Snoop-based Multiprocessor Design

Design Goals

Performance and cost depend on design and implementation too

Goals

- Correctness
- High Performance
- Minimal Hardware

Often at odds

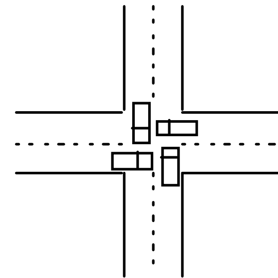
- High Performance => multiple outstanding low-level events
  => more complex interactions
  => more potential correctness bugs

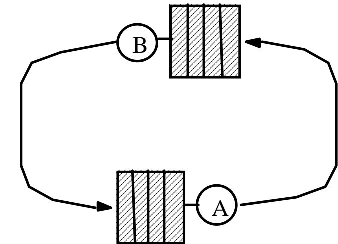
We'll start simply and add concurrency to the design

Correctness Issues

Fulfil 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

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 ...

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)

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

- <u>Shared</u>: asserted if any cache has a copy
- <u>Dirty</u>: asserted if some cache has a dirty copy
  - needn't know which, since it will do what's necessary
- <u>Snoop-valid</u>: 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

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)

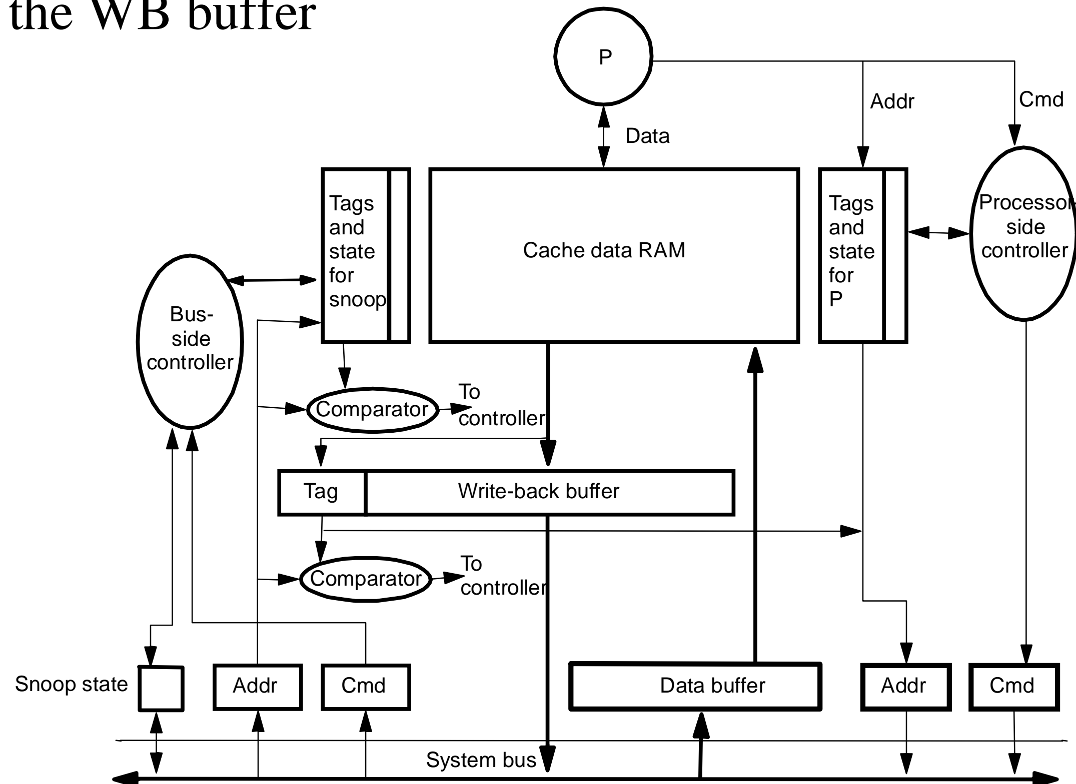
Immediately: Bit-per-block in memory

- Extra hardware complexity in commodity main memory system

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

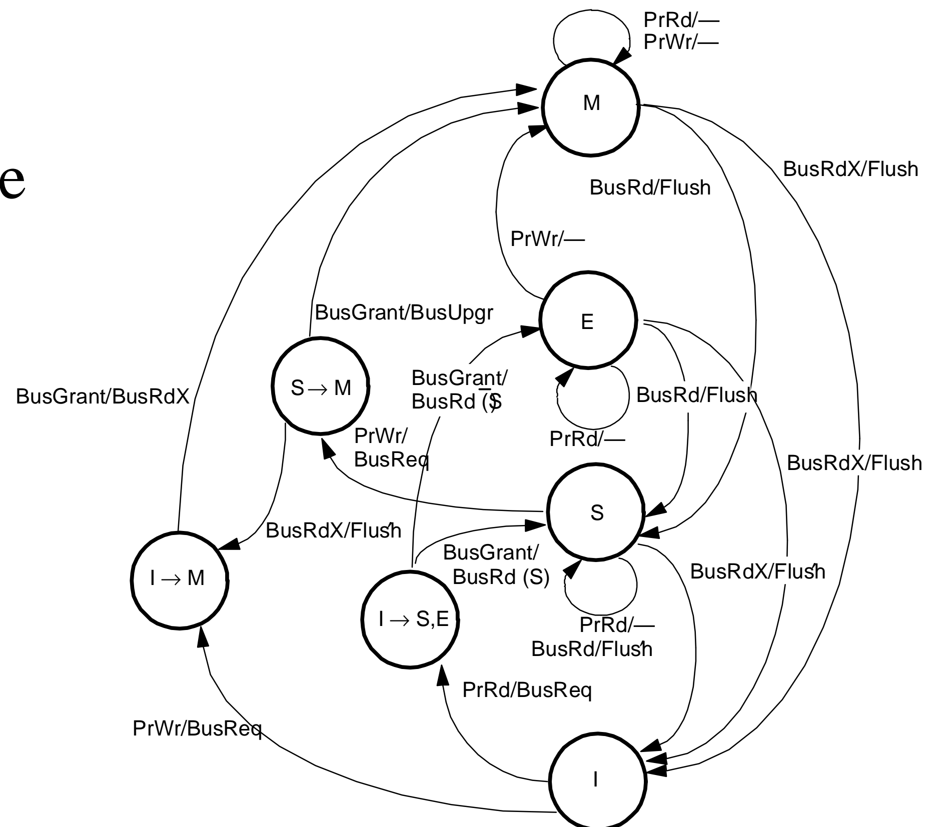
- Each decides to issue BusUpgr to allow S –> M

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

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 actuallly 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

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

Implementing Atomic Operations

Read-modify-write: read component and write component

- Cacheable variable, or perform read-modify-write at memory
  - cacheable 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 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: use backoff

