Snoop-Based Multiprocessor Design I: Base Design

Todd C. Mowry
CS 418
Feb. 23 & 24, 2011

Design Goals

Performance and cost depend on design and implementation

Goals
- Correctness
- High Performance
- Minimal Hardware

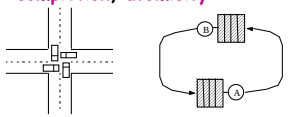
Often at odds
- High Performance => multiple outstanding low-level events
  => more complex interactions
  => more potential correctness bugs

We'll start simply and add concurrency to the design

Correctness Issues

Fulfill conditions for coherence and consistency
- Write propagation, serialization; for SC: completion, atomicity

Deadlock: all system activity ceases
- Cycle of resource dependences



Livelock: no processor makes forward progress although transactions are performed at hardware level
- e.g. simultaneous writes in invalidation-based protocol
  - each requests ownership, invalidating other, but loses it before winning arbitration for the bus

Starvation: one or more processors make no forward progress while others do.
- e.g. interleaved memory system with NACK on bank busy
- often not completely eliminated (not likely, not catastrophic)

Base Cache Coherence Design

Single-level write-back cache

Invalidation protocol

One outstanding memory request per processor

Atomic memory bus transactions
- For BusRd, BusRdX no intervening transactions allowed on bus between issuing address and receiving data
- BusWB: address and data simultaneous and sinked by memory system before any new bus request

Atomic operations within process
- One finishes before next in program order starts

Examine write serialization, completion, atomicity

Then add more concurrency/complexity and examine again

Some Design Issues

1. Design of cache controller and tags
   - Both processor and bus need to look up
2. How and when to present snoop results on bus
3. Dealing with write backs
4. Overall set of actions for memory operation not atomic
   - Can introduce race conditions
5. New issues: deadlock, livelock, starvation, serialization, etc.
6. Implementing atomic operations (e.g. read-modify-write)

Let's examine one by one ...

Cache Controller and Tags

Cache controller stages components of an operation

- itself a finite state machine (but not same as protocol state machine)

Uniprocessor: On a miss:

1. assert request for bus
2. wait for bus grant
3. drive address and command lines
4. wait for command to be accepted by relevant device
5. transfer data

In snoop-based multiprocessor, cache controller must:

- Monitor bus and processor
  - can view as two controllers: bus-side, and processor-side
  - with single-level cache: dual tags (not data) or dual-ported tag RAM
  - must reconcile when updated, but usually only looked up
- Respond to bus transactions when necessary (multiprocessor-ready)

Some Design Issues

1. Design of cache controller and tags
   - Both processor and bus need to look up
2. How and when to present snoop results on bus
3. Dealing with write backs
4. Overall set of actions for memory operation not atomic
   - Can introduce race conditions
5. New issues: deadlock, livelock, starvation, serialization, etc.
6. Implementing atomic operations (e.g. read-modify-write)

Reporting Snoop Results: How?

Collective response from caches must appear on bus

Example: in MESI protocol, need to know

- Is block dirty; i.e. should memory respond or not?
- Is block shared; i.e. transition to E or S state on read miss?

Three wired-OR signals

- Shared: asserted if any cache has a copy
- Dirty: asserted if some cache has a dirty copy
  - needn't know which, since it will do what's necessary
- Snoop-valid: asserted when OK to check other two signals
  - actually inhibit until OK to check

Illinois MESI requires priority scheme for cache-to-cache transfers

- Which cache should supply data when in shared state?
- Commercial implementations allow memory to provide data

Reporting Snoop Results: When?

Memory needs to know what, if anything, to do

Options for when memory should respond:

1. Fixed number of clocks from address appearing on bus
   - Dual tags required to reduce contention with processor
   - Still must be conservative (update both on write: E -> M)
   - examples: Pentium Pro, HP servers, Sun Enterprise
2. Variable delay
   - Memory assumes cache will supply data till all say "sorry"
   - Less conservative, more flexible, more complex
   - Memory can fetch data and hold just in case (SGI Challenge)
3. Immediately
   - Requires one bit of state per block in memory
   - Extra hardware complexity in commodity main memory system

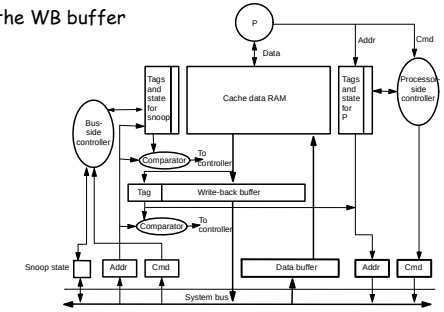
Some Design Issues

1. Design of cache controller and tags
   - Both processor and bus need to look up
2. How and when to present snoop results on bus
3. Dealing with write backs
4. Overall set of actions for memory operation not atomic
   - Can introduce race conditions
5. New issues: deadlock, livelock, starvation, serialization, etc.
6. Implementing atomic operations (e.g. read-modify-write)

Writebacks

To allow processor to continue quickly, we want to service miss first and then process the writeback caused by the miss *asynchronously*

- Need write-back buffer
- Must handle bus transactions relevant to buffered block
  - snoop the WB buffer



Some Design Issues

1. Design of cache controller and tags
   - Both processor and bus need to look up
2. How and when to present snoop results on bus
3. Dealing with write backs
4. Overall set of actions for memory operation not atomic
   - Can introduce race conditions
5. New issues: deadlock, livelock, starvation, serialization, etc.
6. Implementing atomic operations (e.g. read-modify-write)

