Snoop-based Multiprocessor Design

Chapter 6

Design Goals

Performance and cost depend on design and implementation too

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

2

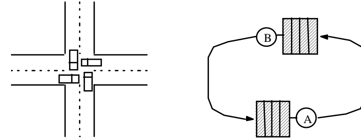
Correctness Issues

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)

3

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

4

Some Design Issues

Design of cache controller and tags

- Both processor and bus need to look up

How and when to present snoop results on bus

Dealing with write backs

Overall set of actions for memory operation not atomic

- Can introduce race conditions

New issues deadlock, livelock, starvation, serialization, etc.

Implementing atomic operations (e.g. read-modify-write)

Let's examine one by one ...

5

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:

- Assert request for bus
- Wait for bus grant
- Drive address and command lines
- Wait for command to be accepted by relevant device
- 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)

6

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

7

Reporting Snoop Results: When?

Memory needs to know what, if anything, to do

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)
- Pentium Pro, HP servers, Sun Enterprise

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)

Immediately: Bit-per-block in memory

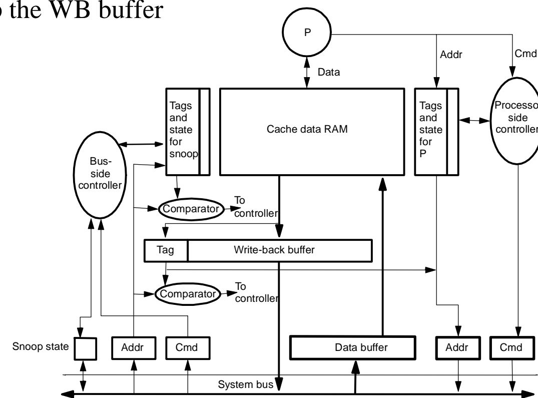
- Extra hardware complexity in commodity main memory system

8

Writebacks

To allow processor to continue quickly, want to service miss first and then process the write back caused by the miss asynchronously

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



Non-Atomic State Transitions

Memory operation involves many actions by many entities, incl. 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

Suppose 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

- Increase complexity, so many seek to avoid
  - e.g. don't use BusUpgr, rather other mechanisms to avoid data transfer

11

Serialization

Processor-cache handshake must preserve serialization of bus order

- e.g. on write to block in S state, mustn't write data in block until ownership is acquired.
  - other transactions that get bus before this one may seem to appear later

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

- Just wait till it gets bus (here, will happen before next bus xaction)
- *Commit* versus *complete*
- Don't know when inval actually inserted in destination process's local order, only that it's before next xaction and in same order for all procs
- Local write hits become visible not before next bus transaction
- Same argument will extend to more complex systems
- What matters is not when written data gets on the bus (write back), 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

12

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

13

Multi-level Cache Hierarchies

How to snoop with multi-level caches?

- independent bus snooping at every level?
- maintain cache inclusion

Requirements for *Inclusion*

- data in higher-level cache is subset of data in lower-level cache
- modified in higher-level => marked modified in lower-level

Now only need to snoop lowest-level cache

- If L2 says not present (modified), then not so in L1 too
- If BusRd seen to block that is modified in L1, L2 itself knows this

Is inclusion automatically preserved

- Replacements: all higher-level misses go to lower level
- Modifications

14

Violations of Inclusion

The two caches (L1, L2) may choose to replace different block

- Differences in reference history
  - set-associative first-level cache with LRU replacement
  - example: blocks m1, m2, m3 fall in same set of L1 cache...
- Split higher-level caches
  - instruction, data blocks go in different caches at L1, but may collide in L2
  - what if L2 is set-associative?
- Differences in block size

But a common case works automatically

- L1 direct-mapped, fewer sets than in L2, and block size same

15

Preserving Inclusion Explicitly

Propagate lower-level (L2) replacements to higher-level (L1)

- Invalidate or flush (if dirty) messages

Propagate bus transactions from L2 to L1

- Propagate all transactions, or use inclusion bits

Propagate modified state from L1 to L2 on writes?

- Write-through L1, or modified-but-stale bit per block in L2 cache

Correctness issues altered?

