Directory-Based Cache Coherence

Mengjia Yan
Computer Science and Artificial Intelligence Lab
M.I.T.
Maintaining Cache Coherence

It is sufficient to have hardware such that

• Only one processor at a time has write permission for a location
• No processor can load a stale copy of the location after a write

⇒ A correct approach could be:

write request:
The address is *invalidated* in all other caches *before* the write is performed

read request:
If a dirty copy is found in some cache, a write-back is performed before the memory is read
Directory-Based Coherence
[Censier and Feautrier, 1978]

Snoopy Protocols

• Snoopy schemes broadcast requests over memory bus
• Difficult to scale to large numbers of processors
• Requires additional bandwidth to cache tags for snoop requests

Directory Protocols

• Directory schemes send messages to only those caches that might have the line
• Can scale to large numbers of processors
• Requires extra directory storage to track possible sharers
An MSI Directory Protocol

- Cache states: Modified (M) / Shared (S) / Invalid (I)
- Directory states:
  - Uncached (Un): No sharers
  - Shared (Sh): One or more sharers with read permission (S)
  - Exclusive (Ex): A single sharer with read & write permissions (M)
- Transient states not drawn for clarity; for now, assume no racing requests
Transitions initiated by processor accesses:

- PrRd / --
- PrWr / --
- PrWr / ExReq
- PrWr / ExReq
- PrRd / --
- PrRd / ShReq

### Actions

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Read</td>
<td>PrRd (PrRd)</td>
</tr>
<tr>
<td>Processor Write</td>
<td>PrWr (PrWr)</td>
</tr>
<tr>
<td>Shared Request</td>
<td>ShReq (ShReq)</td>
</tr>
<tr>
<td>Exclusive Request</td>
<td>ExReq (ExReq)</td>
</tr>
</tbody>
</table>
MSI Protocol: Caches (2/3)

Transitions initiated by directory requests:

- InvReq / InvResp (with data)
- DownReq / DownResp (with data)
- InvReq / InvResp (without data)

Actions:

<table>
<thead>
<tr>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalidation Request (InvReq)</td>
</tr>
<tr>
<td>Downgrade Request (DownReq)</td>
</tr>
<tr>
<td>Invalidation Response (InvResp)</td>
</tr>
<tr>
<td>Downgrade Response (DownResp)</td>
</tr>
</tbody>
</table>
Transitions initiated by evictions:

- **Eviction / WbReq (with data)**
- **Eviction / WbReq (without data)**

**Actions**
- Writeback Request (WbReq)
MSI Protocol: Caches

Transitions initiated by processor accesses
Transitions initiated by directory requests
Transitions initiated by evictions
Transitions initiated by data requests:

ExReq / Sharers = {P}; ExResp

Ex

ExReq / Inv(Sharers), Sharers={P}; ExResp

ShReq / Down(Sharer); Sharers = Sharer + {P}; ShResp

ExReq / Inv(Sharers – {P}); Sharers = {P}; ExResp

Sh

ShReq / Sharers = Sharers + {P}; ShResp

ShReq / Sharers = {P}; ShResp

Un
MSI Protocol: Directory (2/2)

Transitions initiated by writeback requests:

```
Ex

WbReq / Sharers = {}; WbResp
```

```
Sh

WbReq && |Sharers| > 1 /
Sharers = Sharers - {P}; WbResp
```

```
Un

WbReq && |Sharers| == 1 /
Sharers = {}; WbResp
```
MSI Directory Protocol Example

1. LD 0xA

2. ShReq 0xA

3. Mem[0xA] = 3

4. ShResp 0xA, data=3
### MSI Directory Protocol Example

1. **LD 0xA**
   - **Core 0**
     - **Cache 0**
       - **Tag**: 0xA
       - **State**: S
       - **Data**: 3

2. **ShReq 0xA**
   - **Core 1**
     - **Cache 1**
       - **Tag**: 0xA
       - **State**: I
       - **Data**: 3

3. **Mem[0xA] = 3**
   - **Directory**
     - **Tag**: 0xA
     - **State**: Sh
     - **Sharers**: \{0, 2\}

4. **ShResp 0xA, data=3**
   - **Core 2**
     - **Cache 2**
       - **Tag**: 0xA
       - **State**: S
       - **Data**: 3
MSI Directory Protocol Example

```
<table>
<thead>
<tr>
<th>Tag</th>
<th>State</th>
<th>Sharers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xA</td>
<td>Ex</td>
<td>{1}</td>
</tr>
</tbody>
</table>
```

