Cache coherence, etc...
Outline

- The problem of cache coherency
- Examples
The Problem of Cache Coherence

- Shared-Memory Architectures cache both **private data** (used by a single processor) and **shared data** (used by multiple processors to provide communication).
- When shared data are cached, the shared value may be replicated in multiple caches.
- In addition to the reduction in access latency and required memory bandwidth, this replication provides a reduction of shared data contention read by multiple processors simultaneously.
- Private processor caches create a problem
  - Copies of a variable can be present in multiple caches
  - A write by one processor may not become visible to others
- The use of multiple copies of same data introduces a new problem: **cache coherence**.
### Example: 2 CPUs with write-through caches

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
<th>Cache content CPU A</th>
<th>Cache content CPU B</th>
<th>Memory Content for location X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>CPU A reads X</td>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>CPU B reads X</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>CPU A stores 0 into X</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Example Cache Coherence Problem

- Things to note:
  - Processors see different values for u after event 3
  - With write back caches, value written back to memory depends on happenstance of which cache flushes or writes back value
    - Processes accessing main memory may see very stale value
  - Unacceptable to programs, and frequent!
The Problem of Cache Coherence

- Alternatively, accesses to shared data could be forced always to go around the cache to main memory ⇒ slow solution that requires a very high bus bandwidth.
- Maintain coherency has two components: read and write.
- Multiple copies are not a problem when reading, but a processor must have exclusive access to write a word.
- Processors must have the most recent copy when reading an object, so all processors must get new values after a write.
- Coherency protocols must locate all the caches that share an object to be written.
- A write to a shared data can cause either to invalidate all other copies or to update the shared copies.
What Does Coherency Mean?

- Informally:
  - “Any read must return the most recent write”
  - Too strict and too difficult to implement

- Better:
  - “Any write must eventually be seen by a read”
  - All writes are seen in proper order ("serialization")

- Two rules to ensure this:
  1. If P writes x and P1 reads it, P’s write will be seen by P1 if the read and write are sufficiently far apart and no other writes to x occur between the two accesses.
  2. Writes to a single location are serialized: two writes to the same location by any two processors are seen in the same order by all processors.
     - Latest write will be seen
     - Otherwise could see writes in illogical order (could see older value after a newer value)
When a written value will be seen?

- We cannot require that a read of x by P1 can instantaneously see the write of x by another processor that precedes by a small amount of time.

- Problem of memory consistency.

- **Coherence** and **consistency** are complementary:
  - Coherence defines the behavior of reads and writes to the same memory location.
  - Consistency defines the behavior of reads and writes with respect to accesses to other memory locations.

- **Assumptions for now:**
  - A write does not complete (and allow the next write to occur) until all processors have seen the effect of that write.
  - The processor does not change the order of any write with respect to any other memory access.
    - If a processor writes A followed by B any processor that sees the new value of B must also see the new value of A.
Coherent caches

- A program running on multiple processors will normally have copies of the same data in several caches.
- In a coherent multiprocessor the caches provide both migration and replication of shared data items.
- Migration: a data item can be moved to a local cache and used there in a transparent fashion
  - Reduces both the latency to access a shared data item that is allocated remotely and the bandwidth demand on the shared memory
- Replication for shared data that are being simultaneously read: caches make a copy of the data item in the local cache
  - Reduces both latency of access and contention for a read shared
Potential Solutions

- HW-based solutions to maintain coherency: Cache-Coherence Protocols

- Key issues to implement a cache coherent protocol in multiprocessors is tracking the status of any sharing of a data block.

