Hardware Support for CC

Zeljko Zilic
McConnell Engineering Building
Room 546

McGill

Outline

- Synchronization
- Buses
- Split-Transaction Buses

Role of Synchronization

- "A collection of processing elements that <u>cooperate</u> and communicate to solve large problems fast."
- Types of Synchronization
  - *Mutual Exclusion*
  - Event synchronization
    - *point-to-point*
    - group
    - *global (barriers)*
- How much hardware support?
  - high-level operations?
  - atomic instructions?
  - specialized interconnect?

Mini-Instruction Set debate

- Atomic read-modify-write (r-m-w) instructions
  - IBM 370: included atomic compare&swap for multiprogramming
  - x86: any instruction can be prefixed with a lock modifier
  - High-level language advocates want hardware locks/barriers
    - but it's goes against the "RISC" flow,and has other problems
  - SPARC: atomic register-memory ops (swap, compare&swap)
  - MIPS, IBM Power: no atomic operations but pair of instructions
    - load-locked, store-conditional
    - later used by PowerPC and DEC Alpha too
- Rich set of tradeoffs

Other forms of hardware support

- Separate lock lines on the bus
- Lock locations in memory
- Lock registers (Cray Xmp)
- Hardware full/empty bits (Tera)
- Bus support for interrupt dispatch

Components of Synchronization Event

- Acquire method
  - Acquire right to the synch
    - enter critical section, go past event
- Waiting algorithm
  - Wait for synch to become available when it isn't
  - busy-waiting, blocking, or hybrid
- Release method
  - Enable other processors to acquire right to the synch
- Waiting algorithm is independent of type of synchronization
  - Makes no sense to put in hardware

Strawman Lock

Busy-Wait

```
lock:    ld   register, location      /* copy location to register */
         cmp  register, #0            /* compare with 0 */
         bnz  lock                    /* if not 0, try again */
         st   location, #1            /* store 1 to mark it locked */
         ret                          /* return control to caller */

unlock:  st   location, #0            /* write 0 to location */
         ret                          /* return control to caller */
```

Why doesn't the acquire method work?

Release method?

Atomic Instructions

- Specifies a location, register, & atomic operation
  - Value in location read into a register
  - Another value (function of value read or not) stored into location
- Many variants
  - Varying degrees of flexibility in second part
- Simple example: test&set
  - Value in location read into a specified register
  - Constant 1 stored into location
  - Successful if value loaded into register is 0
  - Other constants could be used instead of 1 and 0

Simple Test&Set Lock

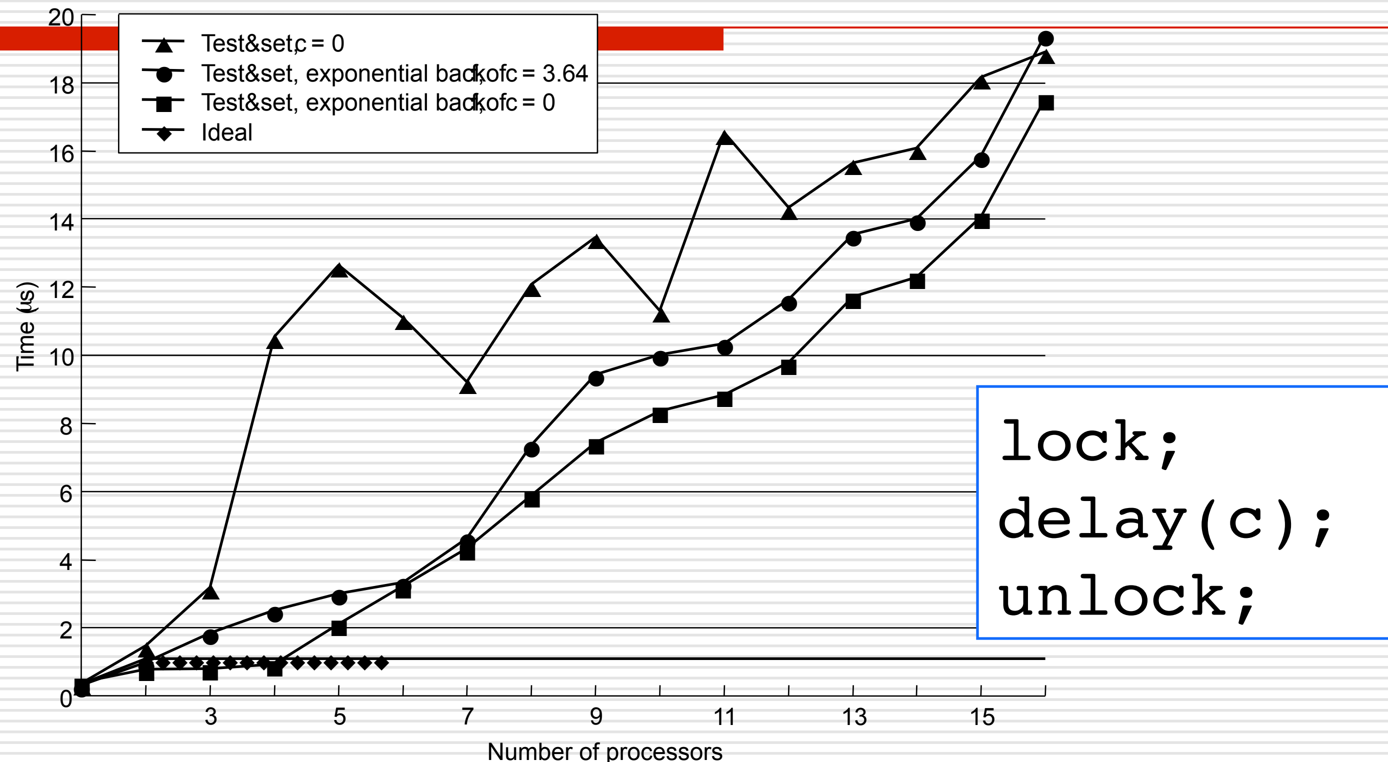
```
lock:      t&s   register, location
           bnz   lock                   /* if not 0, try again */
           ret                          /* return control to caller */
unlock:    st    location, #0           /* write 0 to location */
           ret                          /* return control to caller */
```

- Other read-modify-write primitives
  - Swap
  - Fetch&op
  - Compare&swap
    - Three operands: location, register to compare with, register to swap with
    - Not commonly supported by RISC instruction sets
- Cacheable or uncacheable

Performance Criteria for Synch. Ops

- Latency (time per op)
  - especially when light contention
- Bandwidth (ops per sec)
  - especially under high contention
- Traffic
  - load on critical resources
  - especially on failures under contention
- Storage
- Fairness

T&S Lock Microbenchmark: SGI Chal.



- Why does performance degrade?
  - Bus Transactions on T&S?
  - Hardware support in CC protocol?

Enhancements to Simple Lock

- Reduce frequency of issuing test&sets while waiting
  - *Test&set lock with backoff*
  - Don't back off too much or will be backed off when lock becomes free
  - Exponential backoff works quite well empirically: i^{th} time = $k*c^i$
- Busy-wait with read operations rather than test&set
  - *Test-and-test&set lock*
  - Keep testing with ordinary load
    - cached lock variable will be invalidated upon release
  - When value changes (to 0), try to obtain lock with test&set
    - only 1 attemptor will succeed; others fail and restart test

Improved Hardware Primitives: LL-SC

- Goals:
  - Test with reads
  - Failed read-modify-write attempts don't generate invalidations
  - Nice if single primitive can implement few r-m-w operations
- *Load-Locked* (or -linked), *Store-Conditional*
  - LL reads variable into register
  - Follow with arbitrary instructions to manipulate its value
  - SC tries to store back to location
  - Succeed iff no other write to the variable since this processor's LL
    - indicated by condition codes;
- If SC succeeds, all three steps happened atomically
- If fails, doesn't write or generate invalidations
  - must retry aquire

Simple Lock with LL-SC

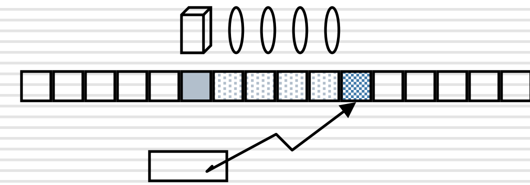
```
lock:        ll       reg1, location      /* LL location to reg1 */
             bnz      reg1, lock
             sclocation, reg2   /* SC reg2 into location*/
             beqz     reg2, lock          /* if failed, start again */
             ret
unlock:      st       location, #0        /* write 0 to location */
             ret
```

- Can do more fancy atomic ops by changing what's between LL & SC
  - But keep it small so SC likely to succeed
  - Don't include instructions that would need to be undone (e.g. stores)
- SC can fail (without putting transaction on bus) if:
  - Detects intervening write even before trying to get bus
  - Tries to get bus but another processor's SC gets bus first
- LL & SC are not lock & unlock, respectively
  - Only guarantee no conflicting write to lock variable between them
  - But can use directly to implement simple operations on shared variables

Trade-offs So Far

- Latency?
- Bandwidth?
- Traffic?
- Storage?
- Fairness?

- What happens when several processors spinning on lock and it is released?
  - Traffic per P lock operations?



Ticket Lock

- Only one r-m-w per acquire
- Two counters per lock (next_ticket, now_serving)
  - *Acquire*: **fetch&inc next_ticket; wait for now_serving == next_ticket**
    - atomic op when arrive at lock, not when it's free (so less contention)
  - *Release*: increment now-serving
- Performance
  - low latency for low-contention - if fetch&inc cacheable
  - O(p) read misses at release, since all spin on same variable
  - FIFO order
    - like simple LL-SC lock, but no inval when SC succeeds, and fair
  - Backoff?
- Wouldn't it be nice to poll different locations ...

