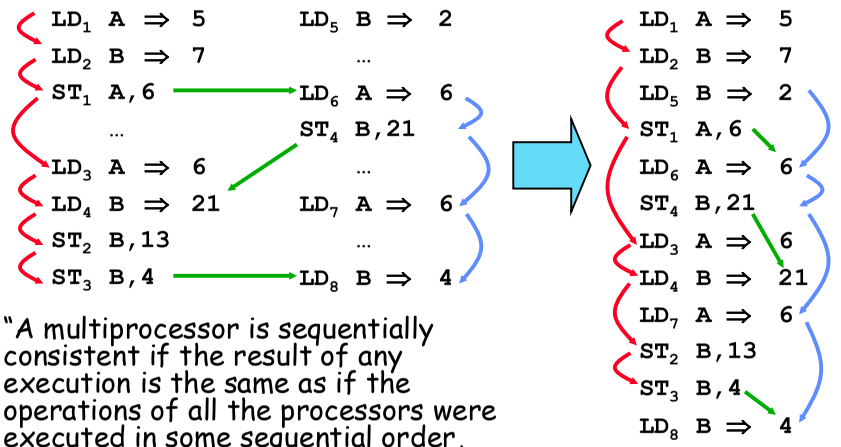
CS 258
Parallel Computer Architecture
Lecture 15

Sequential Consistency and Snoopy Protocols

March 17, 2008

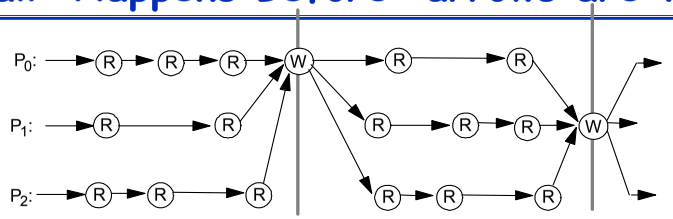
Prof John D. Kubiatowicz
http://www.cs.berkeley.edu/~kubitron/cs258

Recall: Sequential Consistency



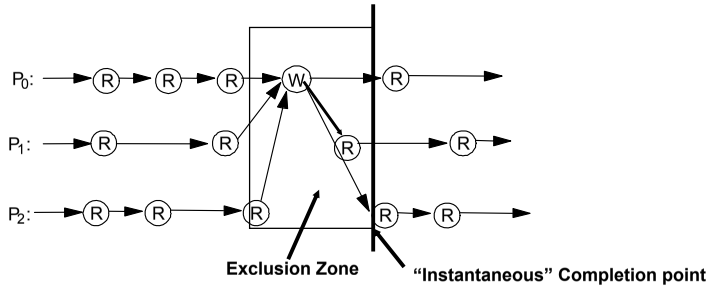
"A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." [Lamport, 1979]

Recall: Happens Before: arrows are time



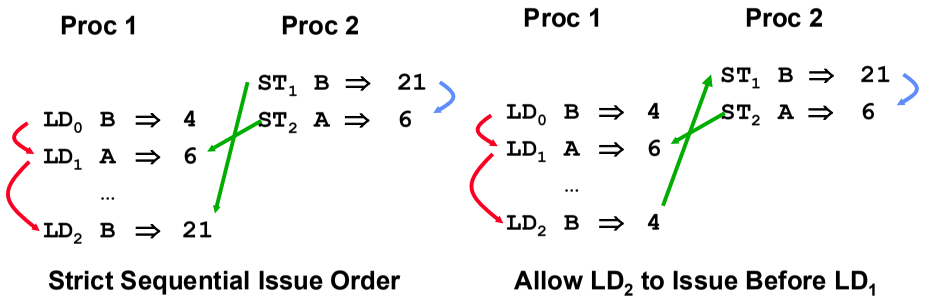
- Tricky part is relationship between nodes with respect to single location
  - Program order adds relationship between locations
- Easy topological sort comes up with sequential ordering assuming:
  - All happens-before relationships are time
  - Then – can't have time cycles (at least not inside classical machine in normal spacetime ☺).
- Unfortunately, writes are not instantaneous
  - What do we do?