- Two classes of protocols:
  - Snooping Protocols
  - Directory-Based Protocols
Snooping Protocols (Snoopy Bus)

- All cache controllers monitor (snoop) on the bus to determine whether or not they have a copy of the block requested on the bus and respond accordingly.
- Every cache that has a copy of the shared block, also has a copy of the sharing status of the block, and no centralized state is kept.
- Send all requests for shared data to all processors.
- Require broadcast, since caching info is at processors.
- Suitable for **Centralized Shared-Memory Architectures**, and in particular for small scale multiprocessors with single shared bus.
Snoopy Cache-Coherence Protocols

- Bus is a broadcast medium & Caches know what they have
- Cache Controller “snoops” all transactions on the shared bus
  - relevant transaction if for a block it contains
  - take action to ensure coherence
    - invalidate, update, or supply value
  - depends on state of the block and the protocol
Snooping Protocols (Snoopy Bus)

- Since every bus transaction checks the cache address tags, this checking can interfere with the processor operations.
- When there is interference, the processor will likely stall because the cache is unavailable.
- To reduce the interference with the processor’s accesses to the cache, we duplicate the address tag portion of the cache (not the whole cache) for snooping activities.
- In practice, an extra read port is added to the address tag portion of the cache.
Snoop Tag

Processor

Snoop Tag

Cache Tag and Data

Processor

Snoop Tag

Cache Tag and Data

... Processor

Snoop Tag

Cache Tag and Data

Snoop Tag

Snoop Tag

Snoop Tag

Cache Tag and Data

Single BUS

Memory

I/O
Basic Snooping Protocols

- Snooping Protocols are of two types depending on what happens on a write operation:
  - Write-Invalidate Protocol
  - Write-Update or Write-Broadcast Protocol
Write-Invalidate Protocol

- The writing processor issues an invalidation signal over the bus to cause all copies in other caches to be invalidated before changing its local copy.
- The writing processor is then free to update the local data until another processor asks for it.
- All caches on the bus check to see if they have a copy of the data and, if so, they must invalidate the block containing the data.
- This scheme allows multiple readers but only a single writer.
Write-Invalidate Protocol

- This scheme uses the bus only on the first write to invalidate the other copies.
- Subsequent writes do not result in bus activity.
- This protocol provides similar benefits to write-back protocols in terms of reducing demands on bus bandwidth.

Read Miss:
- Write-Through: Memory always up-to-date
- Write-Back: Snoop in caches to find the most recent copy.
Write-through Invalidate Protocol

- Basic Bus-Based Protocol
  - Each processor has cache, state
  - All transactions over bus snooped
- Writes invalidate all other caches
  - can have multiple simultaneous readers of block, but write invalidates them
- Two states per block in each cache
  - as in uniprocessor
  - state of a block is a $p$-vector of states
  - Hardware state bits associated with blocks that are in the cache
  - other blocks can be seen as being in invalid (not-present) state in that cache

![Diagram of Write-through Invalidate Protocol]

<table>
<thead>
<tr>
<th>State</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>PrRd</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BusRd</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>PrWr</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BusWr</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>PrWr</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BusWr</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>PrRd</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BusRd</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: Write-thru Invalidate

$P_1$

$u = ?$

$4$

$u : 5$

$I/O$ devices

$P_2$

$u = ?$

$5$

$u : 5$

$P_3$

$3$

$u : 5 u = 7$

$1$

$u : 5$

$u = 7$

Memory

$2$
Write-through vs. Write-back

- Write-through protocol is simple
  - every write is observable
- Every write goes on the bus
  \[\Rightarrow\] Only one write can take place at a time in any processor
- Uses a lot of bandwidth!

Example: 200 MHz dual issue,
CPI = 1, 15% stores of 8 bytes
\[\Rightarrow\] 30 M stores per second per processor
\[\Rightarrow\] 240 MB/s per processor

1GB/s bus can support only about 4 processors without saturating
Write-Update Protocol

- The writing processor broadcasts the new data over the bus; all caches check if they have a copy of the data and, if so, all copies are updated with the new value.
- This scheme requires the continuous broadcast of writes to shared data (while write-invalidate deletes all other copies so that there is only one local copy for subsequent writes)
- This protocol is like write-through because all writes go over the bus to update copies of the shared data.
- This protocol has the advantage of making the new values appear in caches sooner ⇒ reduced latency
- Read Miss: Memory always up-to-date.
Invalidate vs. Update

- Basic question of program behavior:
  - Is a block written by one processor later read by others before it is overwritten?

