Snoop-based Multiprocessor Design

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

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

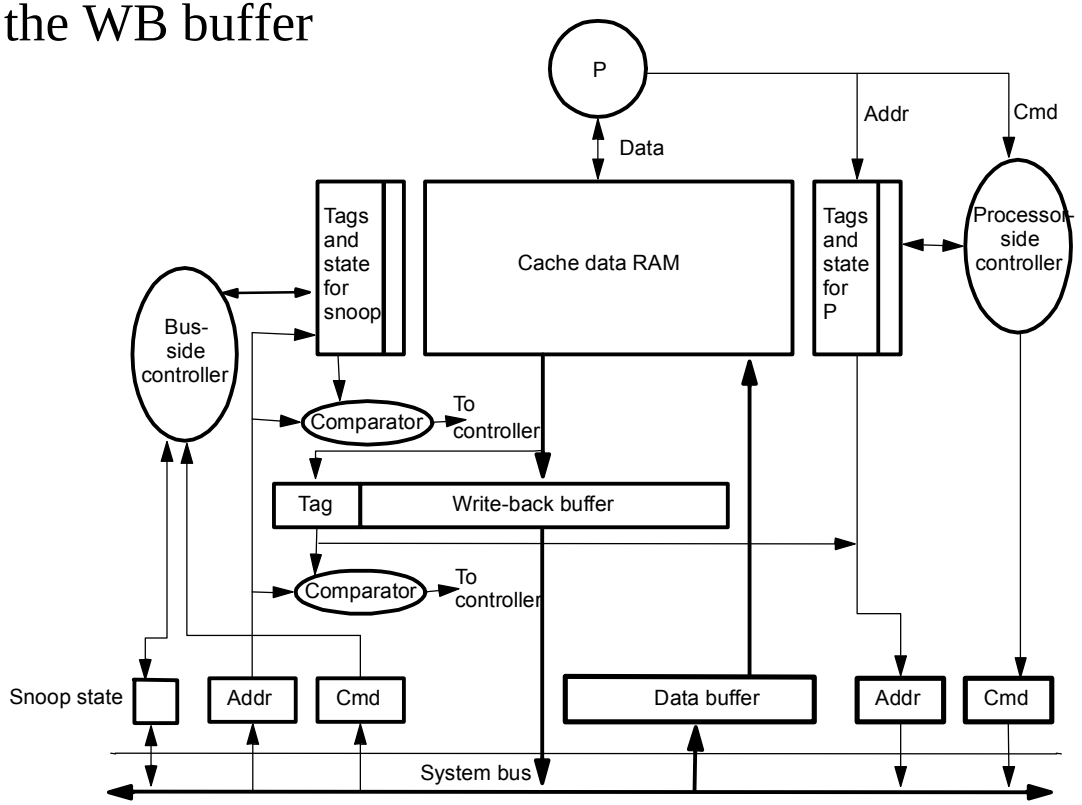
Illinois MESI requires priority scheme for cache-to-cache transfers

