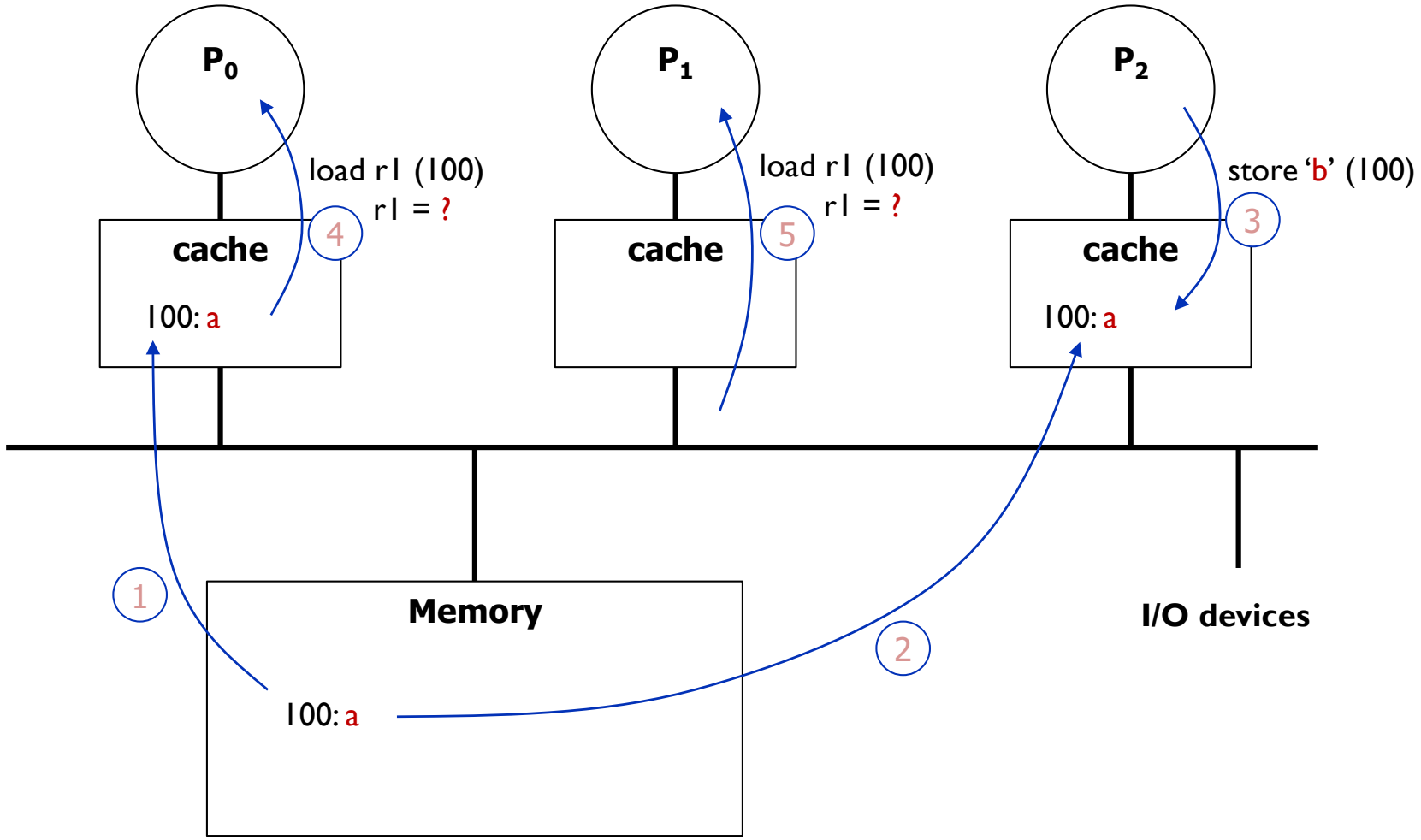
Shared Memory Multiprocessors

Jinkyu Jeong (jinkyu@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu

Cache Coherence Problem



Coherent Memory System: Intuition

- Reading a location should return latest value written by any process

- Easy in uniprocessors
  - Except for I/O (DMA)
    - Coherence between I/O devices and processors
  - But infrequent: SW solutions work
    - Uncacheable operations, flush pages, pass I/O data through caches

- Coherence problem is more pervasive and performance critical in multiprocessors
  - Much higher impact on hardware design

Problems with the Intuition

- The meaning of "last value" should be well-defined

- In sequential case:
  - "last" is defined in terms of program order, not time
    - Program order: order of operations in machine code

- In parallel case:
  - Program order defined within a process (thread), but orders across processes should be defined to make sense

- Meaningful semantics on parallel case
  - Involve both cache coherence and memory consistency model

Coherence and Consistency

- Cache coherence
  - Multiple local caches should work as if they are unified one
  - Read should get the value of the last write to any local cache
- Memory consistency
  - Sequential consistency: execution should be the same result as if ...
    - Some serial order among operations from multiple processors
    - Program order among operations in each individual processor
  - Weak consistency: needed for better performance
    - Written values are available after exit of critical section (synchronization)

<Proc 1>	<Proc 2>
A = 1	x = B
B = 1	y = A

(x,y) = (0,0), (0,1), (1,1) : possible values
But (1,0) is impossible in sequential consistency

