Shared Memory Multiprocessors

Todd C. Mowry
CS 418
February 12, 13, & 19, 2008

Topics
- The Cache Coherence Problem
- Snoopy Coherence Protocols

A Coherent Memory System: Intuition

Reading a location should return latest value written (by any process)

Easy in uniprocessors
- Except for I/O: coherence between I/O devices and processors
- But infrequent so software solutions work
  - uncachable operations, flush pages, pass I/O data through caches

Would like same to hold when processes run on different processors
- E.g. as if the processes were interleaved on a uniprocessor

The coherence problem is more pervasive and performance-critical in multiprocessors
- has a much larger impact on hardware design

Problems with the Intuition

Recall:
- Value returned by read should be last value written
But "last" is not well-defined!

Even in sequential case:
- "last" is defined in terms of program order, not time
  - Order of operations in the machine language presented to processor
  - "Subsequent" defined in analogous way, and well defined

In parallel case:
- program order defined within a process, but need to make sense of orders across processes

Must define a meaningful semantics
- the answer involves both "cache coherence" and an appropriate "memory consistency model" (to be discussed in a later lecture)
Formal Definition of Coherence

Results of a program: values returned by its read operations
A memory system is coherent if the results of any execution of a program are such that for each location, it is possible to construct a hypothetical serial order of all operations to the location that is consistent with the results of the execution and in which:
1. operations issued by any particular process occur in the order issued by that process, and
2. the value returned by a read is the value written by the last write to that location in the serial order

Two necessary features:
- Write propagation: value written must become visible to others
- Write serialization: writes to location seen in same order by all
  - if I see w1 after w2, you should not see w2 before w1
  - no need for analogous read serialization since reads not visible to others

Cache Coherence Solutions

Software Based:
- often used in clusters of workstations or PCs (e.g., “Treadmarks”)
- extend virtual memory system to perform more work on page faults
  - send messages to remote machines if necessary

Hardware Based:
- two most common variations:
  - “snoopy” schemes
    » rely on broadcast to observe all coherence traffic
    » well suited for buses and small-scale systems
    » example: SGI Challenge
  - directory schemes
    » uses centralized information to avoid broadcast
    » scales well to large numbers of processors
    » example: SGI Origin 2000

Shared Caches

- Processors share a single cache, essentially punting the problem.
- Useful for very small machines.
  - Problems are limited cache bandwidth and cache interference
  - Benefits are fine-grain sharing and prefetch effects

Snoopy Cache Coherence Schemes

Basic Idea:
- all coherence-related activity is broadcast to all processors
  - e.g., on a global bus
  - each processor (or its representative) monitors (aka “snoops”) these actions and reacts to any which are relevant to the current contents of its cache
  - examples:
    » if another processor wishes to write to a line, you may need to “invalidate” (i.e. discard) the copy in your own cache
    » if another processor wishes to read a line for which you have a dirty copy, you may need to supply

Most common approach in commercial multiprocessors.
Examples:
- SGI Challenge, SUN Enterprise, multiprocessor PCs, etc.
Implementing a Snoopy Protocol

Cache controller now receives inputs from both sides:
• Requests from processor, bus requests/responses from snooper
In either case, takes zero or more actions
• Updates state, responds with data, generates new bus transactions
Protocol is a distributed algorithm: cooperating state machines
• Set of states, state transition diagram, actions
Granularity of coherence is typically a cache block
• Like that of allocation in cache and transfer to/from cache

Coherence with Write-through Caches

• Key extensions to uniprocessor: snooping, invalidating/updating caches
  - no new states or bus transactions in this case
  - invalidation- versus update-based protocols
• Write propagation: even in inval case, later reads will see new value
  - inval causes miss on later access, and memory up-to-date via write-through

Write-through State Transition Diagram

• Two states per block in each cache, as in uniprocessor
  - state of a block can be seen as p-vector
• Hardware state bits associated with only blocks that are in the cache
  - other blocks can be seen as being invalid (not-present) state in that cache
• Write will invalidate all other caches (no local change of state)
  - can have multiple simultaneous readers of block, but write invalidates them