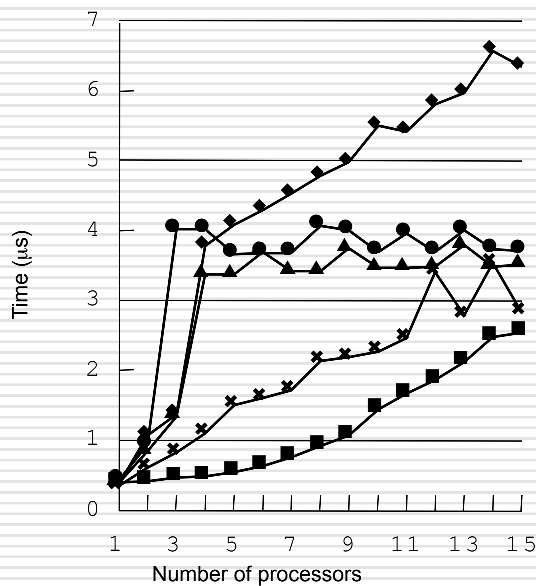
Array-based Queuing Locks

- Waiting processes poll on different locations in an array of size *p*
  - Acquire
    - fetch&inc to obtain address on which to spin (next array element)
    - Ensure that the addresses are in different cache lines or memories
  - Release
    - set next location in array, thus waking up process spinning on it
  - *O(1)* traffic per acquire with coherent caches
  - FIFO ordering, as in ticket lock, but, *O(p)* space per lock
  - Not so great for non-cache-coherent machines with distributed memory
    - array location I spin on not necessarily in my local memory (solution later)

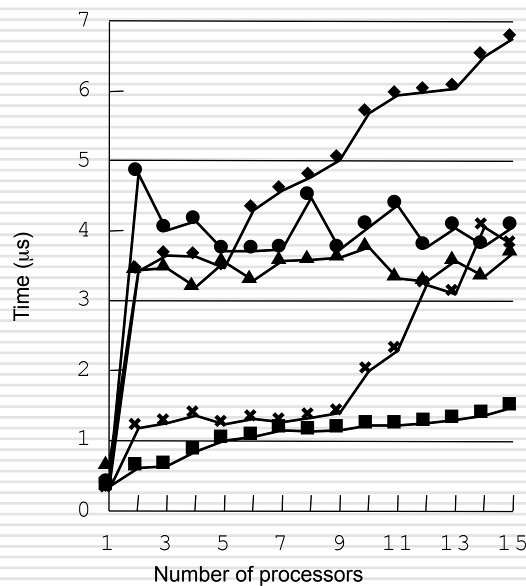
Lock Performance on SGI Challenge

Array-based
LL-SC
LL-SC, exponential
Ticket
Ticket, proportional

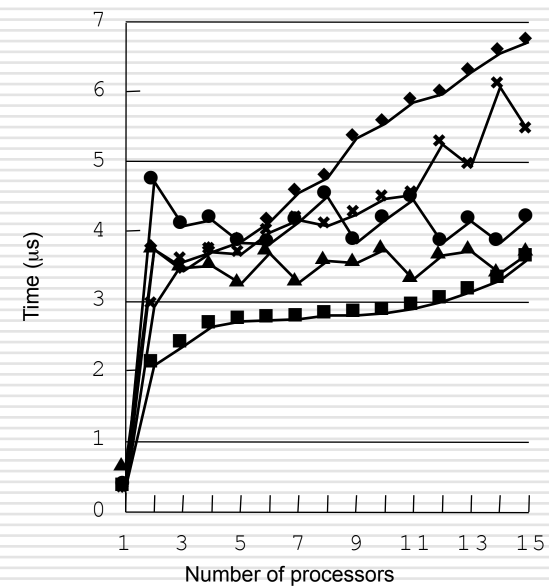
```
Loop: lock;
      delay(c);
      unlock;
      delay(d);
```



(a) Null (c = 0, d = 0)



(b) Critical-section (c = 3.64 µs, d = 0)



(c) Delay (c = 3.64 µs, d = 1.29 µs)

Point to Point Event Synchronization

- Software methods:
  - Interrupts
  - Busy-waiting: use ordinary variables as flags
  - Blocking: use semaphores
- Full hardware support: *full-empty bit* with each word in memory
  - Set when word is “full” with newly produced data (i.e. when written)
  - Unset when word is “empty” due to being consumed (i.e. when read)
  - Natural for word-level producer-consumer synchronization
    - producer: write if empty, set to full; consumer: read if full; set to empty
  - Hardware preserves atomicity of bit manipulation with read or write
  - Problem: flexiblity
    - multiple consumers, or multiple writes before consumer reads?
    - needs language support to specify when to use
    - composite data structures?

Barriers

- Software algorithms implemented using locks, flags, counters
- Hardware barriers
  - Wired-AND line separate from address/data bus
    - Set input high when arrive, wait for output to be high to leave
  - In practice, multiple wires to allow reuse
  - Useful when barriers are global and very frequent
  - Difficult to support arbitrary subset of processors
    - even harder with multiple processes per processor
  - Difficult to dynamically change number and identity of participants
    - e.g. latter due to process migration
  - Not common today on bus-based machines

A Simple Centralized Barrier

- Shared counter maintains number of processes that have arrived
  - increment when arrive (lock), check until reaches numprocs
  - Problem?

```
struct bar_type {int counter; struct lock_type lock;
                                     int flag = 0;} bar_name;
BARRIER (bar_name, p) {
    LOCK(bar_name.lock);
    if (bar_name.counter == 0)
        bar_name.flag = 0;              /* reset flag if first to reach*/
    mycount = bar_name.counter++;       /* mycount is private */
    UNLOCK(bar_name.lock);
    if (mycount == p) {                 /* last to arrive */
        bar_name.counter = 0;           /* reset for next barrier */
        bar_name.flag = 1;              /* release waiters */
    }
    else while (bar_name.flag == 0) {};   /* busy wait for release */
}
```

A Working Centralized Barrier

- Consecutively entering the same barrier doesn't work
  - Must prevent process from entering until all leave previous instance
  - Could use another counter, but increases latency and contention
- Sense reversal: wait for flag to flip value in consecutive times
  - Toggle this value only when all processes reach

```
BARRIER (bar_name, p) {
    local_sense = !(local_sense); /* toggle private sense variable */
    LOCK(bar_name.lock);
    mycount = bar_name.counter++;          /* mycount is private */
    if (bar_name.counter == p)
      UNLOCK(bar_name.lock);
      bar_name.flag = local_sense;         /* release waiters*/
    else
       {     UNLOCK(bar_name.lock);
      while (bar_name.flag != local_sense) {}; }
}
```

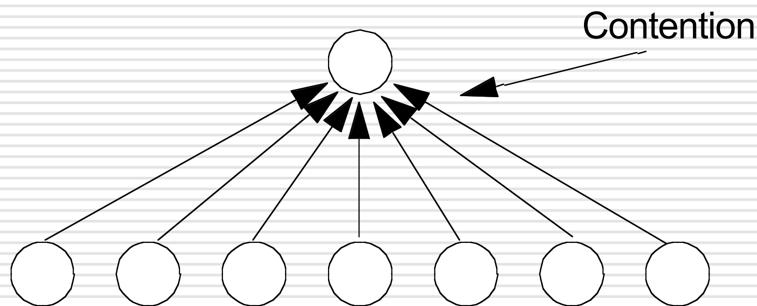
Centralized Barrier Performance

- Latency
  - Centralized has critical path length at least proportional to p
- Traffic
  - About $3p$ bus transactions
- Storage Cost
  - Very low: centralized counter and flag
- Fairness
  - Same processor should not always be last to exit barrier
  - No such bias in centralized
- Key problems for centralized barrier are latency and traffic
  - Especially with distributed memory, traffic goes to same node