* All values are initialized to zeros
* Assume all statements are atomic

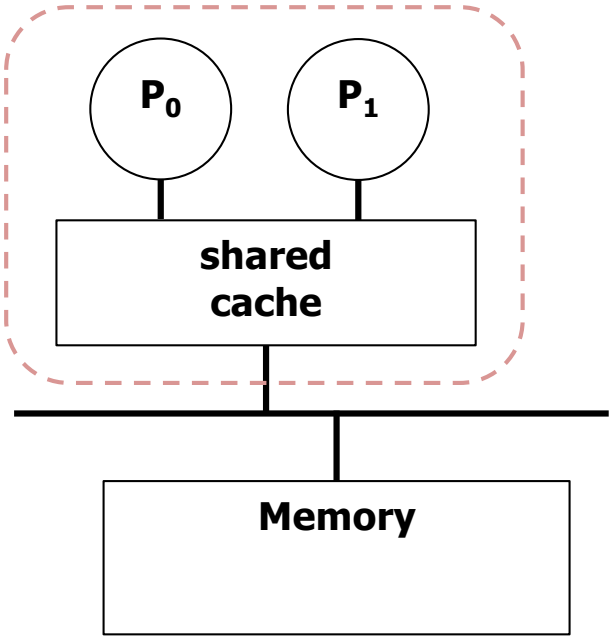
Cache Coherence Solutions

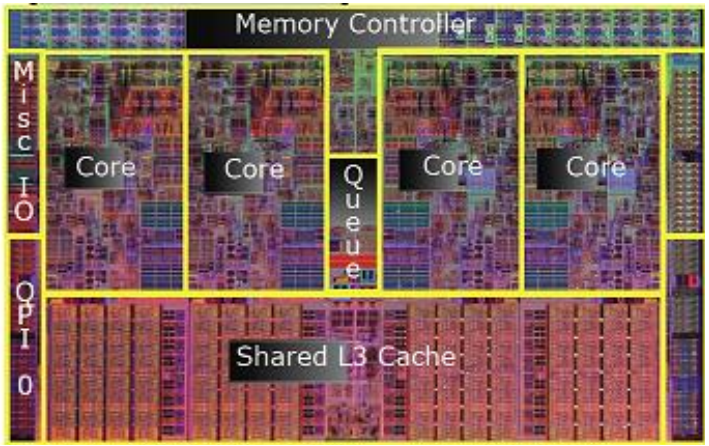
- Software based:
  - Often used in clusters of workstations or PCs (e.g. "Treadmarks")
  - Extend virtual memory system to perform more work on page faults
    - Send messages to remote machines if necessary

- Hardware based: two most common variations exist
  - "Snoopy" schemes
    - Rely on broadcast to observe all coherence traffic
    - Well suited for buses and small-sized systems (e.g. SGI Challenge)
  - "Directory" schemes
    - Uses centralized information to avoid broadcast
    - Scales well to large numbers of processors (e.g. SGI Origin 2000)

Shared Caches

- Processors share a single cache
  - Essentially eliminating the problem
- Useful for very small machines
  - Problems are limited cache bandwidth and cache interference
  - Benefits are fine-grain sharing and prefetch effects





Intel Quad Core i7

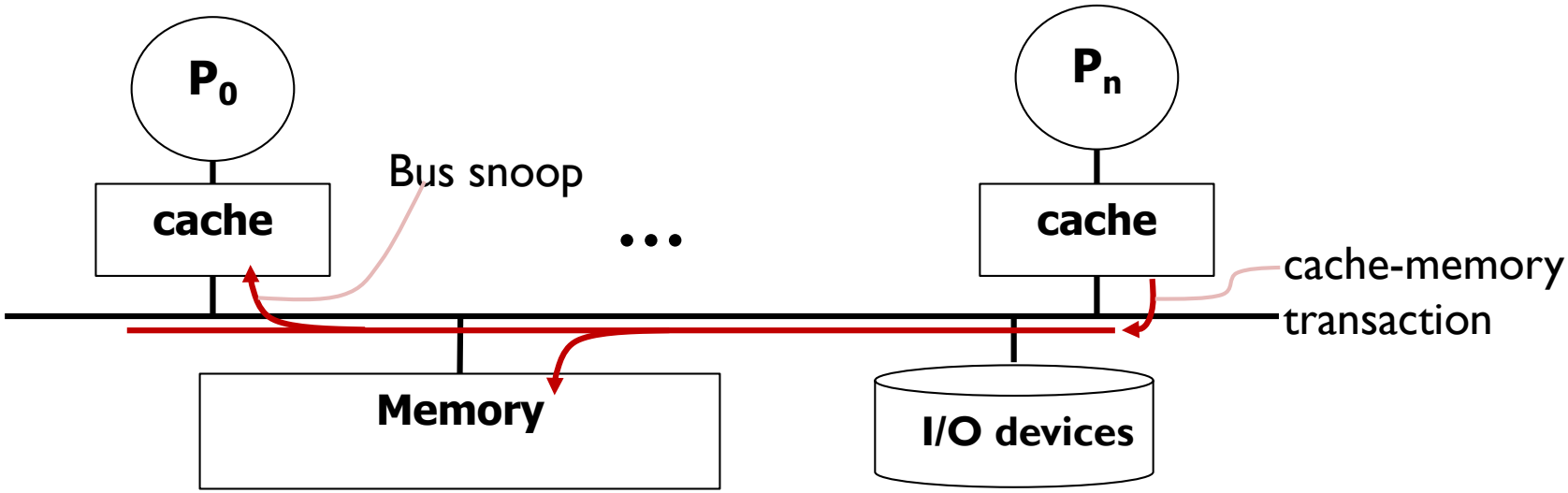
Snoopy Cache Coherence Schemes

- Basic idea
  - All coherence-related activities are broadcast to all processors
    - E.g. on a global bus
  - Each processor (or its representative) monitors (a.k.a. "snoops") these actions and reacts to any which are relevant to the current contents of its cache
    - If another processor wishes to write to a line, you may need to invalidate (i.e. discard) the copy in your own cache
    - If another processor whishes to read a line for which you have a dirty copy, you may need to apply
- Most common approach in multiprocessors
  - SGI Challenge, SUN Enterprise, multiprocessor PCs, etc.

