EECS 570
Lecture 12
Directory & Optimizations
Winter 2018
Subhankar Pal
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

• Milestone 1:
  ❑ Sign up for group meetings on 2/22 and 2/23
  ❑ I will send a link out on Canvas

• Midterm exam on Wednesday, 2/21 in class

• Monday’s lecture will contain the last set of slides relevant for the midterm exam
Readings

• No readings for Monday, 2/19 or Wednesday, 2/21
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 events to home directory
    ○ Home directory only sends events to processors that care
Basic Operation: Read

Node #1 → Directory → Node #2

Load A (miss)

Get-S A

Data A

A: Shared, #1
Basic Operation: Write

- Read A (miss)
- Read A
- Fill A
- Invalidate A
- Inv-Ack A
- Get-M A
- A: Mod., #2
- Data A
- A: Shared, #1
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

![Diagram of distributed directory system](image)
What is in the directory?

- Directory State
  - Invalid, Exclusive, Shared, ... ("stable" states)
  - # outstanding invalidation messages, ... ("transient" states)
- 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.)

![Graph showing number of sharers at an exclusive request (P=32)](OLTP workloads)

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

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

• Any better ideas?
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 without 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

1: Get-S

2: Data
4-hop Read Transaction

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

Diagram:

1: Get-S

L → H

2: Recall

R → H

State: M
Owner: R

3: Data

H → R

4: Data

L → H
3-hop Read Transaction

- L has a cache miss on a load instruction
  - Block was previously in modified state at R
An Example Race: Writeback & Read

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

Race! Put-M & Fwd-Get-S

Race! Final State: S

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.

EECS 570
Store-Store Race

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

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

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

1: Get-M

Get-M

3:

5:

4: Data [ack=0]

6: Fwd-Get-M

8: Data [ack=0]
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...

1: Put-M
2: Get-M
3: Fwd-Get-M
5: 

4: Data [ack=0]
6: Put-Ack

L
H
R
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
CC Protocol Scorecard

- Does the protocol use negative acknowledgments (retries)?
- Is the number of active messages (sent but unprocessed) for one transaction bounded?
- Does the protocol require clean eviction notifications?
- How/when is the directory accessed during transaction?
- How many lanes are needed to avoid deadlocks?
NACKs in a CC Protocol

- Issues: Livelock, Starvation, Fairness
- NACKs as a flow control method ("home node is busy")
  - Really bad idea...
- NACKs as a consequence of protocol interaction...

Race! Put-M & Fwd-Get-S

Race! Final State: S
No need to Ack

1: Put-M
2: Get-S
3: Fwd-Get-S
4: 6:
5: Get-S NACK
Bounded # Msgs / Transaction

- Scalability issue: how much queue space is needed
- Coarse-vector vs. cruise-missile invalidation
Frequency of Directory Updates

- How to deal with transient states?
  - Keep it in the directory: unlimited concurrency
  - Keep it in a pending transaction buffer (e.g., transaction state register file): faster, but limits pending transactions

- Occupancy free: Upon receiving an unsolicited request, can directory determine final state solely from current state?
Required # of lanes

- Need at least 2:
  - More may be needed by I/O, complex forwarding
  - How to assign lane to message type?
    - Secondary (forced) requests must not be blocked by new requests
    - Replies (completing a pending transaction) must not be blocked by new requests
Some more guidelines

• All messages should be ack’d (requests elicit replies)
• Maximum number of potential concurrent messages for one transaction should be small and constant (i.e., independent of number of nodes in system)
• Anticipate *ships passing in the night* effect
• Use context information to avoid NACKs
Optimizing coherence protocols

Read A (miss)

Read latency

Get-S A

Recall A

Data A

Data A
Prefetching

L

Prefetch A

Get-S A

H

Recall A

Data A

R

Data A

Read latency

Read A (miss)
3-hop reads

Read A (miss)
Read latency

Get-S A

Fwd-Get-S A

Data A

Data A
3-hop writes

L - H - R

Store A (miss)

Get-M A

Data [ack=x]

Invalidate A

Inv-Ack A

Store latency
Migratory Sharing

- Each Read/Write pair results in read miss + upgrade miss
- Coherence FSM can detect this pattern
  - Detect via back-to-back read-upgrade sequences
  - Transition to “migratory M” state
  - Upon a read, invalidate current copy, pass in “mig E” state
Producer Consumer Sharing

