EECS 570
Lecture 12
Dir. Optimization & COMA
Winter 2019
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

• Milestone 1:
  - max 2-page write-up of progress and initial results
  - Submit via Canvas by 11:59pm tonight
  - Sign up for group meeting on Piazza (soon)

• Midterm Wednesday
Readings

For today:
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

```
1: Get-S

L

2: Recall

H

3: Data

R

4: Data

State: M
Owner: R
```
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

To make your head really hurt:
Can optimize away SI^ & 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
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
4: Data [ack=0]
5: Put-M from NonOwner
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
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: Race!
5: Get-S NACK
6: Race!
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) → Get-S A → Recall A → Data A

Readlatency

L → H → R
Prefetching

Prefetch A

Get-S A

Recall A

Data A

Read A (miss)

Data A

Read latency
3-hop reads

Read A (miss)

Get-S A

Fwd-Get-S A

Data A

Data A

Read latency
3-hop writes

Store A (miss)

Get-M A

Data [ack=x]

Inv-Ack A

Invalidate A

Store latency
## 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

- 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
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
COMA Alternatives

- 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