Implementing a Snoopy Protocol

- Cache controller now receives inputs from both:
  - Requests from processor, bus requests/response from snooper
- React on inputs by ignoring or taking actions
  - Update states, respond with data, generate new bus transactions
- Protocol is a distributed algorithm
  - Cooperating state machines
  - Set of states, state transition diagram, actions
- Granularity of coherence is typically a cache block
  - Same as that of allocation in cache and transfer to/from cache

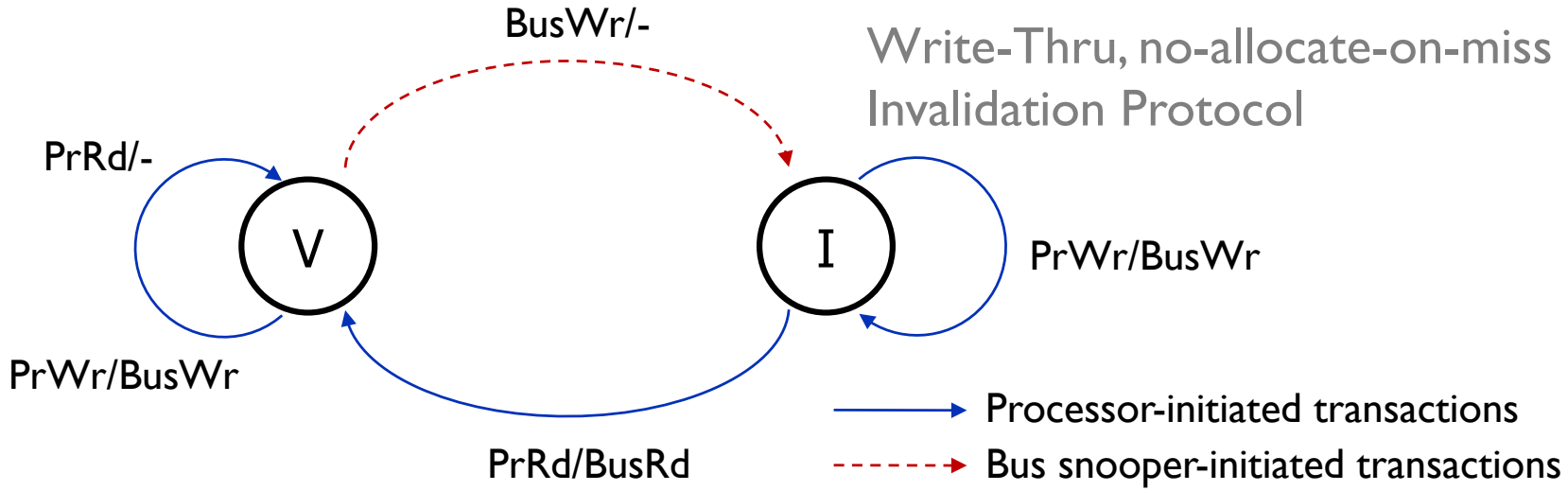
Coherence with Write-through Caches



- Key extensions to uniprocessors
  - Snooping, invalidating/updating caches
  - No new states or bus transactions in this case
  - Invalidation- vs. update-based protocols
- Write propagation
  - Invalid state causes miss on later access
  - Memory is up-to-date via write-through

Write-through State Transition

- Two states per block, as in uniprocessor
  - State of a block can be independently collected to build cache state
- State bits associated with only blocks in the cache
  - Other blocks regarded as being in invalid (not-present) state
- Write will invalidate all other caches, not local one
  - Can have multiple simultaneous readers, but write invalidates them



Problem with Write-Through

- High bandwidth requirements
  - Every write from every processor goes to shared bus and memory
    - E.g., a 3GHz, 1CPI processor, 15% of instructions are 8 byte stores
    - Each processor generates 450M stores or 3.6GB data per second
    - 5GB/s bus can support only 1 processor without saturating
    - Write-through especially unpopular for SMPs

- Write-back caches absorb most writes as cache hits
  - Write hits do not go on bus
  - But need to ensure write propagation and serialization
  - Need more sophisticated protocols: large design space exists

Write-Back Snoopy Protocols

- No need to change processor, main memory, cache
  - Extend cache controller and exploit bus (provides serialization)

- Dirty state now also indicates exclusive ownership
  - Exclusive: only cache with a valid copy (main memory may be so, too)
  - Owner: responsible for supplying block upon a request for it

- Design space
  - Invalidation- vs. Updated-based protocols
  - Set of states

Write-Back, Invalidate-Based Protocol

- Exclusive state means
  - "Can modify without notifying anyone else" – i.e. no bus transaction
  - Must first get block in exclusive state before writing into it
  - Even if already in valid state, need transaction (as if write miss)
- Store to non-dirty block
  - Generates a read-exclusive bus transaction
  - Tell others about impending write, obtain exclusive ownership
    - Makes the write visible, (notify "write is performed" to other processors)
    - May be actually observed later (by other processors via read misses)
    - Write hit made visible, when block updated in writer's cache
  - Only one RdX can succeed at a time for a block – serialized by bus
- Read and Read-exclusive (RdX) bus transactions drive coherence actions

Write-Back, Update-Based Protocol

- A write operation updates values in other caches
  - Update bus transaction – new transaction
- Advantages
  - Other processors don't miss on next access – reduced latency
    - In invalidation protocols, they would miss and cause more transactions
  - Single bus transaction to update several caches can save bandwidth
    - Also, only the word written is transferred, not whole block
- Disadvantages
  - Multiple writes by the same processor cause multiple update transactions
    - In invalidation, first write gets exclusive ownership, other writes are performed locally
- Detailed tradeoffs are more complex

