CIS 429/529

Cache Coherence Protocols

Prof. Michel A. Kinsky
Symmetric Multiprocessors (SMP)
Distributed Memory Multiprocessors

- Processor 1
- Cache
- Memory 1

- Processor 2
- Cache
- Memory 2

... [repeated M-1 times]

- Processor N
- Cache
- Memory M

Interconnection Network
Memory Consistency in SMPs

CPU-1

cache-1

CPU-2

cache-2

CPU-Memory bus

memory
• Suppose CPU-1 updates A to 200
  ‣ write-back: memory and cache-2 have stale values
  ‣ write-through: cache-2 has a stale value
Memory Consistency in SMPs

- Suppose CPU-1 updates A to 200
  - Do these stale values matter?
  - What is the view of shared memory for programming?
### Write-back Caches & SC

**prog T1**
- **ST X, 1**
- **ST Y, 11**

**cache-1**
- **X = 1**
- **Y = 11**

**memory**
- **X = 0**
- **Y = 10**
- **X' =**
- **Y' =**

**cache-2**
- **Y' =**
- **X =**
- **X' =**

---

**prog T2**
- **LD Y, R1**
- **ST Y', R1**
- **LD X, R2**
- **ST X', R2**

**cache-1**
- **X = 1**
- **Y = 11**
- **X' =**
- **Y' =**

**cache-2**
- **Y' =**
- **X =**
- **X' =**

---

- **T1 is executed**
- cache-1 writes back Y
- **T2 executed**
- cache-1 writes back X
- cache-2 writes back X' & Y'

**incoherent**
### Write-through Caches & SC

#### prog T1

<table>
<thead>
<tr>
<th>cache-1</th>
<th>memory</th>
<th>cache-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST X, 1</td>
<td>X = 0</td>
<td>Y = 10</td>
</tr>
<tr>
<td>ST Y, 11</td>
<td>Y' =</td>
<td>X' =</td>
</tr>
</tbody>
</table>

- **T1 is executed**

<table>
<thead>
<tr>
<th>cache-1</th>
<th>memory</th>
<th>cache-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>X = 1</td>
<td>X = 1</td>
<td>Y = 11</td>
</tr>
<tr>
<td>Y = 11</td>
<td>Y' =</td>
<td>X' = 0</td>
</tr>
</tbody>
</table>

- **T2 executed**

<table>
<thead>
<tr>
<th>cache-1</th>
<th>memory</th>
<th>cache-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>X = 1</td>
<td>X = 1</td>
<td>Y = 11</td>
</tr>
<tr>
<td>Y = 11</td>
<td>Y' =</td>
<td>X' = 0</td>
</tr>
</tbody>
</table>

- **Write-through caches don’t preserve sequential consistency either**

```
prog T1
ST X, 1
ST Y, 11

prog T2
LD Y, R1
ST Y', R1
LD X, R2
ST X', R2
```
Maintaining Sequential Consistency

• Motivation: We can do without locks -- SC is sufficient for writing producer-consumer and mutual exclusion codes (e.g., Dekker)

• Problem: Multiple copies of a location in various caches can cause SC to break down.

• Hardware support is required 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
    • cache coherence protocols
A System with Multiple Caches

• Modern systems often have hierarchical caches
• Each cache has exactly one parent but can have zero or more children
• Only a parent and its children can communicate directly
• **Inclusion property** is maintained between a parent and its children, i.e.,
  
  ‣ a in $L_i$ implies that a in $L_{i+1}$
Cache Coherence Protocols for SC

• write request:
  ‣ the address is invalidated in all other caches before the write is performed, or
  ‣ the address is updated in all other caches after 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
  ‣ we will focus on Invalidation protocols as opposed to Update protocols
Warmup: Parallel I/O

- DMA stands for Direct Memory Access

Either Cache or DMA can be the Bus Master and effect transfers

Page transfers occur while the Processor is running
Problems with Parallel I/O

• Memory → Disk: Physical memory may be stale if cache copy is dirty

• Disk → Memory: Cache may have data corresponding to the memory
Snoopy Cache Goodman 1983

- Idea: have the cache watch (or snoop upon) DMA transfers, and then “do the right thing”
- Snoopy cache tags are dual-ported
## Snoopy Cache Actions

<table>
<thead>
<tr>
<th>Observed Bus Cycle</th>
<th>Cache State</th>
<th>Cache Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Cycle, i.e., Memory → Disk</td>
<td>Address not cached</td>
<td>No action</td>
</tr>
<tr>
<td></td>
<td>Cached, unmodified</td>
<td>No action</td>
</tr>
<tr>
<td></td>
<td>Cached, modified</td>
<td>Cache intervenes</td>
</tr>
<tr>
<td>Write Cycle, i.e., Disk → Memory</td>
<td>Address not cached</td>
<td>No action</td>
</tr>
<tr>
<td></td>
<td>Cached, unmodified</td>
<td>Cache purges its copy</td>
</tr>
<tr>
<td></td>
<td>Cached, modified</td>
<td>???</td>
</tr>
</tbody>
</table>
Shared Memory Multiprocessor