- Not really, if all propagation occurs correctly and is waited for
- Writes commit when they reach the bus, acknowledged immediately
- But performance problems, so want to not wait for propagation
- Discuss after split-transaction busses

Dual cache tags less important: each cache is filter for other

16

Split-Transaction Bus

Split bus transaction into request and response sub-transactions

- Separate arbitration for each phase

Other transactions may intervene

- Improves bandwidth dramatically
- Response is matched to request
- Buffering between bus and cache controllers

Reduce serialization down to the actual bus arbitration

17

Complications

New request can appear on bus before previous one serviced

- Even before snoop result obtained
- Conflicting operations to same block may be outstanding on bus
- e.g. P1, P2 write block in S state at same time
  - both get bus before either gets snoop result, so both think they've won
- Note: different from overall non-atomicity discussed earlier

Buffers are small, so may need *flow control*

Buffering implies revisiting snoop issues

- When and how snoop results and data responses are provided
- In order w.r.t. requests? (PPro, DEC Turbolaser: yes; SGI, Sun: no)
- Snoop and data response together or separately?
  - SGI together, SUN separately

Large space, much industry innovation: let's look at one example first

18

Example (based on SGI Challenge)

No conflicting requests for same block allowed on bus

- 8 outstanding requests total, makes conflict detection tractable

Flow-control through *negative acknowledgement (NACK)*

- NACK as soon as request appears on bus, requestor retries
- Separate command (incl. NACK) + address and tag + data buses

Responses may be in different order than requests

- Order of transactions determined by requests
- Snoop results presented on bus with response

Look at

- Bus design, and how requests and responses are matched
- Snoop results and handling conflicting requests
- Flow control
- Path of a request through the system

19

Bus Design and Req-Resp Matching

Essentially two separate buses, arbitrated independently

- "Request" bus for command and address
- "Response" bus for data

Out-of-order responses imply need for matching req-response

- Request gets 3-bit tag when wins arbitration (8 outstanding max)
- Response includes data as well as corresponding request tag
- Tags allow response to not use address bus, leaving it free

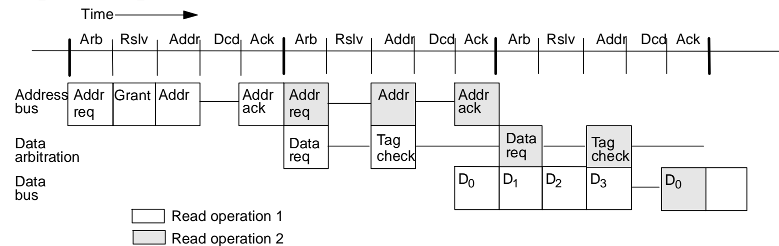
Separate bus lines for arbitration, and for snoop results

20

Bus Design (continued)

Each of request and response phase is 5 bus cycles (best case)

- Response: 4 cycles for data (128 bytes, 256-bit bus), 1 turnaround
- Request phase: arbitration, resolution, address, decode, ack
- Request-response transaction takes 3 or more of these



Cache tags looked up in decode; extend ack cycle if not possible

- Determine who will respond, if any
- Actual response comes later, with re-arbitration

Write-backs have request phase only: arbitrate both data+addr buses

Upgrades have only request part; ack'ed by bus on grant (commit)

21

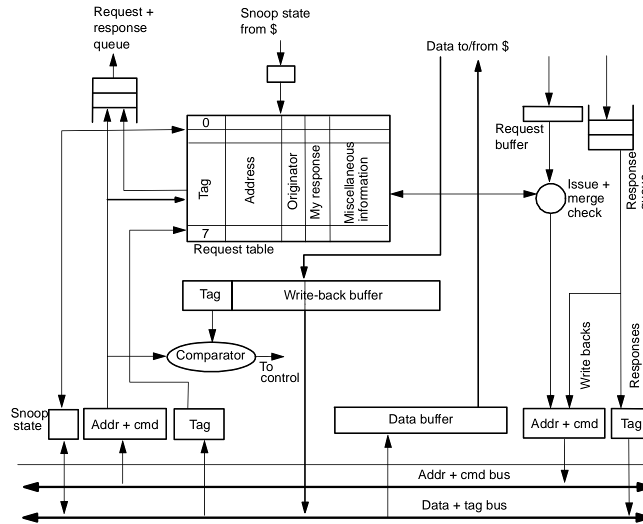
Bus Design (continued)

Tracking outstanding requests and matching responses

- Eight-entry "request table" in each cache controller
- New request on bus added to all at same index, determined by tag
- Entry holds address, request type, state in that cache (if determined already), ...
- All entries checked on bus or processor accesses for match, so fully associative
- Entry freed when response appears, so tag can be reassigned by bus