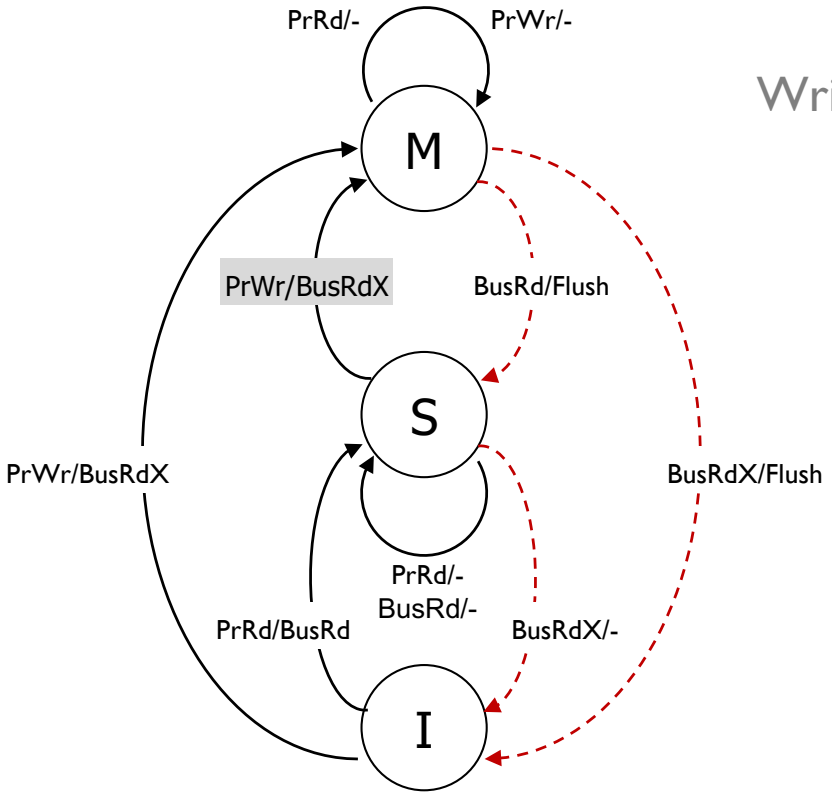
Invalidate vs. Update

- Basic question of program behavior
  - Is a block (written by one) read by others before it is rewritten?
    - Yes: …, P0Wr, P1Rd, P0Wr, …
    - No: …, P0Wr, P0Wr, P1Rd, …
  - Invalidation
    - Yes: readers will take a miss
    - No: multiple writes without additional traffic
  - Update
    - Yes: readers will not miss, if they had a copy previously
      - Single bus transaction to update all copies in other caches
    - No: multiple useless updates, even to dead copies
  - Need to look at program behavior and hardware complexity
- Invalidation protocols much more popular
  - Some systems provide both, or even hybrid

MSI Write-back Invalidation Protocol

- States
  - Invalid (I)
  - Shared (S) – one or more
  - Dirty or Modified (M) – one copy
- Processor events
  - PrRd – read from `load` instruction
  - PrWr – write from `store` instruction
- Bus transactions
  - BusRd – asks for copy with no intent to modify
  - BusRdX – ask for copy with intent to modify
  - BusWr (Flush) – updates memory
- Actions
  - Update state, perform bus transaction, flush value onto bus

State Transition Diagram



Write-Back, Invalidation Protocol

- Write to shared block
  - Already have latest data;
  - can use upgrade (BusUpgr) instead of BusRdx(S→M)
- Replacement changes state of two blocks: outgoing and incoming

Satisfying Coherence

- Write propagation is clear
- Write serialization
  - All writes that appear on the bus (BusRdX) ordered by the bus
    - Write performed in writer's cache before it handles other transactions – ordered in the same way even w.r.t. writer
  - Reads that appear on the bus ordered w.r.t. these
  - Writes that don't appear on the bus
    - Writes in M state from the same processor
    - Reads by all processors see writes in the same order
      - Read by P itself will see them in this order
      - Read by other processors will generate "BusRd" and the snooper in P will respond with "Flush". Thus, the others will see the content

Lower-Level Protocol Choices

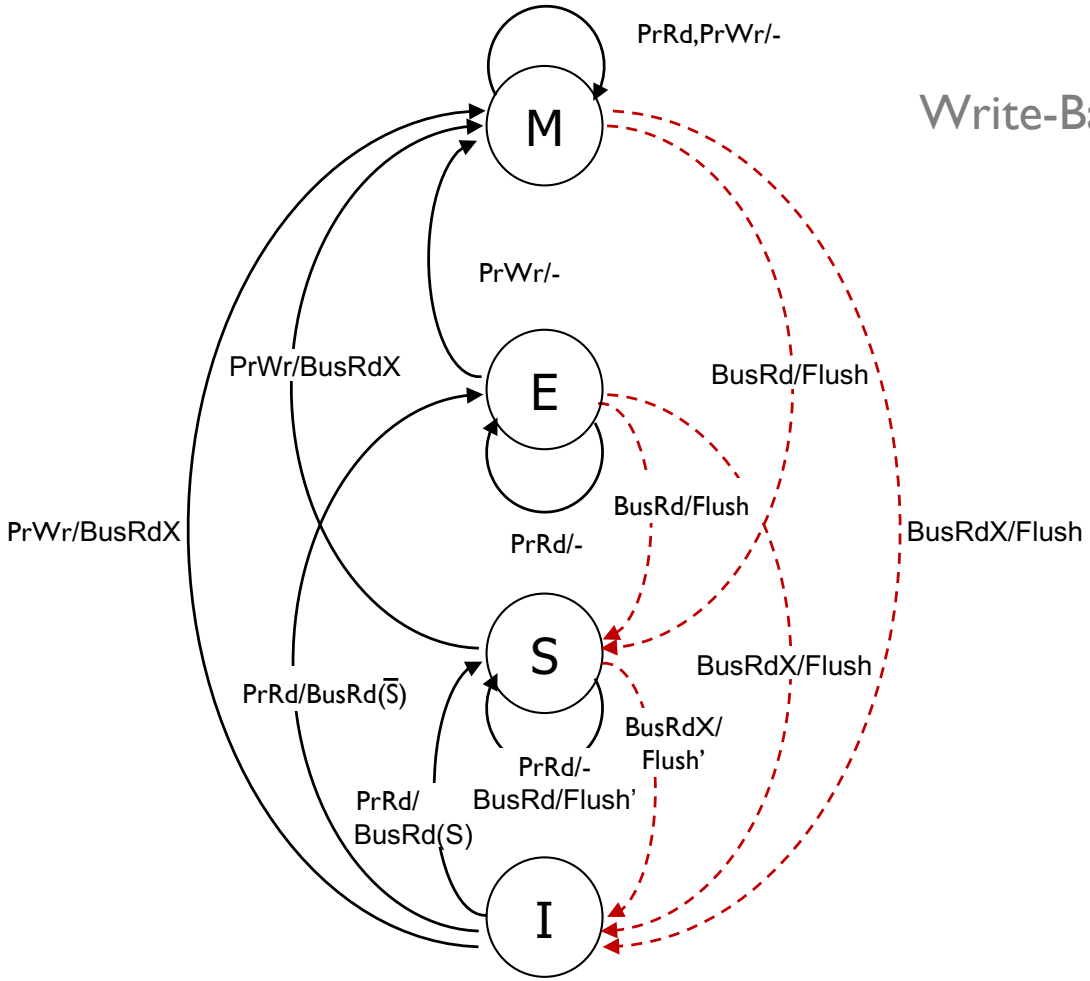
- BusRd observed in M state
  - What transition should be made?

