EECS 570
Lecture 13
Directory & Optimizations
Winter 2016
Prof. Thomas Wenisch
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

• Office hours moved to Tuesday 12-1pm
  ☑ You are welcome to attend Neha Agarwal’s PhD defense in 3725 BBB today at 3pm.
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
4: 

2: Get-S

3: Fwd-Get-S

5: Get-S NACK

6:
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
- Data A
- Recall A
- Data A
Prefetching

L → H:
- Prefetch A
- Get-S A

H → R:
- Recall A
- Data A

Read A (miss)
- Read latency
3-hop reads

Read A (miss)

Get-S A

Fwd-Get-S A

Data A

Read latency
3-hop writes

Store A (miss)

Store latency

Get-M A

Data [ack=x]

Invalidate A

Inv-Ack A
## Migratory Sharing

<table>
<thead>
<tr>
<th>Node 1</th>
<th>Node 2</th>
<th>Node 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read X</td>
<td>Read X</td>
<td>Read X</td>
</tr>
<tr>
<td>Write X</td>
<td>Write X</td>
<td>Write X</td>
</tr>
</tbody>
</table>

- 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**

<table>
<thead>
<tr>
<th>Node 1</th>
<th>Node 2</th>
<th>Node 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write X</td>
<td>Read X</td>
<td></td>
</tr>
<tr>
<td>Read X</td>
<td>Read X</td>
<td></td>
</tr>
<tr>
<td>Read X</td>
<td></td>
<td>Read X</td>
</tr>
</tbody>
</table>

- 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]:

History Table

- (upgrade, P3)

Pattern Table

- (upgrade, P3)
- (read, [P1, P2])
- (upgrade, P3)
- (read, [P1, P2])

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
Last Touch Predictors [ISCA ’00]

- 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
How Does an LTP Work?

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

PC_i : rd/wr X  \quad \text{miss on X}

PC_j : rd/wr X

PC_k : rd/wr X  \quad \text{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