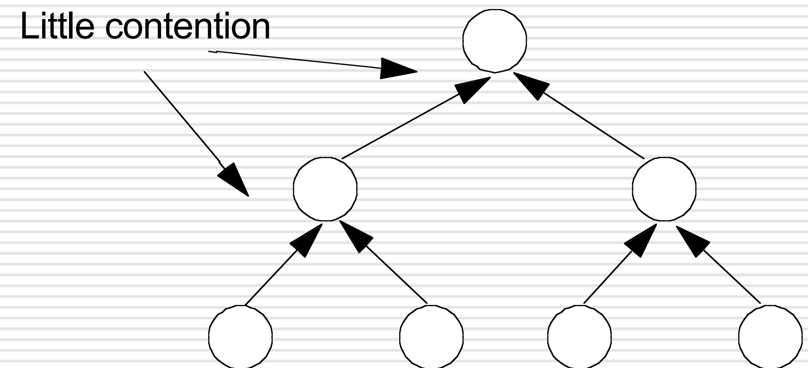
Improved Barrier Algorithms for a Bus

Software combining tree

- Only *k* processors access the same location, where *k* is tree degree



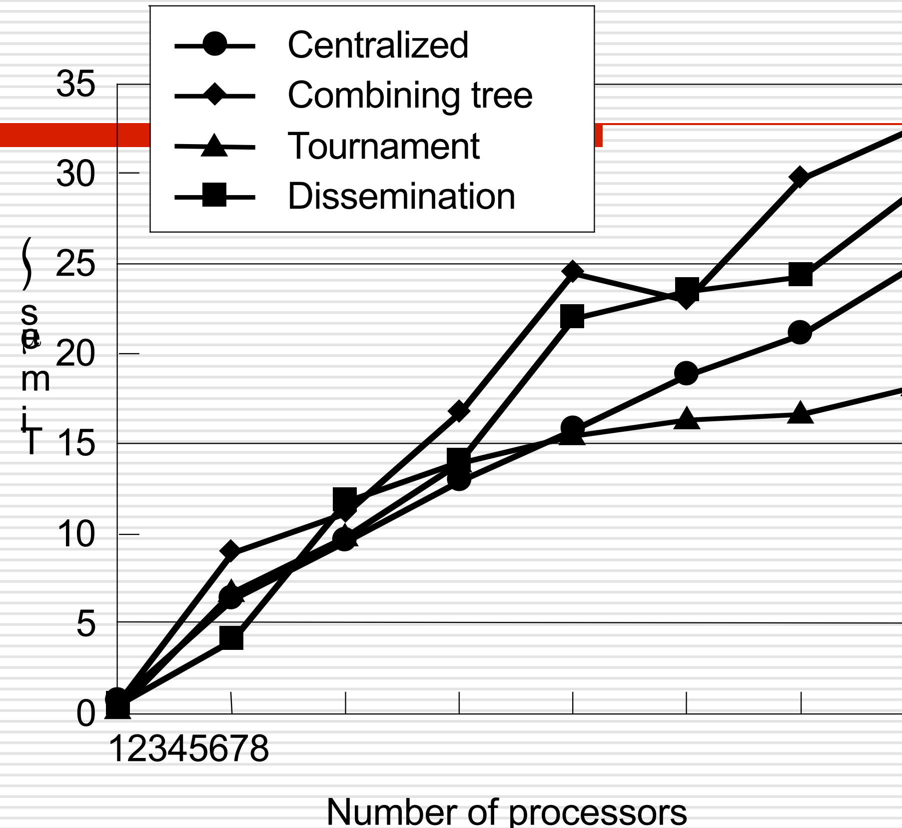
Flat



Tree structured

- Separate arrival and exit trees, and use sense reversal
- Valuable in distributed network: communicate along different paths
- On bus, all traffic goes on same bus, and no less total traffic
- Higher latency (*log p* steps of work, and *O(p)* serialized bus xactions)
- Advantage on bus is use of ordinary reads/writes instead of locks

Barrier Performance on SGI Challenge

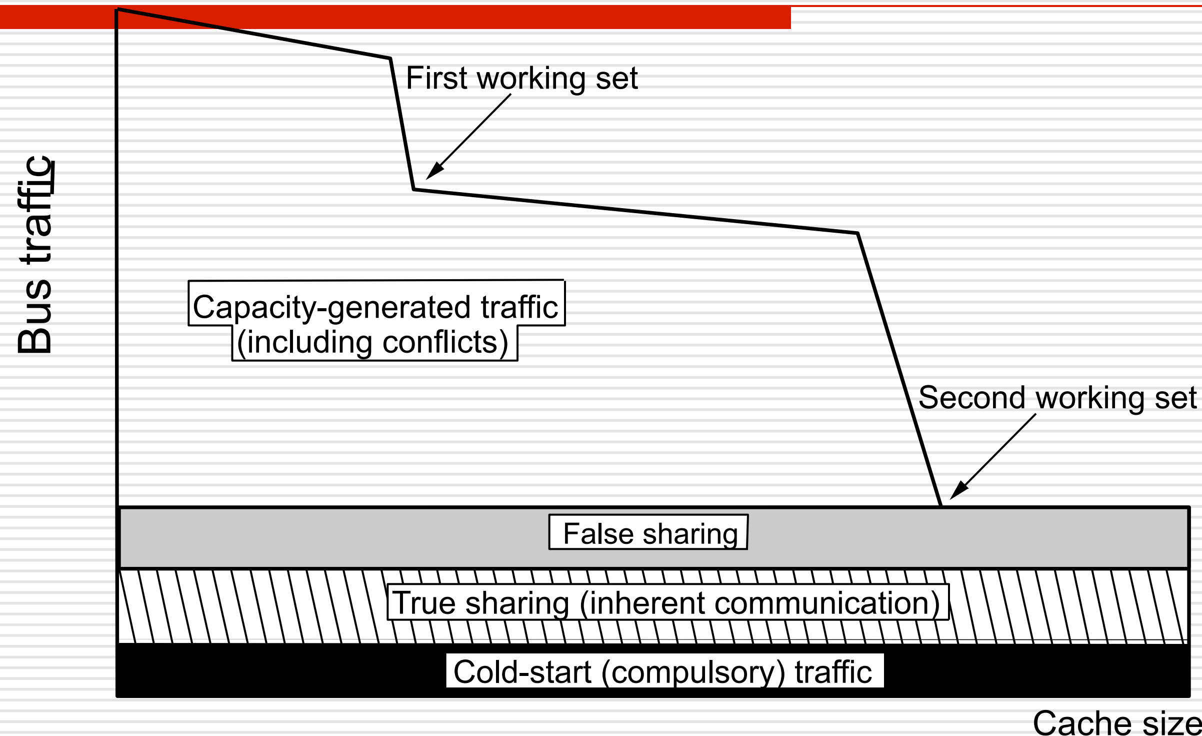


- Centralized does quite well
  - Will need fancier barrier algorithms for distributed machines
- Helpful hw support: piggybacking of reads misses (for flag) on bus
  - Also for spinning on highly contended locks

Synchronization Summary

- Rich interaction of hardware-software tradeoffs
- Must evaluate hardware primitives and software algorithms together
  - primitives determine which algorithms perform well
- Evaluation methodology is challenging
  - Use of delays, microbenchmarks
  - Should use both microbenchmarks and real workloads
- Simple software algorithms with common hardware primitives do well on bus
  - Will see more sophisticated techniques for distributed machines
  - Hardware support still subject of debate
- Research argues for swap or compare&swap, not fetch&op
  - Algorithms that ensure constant-time access, but complex

