**Review: Shared Memory via Bus**

Private caches are necessary for performance, but can introduce coherency and consistency problems. Bus provides a global, serializing broadcast platform. This is a key enabler in building extremely useful snooping protocols.

---

**Review: Coherency**

Coherency ensures that all processors have the same view of memory.

Explicit Requirements:
- operations occur in process order
- value returned by read of address must be same as the value produced by last write to address

Implicit Requirements:
- *write propagation*- value written must become visible to all
- *write serialization*- writes to location seen in same order by all

---

**Administrative Stuff**

**Announcements**
Exam 1 will be on Thursday, 2/3/05
Homework 1 has been posted (check webpage).

**Projects**
Group proposal due 1/25/05 (Tuesday):
- group members
- problem description
- related work (brief summary)
- solution approach
- methodology
- total length <= page
Review: Sequential Consistency

Coherency alone is not enough.

*Memory consistency* constrains the order in which memory operations (with same/different address) must appear to all processors.

*Sequential consistency* requires the ordering to appear as if:
- Operations within process are executed in program order.
- Operations across processes are interleaved in some fashion (atomic operations).

Broadcasts

When a write occurs (locally), must notify other processors and main memory.

Two main options:
- **write invalidate** - store to shared block causes address only to be sent all other caches (and memory) which invalidate their own copy.
- **write update** - Store to shared block causes address and data to be sent to all other caches (and memory) which update their own copy (if present).

Design Space for Snooping Protocols

No need to drastically change processor, main memory, or cache structure.

Extend cache controller and exploit bus (provides serialization).

Design Choices:
- broadcast method (update/invalidate)
- cache states
- state transitions

Last Time: Saw write-through invalidation scheme.

Today: Focus on protocols for write-back caches.

Write-Invalidate/Write-through State Transition Diagram

Simple protocol used on write-through caches with write no allocate.

Two states per block in each cache, as in uniprocessor.

Hardware state bits associated with only blocks that are in the cache (anything else assumed invalid).

Write will invalidate all other caches (no local change of state).

So, we 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.
- Write-through especially unpopular for SMPs.

Write-back caches absorb most writes as cache hits:
- Write hits don’t go on bus.
- But now how do we ensure write propagation and serialization?
- Need more sophisticated protocols...

Closer Look at Invalidation-based Protocols

Recall: Stores to shared data cause broadcasts which invalidate other copies.
Exclusive means can modify without notifying anyone else (without bus transaction).
- Must first get block in exclusive state before writing into it.
- Even if already in valid (but not exclusive) state, need transaction.

Store to non-dirty data generates a read-exclusive bus transaction (assuming write allocate).
- 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.
- Only one RdX can succeed at a time for a block: serialized by bus.

Read and Read-exclusive bus transactions drive coherence actions.

Problem With Writeback

Writeback caches introduce dirty blocks, data which has been modified, but not propagated to rest of system.

This makes the protocol design a bit more challenging.

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

Update-based Protocols

Recall: Store to shared data broadcasts update of values to other caches.
This requires a new update bus transaction.

Comments on write update:
- Other processors don’t miss on next access following write (reduced latency).
- Single bus transaction to update several caches can save bandwidth (especially if only word is written rather than block).
- Multiple writes by same processor cause multiple update transactions.
Invalidation versus Update

Let's say a block is written by one processor and the next access is either:
(1) a read by another processor
(2) another write (by same processor)

Invalidation:
Read  => Readers will take a miss.
Write => Multiple writes without additional traffic (and old copies are cleared out).

Update:
Read  => Readers will not miss if they had a copy previously (single transaction updates all copies).
Write => Multiple useless updates (even to dead copies).

Invalidation protocols are more popular.

Let's take a look at some protocols...

Basic MSI Writeback Invalidate Protocol

States:
- Invalid (I) - entry is not valid (duh!)
- Shared (S) - one or more copies exist in caches, not modified, consistent with memory
- Dirty or Modified (M) - only copy, modified, not consistent with memory

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

MSI State Transition Diagram

Key concepts:
Modified (M) tracks dirty blocks,
Shared (S) blocks may or may not be duplicated in other caches.

Comments:
- For writes to shared block (already have latest data) can use upgrade (BusUpgr) instead of BusRdX.
- Replacement changes state of two blocks: outgoing and incoming.

MSI Example

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>$</td>
<td>$</td>
</tr>
</tbody>
</table>

I/O devices
Memory

Action
1) P1 reads $ u
2) P2 reads $ u
3) P3 writes $ u
4) P1 reads $ u
5) P2 reads $ u
**MSI: Coherence**

To satisfy coherence, must have write propagation and write serialization.

Write propagation: Easy to see.