- Depends on expectations of access patterns
  - S: assuming that I'll read it again soon, rather than other will write
    - Good for mostly read again
  - What about "migratory" data
    - Read-write operations from me, another, the third, ...
    - I state is better – don't have to be invalidated from other's write
  - *Synapse* transitioned to I state
  - *Sequent Symmetry* and *MIT Alewife* use adaptive protocols
- Choices can affect performance of memory system

MESI Invalidation Protocol

- Problem with MSI protocol
  - Read-and-modify requires 2 bus transactions even if not shared data
    - BusRd (I → S) followed by BusRdX or BusUpgr (S → M)
    - E.g. sequential programs will not share data even on multiprocessors
    - E.g. parallel programs also have many private data not to share
- Add exclusive state
  - Allows to write locally without bus transaction, but not modified
  - 4 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 other processor has a copy
    - Need "shared" signal on bus – wired-or asserted in response to BusRd

MESI State Transition Diagram



Write-Back, Invalidation Protocol

- BusRd(S) means shared line asserted on BusRd transaction
- Flush': if cache-to-cache sharing, only one cache flushes data
- Extended to MOESI protocol: Owned state – others may be in shared state, memory may be invalid (Owned state is responsible for memory write-back)

Lower-Level Protocol Choices

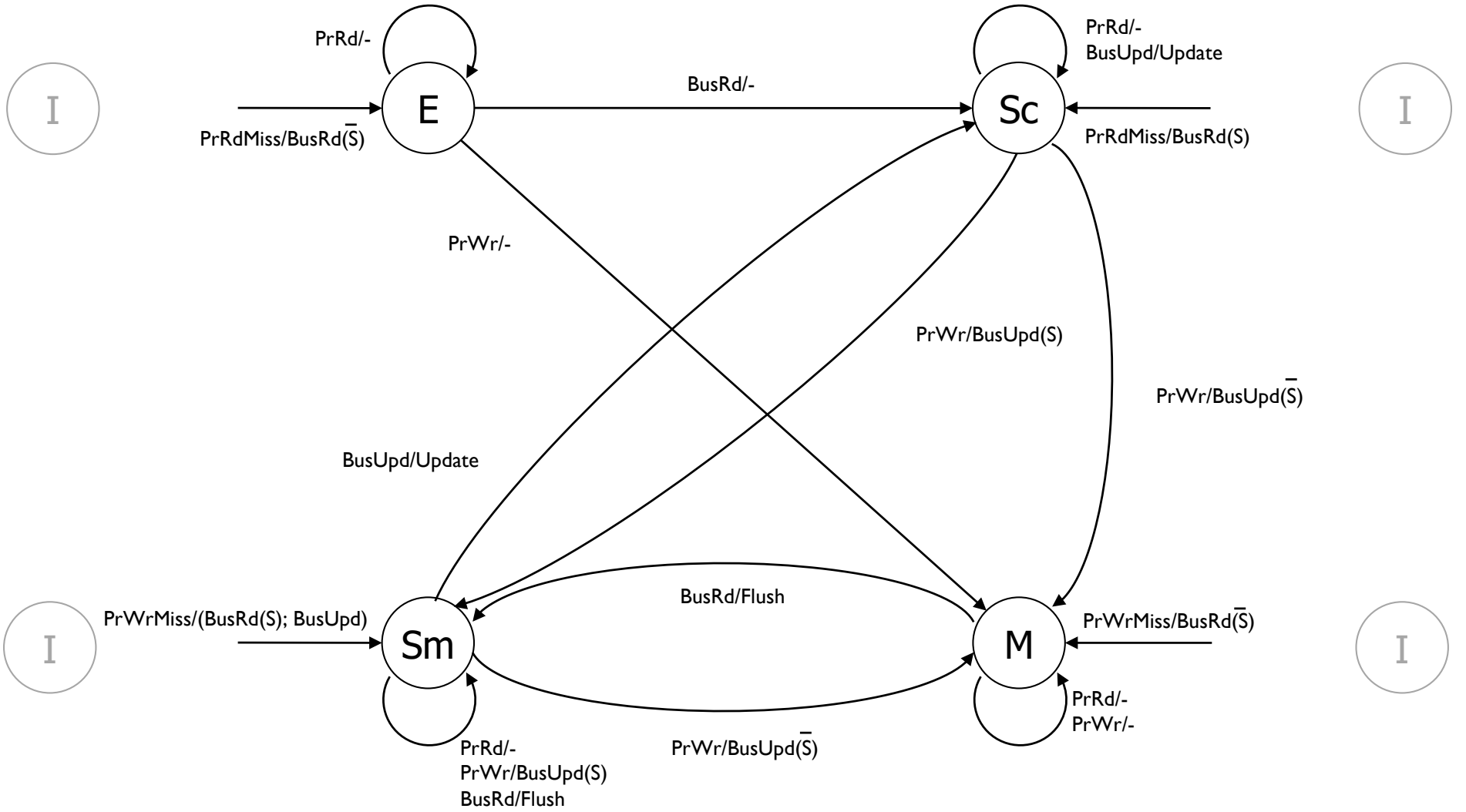
- Who supplies data on miss when not in M state
  - Memory or cache?
- Original (*Illinois* MESI):
  - cache, since assumed faster than memory : cache-to-cache sharing
  - Not true in modern systems – intervening in another cache more expensive than getting from memory
  - Cache-to-cache sharing also adds complexity
    - How does memory know it should supply data – must wait for caches
    - Selection algorithm if multiple caches have valid data
  - But 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*)

Dragon: Write-Back Update Protocol

- 5 states (or 4 states without Invalid)
  - Invalid (I): no data in cache block – if in cache, cannot be invalid
  - 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, but I'm the owner – Sm and Sc can coexist in different caches, with only one Sm
  - Modified or Dirty (D): I and nobody else
- New features
  - New processor events
    - PrRdMiss, PrWrMiss – introduced to specify actions, when block *not present* in cache
  - New bus transactions
    - BusUpd – broadcasts single word written on bus: updates other relevant caches