22

Bus Interface with Request Table



Snoop Results and Conflicting Requests

Variable-delay snooping

Shared, dirty and inhibit wired-OR lines, as before

Snoop results *presented* when response appears

- *Determined* earlier, in request phase, and kept in request table entry
- (Also determined who will respond)
- Writebacks and upgrades don't have data response or snoop result

Avoiding conflicting requests on bus

- easy: don't issue request for conflicting request that is in request table

Recall writes committed when request gets bus

Flow Control

Not just at incoming buffers from bus to cache controller

Cache system's buffer for responses to its requests

- Controller limits number of outstanding requests, so easy

Mainly needed at main memory in this design

- Each of the 8 transactions can generate a writeback
- Can happen in quick succession (no response needed)
- SGI Challenge: separate NACK lines for address and data buses
  - Asserted before ack phase of request (response) cycle is done
  - Request (response) cancelled everywhere, and retries later
  - Backoff and priorities to reduce traffic and starvation
- SUN Enterprise: destination initiates retry when it has a free buffer
  - source keeps watch for this retry
  - guaranteed space will still be there, so only two "tries" needed at most

25

Handling a Read Miss

Need to issue BusRd

First check request table. If hit:

- If prior request exists for same block, want to grab data too!
  - "want to grab response" bit
  - "original requestor" bit
    - non-original grabber must assert sharing line so others will load in S rather than E state
- If prior request incompatible with BusRd (e.g. BusRdX)
  - wait for it to complete and retry (processor-side controller)
- If no prior request, issue request and watch out for race conditions
  - conflicting request may win arbitration before this one, but this one receives bus grant before conflict is apparent
    - watch for conflicting request in slot before own, degrade request to "no action" and withdraw till conflicting request satisfied

26

Upon Issuing the BusRd Request

All processors enter request into table, snoop for request in cache

Memory starts fetching block

1. Cache with dirty block responds before memory ready
    - Memory aborts on seeing response
    - Waiters grab data
        - some may assert inhibit to extend response phase till done snooping
        - memory must accept response as WB (might even have to NACK)
2. Memory responds before cache with dirty block
    - Cache with dirty block asserts inhibit line till done with snoop
    - When done, asserts dirty, causing memory to cancel response
    - Cache with dirty issues response, arbitrating for bus
3. No dirty block: memory responds when inhibit line released
    - Assume cache-to-cache sharing not used (for non-modified data)

27

Handling a Write Miss

Similar to read miss, except:

- Generate BusRdX
- Main memory does not sink response since will be modified again
- No other processor can grab the data

If block present in shared state, issue BusUpgr instead

- No response needed
- If another processor was going to issue BusUpgr, changes to BusRdX as with atomic bus

28

Write Serialization

With split-transaction buses, usually bus order is determined by order of *requests* appearing on bus

- actually, the ack phase, since requests may be NACKed
- by end of this phase, they are committed for visibility in order

A write that follows a read transaction to the same location should not be able to affect the value returned by that read

- Easy in this case, since conflicting requests not allowed
- Read response precedes write request on bus

Similarly, a read that follows a write transaction won't return old value

29

Detecting Write Completion

Problem: invalidations don't happen as soon as request appears on bus

- They're buffered between bus and cache
- Commitment does not imply performing or completion
- Need additional mechanisms

Key property to preserve: processor shouldn't see new value produced by a write before previous writes in bus order are visible to it

1. Don't let certain types of incoming transactions be reordered in buffers
   - in particular, data reply should not overtake invalidation request
   - okay for invalidations to be reordered: only reply actually brings data in
2. Allow reordering in buffers, but ensure important orders preserved at key points
   - e.g. flush incoming invalidations/updates from queues and apply before processor completes operation that may enable it to see a new value

30

Commitment of Writes (Operations)

More generally, distinguish between *performing* and *commitment* of a write *w*:

Performed w.r.t a processor: invalidation actually applied

Committed w.r.t a processor: guaranteed that once that processor sees the new value associated with W, any subsequent read by it will see new values of all writes that were committed w.r.t that processor before W.

Global bus serves as point of commitment, if buffers are FIFO

- benefit of a serializing broadcast medium for interconnect

Note: acks from bus to processor must logically come via same FIFO

- not via some special signal, since otherwise can violate ordering

31

Write Atomicity

Still provided naturally by broadcast nature of bus

Recall that bus implies:

- writes commit in same order w.r.t. all processors
- read cannot see value produced by write before write has committed on bus and hence w.r.t. all processors

Previous techniques allow substitution of “complete” for “commit” in above statements

- that’s write atomicity

32