Directory-Based Cache Coherence

Daniel Sanchez
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
MSI Protocol: Caches (1/3)

Transitions initiated by processor accesses:

Actions

<table>
<thead>
<tr>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Read (PrRd)</td>
</tr>
<tr>
<td>Processor Write (PrWr)</td>
</tr>
<tr>
<td>Shared Request (ShReq)</td>
</tr>
<tr>
<td>Exclusive Request (ExReq)</td>
</tr>
</tbody>
</table>
Transitions initiated by directory requests:

- **M**: Invalidation Request (InvReq) / Invalidation Response (InvResp) (with data)
- **S**: Downgrade Request (DownReq) / Downgrade Response (DownResp) (with data)
- **I**: Invalidation Request (InvReq) / Invalidation Response (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) from M to S
- Eviction / WbReq (without data) from S to I

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:

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

- **Sh**
  - ShReq / Sharers = Sharers + {P}; ShResp

- **Un**
  - ShReq / Sharers = {P}; ShResp
Transitions initiated by writeback requests:

- **Ex**
  - WbReq / Sharers = {}; WbResp

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

- **Un**
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
   - Cache 2
   - Directory

2. **ShReq 0xA**
   - Core 1
   - Directory

3. **Mem[0xA] = 3**
   - Main Memory
   - Directory

4. **ShResp 0xA, data=3**
   - Core 1
   - Core 2
   - Cache 0
   - Cache 1
   - Cache 2

- **Directory Table**
  - Tag: 0xA
  - State: Sh
  - Sharers: {0,2}
MSI Directory Protocol Example

1. ST 0xA
2. ExReq 0xA
3. InvReq 0xA
4. InvResp 0xA
5. Mem[0xA] = 3
6. ExResp 0xA

Core 0
- Cache 0
  - Tag 0xA
  - State I
  - Data 3

Core 1
- Cache 1
  - Tag 0xA
  - State M
  - Data 5

Core 2
- Cache 2
  - Tag 0xA
  - State I
  - Data 3

Directory
- Tag 0xA
- State Ex
- Sharers {1}
Why are 0xA’s wb and 0xB’s req serialized? Possible solutions?
**Miss Status Handling 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>

- **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 Handling 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 Handling Register

MSHR – Holds load misses and writes outside of cache

**Table:**

<table>
<thead>
<tr>
<th>V</th>
<th>X</th>
<th>Addr</th>
<th>Data</th>
<th>per ld/st slots</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>V</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>V</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>V</td>
</tr>
</tbody>
</table>

- **On cache load miss**
  - Look for matching address is MSHR
    - 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

✔ 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
  
<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-3</td>
<td>4-7</td>
<td>8-11</td>
<td>12-15</td>
<td>16-19</td>
<td>20-23</td>
<td></td>
</tr>
</tbody>
</table>

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

  Sharer Set
  
<table>
<thead>
<tr>
<th>0</th>
<th>8</th>
<th>14</th>
<th>33</th>
</tr>
</thead>
<tbody>
<tr>
<td>all</td>
<td>sharer 1</td>
<td>sharer 2</td>
<td>sharer 3</td>
</tr>
</tbody>
</table>

- 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
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

April 8, 2021
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

```
<table>
<thead>
<tr>
<th>Tag</th>
<th>State</th>
<th>Sharers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xA</td>
<td>Ex</td>
<td>{2}</td>
</tr>
</tbody>
</table>
```

1. **ST 0xA**
2. **ExReq 0xA**
3. **ExResp 0xA, data=3**
4. **ExAck 0xA**
5. **ExFwd 0xA, req=2**
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)
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)
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
Implementing Atomic Instructions

• In general, an *atomic read-modify-write* instruction requires two memory 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 often 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