Dragon State Transition Diagram

Write-Back, Update Protocol



Low-Level Protocol Choices

- Can shared-modified state be eliminated?
  - Yes, if update memory as well on BusUpd transactions (DEC Firefly)
  - Dragon protocol doesn't (assume 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 in critical path
- Shouldn't update local copy on write hit before controller gets bus
  - Otherwise, it can mess up serialization
  - Coherence, consistency considerations much like write-through case
  - In general, many subtle race conditions in protocols

Assessing Protocol Tradeoffs

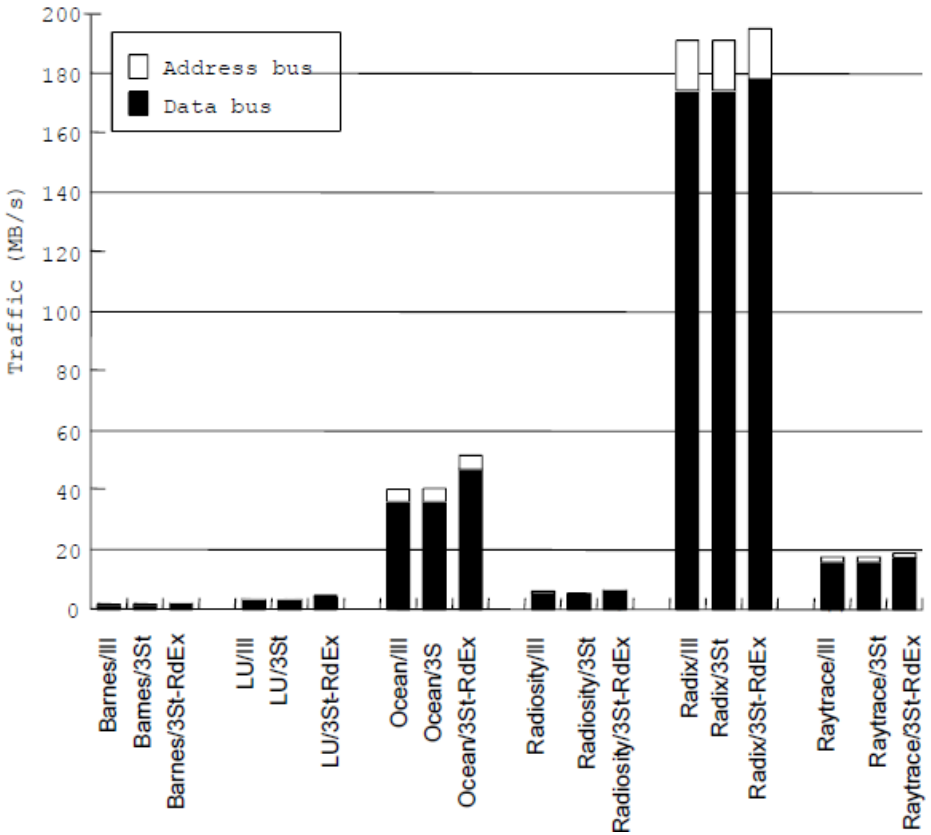
- Tradeoffs affected by performance and organization characteristics

- Part art and part science
  - Art: experience, intuition and aesthetics of designers
  - Science: workload-driven evaluation for cost-performance

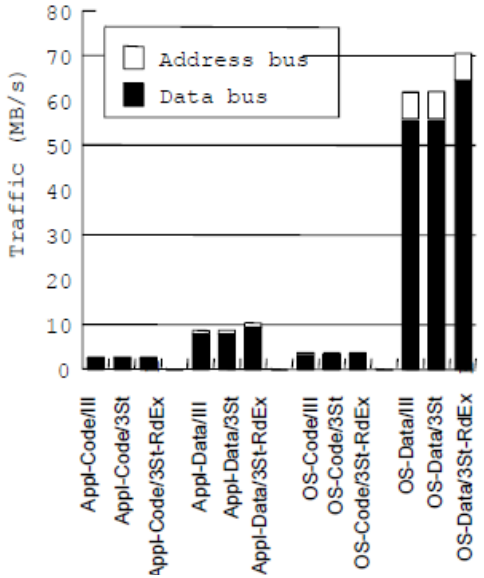
- Methodology
  - Use simulator – choose parameters per earlier methodology
    - e.g. 64KB~1MB, 4-way, 64-byte block, 16 processors
  - Use idealized memory performance model to avoid changes of reference across processors
    - Cheap simulation – no need to model contention

Impact of Protocol Optimizations

- Effect of E state, and of BusUpgr instead of BusRdX
  - MSI vs. MESI doesn't seem to matter for BW for these workloads
  - Upgrades instead of read-exclusive helps



III : MESI
3St: MSI w/ BusUpgr (S→M)
RdEx: MSI w/ BusRdX (S →M)