Non-Atomic State Transitions

Memory operation involves many actions by many entities, including bus

- Look up cache tags, bus arbitration, actions by other controllers, ...
- Even if bus is atomic, overall set of actions is not
- Can have race conditions among components of different operations

Example: P1 and P2 attempt to write cached block A simultaneously

- Each decides to issue BusUpgr to allow S -> M

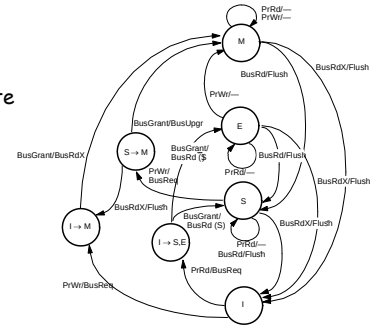
Issues:

- Must handle requests for other blocks while waiting to acquire bus
- Must handle requests for this block A
  - e.g. if P2 wins, P1 must invalidate copy and modify request to BusRdX

Handling Non-Atomicity: Transient States

Two types of states

- Stable (e.g. MESI)
- Transient or Intermediate



- This increases complexity; avoid if possible
  - e.g. don't use BusUpgr, rather other mechanisms to avoid data transfer

Some Design Issues

1. Design of cache controller and tags
   - Both processor and bus need to look up
2. How and when to present snoop results on bus
3. Dealing with write backs
4. Overall set of actions for memory operation not atomic
   - Can introduce race conditions
5. New issues: deadlock, livelock, starvation, serialization, etc.
6. Implementing atomic operations (e.g. read-modify-write)

Serialization

Processor-cache handshake must preserve serialization of bus order

- e.g., on write to block in S state, must not write data in block until ownership is acquired.
  - otherwise, the side-effects of other transactions that get bus before this one will appear later than they should

Write completion for SC: needn't wait for inval to actually happen

- Just wait until it gets bus (in this design, will happen before next bus xaction)
- *Commit* versus *complete*
- Don't know when inval actually inserted in destination processor's local order, only that it's before next xaction and in same order for all processors
- Local write hits do not become visible before next bus transaction
- Same argument will extend to more complex systems
- What matters is not when written data gets on the bus (writeback), but when subsequent reads are guaranteed to see it

Write atomicity: if a read returns value of a write W, W has already gone to bus and therefore completed if it needed to

Deadlock, Livelock, Starvation

Request-reply protocols can lead to protocol-level, *fetch deadlock*
- In addition to buffer deadlock discussed earlier
- When attempting to issue requests, must service incoming transactions
  - e.g. cache controller awaiting bus grant must snoop and even flush blocks
  - else may not respond to request that will release bus: deadlock

Livelock: many processors try to write same line. Each one:
- Obtains exclusive ownership via bus transaction (assume not in cache)
- Realizes block is in cache and tries to write it
- Livelock: I obtain ownership, but you steal it before I can write, etc.
- Solution: don't let exclusive ownership be taken away before write

Starvation: solve by using fair arbitration on bus and FIFO buffers
- May require too much buffering; if retries used, priorities as heuristics

Some Design Issues

1. Design of cache controller and tags
   - Both processor and bus need to look up
2. How and when to present snoop results on bus
3. Dealing with write backs
4. Overall set of actions for memory operation not atomic
   - Can introduce race conditions
5. New issues: deadlock, livelock, starvation, serialization, etc.
6. Implementing atomic operations (e.g. read-modify-write)

Implementing Atomic Operations

Read-modify-write: read component and write component

Cacheable variable vs. perform read-modify-write at memory:
- cacheable variable:
  - has lower latency and bandwidth needs for self-reacquisition
  - also allows spinning in cache without generating traffic while waiting
- at-memory:
  - has lower transfer time
- usually traffic and latency considerations dominate, so *use cacheable*

Natural to implement with two bus transactions: read and write
- can lock down bus: okay for atomic bus, but not for split-transaction
- get exclusive ownership, read-modify-write, only then allow others access
- compare&swap more difficult in RISC machines: two registers+memory

Implementing LL-SC

Lock flag and lock address register at each processor

LL: reads block, sets lock flag, puts block address in register
- Incoming invalidations checked against address: if match, reset flag
- Also reset flag if block is replaced and at context switches

SC: checks lock flag as indicator of intervening conflicting write
- If reset, fail; if not, succeed

Livelock considerations:
- Don't allow replacement of lock variable between LL and SC
  - split or set-assoc. cache, and don't allow memory accesses between LL, SC
  - (also don't allow reordering of accesses across LL or SC)
- Don't allow failing SC to generate invalidations (not an ordinary write)

Performance: both LL and SC can miss in cache
- Prefetch block in exclusive state at LL
- But exclusive request reintroduces livelock possibility
  - one solution: use backoff

Recap: We Have a Working Solution for the Base Design

Properties of the Base Design:

- **Single-level write-back cache**
- **Invalidation protocol**
- **One outstanding memory request per processor**
- **Atomic memory bus transactions**
  - For BusRd, BusRdX no intervening transactions allowed on bus between issuing address and receiving data
  - BusWB: address and data simultaneous and sinked by memory system before any new bus request
- **Atomic operations within process**
  - One finishes before next in program order starts

We examined write serialization, completion, atomicity

Next Step:

- **add more concurrency/complexity and examine again**