Directory-Based Cache Coherence & Sequential Consistency

Daniel Sanchez
Computer Science and Artificial Intelligence Lab M.I.T.

http://www.csg.csail.mit.edu/6.823
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:

- **M**: PrRd / --, PrWr / --
- **S**: PrWr / ExReq, PrRd / --
- **I**: PrRd / ShReq

**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:

- **S** (Secondary)
  - InvReq / InvResp (without data)
  - DownReq / DownResp (with data)

- **I** (Invalid)
  - InvReq / InvResp (with data)

- **M** (Master)
  - InvReq / InvResp (with data)

**Actions**:

- Invalidation Request (InvReq)
- Downgrade Request (DownReq)
- Invalidation Response (InvResp)
- Downgrade Response (DownResp)
Transitions initiated by evictions:

- **M** to **S**: Eviction / WbReq (with data)
- **S** to **I**: 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
MSI Protocol: Directory (1/2)

Transitions initiated by data requests:

ExReq / Sharers = \{P\}; ExResp

\[ \text{Ex} \]

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

\[ \text{Sh} \]

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

\[ \text{Un} \]

ShReq / Sharers = Sharers + \{P\}; ShResp

ShReq / Sharers = \{P\}; ShResp
Transitions initiated by writeback requests:

- **Ex**
  - \( \text{WbReq} \land |\text{Sharers}| = 1 \) / \( \text{Sharers} = \emptyset \); \( \text{WbResp} \)

- **Sh**
  - \( \text{WbReq} \land |\text{Sharers}| > 1 \) / \( \text{Sharers} = \text{Sharers} - \{P\} \); \( \text{WbResp} \)

- **Un**
  - \( \text{WbReq} \land |\text{Sharers}| = 1 \) / \( \text{Sharers} = \emptyset \); \( \text{WbResp} \)
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
2. ShReq 0xA
3. Mem[0xA] = 3
4. ShResp 0xA, data=3
MSI Directory Protocol Example

1. ST 0xA

2. ExReq 0xA

3. InvReq 0xA

4. InvResp 0xA

5. Mem[0xA] = 3

6. ExResp 0xA

data = 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>

Cache 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>

Cache 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>

Cache 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>
MSI Directory Protocol Example

Why are 0xA’s wb and 0xB’s req serialized? Structural dependence
Possible solutions? Buffer outside of cache to hold write data
Miss Status Handling Register

MSHR – Buffer to hold 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 miss or writeback – scan MSHR for entry
  1. Entry not found in MSHR:
     - No free MSHR entry: stall
     - Allocate new MSHR entry - goto step 2
  2. Entry found in MSHR: add ld/st to per ld/st slot

- On data return from memory
  - Forward data to CPU and cache for all ld/st slots
  - Deallocate MSHR
Directory Organization

• How to track contents of shared caches?
  – Flat, memory-based directories
  – Sparse full-map directories
  – Sparse directories with inexact sharer representations
  – In-cache directories
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

![Diagram of Main Memory with State and Sharer Set]

- Simple
- Slow
- Very inefficient with many processors (~P bits / cache 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

```
<table>
<thead>
<tr>
<th>Tag</th>
<th>State</th>
<th>Sharers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xB</td>
<td>Sh</td>
<td>{0}</td>
</tr>
<tr>
<td>0xF</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>0xF</td>
<td>M</td>
<td>1</td>
</tr>
</tbody>
</table>
```

```
Core 2
```

```
<table>
<thead>
<tr>
<th>Tag</th>
<th>State</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xB</td>
<td>S</td>
<td>5</td>
</tr>
</tbody>
</table>
```

```
Main Memory
```

```
Mem[0xB] = 5
```

```
InvReq 0xA
```

```
InvResp 0xA
```

```
ShReq 0xB
```

```
ShResp 0xB, data=5
```

```
LD 0xB
```

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
  \[
  \begin{array}{ccccccc}
  0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0-3 & 4-7 & 8-11 & 12-15 & 16-19 & 20-23
  \end{array}
  \]

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

  Sharer Set
  \[
  \begin{array}{cccc}
  0 & 8 & 14 & 33 \\
  all & sharer 1 & sharer 2 & sharer 3
  \end{array}
  \]

- 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
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!

Example: Both nodes saturate all intermediate buffers with requests to each other, blocking responses from entering the network

- 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
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 13, 2014
Optimization: 3-hop Protocols

- Reduce latency by having a neighbor cache forward data to requester
Consistency
Coherence vs Consistency

• Coherence: What values can a read return?
  – Concerns reads/writes to a single memory location

• Consistency: When do writes become visible to reads?
  – Concerns reads/writes to multiple memory locations
Why Consistency Matters

*Initial memory contents*

- **a**: 0
- **flag**: 0

**Processor 1**
- Store (a), 10;
- Store (flag), 1;

**Processor 2**
- **L**: Load r1, (flag);
- if r₁ == 0 goto L;
- Load r2, (a);

- What value does processor 1’s r2 hold after both processors finish running this code?

  It depends on the order in which processor 2 observes processor 1’s stores!

  10 if Store (flag) > Store (a); 0 or 10 otherwise
"A system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in the order specified by the program"  

Leslie Lamport

Sequential Consistency =

arbitrary order-preserving interleaving of memory references of sequential programs
Sequential Consistency

- In-order instruction execution
- Atomic loads and stores

SC is easy to understand but architects and compiler writers want to violate it for performance
Memory Model Issues

Architectural optimizations that are correct for uniprocessors often violate sequential consistency and result in a new memory model for multiprocessors
Next Lecture:
Relaxed Memory Models