Recall: Ordering: Scheurich and Dubois



- Sufficient Conditions for Sequential Consistency
  - every process issues mem operations in program order
  - after a write operation is issued, the issuing process waits for the write to complete before issuing next memory operation
  - after a read is issued, the issuing process waits for the read to complete and for the write whose value is being returned to complete (gloabaly) before issuing its next operation

What about reordering of accesses?



- Can LD_2 issue before LD_1?
  - Danger of getting CYCLE! (i.e. not sequentially consistent
- What can we do?
  - Go ahead and issue ld early, but watch cache
  - If value invalidated from cache early:
    - » Must squash LD_2 and any instructions that have used its value
- Reordering of Stores
  - Must be even more careful

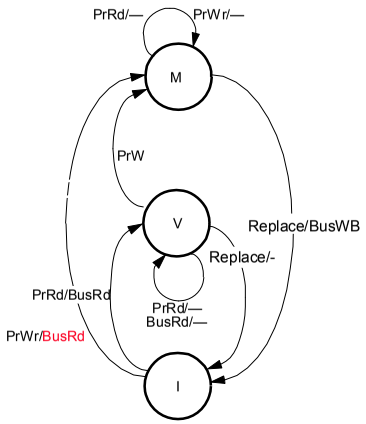
Write-back Caches (Uniprocessor)

- 2 processor operations
  - PrRd, PrWr
- 3 states
  - invalid, valid (clean), modified (dirty)
  - ownership: who supplies block
- 2 bus transactions:
  - read (BusRd), write-back (BusWB)
  - only cache-block transfers

⇒ treat Valid as "shared" and Modified as "exclusive"

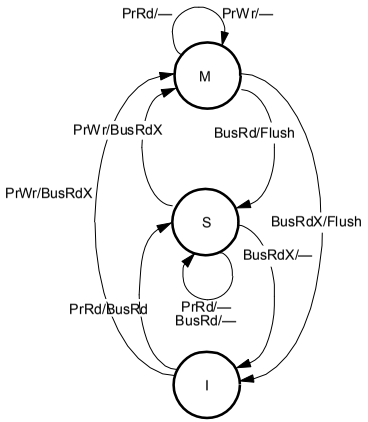
⇒ introduce one new bus transaction

- read-exclusive: read for purpose of modifying (read-to-own)

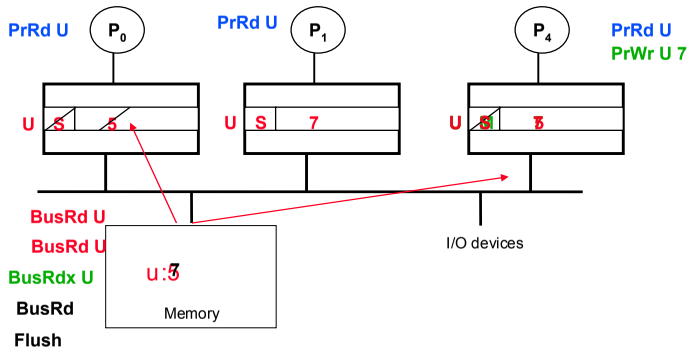


MSI Invalidate Protocol

- Three States:
  - "M": "Modified"
  - "S": "Shared"
  - "I": "Invalid"
- Read obtains block in "shared"
  - even if only cache copy
- Obtain exclusive ownership before writing
  - BusRdx causes others to invalidate (demote)
  - If M in another cache, will flush
  - BusRdx even if hit in S
    - » promote to M (upgrade)
- What about replacement?
  - S->I, M->I as before



Example: Write-Back Protocol



Correctness

- When is write miss performed?
  - How does writer "observe" write?
  - How is it "made visible" to others?
  - How do they "observe" the write?
- When is write hit made visible to others?
- When does a write hit complete globally?

Write Serialization for Coherence

- Writes that appear on the bus (BusRdX) are ordered by bus
  - performed in writer's cache before other transactions, so ordered same w.r.t. all processors (incl. writer)
  - Read misses also ordered wrt these
- Write that don't appear on the bus:
  - P issues BusRdX B.
  - further mem operations on B until next transaction are from P
    - » read and write hits
    - » these are in program order
  - for read or write from another processor
    - » separated by intervening bus transaction
- Reads hits?