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 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

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

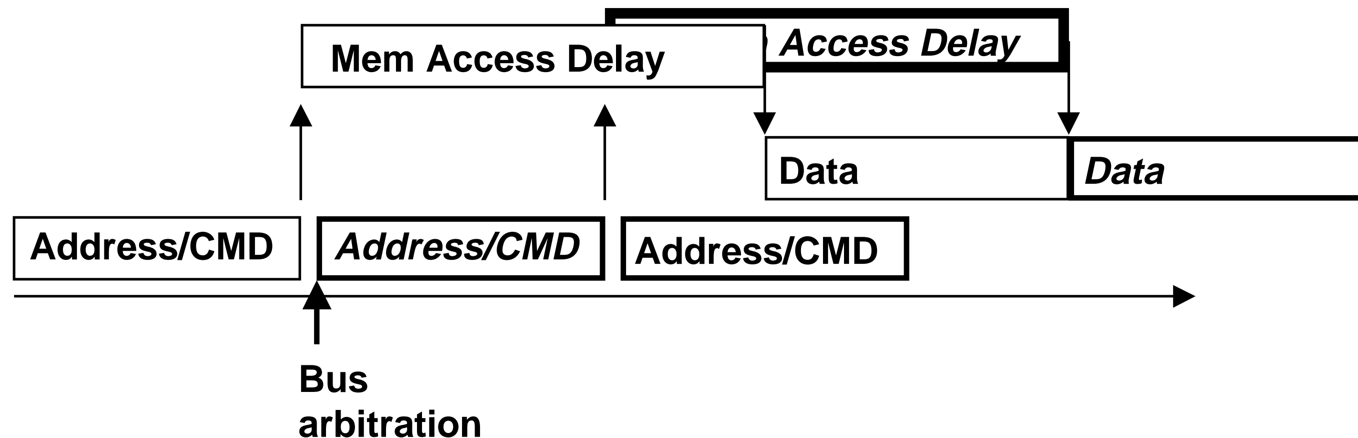
Split bus transaction into request and response sub-transactions

- 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



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

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

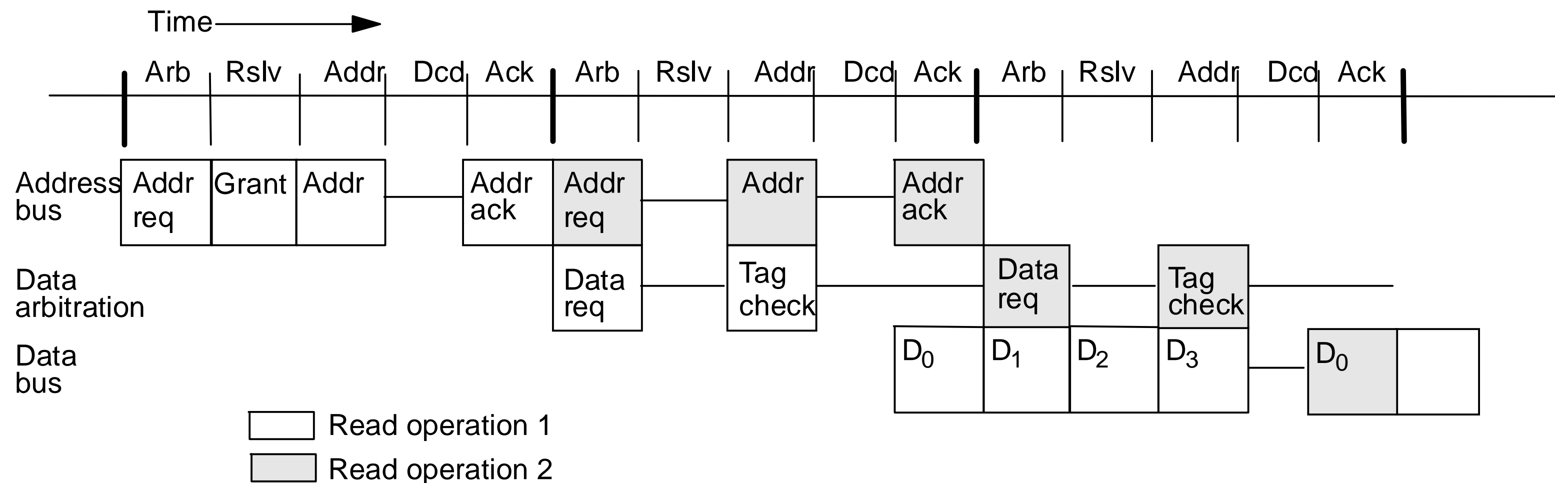
- Request gets 3-bit tag when wins arbitration (8 outstanding max)
- Response includes data as well as corresponding request tag
- Tags allow response to not use address bus, leaving it free

