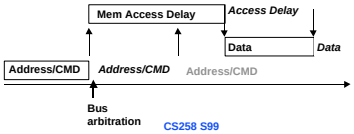
Split-Phase Busses

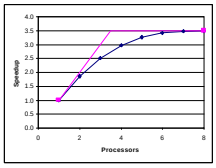
CS 258, Spring 99
David E. Culler
Computer Science Division
U.C. Berkeley

Split-Transaction Bus

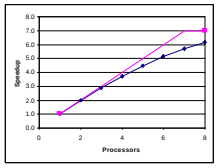
- Split bus transaction into request and response sub-transactions
  - Separate arbitration for each phase
- Other transactions may intervene
  - Improves bandwidth dramatically
  - Response is matched to request
  - Buffering between bus and cache controllers
- Reduce serialization down to the actual bus arbitration



Impact of just 2-stage miss processing

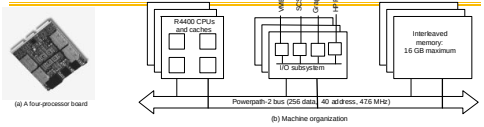


Z = 50 cycles
S = 20 cycles



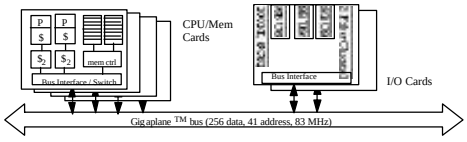
Z = 60 cycles
S = 10 cycles

SGI Challenge Overview



- 36 MIPS R4400 (peak 2.7 GFLOPS, 4 per board) or 18 MIPS R8000 (peak 5.4 GFLOPS, 2 per board)
- 8-way interleaved memory (up to 16 GB)
- 4 I/O busses of 320 MB/s each
- 1.2 GB/s Powerpath-2 bus @ 47.6 MHz, 16 slots, 329 signals
- 128 Bytes lines (1 + 4 cycles)
- Split-transaction with up to 8 outstanding reads

SUN Enterprise Overview



- Up to 30 UltraSPARC processors (peak 9 GFLOPs)
- Gigaplane^{TM} bus has peak bw 2.67 GB/s; upto 30GB memory
- 16 bus slots, for processing or I/O boards
  - 2 CPUs and 1GB memory per board
    - » memory distributed, unlike Challenge, but protocol treats as centralized
  - Each I/O board has 2 64-bit 25Mhz SBUSes

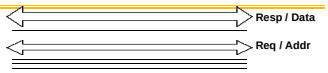
Complications

- 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
- 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

Example (based on SGI Challenge)

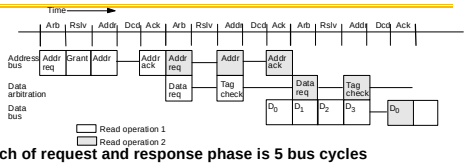
- No conflicting requests for same block allowed on bus
  - 8 outstanding requests total, makes conflict detection tractable
- Flow-control through negative acknowledgement (NACK)
  - NACK as soon as request appears on bus, requestor retries
  - Separate command (incl. NACK) + address and tag + data 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
    - » max 8 outstanding
  - 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)

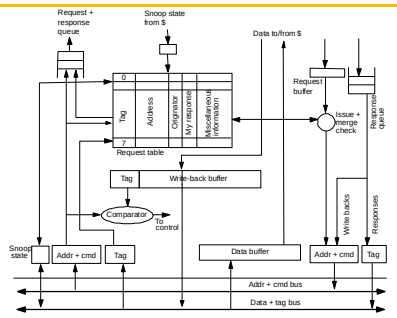


- Each of request and response phase is 5 bus cycles
  - Response: 4 cycles for data (128 bytes, 256-bit bus), 1 turnaround
  - Request phase: arbitration, resolution, address, decode, ack
  - Request-response transaction takes 3 or more of these
- Cache tags looked up in decode; extend ack cycle if not possible
  - Determine who will respond, if any
  - Actual response comes later, with re-arbitration
- Write-backs only request phase : 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



Snoop Results and Conflicting Requests

- Variable-delay snooping
- Shared, dirty and inhibit wired-OR lines
- 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
  - don't issue request for conflicting request that is in request table
  - adds delay to issue logic
- Recall writes committed when request gets bus

Flow Control

- Where?
  - incoming request buffers from bus to cache controller
  - response buffer
    - » Controller limits number of outstanding requests
- 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

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

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 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

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
- Performance problems possible at interleaved memory
- Allow conflicting requests more easily

Handling Conflicting Requests

- Two BusRdX requests one after the other on bus for same block
  - latter controller invalidates its block, as before, but earlier requestor sees later request before its own data response
- with out-of-order response, not known which response will appear first
- with in-order, known, and 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