Problem with Write-Through

High bandwidth requirements
• Every write from every processor goes to shared bus and memory
• Consider a 3GHz, 1CPI processor, where 15% of instructions are 8-byte stores
• Each processor generates 450M stores or 3.6GB data per second
• 5GB/s bus can support only 1 processor without saturating
• Write-through especially unpopular for SMPs
Write-back caches absorb most writes as cache hits
• Write hits don’t go on bus
• But how do we ensure write propagation and serialization?
• Need more sophisticated protocols: large design space
Write-Back Snoopy Protocols

No need to change processor, main memory, cache...
- Extend cache controller and exploit bus (provides serialization)

Dirty state now also indicates exclusive ownership
- Exclusive: only cache with a valid copy (main memory may be too)
- Owner: responsible for supplying block upon a request for it

Design space
- Invalidation versus Update-based protocols
- Set of states

Invalidation-Based Protocols

"Exclusive" state means can modify without notifying anyone else
- i.e., without bus transaction
- Must first get block in exclusive state before writing into it
- Even if already in valid state, need transaction, so called a write miss

Store to non-dirty data generates a read-exclusive bus transaction
- Tells others about impending write, obtains exclusive ownership
  - makes the write visible, i.e., write is performed
  - may be actually observed (by a read miss) only later
  - write hit made visible (performed) when block updated in writer's cache
- Only one RdX can succeed at a time for a block: serialized by bus

Read and Read-exclusive bus transactions drive coherence actions
- Writeback transactions also, but not caused by memory operation and quite incidental to coherence protocol
  - note: replaced block that is not in modified state can be dropped

Update-Based Protocols

A write operation updates values in other caches
- New, update bus transaction

Advantages
- Other processors don't miss on next access: reduced latency
  - In invalidation protocols, they would miss and cause more transactions
- Single bus transaction to update several caches can save bandwidth
  - Also, only the word written is transferred, not whole block

Disadvantages
- Multiple writes by same processor cause multiple update transactions
  - In invalidation, first write gets exclusive ownership, others local

Detailed tradeoffs more complex

Invalidation versus Update

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

Invalidation:
- Yes => readers will take a miss
- No => multiple writes without additional traffic
  - and clears out copies that won't be used again

Update:
- Yes => readers will not miss if they had a copy previously
  - single bus transaction to update all copies
- No => multiple useless updates, even to dead copies

Need to look at program behavior and hardware complexity

Invalidation protocols much more popular
- Some systems provide both, or even hybrid
Basic MSI Writeback Inval Protocol

States
- Invalid (I)
- Shared (S): one or more
- Dirty or Modified (M): one only

Processor Events:
- PrRd (read)
- PrWr (write)

Bus Transactions
- BusRd: asks for copy with no intent to modify
- BusRdX: asks for copy with intent to modify
- BusWB: updates memory

Actions
- Update state, perform bus transaction, flush value onto bus

Satisfying Coherence

Write propagation is clear
Write serialization?
- All writes that appear on the bus (BusRdX) ordered by the bus
  - Write performed in writer's cache before it handles other transactions, so ordered in same way even w.r.t. writer
- Reads that appear on the bus ordered w.r.t these
- Writes that don't appear on the bus:
  - sequence of such writes between two bus actions for the block must come from same processor, say P
  - in serialization, the sequence appears between these two bus actions
  - reads by P will see them in this order w.r.t. other bus transactions
  - reads by other processors separated from sequence by a bus action, which places them in the serialized order w.r.t. the writes
  - so reads by all processors see writes in same order

Lower-Level Protocol Choices

BusRd observed in M state: what transition to make?