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

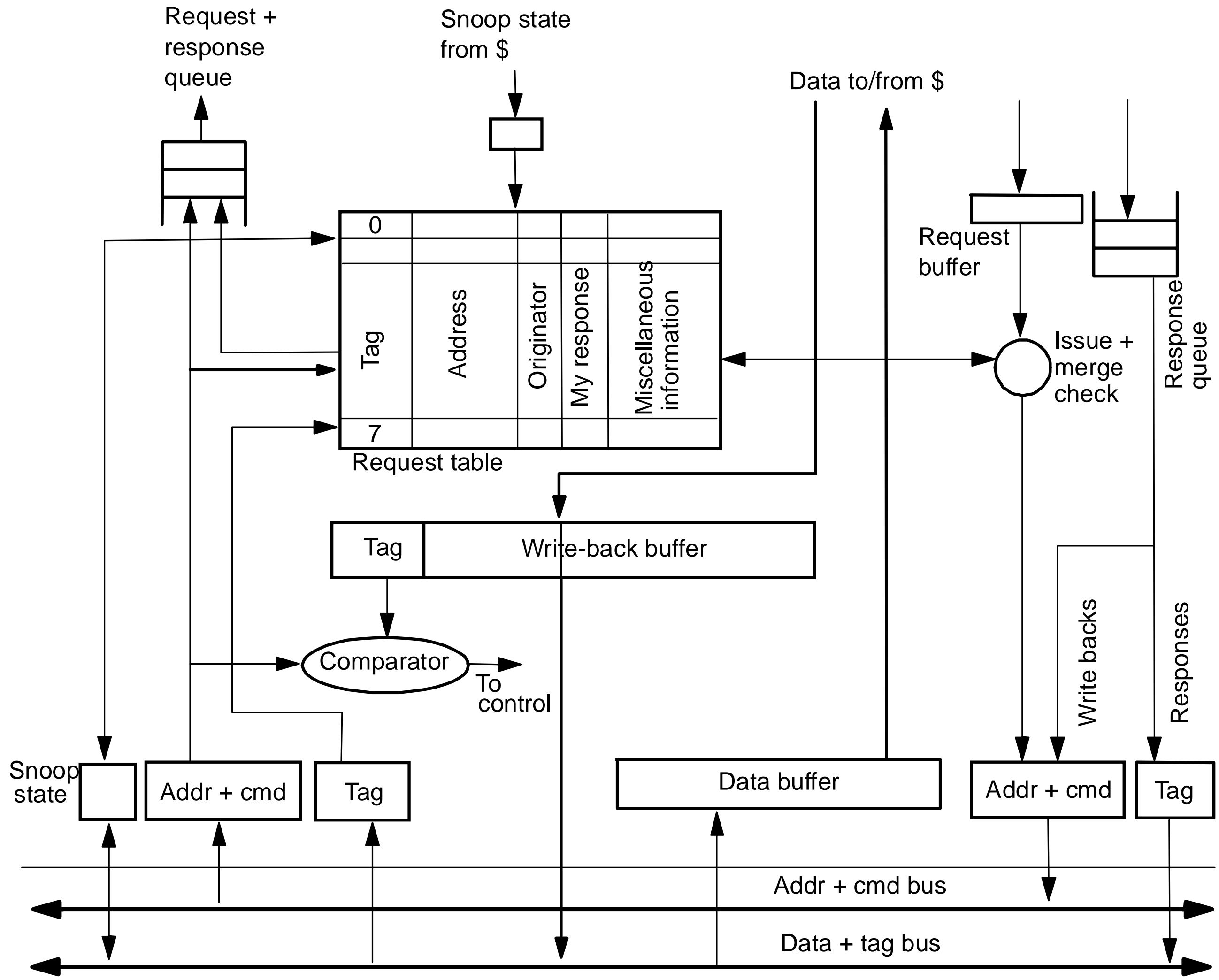
Upgrades have only request part; ack'ed by bus on grant (commit)

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



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

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

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 later 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

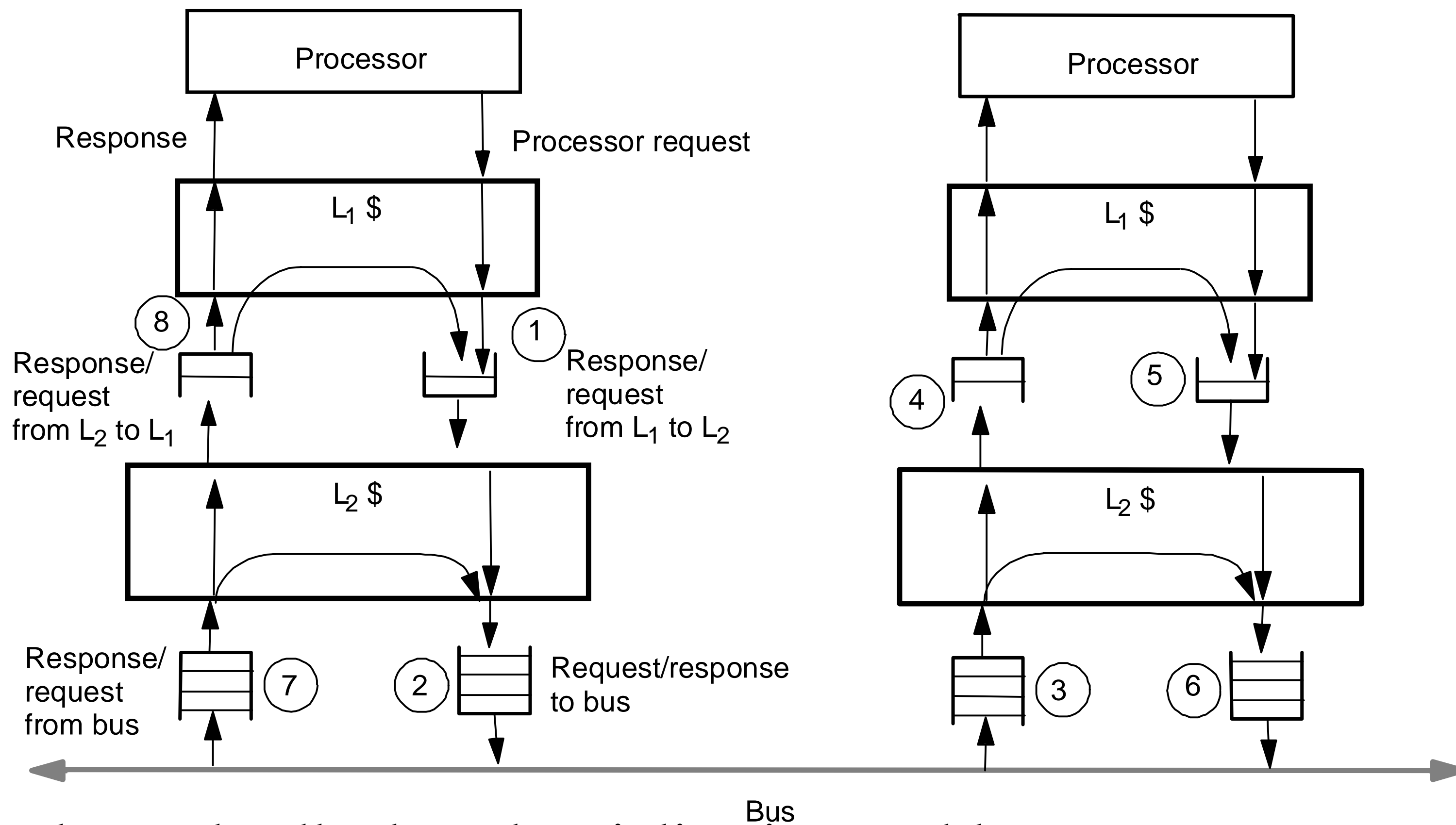
Don't even need request to go on same bus, as long as order is well-defined

- 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)

Case Studies of Bus-based Machines

SGI Challenge, with Powerpath bus

SUN Enterprise, with Gigaplane bus

- Take very different positions on the design issues discussed above

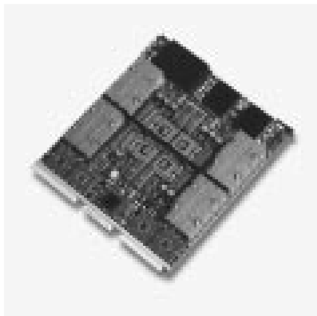
Overview

For each system:

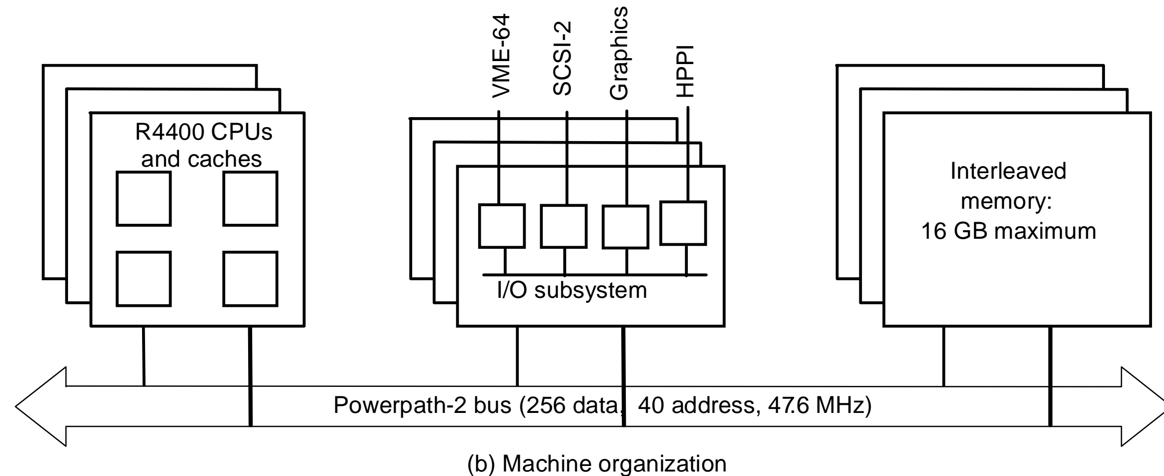
- Bus design
- Processor and Memory System
- Input/Output system
- Microbenchmark memory access results

Application performance and scaling (SGI Challenge)

SGI Challenge Overview



(a) A four-processor board



(b) Machine organization

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

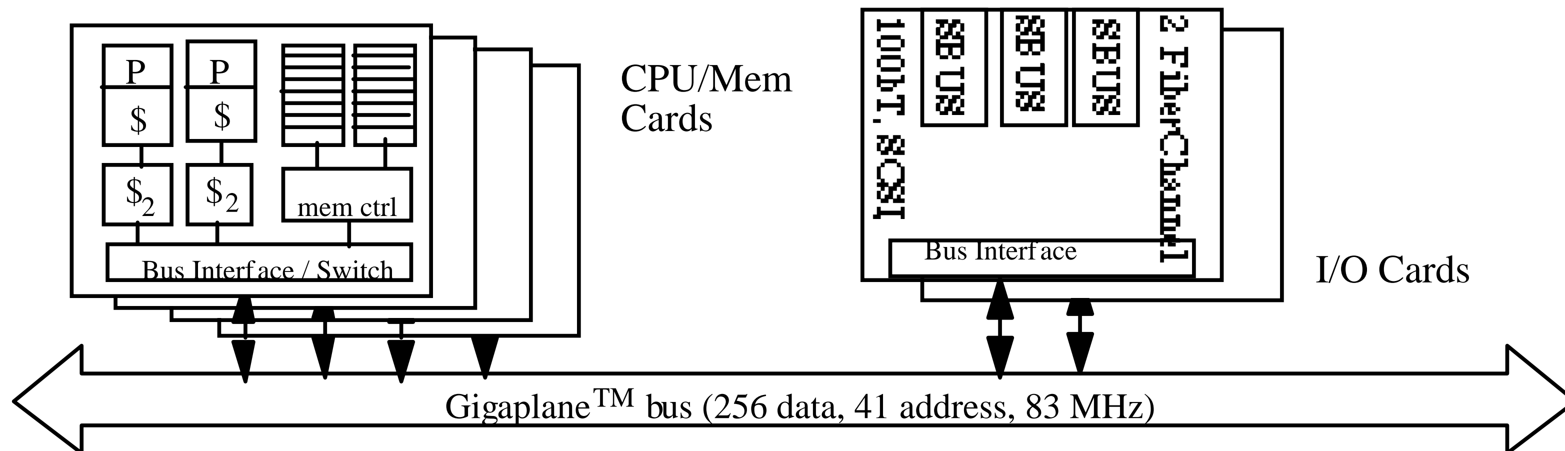
1.2 GB/s Powerpath-2 bus @ 47.6 MHz, 16 slots, 329 signals

128 Bytes lines (1 + 4 cycles)

Split-transaction with up to 8 outstanding reads

- all transactions take five cycles

SUN Enterprise Overview



Up to 30 UltraSPARC processors (peak 9 GFLOPs)

Gigaplane™ 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

Bus Design Issues

Multiplexed versus non-multiplexed (separate addr and data lines)

Wide versus narrow data busses

Bus clock rate

- Affected by signaling technology, length, number of slots...

Split transaction versus atomic

Flow control strategy

SGI Powerpath-2 Bus

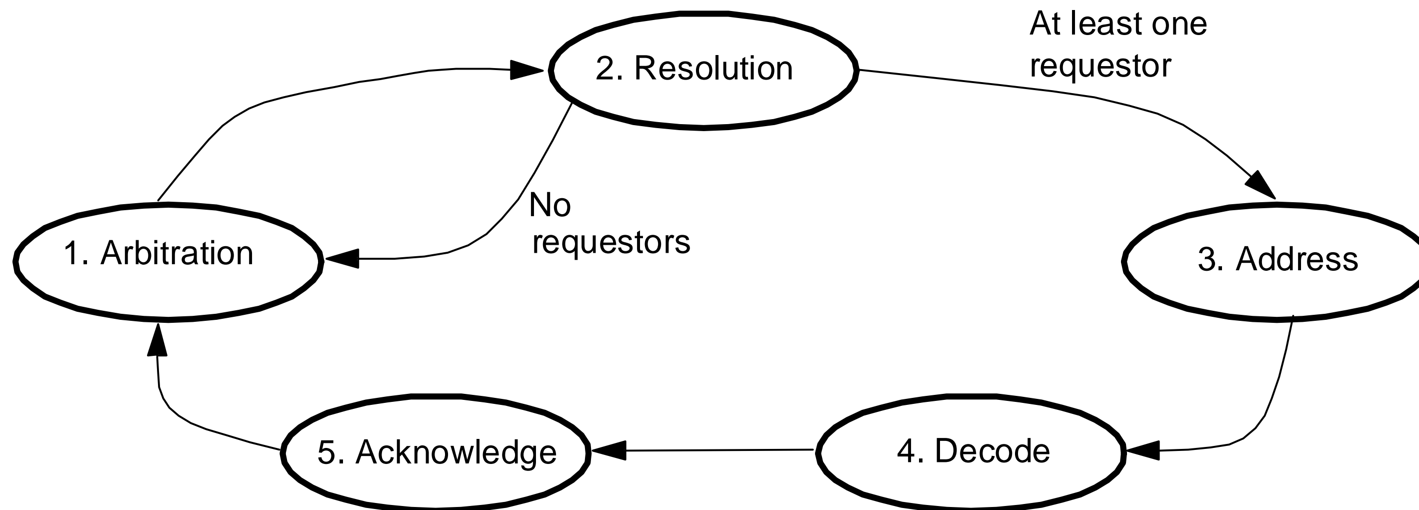
Non-multiplexed, 256-data/40-address, 47.6 MHz, 8 o/s requests