- Invalidate
  - yes: readers will take a miss
  - no: multiple writes without addition traffic
    - also clears out copies that will never be used again

- Update
  - yes: avoids misses on later references
  - no: multiple useless updates

- Need to look at program reference patterns and hardware complexity
- Can we tune this automatically???
  but first - correctness
Snooping Protocols

- Most part of commercial cache-based multiprocessors uses:
  - **Write-Back Caches** to reduce bus traffic ⇒ they allow more processors on a single bus.
  - **Write-Invalidate Protocol** to preserve bus bandwidth

- Write serialization due to bus serializing request: bus is single point of arbitration
  - A write to a shared data item cannot actually complete until it obtains bus access
Write back cache

- How to identify the most recent data value of a cache block in case of cache miss?
  - It can be in a cache rather in a memory

- Can use the same snooping scheme both for cache misses and writes
  - Each processor snoops every address placed on the bus
  - If a processor finds that it has a dirty copy of the requested cache block, it provides the cache block in response to the read request
  - memory access is aborted
Snooping Protocols: An Example

- Write-Invalidate Protocol, Write-Back Cache
- Each block of memory is in one of three states:
  - **Clean** in all caches and up-to-date in memory (Shared)
  - OR **Dirty** in exactly one cache (Exclusive)
  - OR Not in any caches
- Each cache block can be in one of three states:
  - **Clean (or Shared)** (read only): the block is clean (not modified) and can be read
  - **Dirty (or Modified or Exclusive)**: cache has only copy, its writeable, and dirty (block cannot be shared)
  - **Invalid**: block contains no valid data
### MSI Invalidate Protocol

- **Three States:**
  - “M”: “Modified”
  - “S”: “Shared”
  - “I”: “Invalid”

- **Read obtains block in “shared”**
  - even if only cache copy

- **Obtain exclusive ownership before writing**
  - BusRdx causes others to invalidate (demote)
  - If M in another cache, will flush
  - BusRdx even if hit in S
    - promote to M (upgrade)

- **What about replacement?**
  - S->I, M->I
Snooping Cache Variations

- MESI Protocol: Write-Invalidate
- Each cache block can be in one of four states:
  - Modified: the block is dirty and cannot be shared; cache has only copy, its writeable.
  - Exclusive: the block is clean and cache has only copy;
  - Shared: the block is clean and other copies of the block are in cache;
  - Invalid: block contains no valid data
- Add exclusive state to distinguish exclusive (writable) and owned (written)
All cache controllers snoop on BusRd
Assert ‘shared’ if present (S? E? M?)
Issuer chooses between S and E
# States of cache lines with MESI

<table>
<thead>
<tr>
<th></th>
<th>Modified</th>
<th>Exclusive</th>
<th>Shared</th>
<th>Invalid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line valid?</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Copy in memory...</td>
<td>Has to be updated</td>
<td>Valid</td>
<td>Valid</td>
<td>-</td>
</tr>
<tr>
<td>Other copies in other caches?</td>
<td>No</td>
<td>No</td>
<td>Maybe</td>
<td>Maybe</td>
</tr>
<tr>
<td>A write on this line...</td>
<td>Access the BUS</td>
<td>Access the BUS</td>
<td>Access the BUS and Update the cache</td>
<td>Direct access to the BUS</td>
</tr>
</tbody>
</table>
**MESI State Transition Diagram**

- **BusRd(S)** means shared line asserted on BusRd transaction
- **Flush’**: if cache-to-cache xfers
  - only one cache flushes data
- **Replacement**:
  - S→I can happen without telling other caches
  - E→I, M→I
MESI State Transition Diagram V2

Cache line in the "acting" processor

Transaction due to events snooped on the common BUS

BUS Transactions

- RH = Read Hit
- RMS = Read Miss, Shared
- RME = Read Miss, Exclusive
- WH = Write Hit
- WM = Write Miss
- SHR = Snoop Hit on a Read
- SHW = Snoop Hit on a Write or Read-with-Intent-to-Modify

= Snoop Push
× = Invalidate Transaction
= Read-with-Intent-to-Modify
= Cache Block Fill
In both S and E, the memory has an up-to-date version of the data.