Depends on expectations of access patterns
- S: assumption that I’ll read again soon, rather than other will write
- good for mostly read data
- what about “migratory” data
  - I read and write, then you read and write, then X reads and writes...
  - better to go to I state, so I don’t have to be invalidated on your write
- Synapse transitioned to I state
- Sequent Symmetry and MIT Alewife use adaptive protocols

Choices can affect performance of memory system
**MESI (4-state) Invalidation Protocol**

Problem with MSI protocol
- Reading and modifying data is 2 bus transactions, even if no sharing
  - e.g. even in sequential program
  - BusRd (I→S) followed by BusRdX or BusUpgr (S→M)

Add exclusive state: write locally without transaction, but not modified
- Main memory is up to date, so cache not necessarily owner
- States
  - invalid
  - exclusive or exclusive-clean (only this cache has copy, but not modified)
  - shared (two or more caches may have copies)
  - modified (dirty)
- I → E on PrRd if no other processor has a copy
  - needs "shared" signal on bus: wired-or line asserted in response to BusRd

**MESI State Transition Diagram**

- BusRd(S) means shared line asserted on BusRd transaction
- Flush': if cache-to-cache sharing (see next), only one cache flushes data
- MOESI protocol: Owned state: exclusive but memory not valid

**Lower-level Protocol Choices**

Who supplies data on miss when not in M state: memory or cache?
- Original, Illinois MESI: cache, since assumed faster than memory
  - Cache-to-cache sharing
Not true in modern systems
- Intervening in another cache more expensive than getting from memory
- Cache-to-cache sharing also adds complexity
  - How does memory know it should supply data (must wait for caches)
  - Selection algorithm if multiple caches have valid data
But valuable for cache-coherent machines with distributed memory
- May be cheaper to obtain from nearby cache than distant memory
- Especially when constructed out of SMP nodes (Stanford DASH)

**Dragon Write-Back Update Protocol**

4 states
- Exclusive-clean or exclusive (E): I and memory have it
- Shared clean (Sc): I, others, and maybe memory, but I’m not owner
- Shared modified (Sm): I and others but not memory, and I’m the owner
  - Sm and Sc can coexist in different caches, with only one Sm
- Modified or dirty (D): I and nobody else

No invalid state
- If in cache, cannot be invalid
- If not present in cache, can view as being in not-present or invalid state

New processor events: PrRdMiss, PrWrMiss
- Introduced to specify actions when block not present in cache
New bus transaction: BusUpd
- Broadcasts single word written on bus; updates other relevant caches
**Lower-level Protocol Choices**

**Can shared-modified state be eliminated?**
- If update memory as well on BusUpd transactions (DEC Firefly)
- Dragon protocol doesn’t (assumes DRAM memory slow to update)

**Should replacement of an Sc block be broadcast?**
- Would allow last copy to go to E state and not generate updates
- Replacement bus xaction is not in critical path, later update may be
- Shouldn’t update local copy on write hit before controller gets bus
  - Can mess up serialization

Consistency considerations much like write-through case

In general, many subtle race conditions in protocols
But first, let’s illustrate quantitative assessment at logical level

**Assessing Protocol Tradeoffs**

Tradeoffs affected by performance and organization characteristics
Decisions affect pressure placed on these
Part art and part science
- Art: experience, intuition, and aesthetics of designers
- Science: Workload-driven evaluation for cost-performance
  - Want a balanced system: no expensive resource heavily underutilized

**Methodology:**
- Use simulator; choose parameters per earlier methodology (default 1MB, 4-way cache, 64-byte block, 16 processors; 64K cache for some)
- Focus on frequencies, not end performance for now
  - Transcends architectural details, but not what we’re really after
- Use idealized memory performance model to avoid changes of reference
  - Transcends architectural details, but not what we’re really after
- Use idealized memory performance model to avoid changes of reference
  - Transcends architectural details, but not what we’re really after
- Use idealized memory performance model to avoid changes of reference
  - Transcends architectural details, but not what we’re really after
- Use idealized memory performance model to avoid changes of reference
  - Transcends architectural details, but not what we’re really after