Wide => more interface chips so higher latency, but more bw at slower clock

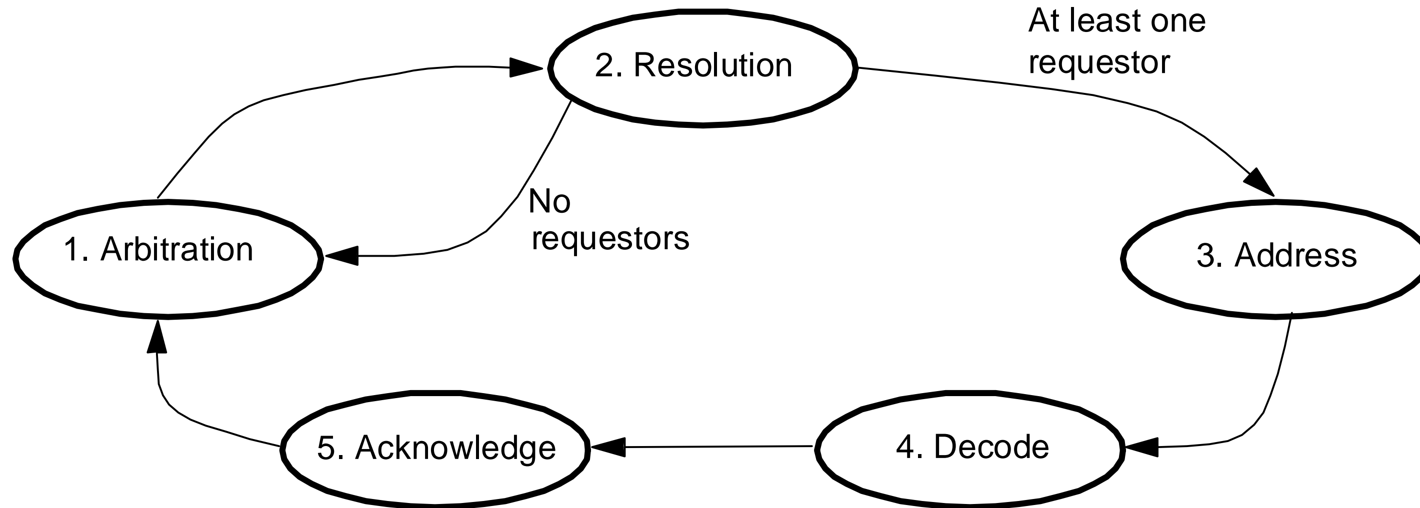
Large block size also calls for wider bus

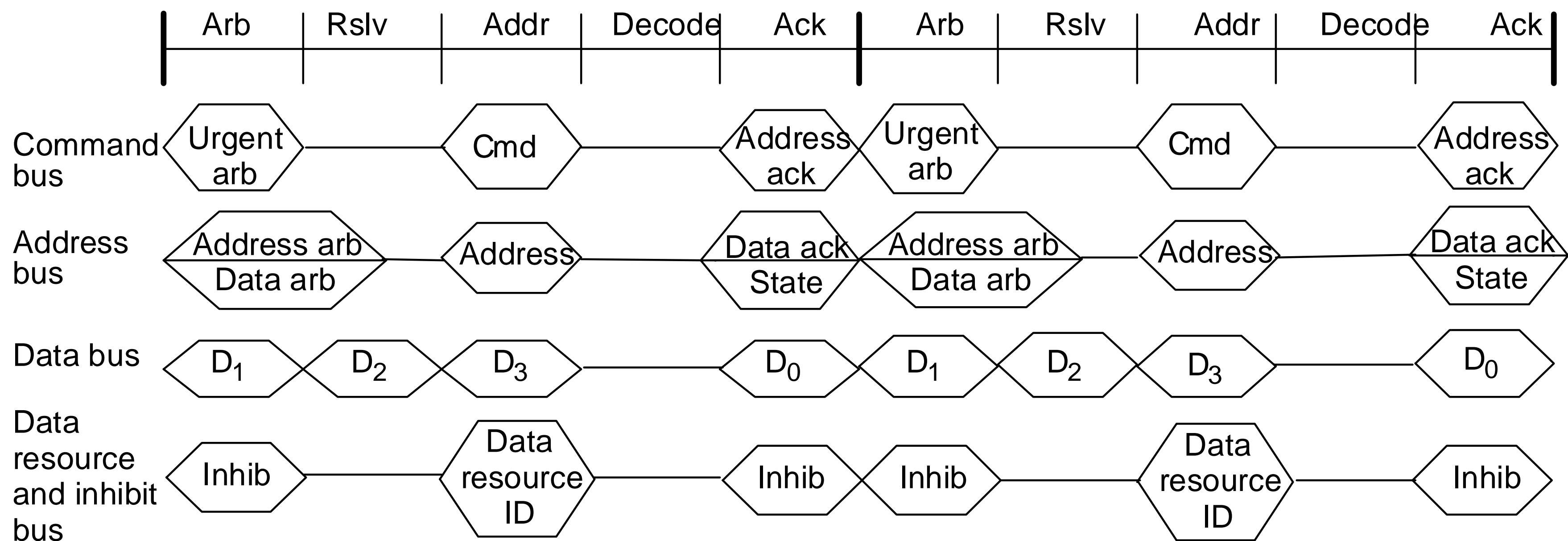
Uses Illinois MESI protocol (cache-to-cache sharing)

More detail in chapter



Bus Timing





Processor and Memory Systems

4 MIPS R4400 processors per board share A and D chips
A chip has address bus interface, request table, control logic
CC chip per processor has duplicate set of tags
Processor requests go from CC chip to A chip to bus
4 bit-sliced D chips interface CC chip to bus