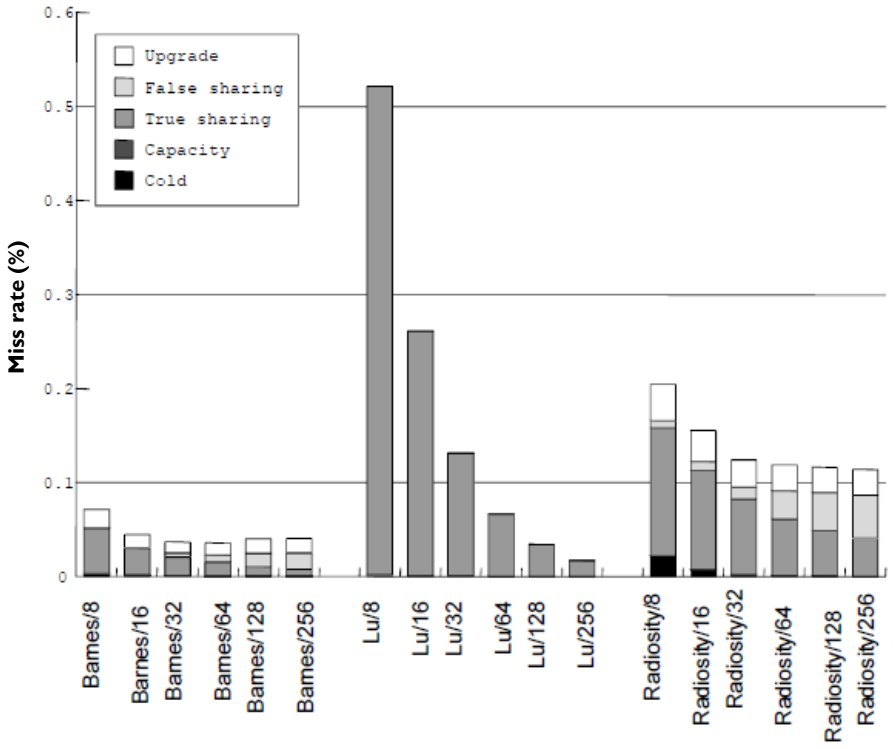


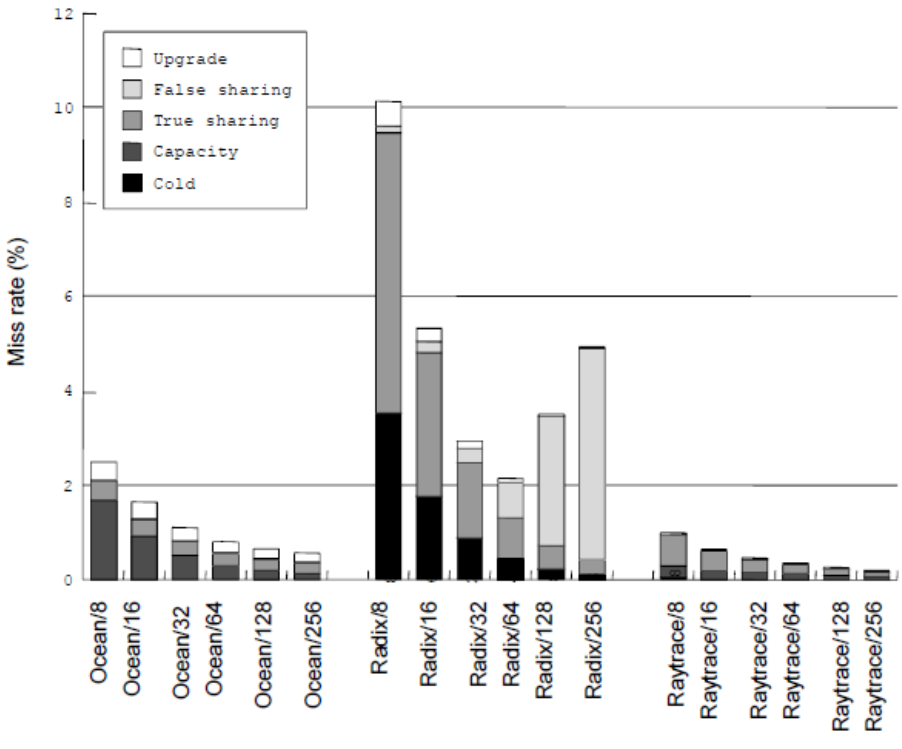
Impact of Cache Block Size

- Multiprocessors add new kind of misses
  - Coherence misses – in addition to cold, capacity, conflict
  - Both miss rate and traffic matter
- Reducing misses in invalidation protocol
  - Capacity – enlarge cache; increase block size (if spatial locality)
  - Conflict – increase associativity
  - Cold and coherence – only block size
- Increasing block size has pros and cons
  - Can reduce misses if spatial locality is good
  - Can hurt too
    - Increase misses due to false sharing, if spatial locality is not good
    - Increase misses due to conflicts in fixed-size cache
    - Increase traffic due to fetching unnecessary data and false sharing
    - Can increase miss penalty and perhaps hit cost

Impact of Block Size on Miss Rate

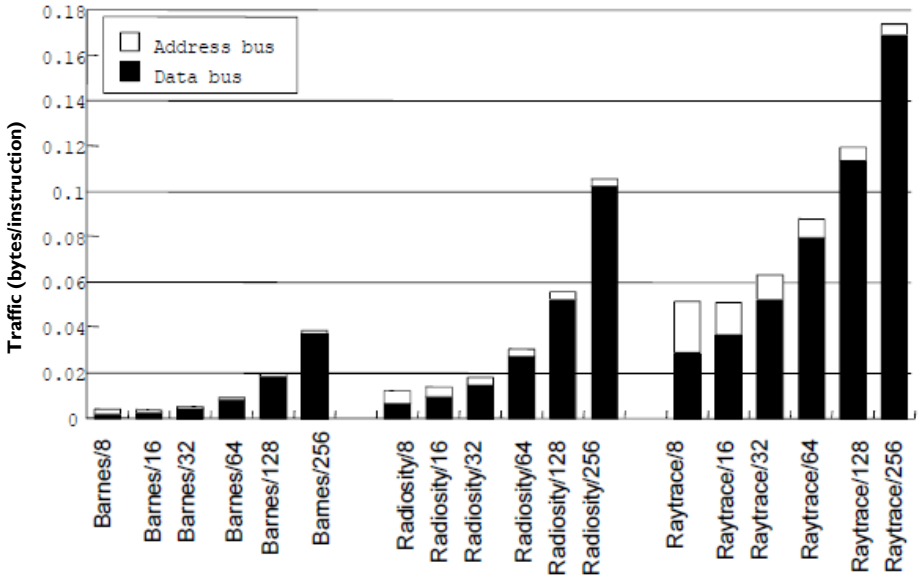
- Results shown here only for default problem size
  - Varied behavior for different problem size and *p* as well
- False sharing increase for some as block size larger
  - Radiosity, Radix for 64 bytes and up