**Impact of Protocol Optimizations**

(Computing traffic from state transitions discussed in book)
Effect of E state, and of BusUpgr instead of BusRdX

- MSI versus MESI doesn’t seem to matter for bw for these workloads
- Upgrades instead of read-exclusive helps
- Same story when working sets don’t fit for Ocean, Radix, Raytrace
Impact of Cache Block Size

Multiprocessors add new kind of miss to cold, capacity, conflict
- Coherence misses: true sharing and false sharing
  - latter due to granularity of coherence being larger than a word
- Both miss rate and traffic matter

Reducing misses architecturally in invalidation protocol
- Capacity: enlarge cache; increase block size (if spatial locality)
- Conflict: increase associativity
- Cold and Coherence: only block size

Increasing block size has advantages and disadvantages
- Can reduce misses if spatial locality is good
- Can hurt too
  - increase misses due to false sharing if spatial locality not good
  - increase misses due to conflicts in fixed-size cache
  - increase traffic due to fetching unnecessary data and due to false sharing
  - can increase miss penalty and perhaps hit cost

Impact of Block Size on Miss Rate

Results shown only for default problem size: varied behavior
- Need to examine impact of problem size and p as well (see text)

Cold
Capacity
True sharing
False sharing
Upgrade

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>Result</th>
<th>Miss Rate (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barnes/8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Barnes/16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Barnes/32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Barnes/64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Barnes/128</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Barnes/256</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lu/8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lu/16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lu/32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lu/64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lu/128</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lu/256</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Radiosity/8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Radiosity/16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Radiosity/32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Radiosity/64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Radiosity/128</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Radiosity/256</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ocean/8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ocean/16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ocean/32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ocean/64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ocean/128</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ocean/256</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Radix/8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Radix/16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Radix/32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Radix/64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Radix/128</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Radix/256</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Raytrace/8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Raytrace/16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Raytrace/32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Raytrace/64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Raytrace/128</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Raytrace/256</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Impact of Block Size on Traffic

Traffic affects performance indirectly through contention

- Results different than for miss rate: traffic almost always increases
- When working sets fits, overall traffic still small, except for Radix
- Fixed overhead is significant component
  - So total traffic often minimized at 16-32 byte block, not smaller
- Working set doesn’t fit: even 128-byte good for Ocean due to capacity

Making Large Blocks More Effective

Software
- Improve spatial locality by better data structuring
- Compiler techniques

Hardware
- Retain granularity of transfer but reduce granularity of coherence
  - use subblocks: same tag but different state bits
  - one subblock may be valid but another invalid or dirty
- Reduce both granularities, but prefetch more blocks on a miss
- Proposals for adjustable cache block size
- More subtle: delay propagation of invalidations and perform all at once
  - But can change consistency model: discuss later in course
- Use update instead of invalidate protocols to reduce false sharing effect
Update versus Invalidate

Much debate over the years: tradeoff depends on sharing patterns

Intuition:
- If those that used continue to use, and writes between use are few, update should do better
  - e.g. producer-consumer pattern
- If those that use unlikely to use again, or many writes between reads, updates not good
  - "pack rat" phenomenon particularly bad under process migration
  - useless updates where only last one will be used

Can construct scenarios where one or other is much better
Can combine them in hybrid schemes (see text)
- E.g. competitive: observe patterns at runtime and change protocol

Let's look at real workloads

Upgrade and Update Rates (Traffic)

- Update traffic is substantial
- Main cause is multiple writes by a processor before a read by other
  - many bus transactions versus one in invalidation case
  - could delay updates or use merging
- Overall, trend is away from update based protocols as default
  - bandwidth, complexity, large blocks trend, pack rat for process migration
- Will see later that updates have greater problems for scalable systems

• Lots of coherence misses: updates help
• Lots of capacity misses: updates hurt (keep data in cache uselessly)
• Updates seem to help, but this ignores upgrade and update traffic