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.
  - Hence ordered in same way even with respect to writer.
- Reads that appear on the bus ordered with respect to these.
- Write that don't appear on the bus:
  - Sequence of such writes between two bus transactions for the block must come from same processor, say P.
  - In serialization, the sequence appears between these two bus transactions.
  - Reads by P will see them in this order with respect to other bus transactions.
  - Reads by other processors separated from sequence by a bus transaction, which places them in the serialized order with respect to the writes.
  - Hence reads by all processors see writes in same order.

**MSI: Sequential Consistency**

- Bus imposes total order on bus transactions for all locations.
- Between transactions, processors perform reads/writes locally in program order.
- So any execution defines a natural partial order, M, subsequent to M, if:
  - (i) follows in program order on same processor
  - (ii) M generates bus transaction that follows the memory operation for M
- In segment between two bus transactions, any interleaving of operations from different processors leads to consistent total order.
- In such a segment, writes observed by processor P serialized as follows:
  - Writes from other processors by the previous bus transaction P issued.
  - Writes from P by program order.
- Can prove via sufficient conditions.

**MSI: 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.
- I: design choice for Synapse
  - adaptive: used by Sequent Symmetry and MIT Alewife

Choices can affect performance of memory system.

**MESI (4-state) Invalidation Protocol**

Here's a problem with MSI protocol:
- Reading and modifying data is 2 bus transactions, even if no one is sharing (even in sequential program).
- BusRd (I->S) followed by BusRdX or BusUpgr (S->M)

Solution: Add exclusive state so writes happen locally without additional transaction.

States:
- invalid (I) = same as before
- exclusive (E) = only this cache has copy (non-modified)
- shared (S) = two or more caches may have copies (non-modified)
- modified (M) = dirty blocks must be written back

Bus Signals:
- What happens on PrRd from I? Go to E? Go to S?
- Need "shared" signal on bus to determine (wired-OR line).
**MESI State Transition Diagram**

New Issues:
- BusRd(S): Means shared line asserted on BusRd transaction.
- Flush: If cache-to-cache sharing, only one cache flushes data.

MOESI protocol: Adds another state (Owner O) - 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 (introduces cache-to-cache sharing).

Not true in modern systems since 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?
- Selection algorithm needed 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 (4-state) Write-back Update Protocol**

States:
- Exclusive-clean or exclusive (E): only this cache and memory have copy
- Shared clean (Sc): copy in this cache, may others, may or may not be consistent with memory, not owner
- Shared modified (Sm): copy in this cache and at least one other, not consistent with memory, owner of block
- Modified or dirty (D): only copy in this cache, dirty

Events:
- PrRdMiss - differentiates read miss from hit
- PrWrMiss - differentiates write miss from hit
- BusUpd - broadcasts single word written onto bus

Hey, where's the invalid state (I)?
- If in cache, cannot be invalid.
- If not present in cache, can view as being in special not-present or invalid state.

**Dragon State Transition Diagram**

Key concept is tracking ownership.
Dragon Example

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:
- Simulation - default 1MB, 4-way cache, 64-byte block, 16 processors (64K cache for some)
- Focus on frequencies, not end performance for now.
- Use idealized memory performance model to avoid changes of reference interleaving across processors with machine parameters.

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 transaction 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).

Coherence, 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...

Impact of Protocol Optimizations

- 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* - actual data dependence
  - *false sharing* - due to granularity of coherence being larger than word

Reducing misses by tuning cache parameters:
- Capacity: enlarge cache, increase block size (if spatial locality present)
- 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 by:
  - increasing misses due to false sharing if spatial locality not good
  - increasing misses due to conflict in fixed-size cache
  - increasing traffic due to fetching unnecessary data and due to false sharing
  - increasing miss penalty and perhaps hit cost

Impact of Block Size on Miss Rate

Results shown only for default problem size, mileage may vary for different problem sizes and number of processors.

Working set doesn't fit: impact on capacity misses much more critical.

A Classification of Cache Misses

Traffic affects performance indirectly through contention.

- Results different than for miss rate: traffic almost always increases.
- When working sets fit, overall traffic still small, except for Radix.
- Fixed overhead is significant component (total traffic often minimized at 16-32 byte block, not smaller).
Making Large Blocks More Effective

Software
- Improve spatial locality by better data structuring.
- Compilation techniques can be aimed at this.

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 size.
- More subtle: delay propagation of invalidations and perform all at once
  (this may change consistency model).
- 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,
You can construct scenarios where one or other is much better.
You can combine them in hybrid schemes (e.g. hybrids or changing patterns at runtime).

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 due to bandwidth, complexity, large blocks trend, pack rat for process migration.
Will see later that updates have greater problems for scalable systems.
Summary

Design space for coherency protocols is rich.
Key design choice: write update or write invalidate.
Covered several protocols:
  Write-Through [Write Invalidate]
  MSI [Write Invalidate]
  MESI [Write Invalidate]
  Dragon [Write Update]
Protocol choices as well as block size have significant impact on miss rate and traffic.

Next Time: More realistic look at snooping protocols.

Don't forget: Group proposal due 1/25/05 (Tuesday).