Sequential Consistency

- Bus imposes total order on bus xactions for all locations
- Between xactions, procs perform reads/writes (locally) in program order
- So any execution defines a natural partial order
  - M_j subsequent to M_i if
    - » (i) M_j follows M_i in program order on same processor,
    - » (ii) M_j generates bus xaction that follows the memory operation for M_i
- In segment between two bus transactions, any interleaving of local program orders leads to consistent total order
- w/i segment writes observed by proc P serialized as:
  - Writes from other processors by the previous bus xaction P issued
  - Writes from P by program order
  - Insight: only one cache may have value in "M" state at a time

Sufficient conditions

- Sufficient Conditions
  - issued in program order
  - after write issues, the issuing process waits for the write to complete before issuing next memory operation
  - after read is issues, the issuing process waits for the read to complete and for the write whose value is being returned to complete (globally) before issuing its next operation
- Write completion
  - can detect when write appears on bus
- Write atomicity:
  - if a read returns the value of a write, that write has already become visible to all others already

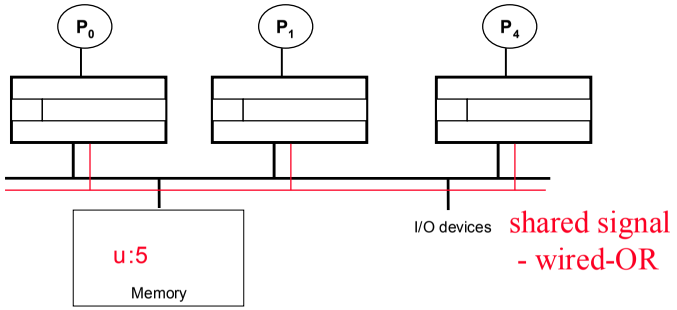
Lower-level Protocol Choices

- BusRd observed in M state: what transition to make?
  - M ----> I
  - M ----> S
  - Depends on expectations of access patterns
- How does memory know whether or not to supply data on BusRd?
- Problem: Read/Write is 2 bus xactions, even if no sharing
  - » BusRd (I->S) followed by BusRdX or BusUpgr (S->M)
  - » What happens on sequential programs?

MESI (4-state) Invalidation Protocol

- Four States:
  - "M": "Modified"
  - "E": "Exclusive"
  - "S": "Shared"
  - "I": "Invalid"
- Add *exclusive* state
  - distinguish exclusive (writable) and owned (written)
  - Main memory is up to date, so cache not necessarily owner
  - can be written locally
- States
  - invalid
  - exclusive or *exclusive-clean* (only this cache has copy, but not modified)
  - shared (two or more caches may have copies)
  - modified (dirty)
- I -> E on PrRd if no cache has copy
  - => How can you tell?

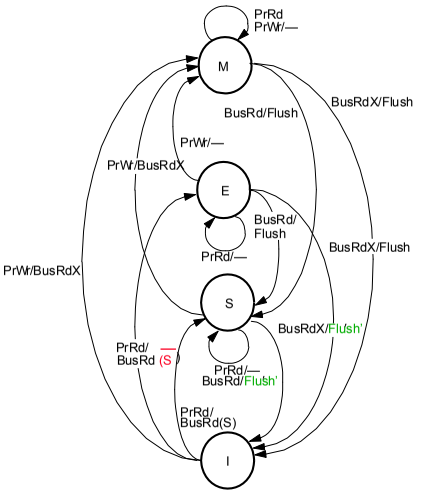
Hardware Support for MESI



- All cache controllers snoop on BusRd
- Assert 'shared' if present (S? E? M?)
- Issuer chooses between S and E
  - how does it know when all have voted?

