EC 513
Computer Architecture

Cache Coherence - Directory Cache Coherence

Prof. Michel A. Kinsky

Department of Electrical & Computer Engineering
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

Other processor reads  
P$_1$ writes back

P$_1$ reads or writes  
Write miss

P$_1$ intends to write  
Other processor intends to write  
P$_1$ writes back

Read miss  
Other processor reads  
Read by any processor

Cache state in processor P$_1$
MESI: An Enhanced MSI protocol

Each cache line has a tag

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

Cache state in processor $P_1$

- **M**: $P_1$ write or read
- **E**: $P_1$ write
- **S**: $P_1$ intent to write
- **I**: $P_1$ read

Other processor reads
- $P_1$ writes back
- Read miss, shared
- Read by any processor

Other processor intent to write
- Write miss

$P_1$ writes back

$P_1$ reads

$P_1$ writes, not shared

Other processor

$P_1$ intent to write

Cache state in processor $P_1$
2 Processor Example

Block b

P₁

P₁ write or read
P₂ reads, P₁ writes back
Read miss

P₁ write

S

P₂ intent to write

P₁ intent to write

I

P₂ intent to write

P₁ write

E

P₁ read

Write miss

P₂ write or read
P₁ reads, P₂ writes back
Read miss

P₂ write

S

P₂ intent to write

P₁ intent to write

I

P₂ read

Write miss

P₁ intent to write

P₂ writes back

P₁ writes back

Block b

P₂

Department of Electrical & Computer Engineering
Cache Coherence State Encoding

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!
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
Synchronization and Caches

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):**
  
  \[ <\text{flag}, \text{adr}> \leftarrow <1, a>; \]
  
  \[ R \leftarrow M[a]; \]

  **Store-conditional (a), R:**
  
  \[ \text{if } <\text{flag}, \text{adr}> = <1, a> \]
  
  \[ \text{then cancel other procs’ reservation on } a; \]
  
  \[ M[a] \leftarrow <R>; \]
  
  \[ \text{status } \leftarrow \text{succeed}; \]
  
  \[ \text{else status } \leftarrow \text{fail}; \]

- 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 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-Based Coherence

- By Censier and Feautrier, 1978

- 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 \((P_i, A)\)

Remote cache \( P_k \)

Write miss \((P_i, A)\)

Data reply \( (data) \)

Directory D, Memory

Fetch \((A)\)

Fetch/Invalidate \((A)\)

Invalidation \((A)\)

Data write back \((A, data)\)

Data reply \( (data) \)
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

| state bits | Address tag |

M: Modified
S: Shared
I: Invalid

Cache state in processor $P_1$

- **M** (Modified)
  - $P_1$ reads or writes
  - $P_1$ write miss

- **S** (Shared)
  - $P_1$ read miss
  - Send write miss
  - Invalidate from directory

- **I** (Invalid)
  - $P_1$ read miss, Send read miss message
  - Write miss, Send write miss message
  - Invalidate from directory
  - $P_1$ writes back

Fetch from directory
$P_1$ writes back

Fetch invalidate from directory
$P_1$ writes back
State in directory for individual cache block
Q: Owner of address

Read miss from P
Fetch from Q, data value reply
Sh = Sh + {P}

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

Data write back, Sh = {}

Read miss from P
Data value reply, Sh = {P}
2 Processors + Directory

- **Fetch**: P₁ Write Back
- **P₁ read miss**: P₁ write miss
- **Sh = Sh + {P₂}**
- **Data value reply**
- **Sh = Sh + {P₂}**
- **Sh = Sh + {P₁}**
- **Invalidate**: P₁ reads P₂ reads P₁ writes P₂ writes P₂ evicts
Message Catalog Including Optimization

Local cache $P_i$

Read miss $(P_i, A)$

Write miss $(P_i, A)$

Directory $D$

Fetch $(A)$

Fetch/Invalidate $(A)$

Invalidate $(A)$

Remote cache $P_k$

Data write back $(A, data)$

Data reply $(data)$

Data reply $(A, data)$

Data reply $(data)$
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.

- **PWB**: Pending Write Back
- **PR**: Pending Read
- **PWM**: Pending Write Miss

- **I**: Initial state
  - P1 read
  - Bus avail WB block
- **M**: Mr. Micawber
  - P1 read, write
  - Bus avail WM block on bus
- **PWB1**: Pending Write Block
  - P1 read miss (diff block)
  - Pk write miss
- **PWB2**: Pending Write Block
  - P1 write
  - P1 read
  - Bus avail WB block
- **PWB3**: Pending Write Block
  - P1 read
  - Bus avail WB block
  - P1 write
  - Miss (diff block)
- **PWM**: Pending Write Miss
  - Bus avail WB block
  - P1 write
  - P1 read miss (diff block)
- **S**: Shared
  - P1 read
  - Bus avail WB block
  - PR: Pending Read
  - PWM: Pending Write Miss

- **PK**: Pending Miss
- **WB**: Write Block
- **WM**: Write Miss
- **RM**: Read Miss
- **diff block**: Different block

Department of Electrical & Computer Engineering
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
- 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

- On-chip Network Architectures