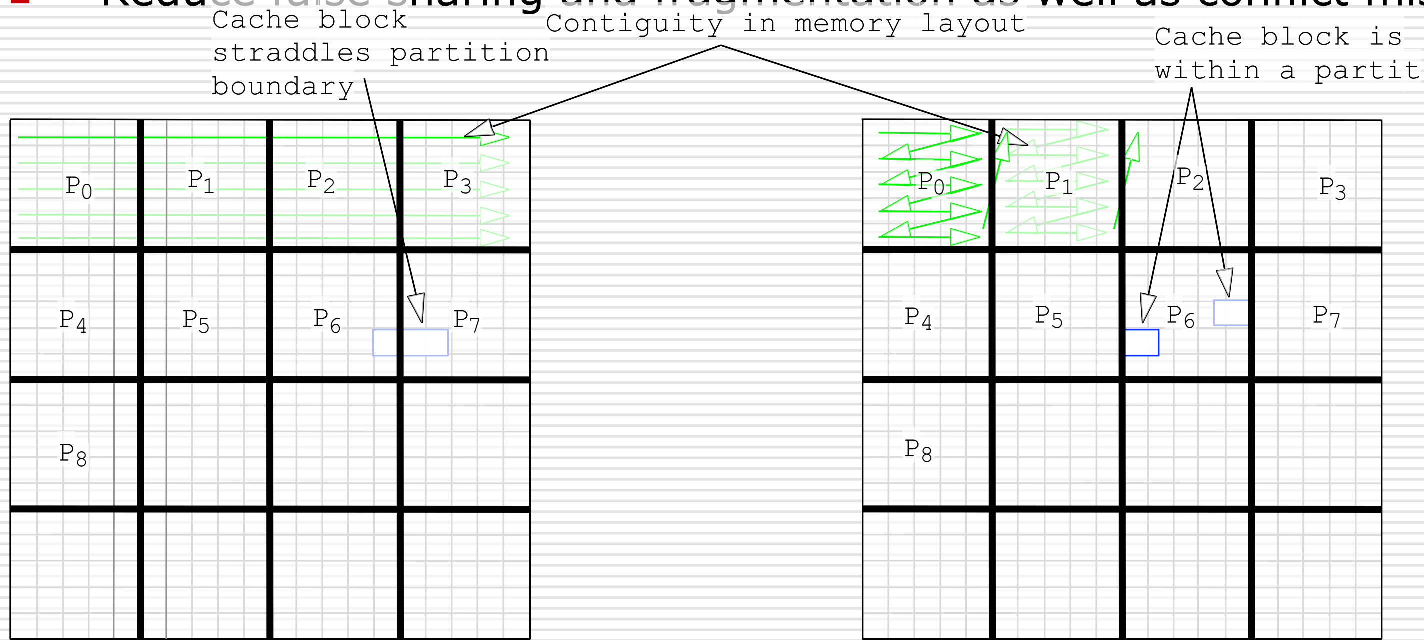
Implications for Software



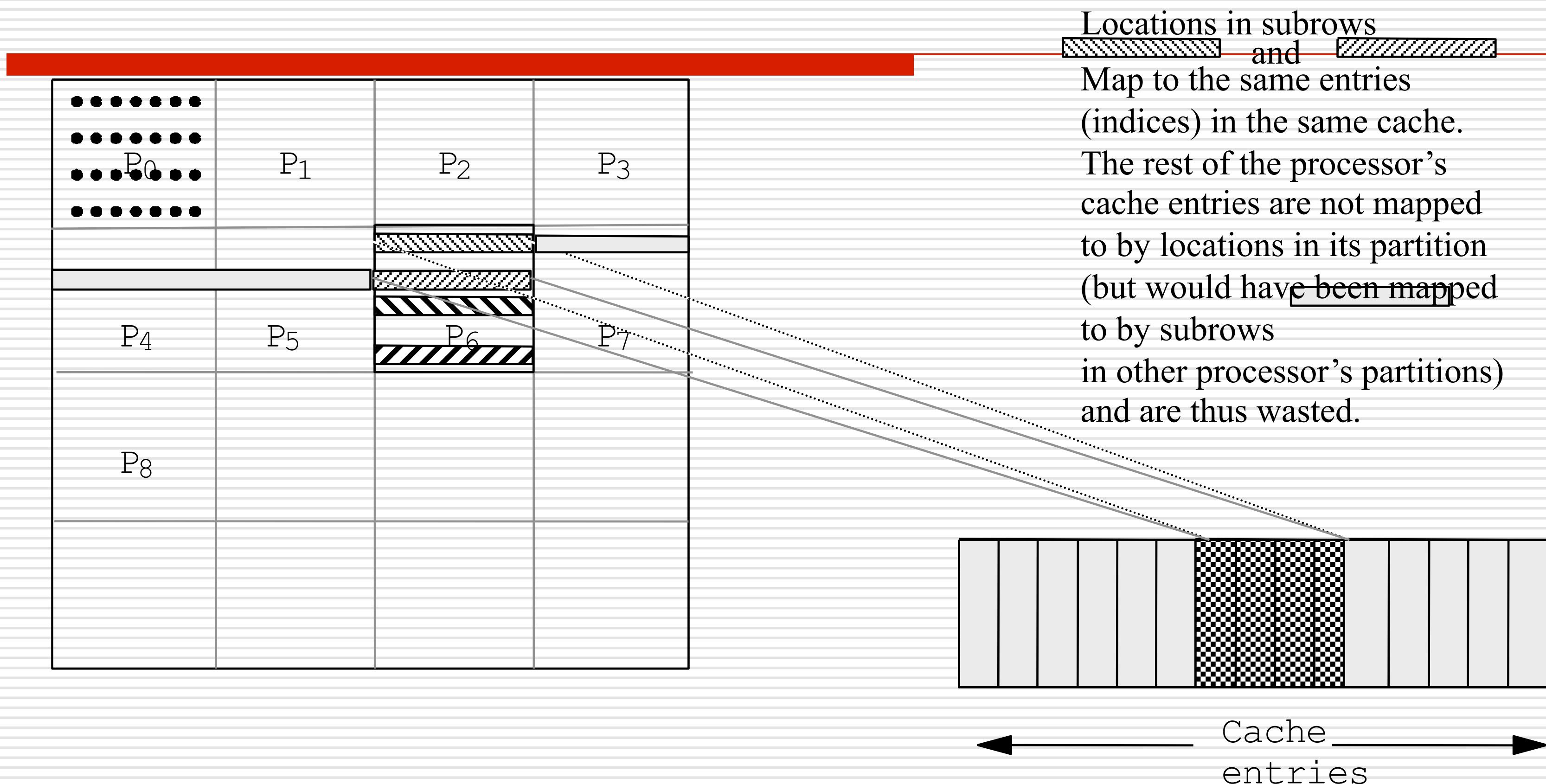
- Processor caches do well with temporal locality
- Synch. algorithms reduce inherent communication
- Large cache lines (spatial locality) less effective

Bag of Tricks for Spatial Locality

- Assign tasks to reduce spatial interleaving of accesses from procs
  - Contiguous rather than interleaved assignment of array elements
- Structure data to reduce spatial interleaving of accesses
  - Higher-dimensional arrays to keep partitions contiguous
  - Reduce false sharing and fragmentation as well as conflict misses



Conflict Misses in a 2-D Array Grid



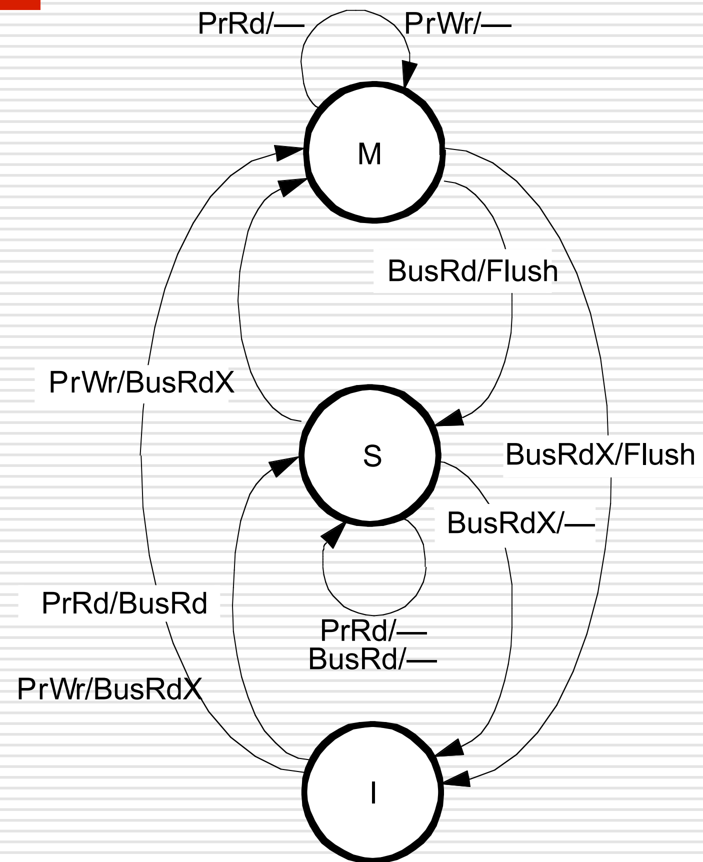
- Consecutive subrows of partition are not contiguous
- Problem when both array and cache size is power of 2

Bag of Tricks (contd.)

- Beware conflict misses more generally
  - Allocate non-power-of-2 even if application needs power-of-2
  - Conflict misses across data structures: ad-hoc padding/alignment
  - Conflict misses on small, seemingly harmless data
- Use per-processor heaps for dynamic memory allocation
- Copy data to increase locality
  - If noncontiguous data are to be reused
  - Must trade off against cost of copying
- Pad and align arrays: can have false sharing v. fragmentation tradeoff
- Organize arrays of records for spatial locality
  - E.g. particles with fields: organize by particle or by field
  - In vector programs by field for unit-stride, in parallel often by particle
  - Phases of program may have different access patterns and needs