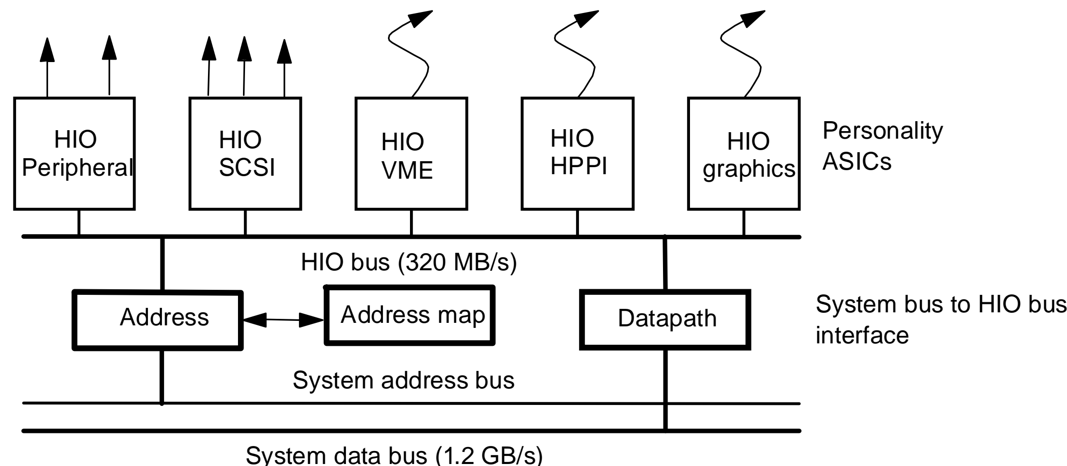
Memory Access Latency

250ns access time from address on bus to data on bus

But overall latency seen by processor is 1000ns!

- 300 ns for request to get from processor to bus
  - down through cache hierarchy, CC chip and A chip
- 400ns later, data gets to D chips
  - 3 bus cycles to address phase of request transaction, 12 to access main memory, 5 to deliver data across bus to D chips
- 300ns more for data to get to processor chip
  - up through D chips, CC chip, and 64-bit wide interface to processor chip, load data into primary cache, restart pipeline

Challenge I/O Subsystem



Multiple I/O cards on system bus, each has 320MB/s HIO bus

- Personality ASICs connect these to devices (standard and graphics)

Proprietary HIO bus

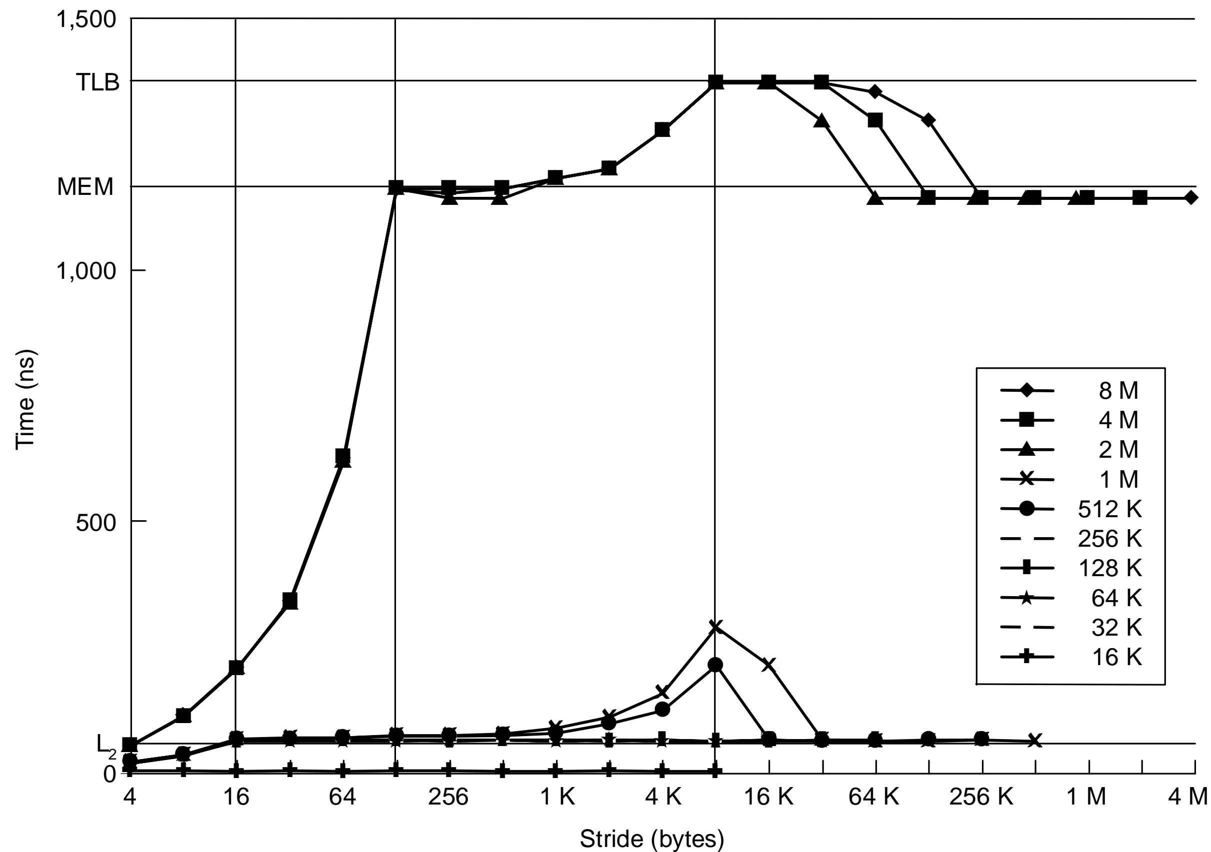
- 64-bit multiplexed address/data, same clock as system bus
- Split read transactions, up to 4 per device
- Pipelined, but centralized arbitration, with several transaction lengths
- Address translation via mapping RAM in system bus interface

Why the decouplings? (Why not connect directly to system bus?)

I/O board acts like a processor to memory system

Challenge Memory System Performance

Read microbenchmark with various strides and array sizes



Ping-pong flag-spinning microbenchmark: round-trip time 6.2 µs.

Sun Gigaplane Bus

Non-multiplexed, split-transaction, 256-data/41-address, 83.5 MHz

- Plus 32 ECC lines, 7 tag, 18 arbitration, etc. Total 388.