Node 1  
Read X  
Write X

Node 2  
Read X

Node 3  
Read X  
Read X  
Write X

• Upon read miss, downgrade instead of invalidate
  □ Detect because there are 2+ readers between writes

• More sophisticated optimizations
  □ Keep track of prior readers
  □ Forward data to all readers upon downgrade
Shortcomings of Protocol Optimizations

- Optimizations built directly into coherence state machine
  - Complex! Adds more transitions, races
  - Hard to verify even basic protocols
  - Each optimization contributes to state explosion
  - Can target only simple sharing patterns
  - Can learn only one pattern per address at a time
Table-based protocol predictors

- Decouple predictor from protocol
  - Learn multiple sharing patterns simultaneously
  - Protocol hints $\rightarrow$ no impact on state machine
  - But, may require significant storage
**Memory Sharing Predictor [ISCA’99]:**

2-level table-based predictor at each dir.

- Keeps history of prior messages
- For each history, keeps a sharing outcome
- E.g., an upgrade by P3 leads to reads by P1, P2

![Diagram showing history and pattern tables]

- History Table:
  - (upgrade, P3)
  - (read, [P1, P2])
  - (read, [P1, P2])

- Pattern Table:
  - (upgrade, P3)
  - (read, [P1, P2])
  - (upgrade, P3)

Block 0x100
• Predict last access
• Release block
• 3-hop misses $\rightarrow$ 2-hop

Self-Invalidations are
+ Timely
  early as possible
+ Accurate
  only last-touched block
+ No protocol changes
– Requires more storage
An LTP per processor
- collects trace per block
- upon invalidation
  - records trace
- upon every rd/wr
  - compares trace
- e.g. \{PC_i, PC_j, PC_k\}
  - is a last-touch trace

How Does an LTP Work?

**PC_i**: rd/wr X  
miss on X

**PC_j**: rd/wr X

**PC_k**: rd/wr X  
last touch

Dynamic Instruction Stream

fetch/ invalidate X
Cache Only Memory Architecture (COMA)
Big Picture

- Centralized shared memory
- Uniform access

- Distributed shared memory
- Non-uniform access latency

- No notion of “home” node; data moves to wherever it is needed
- Individual memories behave like caches
Cache Only Memory Architecture (COMA)

• Make all memory available for migration/replication
• All memory is DRAM cache called attraction memory

• Example systems
  ▐ Data Diffusion Machine (next slide)
  ▐ Flat COMA (fixed home node for directory, but not data)
  ▐ KSR-1 (hierarchical snooping via ring interconnects)

• Key questions:
  ▐ How to find data?
  ▐ How to deal with replacements?
  ▐ Memory overhead
Data Diffusion Machine

• All-hardware COMA
• Attraction memory → one giant hardware cache
• Maintains both address tags and state
• Data addressed, allocated, kept coherent in blocks (“items”)
• Directory info on a per-block basis
• Not home based
  ■ Data is migratory → read requests “attract” data
  ■ Must find a “home” during replacement
  ■ Must find the directory entry before finding the data
DDM Directory

- Directory organized in a hierarchical tree
- Each is a set-associative cache of directory info
- Tree maintains inclusion
- Higher levels keep replica of lower sub-trees
DDM Coherence / Placement Protocol

- Simple write-invalidate protocol
- Cache states: Invalid, Exclusive, Shared
- Must traverse the directory
  - To find a copy on a read or write miss
  - To invalidate on a write to Shared
- Directory is a hierarchical set-associative cache
  - Q1: Is the block in my sub-tree?
  - Q2: Does the block exist outside my sub-tree?
  - Request goes up till Q2==no and then down
  - Request goes down till Q1==no or leaf
- On a replacement
  - For an Exclusive copy, must find another home (HARD!)
  - For a Shared copy, must make sure other copies exist...
  - ...Else must find another home
Alternatives to COMA/DDM

• Flat-COMA
  ❑ Blocks (data) are free to migrate
  ❑ Fixed directory location (home node) for a physical address

• Simple-COMA
  ❑ Allocation managed by OS and done at page granularity

• Reactive-NUMA
  ❑ Switches between S-COMA and NUMA with remote cache on per-page basis