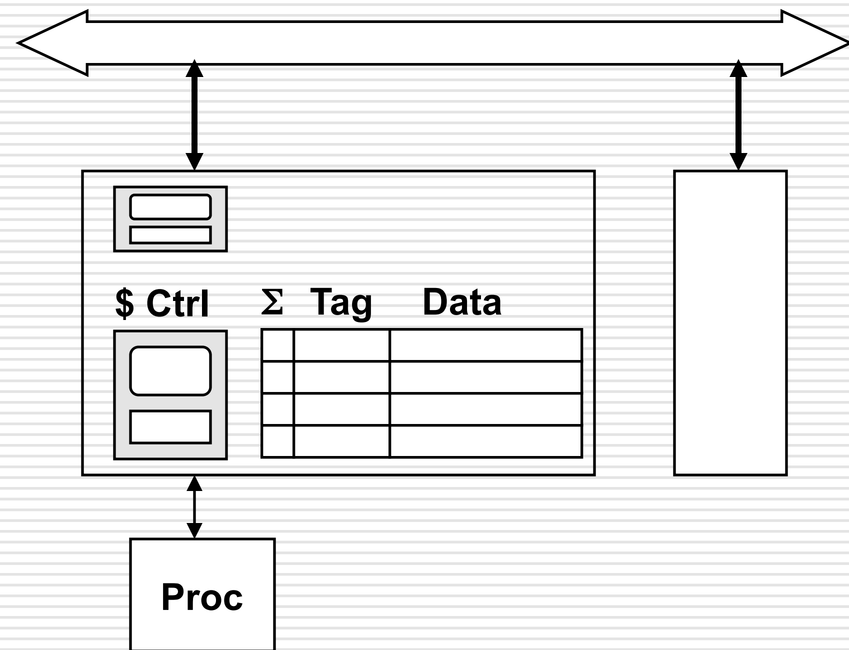
Logical Protocol Algorithm

- Set of States
- Events causing state transitions
- Actions on Transition

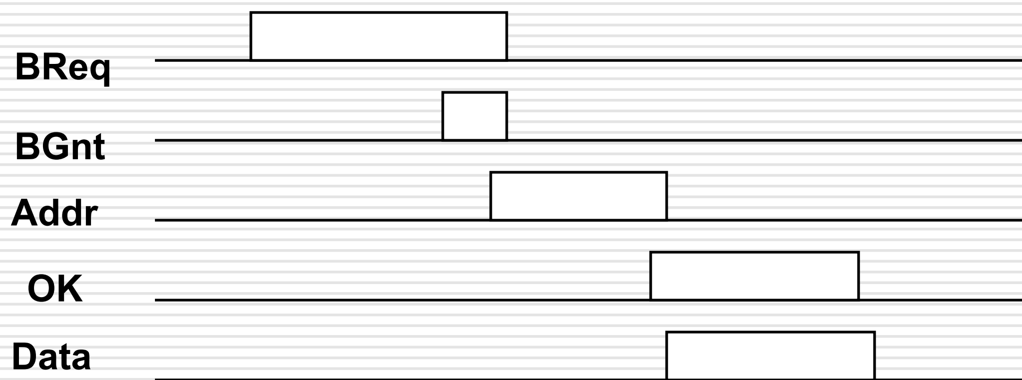


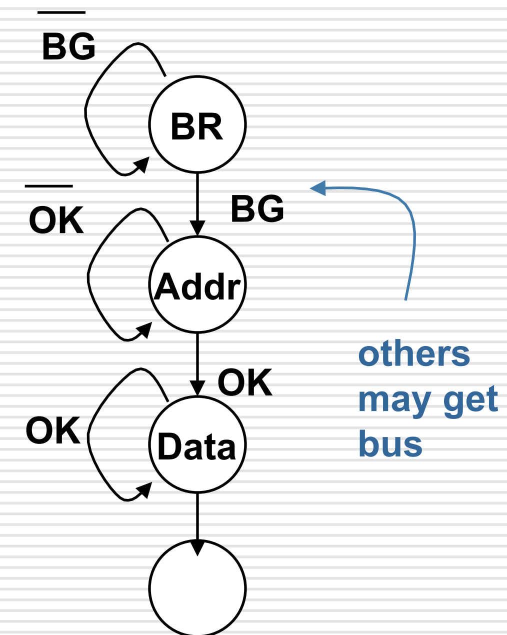
Reality

- Protocol defines logical FSM for each block
- Cache controller FSM
  - multiple states per miss
- Bus controller FSM
- Other $Ctrls Get bus
- Multiple Bus transactions
  - write-back
- Multi-Level Caches
- Split-Transaction Busses



Typical Bus Protocol

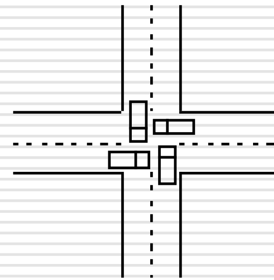


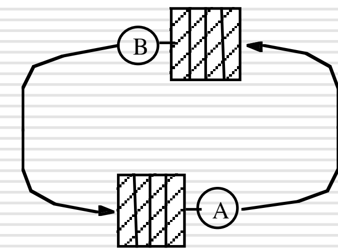


- Bus state machine
  - Assert request for bus
  - Wait for bus grant
  - Drive address and command lines
  - Wait for command to be accepted by relevant device
  - Transfer data

Correctness Issues

- Fulfill conditions for coherence and consistency
  - write propagation and 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 getting bus
- Starvation: some processors make no progress while others do.
  - e.g. interleaved memory system with NACK on bank busy
  - Often not completely eliminated (not likely, not catastrophic)

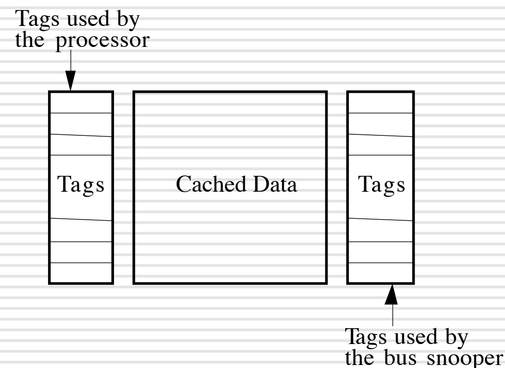
Preliminary Design Issues

- 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
- Atomic operations

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

Contention for Cache Tags

- 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



Reporting Snoop Results: How?

- Collective response from $'s 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

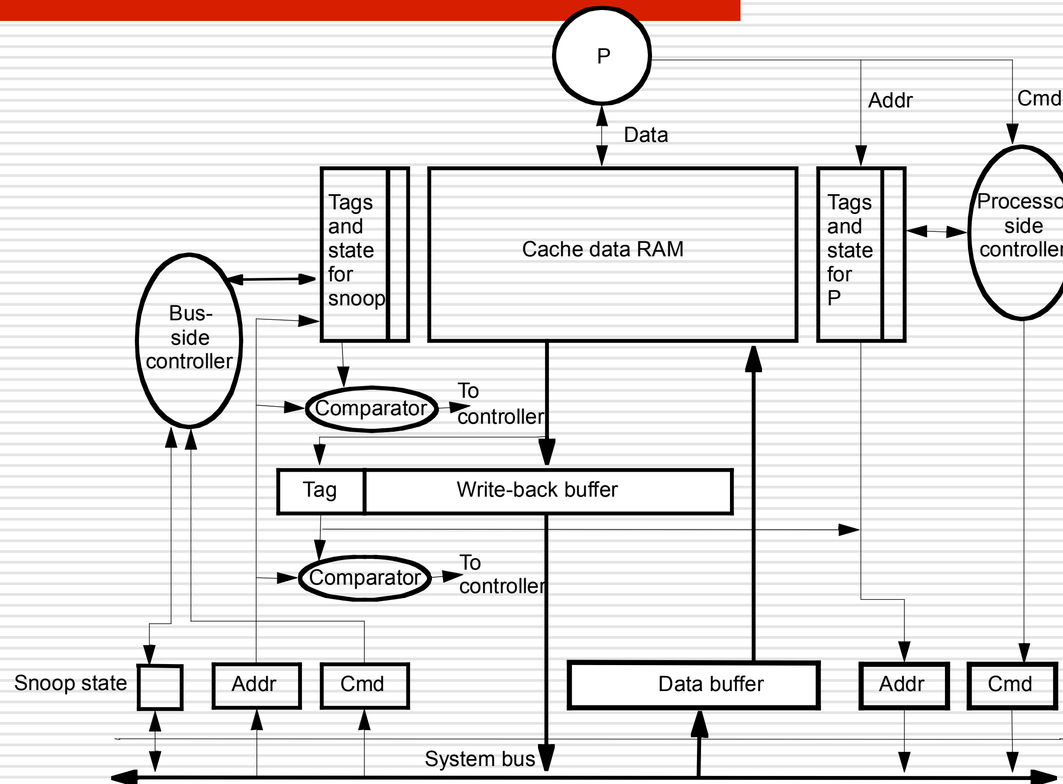
Reporting Snoop Results: When?

- 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

- 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

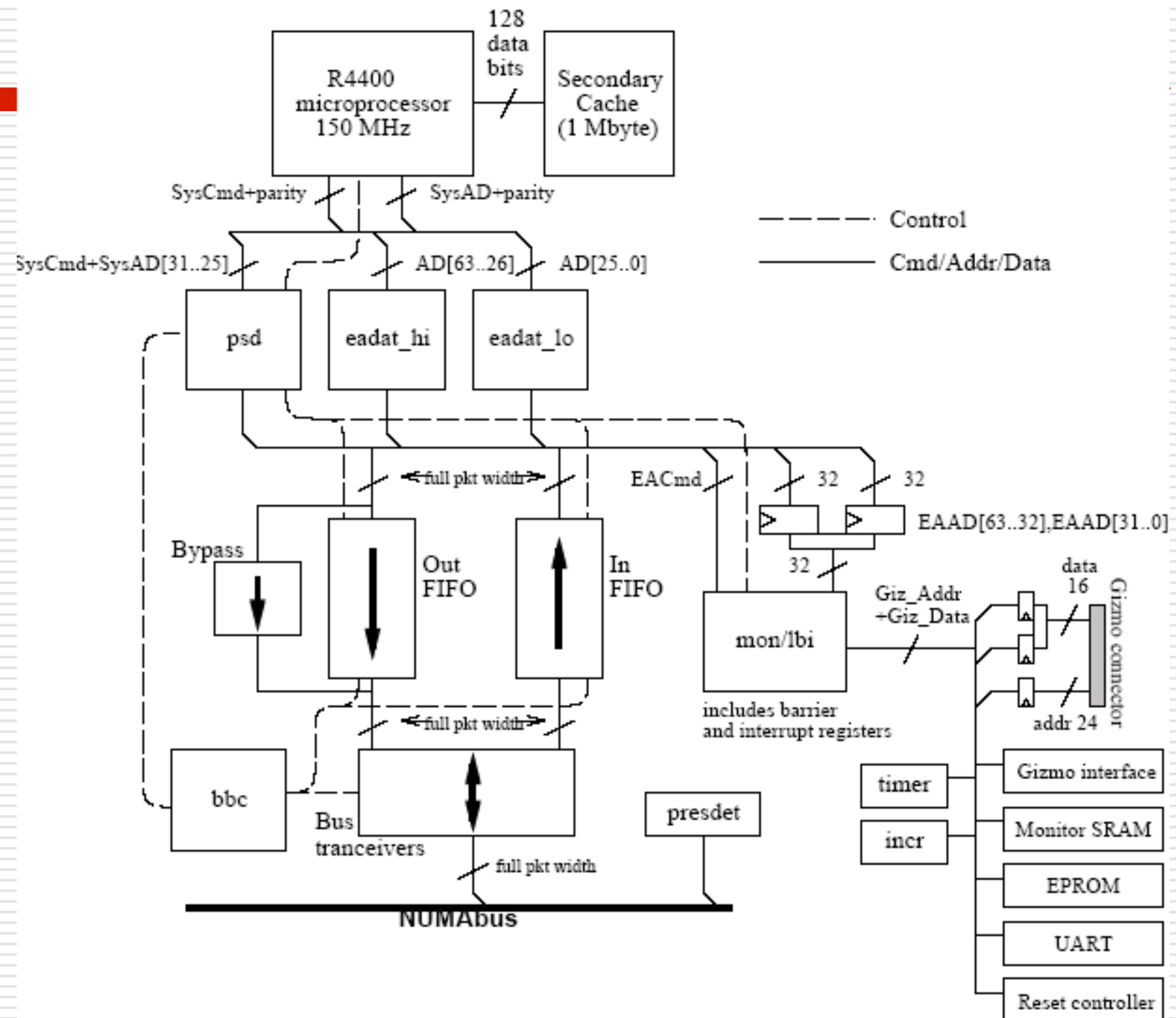
Basic design



- Comparison to NUMAchine (later)