Cards plug in on both sides: 8 per side

112 outstanding transactions, up to 7 from each board

- Designed for multiple outstanding transactions per processor

Emphasis on reducing latency, unlike Challenge

- Speculative arbitration if address bus not scheduled from prev. cycle
- Else regular 1-cycle arbitration, and 7-bit tag assigned in next cycle

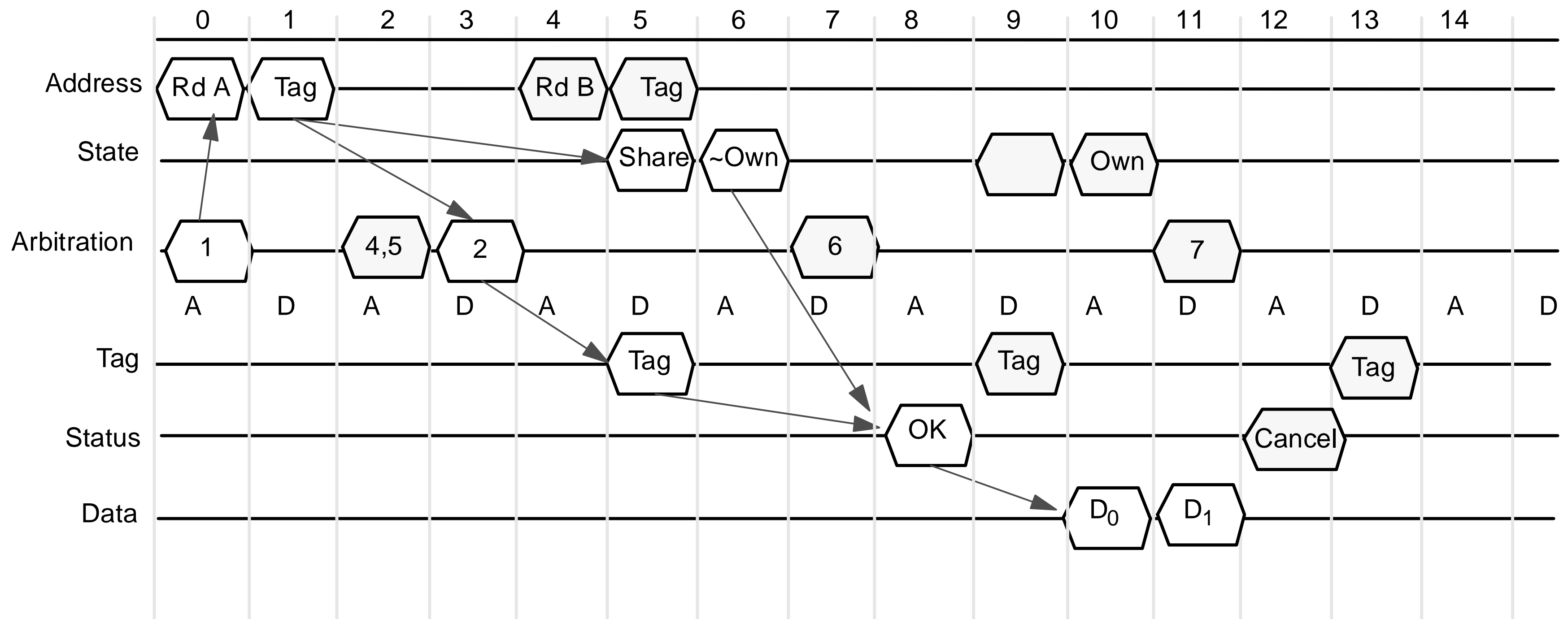
Snoop result associated with request phase (5 cycles later)

Main memory can stake claim to data bus 3 cycles into this, and start memory access speculatively

- Two cycles later, asserts tag bus to inform others of coming transfer

MOESI protocol (owned state for cache-to-cache sharing)

Gigaplane Bus Timing



Enterprise Processor and Memory System

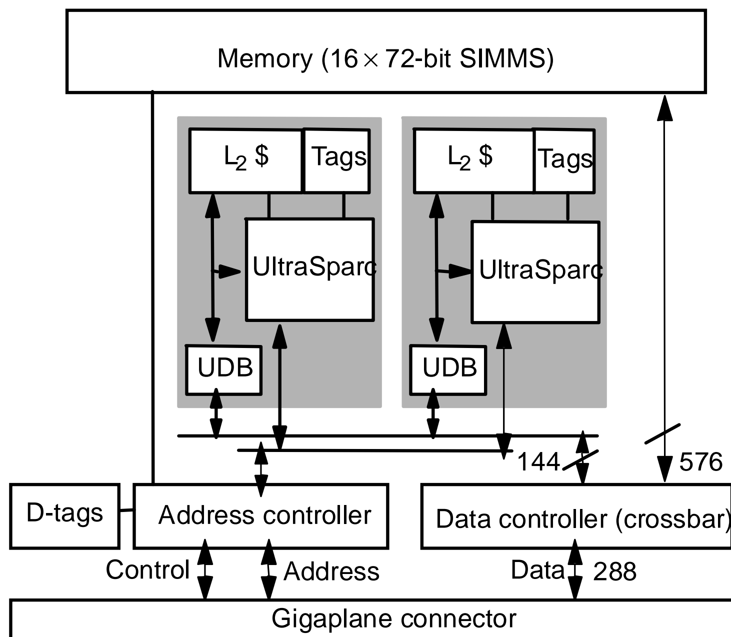
2 procs per board, external L2 caches, 2 mem banks with x-bar

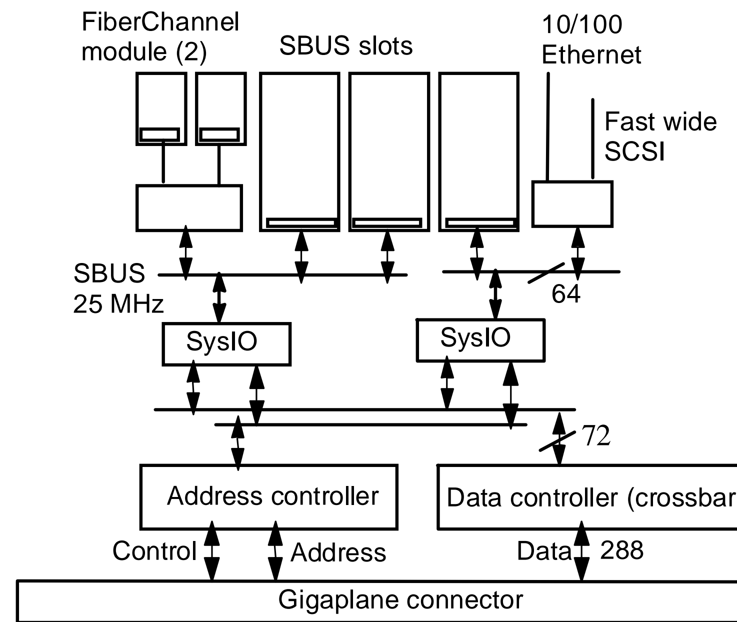
Data lines buffered through UDB to drive internal 1.3 GB/s UPA bus

Wide path to memory so full 64-byte line in 1 mem cycle (2 bus cyc)

Addr controller adapts proc and bus protocols, does cache coherence

- its tags keep a subset of states needed by bus (e.g. no M/E distinction)