- Use snoopy mechanism to keep all processors’ view of memory coherent
Cache State Transition Diagram

- The MSI protocol

Each cache line has a tag

<table>
<thead>
<tr>
<th>State Bits</th>
<th>Address Tag</th>
</tr>
</thead>
</table>

M: Modified
S: Shared
I: Invalid

Read miss

Other processor reads
P₁ writes back

P₁ intends to write

Other processor intends to write
P₁ writes back

Cache state in processor P₁

Write miss

Read by any processor
2 Processor Example

P₁ reads
P₂ reads
P₁ writes
P₂ writes
P₁ reads
P₁ writes
P₂ writes
P₂ writes
• If a line is in the M state then no other cache can have a copy of the line!
  ∗ Memory stays coherent, multiple differing copies cannot exist
MESI: An Enhanced MSI protocol

Each cache line has a tag

- **M**: Modified Exclusive
- **E**: Exclusive, unmodified
- **S**: Shared
- **I**: Invalid

<table>
<thead>
<tr>
<th>state bits</th>
<th>Address tag</th>
</tr>
</thead>
</table>

- **P₁ write or read**
- **Other processor reads**
  - **P₁ writes back**
- **Read miss, shared**
- **Read by any processor**

- **P₁ write**
- **P₁ intent to write**
- **P₁ read**
- **Write miss**
  - **Other processor intent to write**
- **Read miss, not shared**

Cache state in processor P₁
2 Processor Example

Block b

P₁ write or read

P₁ write

P₂ reads,
P₁ writes back

Read miss

P₂ reads

P₂ write

P₁ intent to write

P₁ writes back

P₁ writes back

P₂ writes back

P₂ intent to write

P₁ read

Write miss

Block b

P₂ write or read

P₂ write

P₂ writes back

P₂ reads

P₁ reads,
P₂ writes back

Read miss

P₂ writes back

P₁ writes back

P₁ writes back

P₂ writes back

P₁ intent to write

P₂ intent to write

P₁ reads

Write miss

P₂ read

Write miss

P₁ reads

Write miss
Valid and dirty bits can be used to encode S, I, and (E, M) states
- V=0, D=x → Invalid
- V=1, D=0 → Shared (not dirty)
- V=1, D=1 → Exclusive (dirty)
2-Level On-chip Caches

- Inclusion property: entries in L1 must be in L2
  - invalidation in L2 $\rightarrow$ invalidation in L1
- Does snooping on L2 affect CPU-L1 bandwidth?
  - yes -- to check if a dirty copy is stored in L1
- How can this be avoided?
  - Write-through L1 cache
• When a read-miss for A occurs in cache-2, a read request for A is placed on the bus

  ▸ Cache-1 needs to supply & change its state to shared
  ▸ The memory may respond to the request also!
Intervention

• Does memory know it has stale data?
  ▸ Cache-1 needs to intervene through memory controller to supply correct data to cache-2
False Sharing

• A cache block contains more than one word
• Cache-coherence is done at the block-level and not word-level
• Suppose $M_1$ writes word $i$ and $M_2$ writes word $k$ and both words have the same block address.
• What can happen?
  › Block may be invalidated many times unnecessarily
• Performance Issues

- 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.
Performance Related to Bus Occupancy

• In general, a read-modify-write instruction requires two memory (bus) operations without intervening memory operations by other processors

• In a multiprocessor setting, bus needs to be locked for the entire duration of the atomic read and write operation
  ▸ expensive for simple buses
  ▸ very expensive for split-transaction buses

• modern processors use
  ▸ load-reserve and 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> ← <l, a>;
  R ← M[a];

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

• If the snooper sees a store transaction 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
Performance

• Load-reserve & Store-conditional

  ▸ The total number of memory (bus) transactions is not necessarily reduced, but splitting an atomic instruction into load-reserve & store-conditional:

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

  • reduces cache ping-pong effect because processors trying to acquire a semaphore do not have to perform stores each time
Maintaining Cache Coherence

- Hardware support is required 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

- 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
Software Cache Coherence

• Exclude hardware support for cache coherence, e.g., Cray T3D
• Systems with caches mark shared data as uncachable
• Software can explicitly cache value of stored data
• Disadvantages?
  ‣ Compiler mechanisms for CC very limited
  ‣ Need to be conservative: every block that might be shared is treated as shared
  ‣ Doing things at the cache block level more efficient
Directory-Based Coherence

• By Censier and Feautrier, 1978

Snoopy Protocols

- Snoopy schemes broadcast requests over memory bus
  ‣ Totally ordered interconnect
- 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
- Directory may become bottleneck
Directory Protocols

• Directory keeps the state of every block that is cached
  ‣ Which caches have copies, which do not?

