• Code and read-only objects
  □ No problem since rarely written

• Migratory objects
  □ Even as number of caches grows, only 1-2 invalidations

• Mostly-read objects
  □ Invalidations are expensive but infrequent, so OK

• Frequently read/written objects (e.g., task queues)
  □ Invalidations frequent, hence sharer list usually small

• Synchronization objects
  □ Low-contention locks result in few invalidations
  □ High contention locks may need special support (e.g. MCS)

• Badly-behaved objects
Designing a Directory Protocol: Nomenclature

- Local Node (L)
  - Node initiating the transaction we care about

- Home Node (H)
  - Node where directory/main memory for the block lives

- Remote Node (R)
  - Any other node that participates in the transaction
Read Transaction

- L has a cache miss on a load instruction
4-hop Read Transaction

- L has a cache miss on a load instruction
  - Block was previously in modified state at R
3-hop Read Transaction

- L has a cache miss on a load instruction
  - Block was previously in modified state at R

**Diagram:**

1. L sends a Get-S request to H.
2. H forwards the Get-S request to R.
3. R sends the data back to H.
4. H sends the data back to L.

**Notes:**
- State: M
- Owner: R
An Example Race: Writeback & Read

- L has dirty copy, wants to write back to H
- R concurrently sends a read to H

```
1: Put-M+Data
2: Get-S
3: Fwd-Get-S
4:
5: Data
6:
7: Put-Ack
```

To make your head really hurt:

Can optimize away SI^A & Put-Ack!

L and H each know the race happened, don’t need more msgs.
Store-Store Race

- Line is invalid, both L and R race to obtain write permission

1: Get-M

3: Data [ack=0]

5: Data [ack=0]

6: Fwd-Get-M

7: Fwd-Get-M to L; New Owner: R

8: Data [ack=0]

Race! Stall for Data, do 1 store, then Fwd to R

IMAD
Worst-case scenario?

- L evicts dirty copy, R concurrently seeks write permission

Race! Put-M floating around! Wait till its gone…

Put-M from NonOwner: Race! L waiting to ensure Put-M gone…

4: Data [ack=0]

1: Put-M
2: Get-M
3: Fwd-Get-M
5: 
6: Put-Ack
Design Principles

• Think of sending and receiving messages as separate events

• At each “step”, consider what new requests can occur
  ❑ E.g., can a new writeback overtake an older one?

• Two messages traversing same direction implies a race
  ❑ Need to consider both delivery orders
    ❑ Usually results in a “branch” in coherence FSM to handle both orderings
  ❑ Need to make sure messages can’t stick around “lost”
    ❑ Every request needs an ack; extra states to clean up messages
  ❑ Often, only one node knows how a race resolves
    ❑ Might need to send messages to tell others what to do