**Core 0**

```
<table>
<thead>
<tr>
<th>Tag</th>
<th>State</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xA</td>
<td>I</td>
<td>3</td>
</tr>
</tbody>
</table>
```

**Core 1**

```
<table>
<thead>
<tr>
<th>Tag</th>
<th>State</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xA</td>
<td>M</td>
<td>5</td>
</tr>
</tbody>
</table>
```

**Core 2**

```
<table>
<thead>
<tr>
<th>Tag</th>
<th>State</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xA</td>
<td>I</td>
<td>3</td>
</tr>
</tbody>
</table>
```

**Main Memory**

```
Mem[0xA] = 3
```

**Directory**

```
<table>
<thead>
<tr>
<th>Tag</th>
<th>State</th>
<th>Sharers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xA</td>
<td>Ex</td>
<td>{1}</td>
</tr>
</tbody>
</table>
```

**ST 0xA**

3. InvReq 0xA
6. ExResp 0xA
   data = 3
2. ExReq 0xA
3. InvReq 0xA
4. InvResp 0xA
4. InvResp 0xA
5. Mem[0xA] = 3
MSI Directory Protocol Example

Why are 0xA’s wb and 0xB’s req serialized?

Possible solutions?

Structural dependence
Buffer outside of cache to hold write data
Miss Status Holding Register

MSHR – Holds load misses and writes outside of cache

<table>
<thead>
<tr>
<th>V</th>
<th>X</th>
<th>Addr</th>
<th>Data</th>
</tr>
</thead>
</table>

- On eviction/writeback
  - No free MSHR entry: stall
  - Allocate new MSHR entry
  - When channel available send WBReq and data
  - Deallocate entry on WBResp
Miss Status Holding Register

MSHR – Holds load misses and writes outside of cache

- On cache load miss
  - No free MSHR entry: stall
  - Allocate new MSHR entry
  - Send ShReq (or ExReq)
  - On *Resp forward data to CPU and cache
  - Deallocate MSHR
Miss Status Holding Register

MSHR – Holds load misses and writes outside of cache

MSHR entry

<table>
<thead>
<tr>
<th>V</th>
<th>X</th>
<th>Addr</th>
<th>Data</th>
</tr>
</thead>
</table>

per ld/st slots

<table>
<thead>
<tr>
<th>V</th>
<th>L/S</th>
<th>Inum</th>
<th>Block Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>L/S</td>
<td>Inum</td>
<td>Block Offset</td>
</tr>
<tr>
<td>V</td>
<td>L/S</td>
<td>Inum</td>
<td>Block Offset</td>
</tr>
</tbody>
</table>

- On cache load miss
  - Look for matching address in MSHRs
    - If not found
      - If no free MSHR entry: stall
      - Allocate new MSHR entry and fill in
    - If found, just fill in per ld/st slot
  - Send ShReq (or ExReq)
  - On *Resp forward data to CPU and cache
  - Deallocate MSHR

Per ld/st slots allow servicing multiple requests with one entry
Directory Organization

• Requirement: Directory needs to keep track of all the cores that are sharing a cache block

• Challenge: For each block, the space needed to hold the list of sharers grows with number of possible sharers...
Flat, Memory-based Directories

• Dedicate a few bits of main memory to store the state and sharers of every line
• Encode sharers using a bit-vector

✓ Simple
✗ Slow
✗ Very inefficient with many processors (~P bits/line)
Sparse Full-Map Directories

- Not every line in the system needs to be tracked – only those in private caches!
- Idea: Organize directory as a cache

Directory Entry Format

- Low latency, energy-efficient
- Bit-vectors grow with # cores → Area scales poorly
- Limited associativity → Directory-induced invalidations
Directory-Induced Invalidations

- To retain inclusion, must invalidate all sharers of an entry before reusing it for another address
- Example: 2-way set-associative sparse directory

How many entries should the directory have?
Inexact Representations of Sharer Sets

- Coarse-grain bit-vectors (e.g., 1 bit per 4 cores)

![Sharer Set](image)

- Limited pointers: Maintain a few sharer pointers, on overflow mark ‘all’ and broadcast (or invalidate another sharer)