NUMAchine Processor Board

- Much more realistic support for
  - CC
  - Split-tx bus
  - Scalability
  - Development
  - Debug
  - Multi-clocks



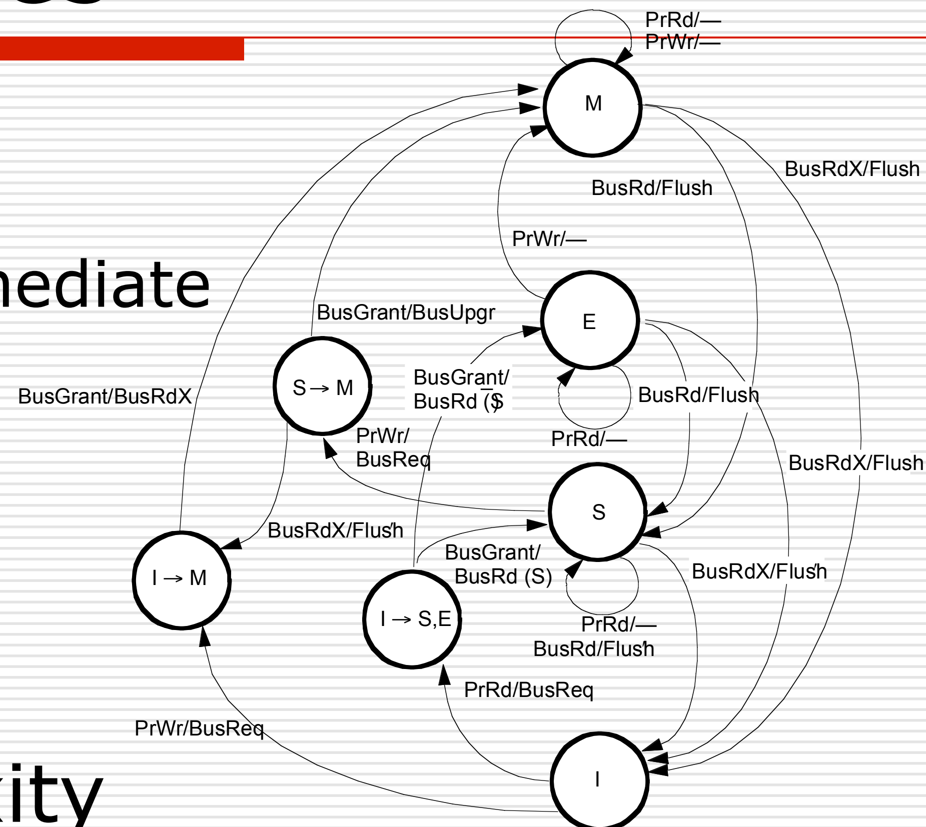
Non-Atomic State Transitions

- Memory operation involves 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



- Increases complexity
  - e.g. don't use BusUpgr, rather other mechanisms to avoid data transfer

Serialization

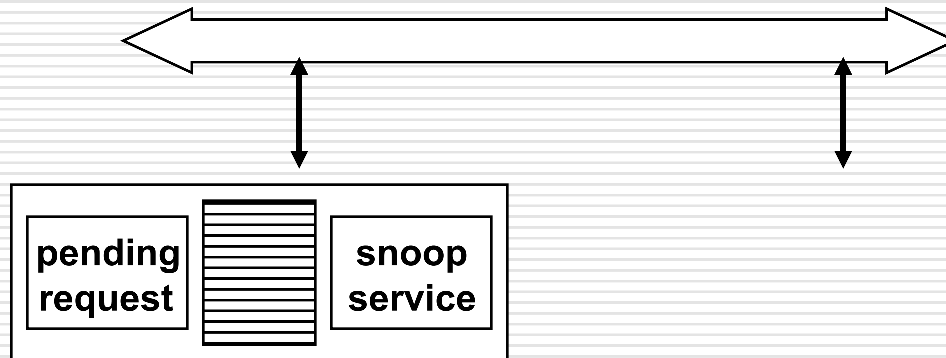
- Processor-cache handshake must preserve serialization of bus order
  - e.g. on write to block in S state, mustn't write data in block until ownership is acquired.
    - Other transactions that get bus before this one may seem to appear later

Write completion for SC?

- Needn't wait for inval to actually happen
  - Just wait till it gets bus
- *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

- 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
    - Cache controller awaiting bus grant must snoop and even flush blocks
    - else may not respond to request that will release bus



Livelock, Starvation

- 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 is done
- Starvation: Solve by using fair arbitration on bus and FIFO buffers

Implementing Atomic Operations

- In cache or memory?
  - Cacheable
    - Better latency and bandwidth on self-reacquisition
    - Allows spinning in cache without making traffic while waiting
  - At-memory
    - Lower transfer time
    - Used to be done with “locked” read-write pair of bus transitions
    - Not viable with modern, pipelined busses
  - Usually traffic and latency considerations dominate, so use cacheable
    - What is the implementation strategy?

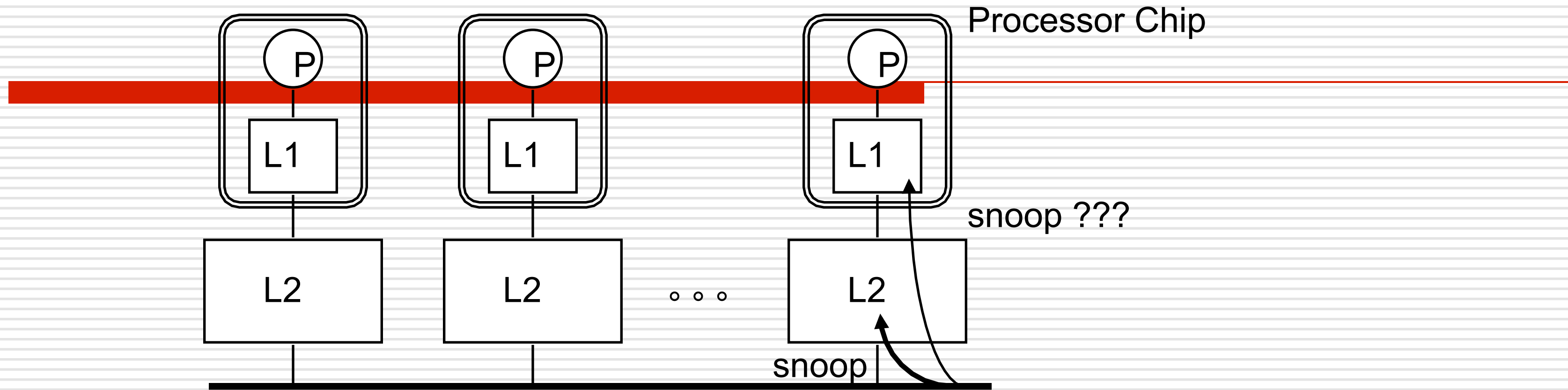
Use Cache Exclusivity for Atomicity

- Get exclusive ownership, read-modify-write
  - Error/don't allow conflicting bus transactions (Read or ReadEx)
  - Can actually buffer request if R-W is committed

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

Multilevel Cache Hierarchies



- Independent snoop hardware for each level?
  - processor pins for shared bus
  - contention for processor cache access ?
- Snoop only at L2 and propagate relevant transactions
- Inclusion property

  (1) contents L1 is a subset of L

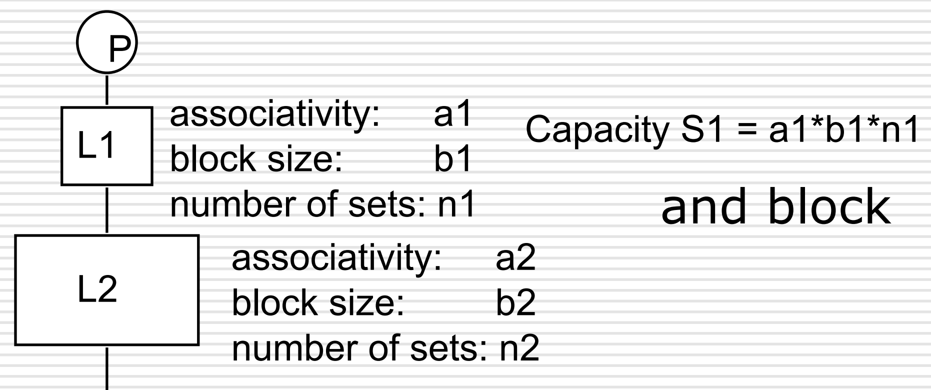
  (2) any block in modified state in L1 is in modified state in L2

  1 => all transactions relevant to L1 are relevant to L2

  2 => on BusRd L2 can wave off memory access and inform L1

Maintaining 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
- Common case - automatical
  - L1 direct-mapped, fewer sets than in L2, and block size same