MESI State Transition Diagram

- BusRd(S) means shared line asserted on BusRd transaction
- Flush': if cache-to-cache xfers
  - only one cache flushes data
- Replacement:
  - S→I can happen without telling other caches
  - E→I, M→I
- MOESI protocol: Owned state: exclusive but memory not valid



Lower-level Protocol Choices

- Who supplies data on miss when not in M state: memory or cache?
  - Original, Illinois MESI: cache, since assumed faster than memory
  - Not true in modern systems
    - » Intervening in another cache more expensive than getting from memory
- Cache-to-cache sharing adds complexity
  - How does memory know it should supply data (must wait for caches)
  - Selection algorithm if multiple caches have valid data
- Valuable for cache-coherent machines with distributed memory
  - May be cheaper to obtain from nearby cache than distant memory, Especially when constructed out of SMP nodes (Stanford DASH)

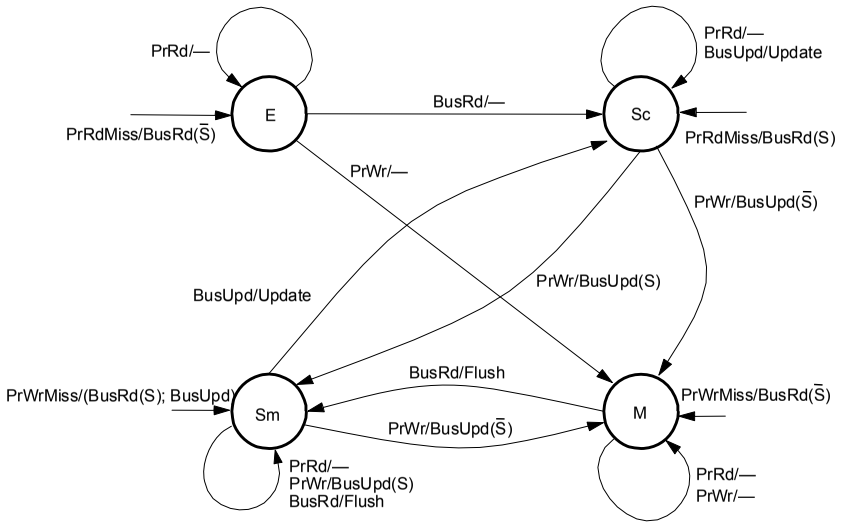
Update Protocols

- If data is to be communicated between processors, invalidate protocols seem inefficient
- consider shared flag
  - p0 waits for it to be zero, then does work and sets it one
  - p1 waits for it to be one, then does work and sets it zero
- how many transactions?

Dragon Write-back Update Protocol

- 4 states
  - Exclusive-clean or exclusive (E): I and memory have it
  - Shared clean (Sc): I, others, and maybe memory, but I'm not owner
  - Shared modified (Sm): I and others but not memory, and I'm the owner
    - » Sm and Sc can coexist in different caches, with only one Sm
  - Modified or dirty (D): I and, noone else
- No invalid state
  - If in cache, cannot be invalid
  - If not present in cache, view as being in not-present or invalid state
- New processor events: PrRdMiss, PrWrMiss
  - Introduced to specify actions when block not present in cache
- New bus transaction: BusUpd
  - Broadcasts single word written on bus; updates other relevant caches

Dragon State Transition Diagram



Lower-level Protocol Choices

- Can shared-modified state be eliminated?
  - If update memory as well on BusUpd transactions (DEC Firefly)
  - Dragon protocol doesn't (assumes DRAM memory slow to update)
- Should replacement of an Sc block be broadcast?
  - Would allow last copy to go to E state and not generate updates
  - Replacement bus transaction is not in critical path, later update may be
- Can local copy be updated on write hit before controller gets bus?
  - Can mess up serialization
- Coherence, consistency considerations much like write-through case

Assessing Protocol Tradeoffs

- Tradeoffs affected by technology characteristics and design complexity
- Part art and part science
  - Art: experience, intuition and aesthetics of designers
  - Science: Workload-driven evaluation for cost-performance
    - » want a balanced system: no expensive resource heavily underutilized

Summary

- Shared-memory machine
  - All communication is implicit, through loads and stores
  - Parallelism introduces a bunch of overheads over uniprocessor
- Memory Coherence:
  - Writes to a given location eventually propagated
  - Writes to a given location seen in same order by everyone
- Memory Consistency:
  - Constraints on ordering between processors and locations
- Sequential Consistency:
  - For every parallel execution, there exists a serial interleaving