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

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

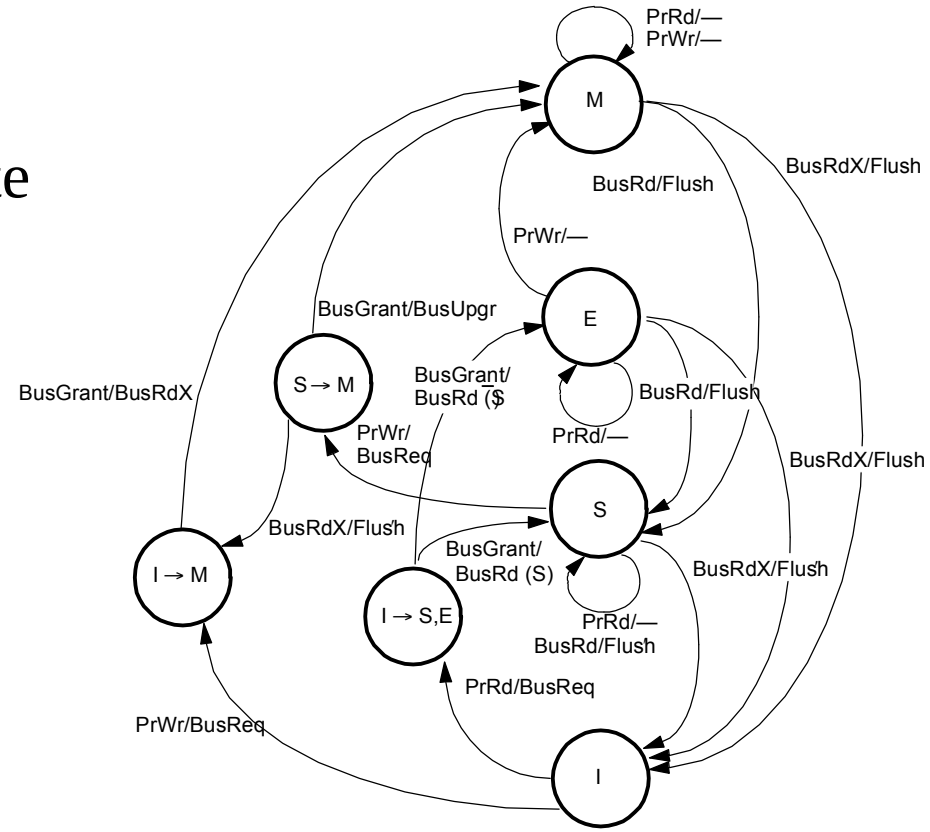
Issues

- 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

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

Split-Transaction Bus

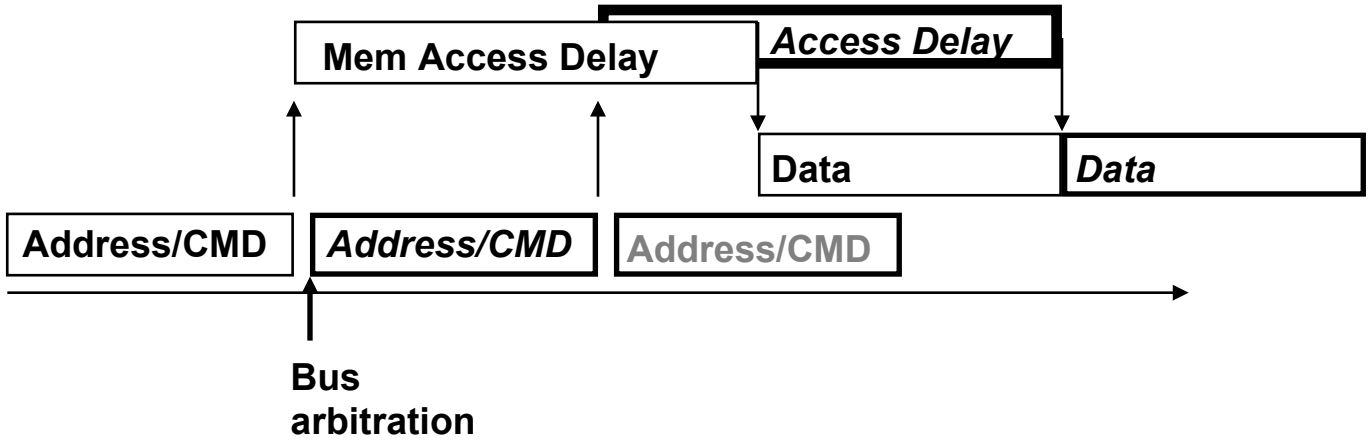
Split bus transaction into request and response sub-transactions

- 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



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

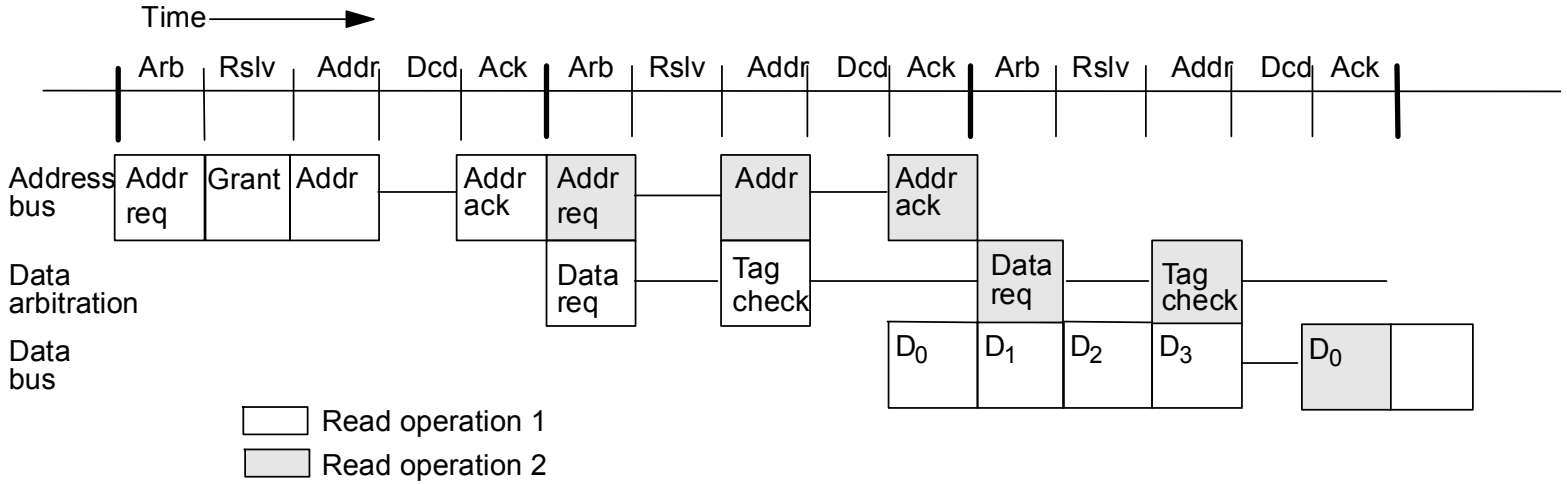
- Request gets 3-bit tag when wins arbitration (8 outstanding max)
- Response includes data as well as corresponding request tag
- Tags allow response to not use address bus, leaving it free

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

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

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

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

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