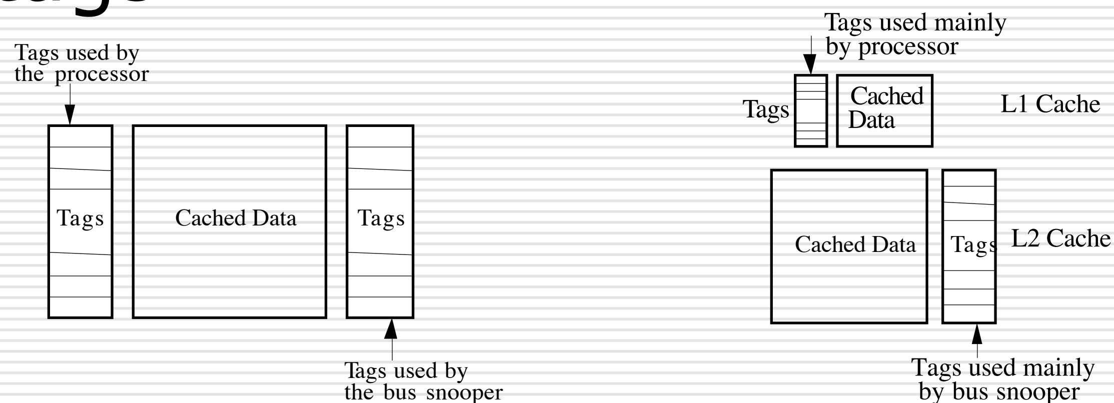


Preserving Inclusion Explicitly

- Propagate lower-level (L2) replacements to higher-level (L1)
  - Invalidate or flush (if dirty) messages
- Propagate bus transactions from L2 to L1
  - Propagate all L2 transactions?
  - use inclusion bits?
- Propagate modified state from L1 to L2 on writes?
  - if L1 is write-through, just invalidate
  - if L1 is write-back
    - add extra state to L2 (dirty-but-stale)
    - request flush from L1 on Bus Rd

Contention of Cache Tags

- L2 filter reduces contention on L1 tags

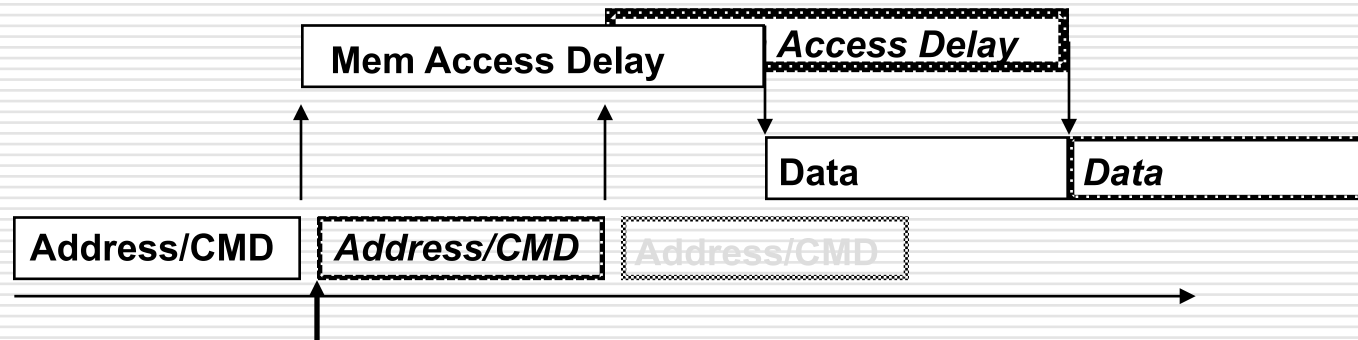


Correctness

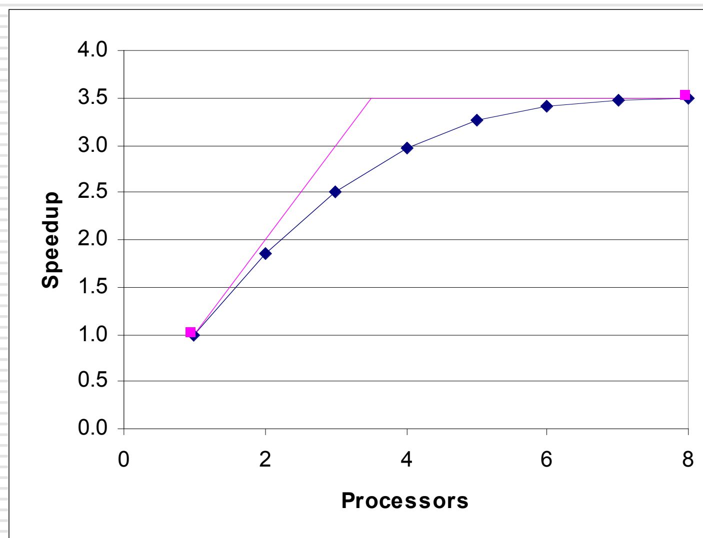
- Issues altered?
  - Not really, if all propagation occurs correctly and is waited for
  - Writes commit when they reach the bus, acknowledged immediately
  - But performance problems, so want to not wait for propagation
  - Same issues as split-transaction busses

Split-Transaction Bus

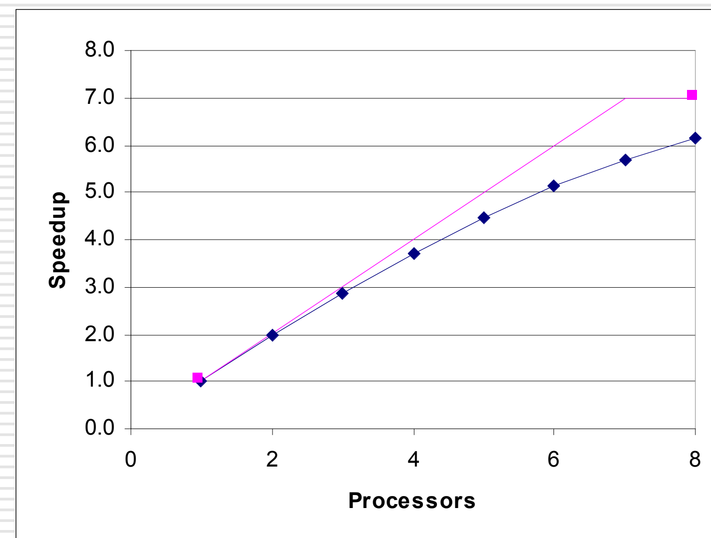
- Split bus transaction into request and response xactions
  - Separate arbitration for each phase
- Other transactions may intervene
  - Improves bandwidth dramatically
  - Response is matched to request
  - Buffering between bus and cache controllers
- Reduce serialization down to the actual bus arbitration