A write to a E block does not require to send the invalidation signal on the bus, since no other copies of the block are in cache.

A write to a S block implies the invalidation of the other copies of the block in cache.
The Problem of Memory Consistency

What is consistency? When must a processor see the new value of a data updated by another processor?

P1:  A = 0;  
     .....  
     A = 1; 

P2:  B = 0;  
     .....  
     B = 1; 

L1:  if (B == 0) ...  

L2:  if (A == 0) ...

Impossible for both if statements L1 & L2 to be true?

What if write invalidate is delayed & processor continues?

Memory consistency models: what are the rules for such cases?
The Problem of Memory Consistency

- In what order must a processor observe the data writes of another processor?
- **Sequential consistency**: result of any execution is the same as if the accesses of each processor were kept in order and the accesses among different processors were interleaved.
- The simplest way to implement sequential consistency is to require a processor to delay the completion of any memory access until all the invalidations caused by that access are completed.
The Problem of Memory Consistency

- Schemes faster execution to sequential consistency
- Not really an issue for most programs; they are **synchronized**
  - A program is synchronized if all access to shared data are ordered by synchronization operations
    - write (x)
    - ...
    - release (s) \{unlock\}
    - ...
    - acquire (s) \{lock\}
    - ...
    - read(x)
Taxonomy of Large Multiprocessors

Larger multiprocessors

Shared address space

- Symmetric shared memory (SMP)
  - Examples: IBM eserver, SUN Sunfire

- Distributed shared memory (DSM)

Distributed address space

- Commodity clusters: Beowulf and others
- Custom cluster
  - Uniform cluster: IBM BlueGene
  - Constellation cluster of DSMs or SMPs
    - SGI Altix, ASC Purple

Cache coherent:
- ccNUMA:
  - SGI Origin/Altix

Noncache coherent:
- Cray T3E, X1
Portable Parallel Programming?

- Most large scale commercial installations emphasize throughput
  - database servers, web servers, file servers
  - independent transactions

- Wide variety of parallel systems
  - message passing
  - shared memory
  - shared memory within node, message passing between nodes

⇒ *Little commercial software support for portable parallel programming*

Message Passing Interface (MPI) standard widely used for portability
  - lowest common denominator
  - “assembly” language level of parallel programming

- Target market is wireless basestations
- 430 cores on one die in 130nm
- Each core is a 3-issue VLIW
The SoC is made up of 8 independent processors, up from 7 in the original Tegra.

- A pair of ARM Cortex A9 cores
- Dual-issue out of order cores from ARM running @ 1GHz
Heterogeneous Multicore

- An Intel Atom E6XX processor
  - # Cores: 1
  - # Threads: 2
  - L2 Cache: 512 KB
- An Altera Field Program
IBM Cell Processor (Playstation-3)

One 2-way threaded PowerPC core (PPE), plus eight specialized short-SIMD cores (SPE)
Each of 16 cores similar to a vector processor with 8 lanes (128 stream processors total)
- Processes threads in SIMD groups of 32 (a “warp”)
- Some stripmining done in hardware
- Threads can branch, but loses performance compared to when all threads are running same code
- Only attains high efficiency on very data-parallel code (10,000s operations)
If and how will these converge?

- General-purpose multicores organized as traditional SMPs
- Embedded manycores with exposed and customized memory hierarchies
- Biggest current issue in computer architecture - will mainly be decided by applications and programming models
End...