• Directory entries can be distributed, so different directory accesses go to different locations
  ‣ Sharing status of a block always in single known location
  ‣ Will not consider this in this class
Directory Protocol Basics

- Handling a read miss
- Handling a write to a shared, clean block
- Handling a write miss

Directory state for address A

- **S**: One or more processors have the block cached and value in memory is up to date
- **U**: No processor has a copy of the cache block
- **M**: Exactly one processor has a copy of the cache block and it has written the block
Snoopy vs. Directory

• States and transitions on the cache side are similar but actions on transition are different

• Cannot use interconnect as a single point of arbitration in directory scheme

• Interconnect is message-oriented (rather than a transaction-oriented bus) and many messages have explicit responses
Message Catalog

Local cache \( P_i \)

Read miss \( (P_i, A) \)

Write miss \( (P_i, A) \)

Directory D, Memory

Fetch (A)
Fetch/Invalidate (A)

Data reply (data)

Invalidate (A)

Remote cache \( P_k \)

Data write back \( (A, \text{data}) \)
Some Assumptions

• Attempts to write data that is not exclusive in the cache always generate write misses

• Processors block until access completes

• Messages will be received and acted upon in the same order that they are sent
  ‣ Invalidates sent by a processor are honored immediately
Each cache line has a tag

<table>
<thead>
<tr>
<th>state bits</th>
<th>Address tag</th>
</tr>
</thead>
</table>

- **M**: Modified
- **S**: Shared
- **I**: Invalid

Cache state in processor $P_1$

- **P$_1$ read miss**
- **P$_1$ write miss**
- **P$_1$ write miss**
- **P$_1$ read miss, Send write miss message**
- **Read miss or hit by any processor**
- **Invalidate from directory**
- **Fetch from directory**
- **$P_1$ writes back**
- **$P_1$ writes back**
- **Send read miss message**
- **Fetch invalidate from directory**
- **Write miss, Send write miss message**
State in directory for individual cache block

Q: Owner of address

- Read miss from P
  - Fetch from Q, data value reply
  - $S_h = S_h + \{P\}$

- Read miss from P, Data value reply
  - $S_h = S_h + \{P\}$

- Write miss from P
  - Invalidate $S_h - \{P\}$
  - Sh = \{P\}
  - (Data value reply)

- Write miss from P, Fetch/Invalidate from Q
  - Data value reply
  - $S_h = \{P\}$

- Data write back, $S_h = \{\}$
2 Processors + Directory

- **Fetch**
  - P₁ Write Back
  - P₁ read miss

- **Invalidate**
  - P₁ write miss
  - P₂ read miss
  - P₂ write miss

- **Data write back**
  - Sh = Sh + {P₂}

- **Data value reply**
  - Sh = Sh + {P₂}

- **Sh = {}**
  - Data write
  - Sh = {P₁}

- **P₁ reads**
  - P₂ reads
  - P₁ writes
  - P₂ reads
  - P₂ writes
  - P₂ evicts
Message Catalog Including Optimization

Local cache $P_i$

Write miss $(P_i, A)$

Data reply (data)

Fetch (A)

Invalidate (A)

Data write back (A, data)

Remote cache $P_k$

Read miss $(P_i, A)$

Data reply (data)

Fetch/Invalidate (A)
Two Remaining Implementation Issues

• In snoopy protocol, we assumed bus transactions were atomic
  ‣ Write miss cannot be atomic
  ‣ This assumption is impossible to maintain in a general interconnection network that is message-oriented

• Assumed infinite buffers
  ‣ Finite buffering to hold message requests and replies causes additional possibilities for deadlock
Write Misses (in Snoopy Protocol)

• Two steps in a write miss

  1. Detect the miss and request the bus
  2. Acquire the bus, place the miss on the bus, get the data and complete the write

• Do not change the block to exclusive or allow the cache update to proceed before bus is acquired

  ‣ Writes to the same cache block will serialize at second step assuming bus transactions are atomic once the bus is acquired (what does this imply?)

  • No split transactions

• 2-step write implies more complex protocol!
"Real" Snoopy Cache Protocol

The states/bits correspond to different cache blocks at different times.
Finite Buffering

- Finite buffers could cause deadlock if there are no buffers to send replies in a directory protocol
  - P1 waiting for reply from directory on write miss
  - Directory waiting for data reply from P1 on P2 read of P1 modified data
Finite Buffering

• Don’t initiate transaction unless resources are available
  ‣ Separate network for requests and replies
  ‣ Every request that expects a reply allocates space for the reply
  ‣ Controller can reject a request but never a reply
  ‣ Any request that is rejected is retried
Next Class

• Interconnect Network