Enterprise I/O System

I/O board has same bus interface ASICs as processor boards

But internal bus half as wide, and no memory path

Only cache block sized transactions, like processing boards

- Uniformity simplifies design
- ASICs implement single-block cache, follows coherence protocol

Two independent 64-bit, 25 MHz Sbuses

- One for two dedicated FiberChannel modules connected to disk
- One for Ethernet and fast wide SCSI
- Can also support three SBUS interface cards for arbitrary peripherals

Performance and cost of I/O scale with no. of I/O boards

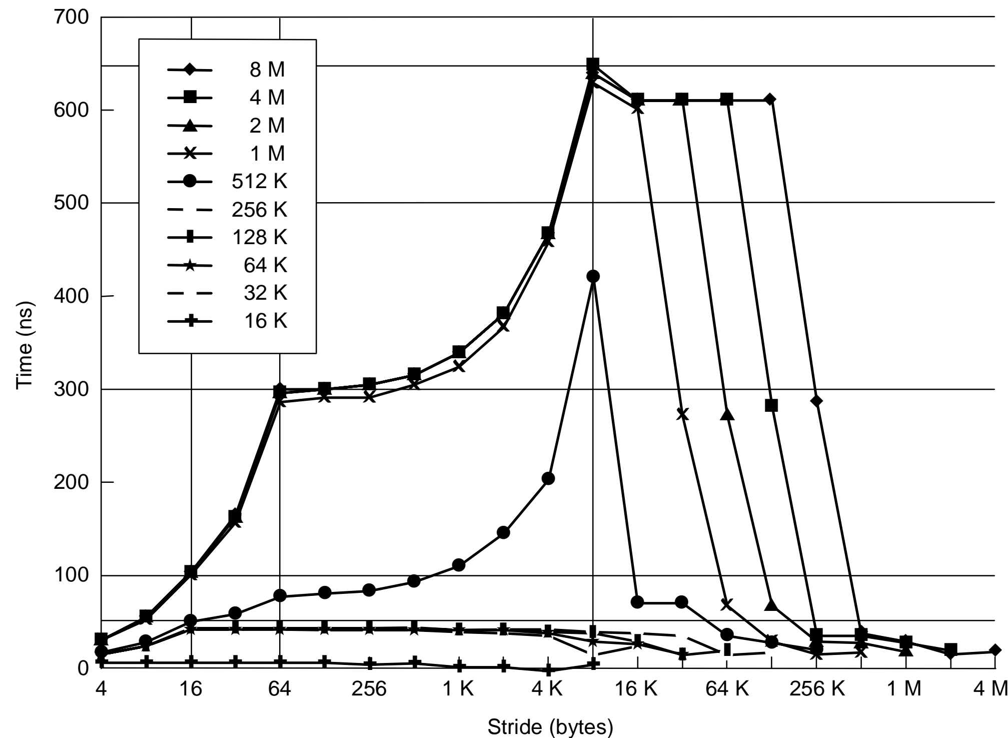
Memory Access Latency

300ns read miss latency

11 cycle min bus protocol at 83.5 Mhz is 130ns of this time

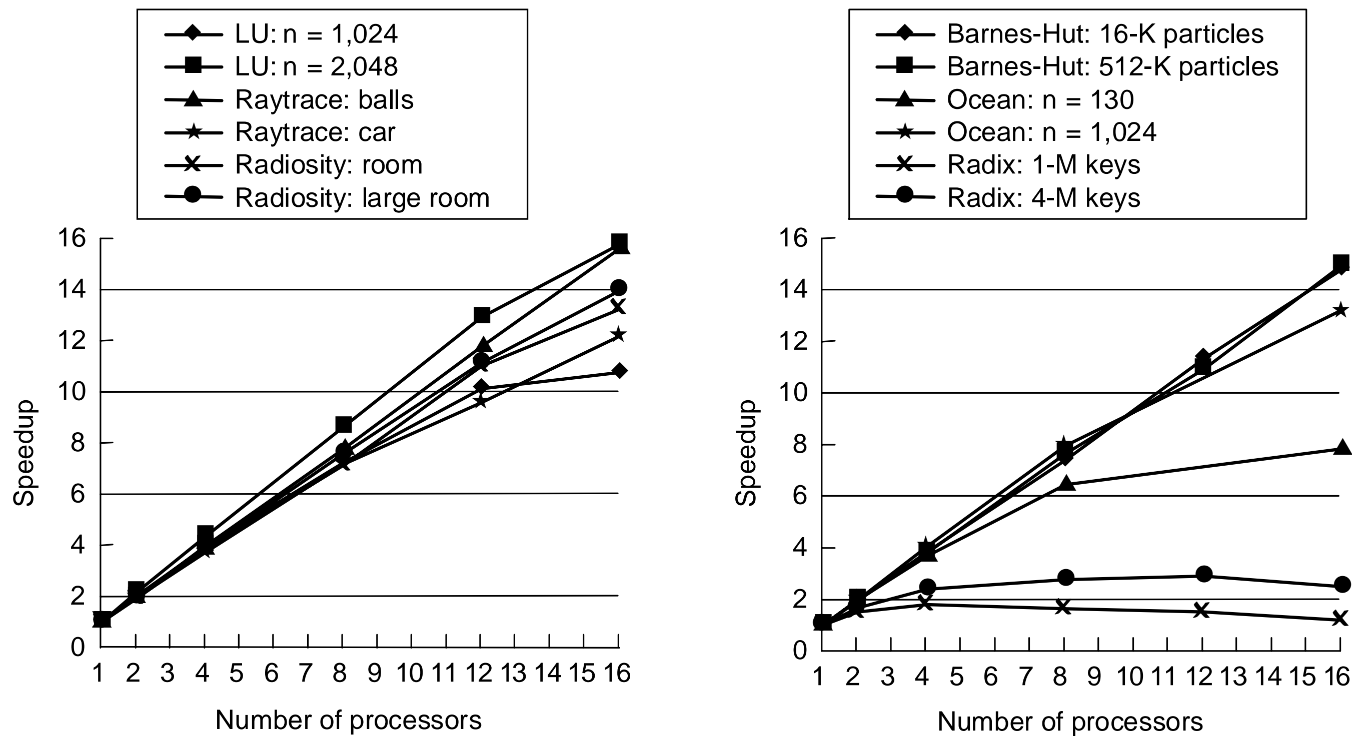
Rest is path through caches and the DRAM access

TLB misses add 340 ns



Ping-pong microbenchmark is 1.7 μs round-trip (5 mem accesses)

Application Speedups (Challenge)



- Problem in Ocean with small problem: communication and barrier cost
- Problem in Radix: contention on bus due to very high traffic
  - also leads to high imbalances and barrier wait time

Application Scaling under Other Models