Snoop-Based Multiprocessor Design II: More Sophisticated Design

Todd C. Mowry
CS 418
March 1, 2011

Roadmap

So far, we have covered a simple base design

Today, we will explore a more realistic design:

- multi-level cache hierarchies
- split-transaction buses

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?

Violations of Inclusion

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

- 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

One case that works automatically

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

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

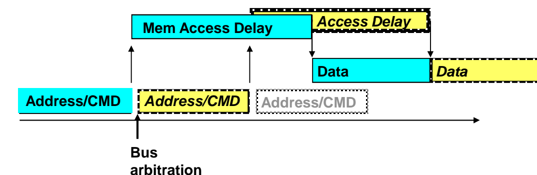
Dual cache tags less important: each cache is filter for other

Split-Transaction Bus

Split bus transaction into request and response sub-xactions
- 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



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 innovation: let's look at one example first

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

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

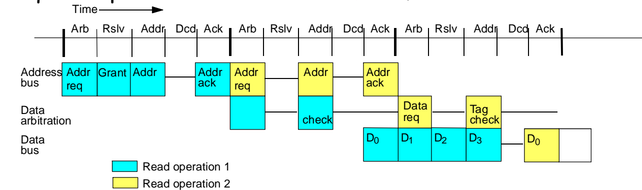
- Request gets 3-bit tag when wins arbitration (8 outstanding max)
- Response includes data as well as corresponding request tag
- Tags allow response to not use address bus, leaving it free

Separate bus lines for arbitration, and for snoop results

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

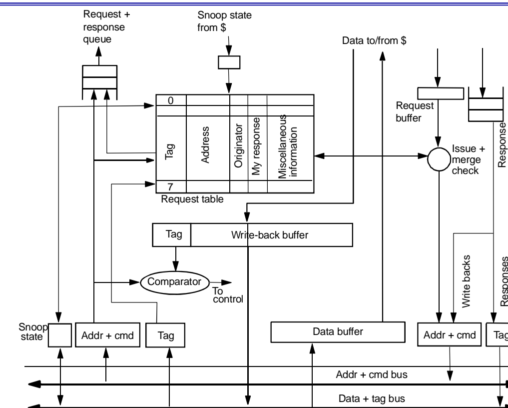
Upgrades have only request part; ack'ed by bus on grant (commit)

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

Bus Interface with Request Table



Outline (for SGI Challenge example)

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

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 that writes are committed when request gets bus

Outline (for SGI Challenge example)

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

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

Outline (for SGI Challenge example)

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

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

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)

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

Correctness Issues

For Coherence:
- Write propagation
- Write serialization

For Memory Consistency:
- Write completion
- Write atomicity

Other Correctness Issues:
- Deadlock
- Livelock
- Starvation

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

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

Commitment of Writes (Operations)

More generally, distinguish between *performing* and *committing* 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

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
- Will discuss deadlock, livelock, starvation after multilevel caches plus split transaction bus

Alternatives: In-Order Responses

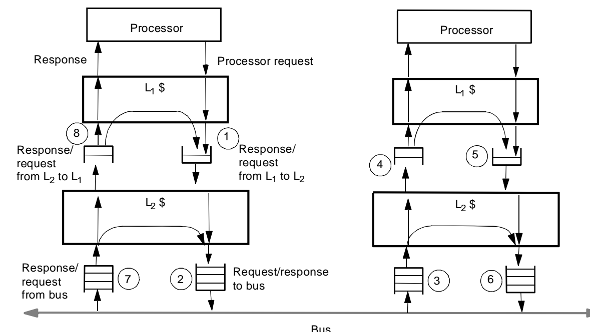
- FIFO request table suffices
- Dirty cache does not release inhibit line till it is ready to supply data
  - No deadlock problem since does not rely on anyone else
- But performance problems possible at interleaved memory
  - Major motivation for allowing out-of-order responses
- Allow conflicting requests more easily
  - Two BusRdX requests one after the other on bus for same block
    - latter controller invalidates its block, as before
    - but earlier requestor sees latter request before its own data response
    - with out-of-order response, not known which response will appear first
    - with in-order, known, and actually can use performance optimization
    - earlier controller responds to latter request by noting that latter is pending
    - when its response arrives, updates word, short-cuts block back on to bus, invalidates its copy (reduces ping-pong latency)

Other Alternatives

- Fixed delay from request to snoop result also makes it easier
  - Can have conflicting requests even if data responses not in order
  - e.g. SUN Enterprise
    - 64-byte line and 256-bit bus => 2 cycle data transfer
    - so 2-cycle request phase used too, for uniform pipelines
    - too little time to snoop and extend request phase
    - snoop results presented 5 cycles after address (unless inhibited)
    - by later data response arrival, conflicting requestors know what to do
- Don't even need request to go on same bus, as long as order is well-defined
  - SUN SparcCenter2000 had 2 busses, Cray 6400 had 4
  - Multiple requests go on bus in same cycle
  - Priority order established among them is logical order

Multi-Level Caches with ST Bus

- Key new problem: many cycles to propagate through hierarchy
  - Must let others propagate too for bandwidth, so queues between levels



Introduces deadlock and serialization problems

Deadlock Considerations

Fetch deadlock:

- Must buffer incoming requests/responses while request outstanding
- One outstanding request per processor => need space to hold *p* requests plus one reply (latter is essential)
- If smaller (or if multiple o/s requests), may need to NACK
- Then need priority mechanism in bus arbiter to ensure progress

Buffer deadlock:

- L1 to L2 queue filled with read requests, waiting for response from L2
- L2 to L1 queue filled with bus requests waiting for response from L1
- Latter condition only when cache closer than lowest level is write back
- Could provide enough buffering, or general solutions discussed later

If # o/s bus transactions smaller than total o/s cache misses, response from cache must get bus before new requests from it allowed

- Queues may need to support bypassing

Sequential Consistency

Separation of commitment from completion even greater now

- More performance-critical that commitment replace completion

Fortunately techniques for single-level cache and ST bus extend

- Just use them at each level
- i.e. either
  - don't allow certain reorderings of transactions at any level, OR
  - don't let outgoing operation proceed past level before incoming invalidations/updates at that level are applied

Multiple Outstanding Processor Requests

So far assumed only one: not true of modern processors

Danger: operations from same processor can complete out of order

- e.g. write buffer: until serialized by bus, should not be visible to others
- Uniprocessors use write buffer to insert multiple writes in succession
  - multiprocessors usually can't do this while ensuring consistent serialization
  - exception: writes are to same block, and no intervening ops in program order

Key question: who should wait to issue next op till previous completes

- Key to high performance: processor needn't do it (so can overlap)
- Queues/buffers/controllers can ensure writes not visible to external world and reads don't complete (even if back) until allowed (more later)

Other requirement: caches must be lockup free to be effective

- Merge operations to a block, so rest of system sees only one o/s to block

All needed mechanisms for correctness available (deeper queues for performance)