![Sharer Set](image)

- Allow false positives (e.g., Bloom filters)

- Reduced area & energy
- Overheads still not scalable (these techniques simply play with constant factors)
- Inexact sharers → Broadcasts, invalidations or spurious invalidations and downgrades
In-Cache Directories

- Common multicore memory hierarchy:
  - 1+ levels of private caches
  - A shared last-level cache
  - Need to enforce coherence among private caches

- Idea: Embed the directory information in shared cache tags
  - Shared cache must be inclusive

✓ Avoids tag overheads & separate lookups
✗ Can be inefficient if shared cache size $>>$ sum(private cache sizes)
Extra Hops and 3-Hop Protocols
Reducing Protocol Latency

- Problem: Data in another cache needs to pass through the directory, adding latency
- Optimization: Forward data to requester directly

**Diagram:**

1. **ST 0xA**
2. **ExReq 0xA**
3. **ExFwd 0xA, req=2**
4. **ExAck 0xA**
Protocol Races

- Directory serializes multiple requests for the same address
  - Same-address requests are queued or NACKed and retried
- But races still exist due to conflicting requests
- Example: Upgrade race

Caches 0 and 1 issue simultaneous ExReqs
Directory starts serving cache 0’s ExReq, queues cache 1’s
Cache 1 expected ExResp, but got InvReq!
Cache 1 should transition from S->M to I->M and send InvResp
Avoiding Protocol Deadlock

- Protocols can cause deadlocks even if network is deadlock-free! *(more on this later)*

  - Solution: Separate *virtual networks*
    - Different sets of virtual channels and endpoint buffers
    - Same physical routers and links

  - Most protocols require at least 2 virtual networks (for requests and replies), often \( >2 \) needed

Example: Both nodes saturate all intermediate buffers with requests to each other, blocking responses from entering the network.
Coherence in Multi-Level Hierarchies

- Can use the same or different protocols to keep coherence across multiple levels
- Key invariant: Ensure sufficient permissions in all intermediate levels
- Example: 8-socket Xeon E7 (8 cores/socket)
Coherence and False Sharing
Performance Issue #1

A cache block contains more than one word and cache coherence is done at the block-level and not word-level.

Suppose $P_1$ writes $\text{word}_i$ and $P_2$ writes $\text{word}_k$ and both words have the same block address.

*What can happen?* The block may be invalidated (ping-pong) many times unnecessarily because addresses are in the same block.

*How to address this problem?*
Our cache coherence protocol will introduce a performance issue here. What is the problem?
Cache coherence protocols will cause `mutex` to ping-pong between P1’s and P2’s caches.

Ping-ponging can be reduced by first reading the `mutex` location (*non-atomically*) and executing a swap only if it is found to be zero (*test&test&set*).
In general, an *atomic read-modify-write* instruction requires two memory (bus) operations without intervening memory operations by other processors.

Implementation options:
- *With snoopy coherence, lock the bus* → expensive
- *With directory-based coherence, lock the line in the cache (prevent invalidations or evictions until atomic op finishes)* → complex

Modern processors use
- *load-reserve*
- *store-conditional*
Load-reserve & Store-conditional

Special register(s) to hold reservation flag and address, and the outcome of store-conditional

Load-reserve R, (a):
<flag, adr> ← <1, a>;
R ← M[a];

Store-conditional (a), R:
if <flag, adr> == <1, a>
then cancel other procs' reservation on a;
M[a] ← <R>;
status ← succeed;
else status ← fail;

If the cache receives an invalidation to the address in the reserve register, the reserve bit is set to 0

- Several processors may reserve ‘a’ simultaneously
- These instructions are like ordinary loads and stores with respect to the bus traffic
Load-Reserve/Store-Conditional

Swap implemented with Ld-Reserve/St-Conditional

# Swap(R1, mutex):

L: Ld-Reserve R2, (mutex)
    St-Conditional (mutex), R1
    if (status == fail) goto L
    R1 <- R2
The total number of coherence transactions is not necessarily reduced, but splitting an atomic instruction into load-reserve & store-conditional:

- *increases utilization* (and reduces processor stall time), especially in split-transaction buses and directories

- *reduces cache ping-pong effect* because processors trying to acquire a semaphore do not have to perform stores each time
Thank you!

Next Lecture:
Consistency and Relaxed Memory Models