Protocol Design Space of Snooping Cache Coherent Multiprocessors

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

Recap

- Snooping cache coherence
  - solve difficult problem by applying extra interpretation to naturally occuring events
    - » state transitions, bus transactions
  - write-thru cache
    - » 2-state: invalid, valid
    - » no new transaction, no new wires
    - » coherence mechanism provides consistency, since all writes in bus order
    - » poor performance
- Coherent memory system
- Sequential Consistency

Sequential Consistency

- Memory operations from a proc become visible (to itself and others) in program order
- There exist a total order, consistent with this partial order - i.e., an interleaving
  - the position at which a write occurs in the hypothetical total order should be the same with respect to all processors
- Sufficient Conditions
  - 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) befor issuing its next operation
- How can compilers violate SC? Architectural enhancements?

Outline for Today

- Design Space of Snoopy-Cache Coherence Protocols
  - write-back, update
  - protocol design
  - lower-level design choices
- Introduction to Workload-driven evaluation
- Evaluation of protocol alternatives

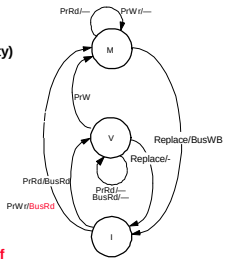
Write-back Caches

- 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"

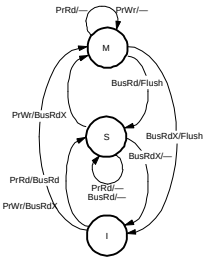
=> introduce one new bus transaction

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

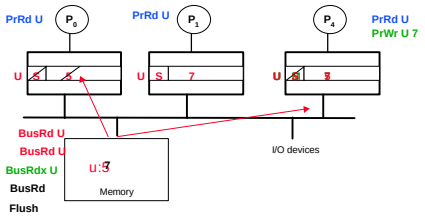


MSI Invalidate Protocol

- 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?

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) follows 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

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 (gloabaly) befor 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

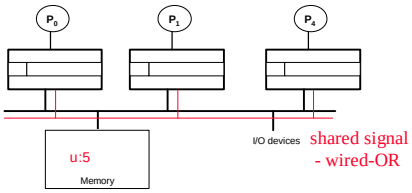
Lower-level Protocol Choices

- BusRd observed in M state: what transitition 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

- 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?

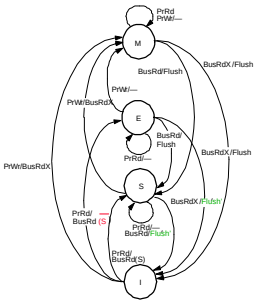
Hardware Support for MESI



- 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
- 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)

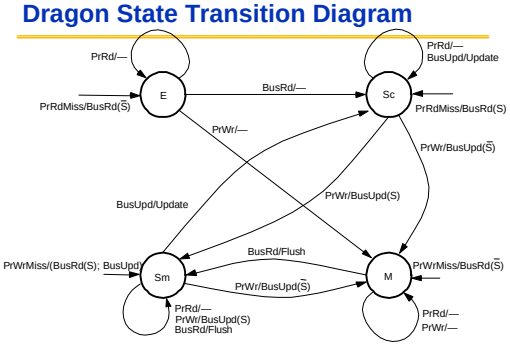
Update Protocols

- 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

Break?

Workload-Driven Evaluation

- Evaluating real machines
- Evaluating an architectural idea or trade-offs

=> need good metrics of performance

=> need to pick good workloads

=> need to pay attention to scaling
  - many factors involved

- Today: narrow architectural comparison
- Set in wider context

Evaluation in Uniprocessors

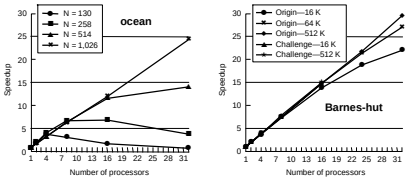
- Decisions made only after quantitative evaluation
- For existing systems: comparison and procurement evaluation
- For future systems: careful extrapolation from known quantities
- Wide base of programs leads to standard benchmarks
  - Measured on wide range of machines and successive generations
- Measurements and technology assessment lead to proposed features
- Then simulation
  - Simulator developed that can run with and without a feature
  - Benchmarks run through the simulator to obtain results
  - Together with cost and complexity, decisions made

More Difficult for Multiprocessors

- What is a representative workload?
- Software model has not stabilized
- Many architectural and application degrees of freedom
  - Huge design space: no. of processors, other architectural, application
  - Impact of these parameters and their interactions can be huge
  - High cost of communication
- What are the appropriate metrics?
- Simulation is expensive
  - Realistic configurations and sensitivity analysis difficult
  - Larger design space, but more difficult to cover
- Understanding of parallel programs as workloads is critical
  - Particularly interaction of application and architectural parameters

A Lot Depends on Sizes

- Application parameters and no. of procs affect inherent properties
  - Load balance, communication, extra work, temporal and spatial locality
- Interactions with organization parameters of extended memory hierarchy affect artifactual communication and performance
- Effects often dramatic, sometimes small: application-dependent



Understanding size interactions and scaling relationships is key

Scaling: Why Worry?

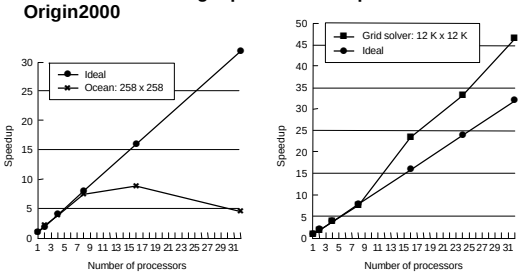
- Fixed problem size is limited
- Too small a problem:
  - May be appropriate for small machine
  - Parallelism overheads begin to dominate benefits for larger machines
    - » Load imbalance
    - » Communication to computation ratio
  - May even achieve slowdowns
  - Doesn't reflect real usage, and inappropriate for large machines
    - » Can exaggerate benefits of architectural improvements, especially when measured as percentage improvement in performance
- Too large a problem
  - Difficult to measure improvement (next)

Too Large a Problem

- Suppose problem realistically large for big machine
- May not "fit" in small machine
  - Can't run
  - Thrashing to disk
  - Working set doesn't fit in cache
- Fits at some *p*, leading to *superlinear speedup*
- Real effect, but doesn't help evaluate effectiveness
- Finally, users want to scale problems as machines grow
  - Can help avoid these problems

Demonstrating Scaling Problems

- Small Ocean and big equation solver problems on SGI Origin2000

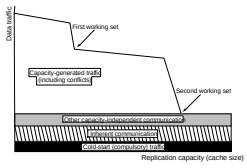


Communication and Replication

- View parallel machine as extended memory hierarchy
  - Local cache, local memory, remote memory
  - Classify "misses" in "cache" at any level as for uniprocessors
    - » compulsory or cold misses (no size effect)
    - » capacity misses (yes)
    - » conflict or collision misses (yes)
    - » communication or coherence misses (no)
- Communication induced by finite capacity is most fundamental artifact
  - Like cache size and miss rate or memory traffic in uniprocessors

Working Set Perspective

•At a given level of the hierarchy (to the next further one)



- Hierarchy of working sets
- At first level cache (fully assoc, one-word block), inherent to algorithm
  - » *working set curve* for program
- Traffic from any type of miss can be local or nonlocal (communication)