Impact of 2-stage Miss Processing

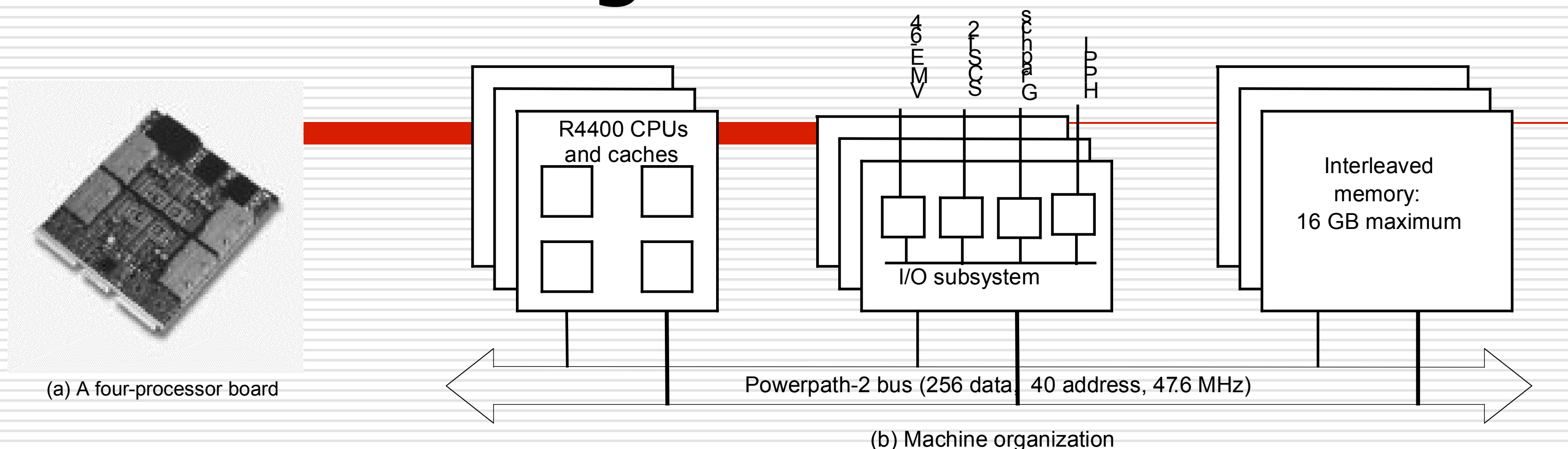


Z = 50 cycles
S = 20 cycles



Z = 60 cycles
S = 10 cycles

SGI Challenge Overview

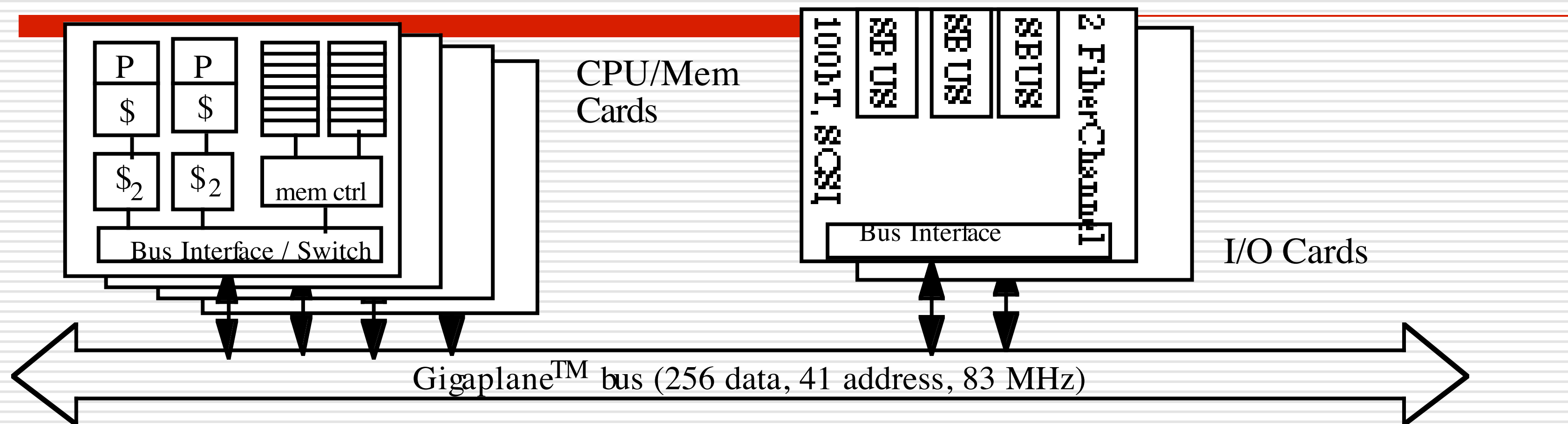


(a) A four-processor board

(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

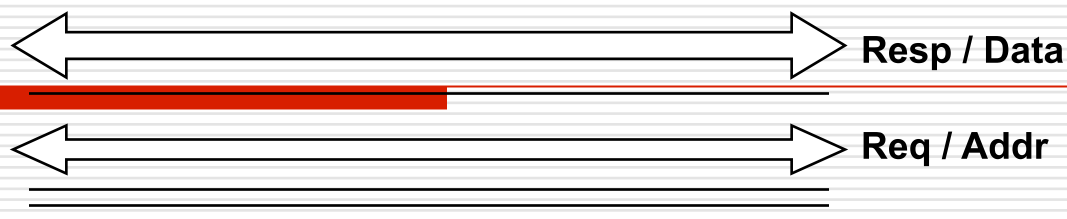
Complications

- 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 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)

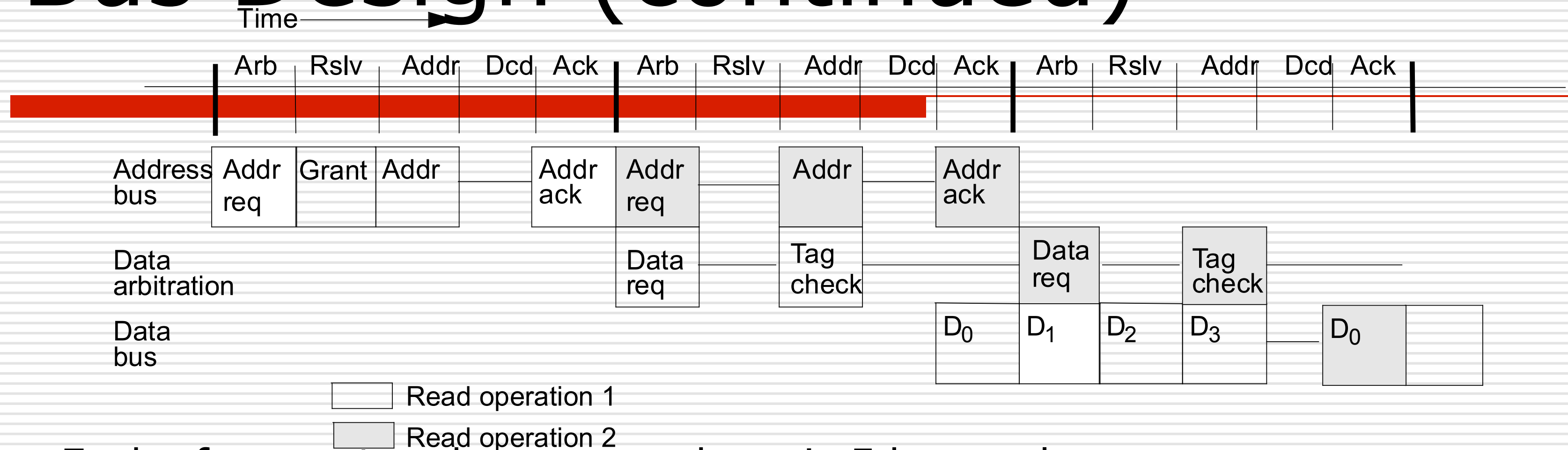
- No conflicting requests for same block allowed on bus
  - 8 outstanding requests total, makes conflict detection tractable
- Flow-control through negative acknowledgement (NACK)
  - NACK as soon as request appears on bus, requestor retries
  - Separate command (incl. NACK) + address and tag + data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

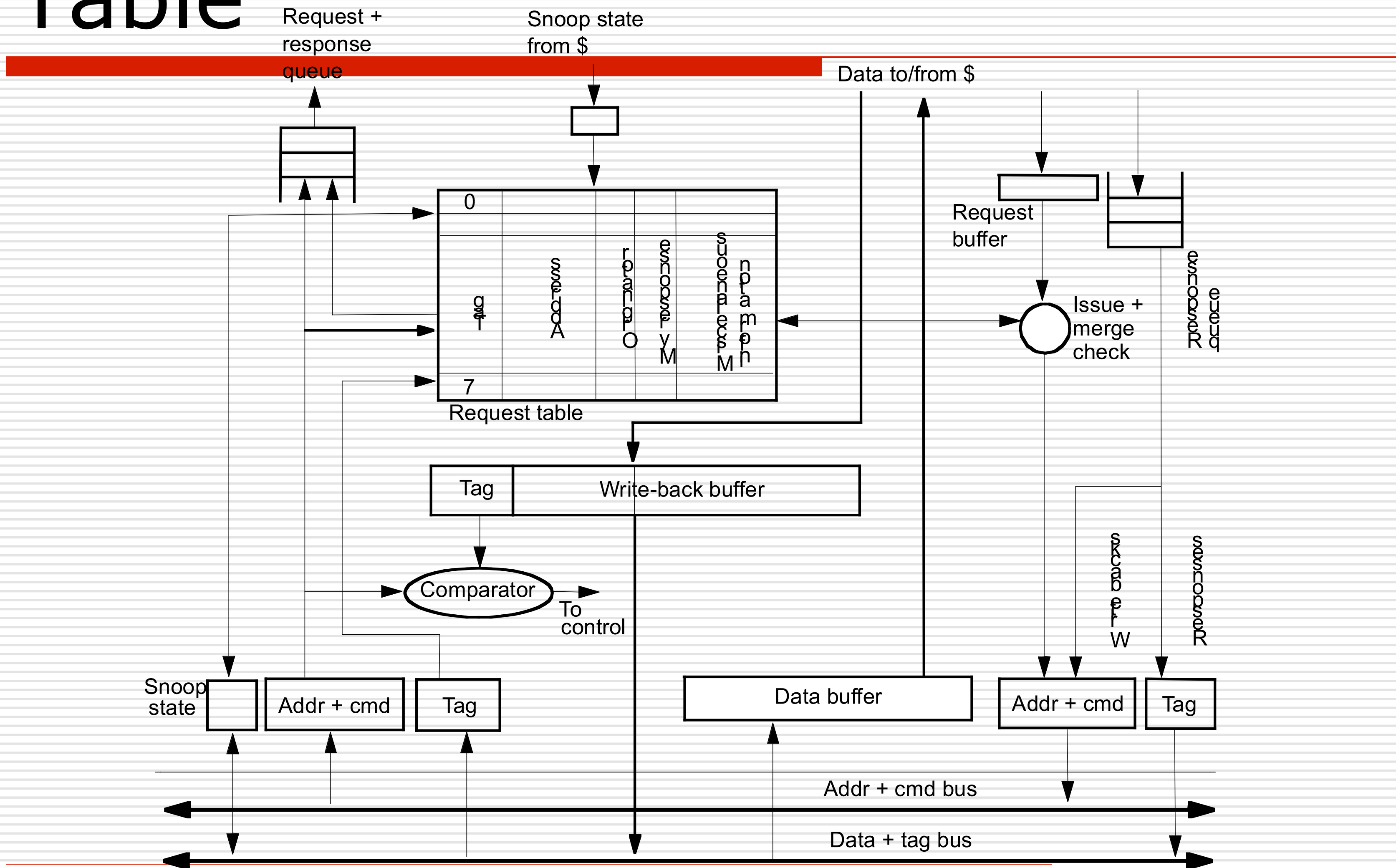


- Each of request and response phase is 5 bus cycles
  - Response: 4 cycles for data (128 bytes, 256-bit bus), 1 turnaround
  - Request phase: arbitration, resolution, address, decode, ack
  - Request-response transaction takes 3 or more of these
- Cache tags looked up in decode; extend ack cycle if 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, 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