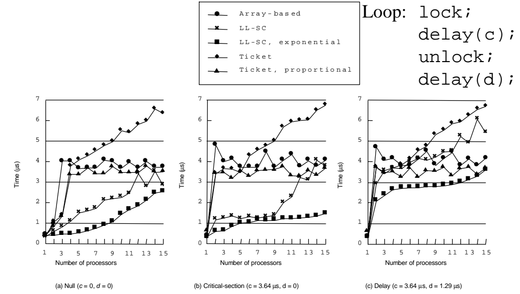
Logical Protocol to Physical Design

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

Lock Performance on SGI Challenge

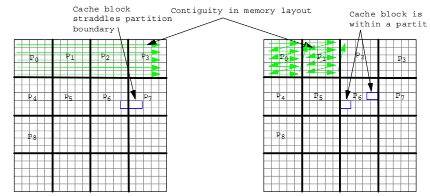


Barriers

- Single flag has problems on repeated use
  - only one when every one has reached the barrier, not whe they have left it
  - use two barriers
  - two flags
  - sense reversal
- Barrier complexity is linear on a bus, regardless of algorithm
  - tree-based algorithm to reduce contention

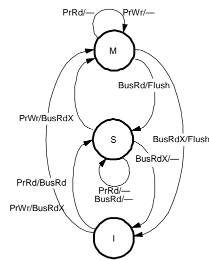
Bag of Tricks for Spatial Locality

- 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



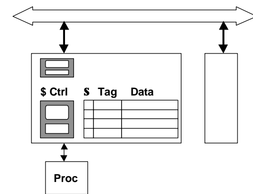
Logical Protocol Algorithm

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

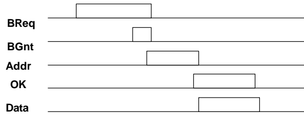


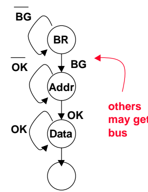
Reality

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



Typical Bus Protocol

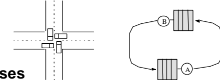




- Bus state machine
  - Assert request for bus
  - Wait for bus grant
  - Drive address and command lines
  - Wait for command to be accepted by relevant device
  - Transfer data

Correctness Issues

- Fulfill conditions for coherence and consistency
  - write propagation and atomicity



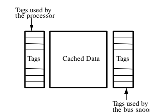
- Deadlock: all system activity ceases
  - Cycle of resource dependences
- Livelock: no processor makes forward progress although transactions are performed at hardware level
  - e.g. simultaneous writes in invalidation-based protocol
    - » each requests ownership, invalidating other, but loses it before 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)

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

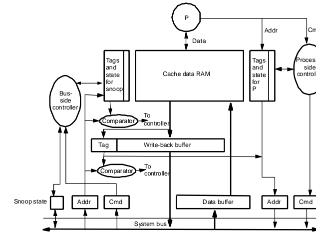
Reporting Snoop Results: When?

- 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

- 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

Basic design

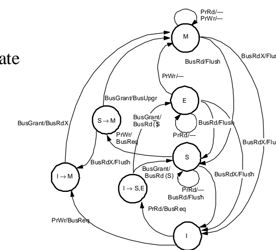


Non-Atomic State Transitions

- 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



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

Serialization

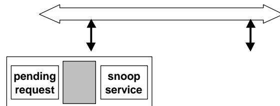
- Processor-cache handshake must preserve serialization of bus order
  - e.g. on write to block in S state, mustn't write data in block until ownership is acquired.
    - » other transactions that get bus before this one may seem to appear later

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 generating traffic while waiting
  - at-memory
    - » lower transfer time
    - » used to be implemented with "locked" read-write pair of bus transitions
    - » not viable with modern, pipelined busses
  - usually traffic and latency considerations dominate, so use cacheable
    - » what the implementation strategy?

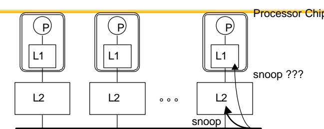
Use cache exclusivity for atomicity

- get exclusive ownership, read-modify-write
  - error 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

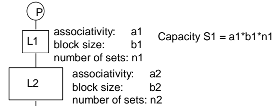


- Independent snoop hardware for each level?
  - processor pins for shared bus
  - contention for processor cache access ?
- Snoop only at L2 and propagate relevant transactions
- Inclusion property
  - (1) contents L1 is a subset of L
  - (2) any block in modified state in L1 is in modified state in L2
  - 1 => all transactions relavant to L1 are relavant to L2
  - 2 => on BusRd L2 can wave off memory access and inform L1

NOW Handout Page 4

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
- But a common case works automatically
  - L1 direct-mapped, fewer sets than in L2, and block size same

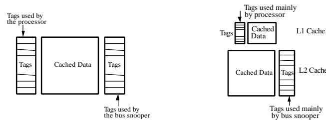


Preserving Inclusion Explicitly

- Propagate lower-level (L2) replacements to higher-level (L1)
  - Invalidate or flush (if dirty) messages
- Propagate bus transactions from L2 to L1
  - Propagate all 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