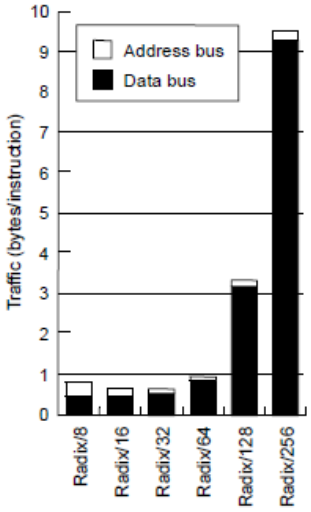


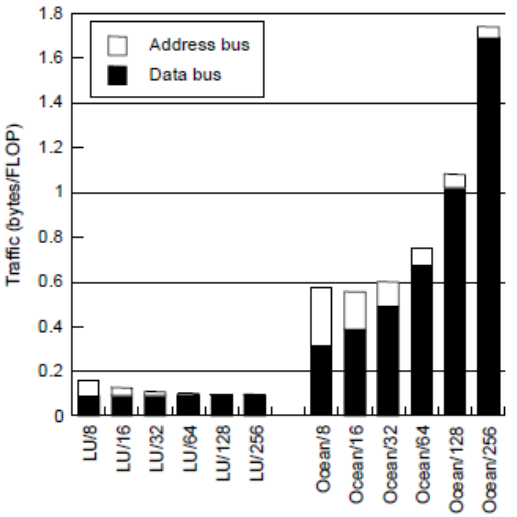


Impact of Block Size on Traffic

- Traffic affects performance indirectly – contention
  - Results different from miss rate – traffic almost always increase
  - When working set fits, overall traffic still small, except Radix (due to false sharing)
  - Fixed overhead is significant component – 16~32B block is optimal rather than 8B
  - Working set doesn't fit (on 64KB cache) – even 128B good for Ocean due to many capacity misses







Making Large Blocks More Effective

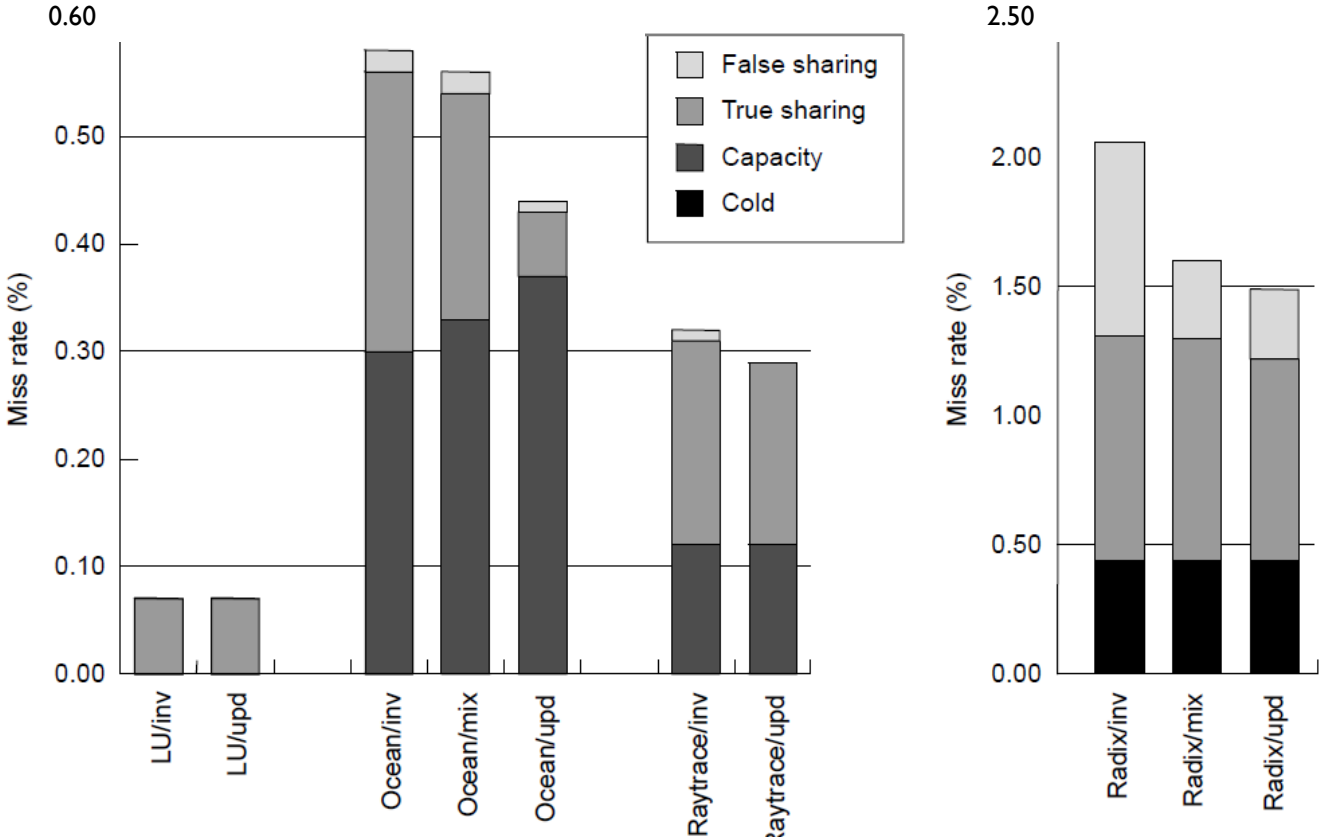
- Software
  - Improve spatial locality by better data structuring
  - Compiler techniques
- Hardware
  - Retain granularity of transfer but reduce granularity of coherence
    - Use subblocks: same tag but different state bits
  - Reduce both granularities, but prefetch more blocks on a miss
  - Proposals for adjustable cache block size
  - More subtle approach – delay propagation of invalidations and perform all at once, but can change consistency model
  - Use update instead of invalidate protocol to reduce false sharing effect

Update vs. Invalidate

- Much debate over the years
  - Tradeoff depends on sharing patterns
- Intuition
  - If processor that was using the data before write wants to see the new value in the future, update should do better – e.g. producer-consumer pattern
  - If processor holding the old data unlikely to use again, updates not good – useless update traffic consumes interconnect and controller resource
  - Can construct scenarios where one or the other is much better
- Can combine them in hybrid schemes
  - Competitive: observe patterns at runtime and change protocol

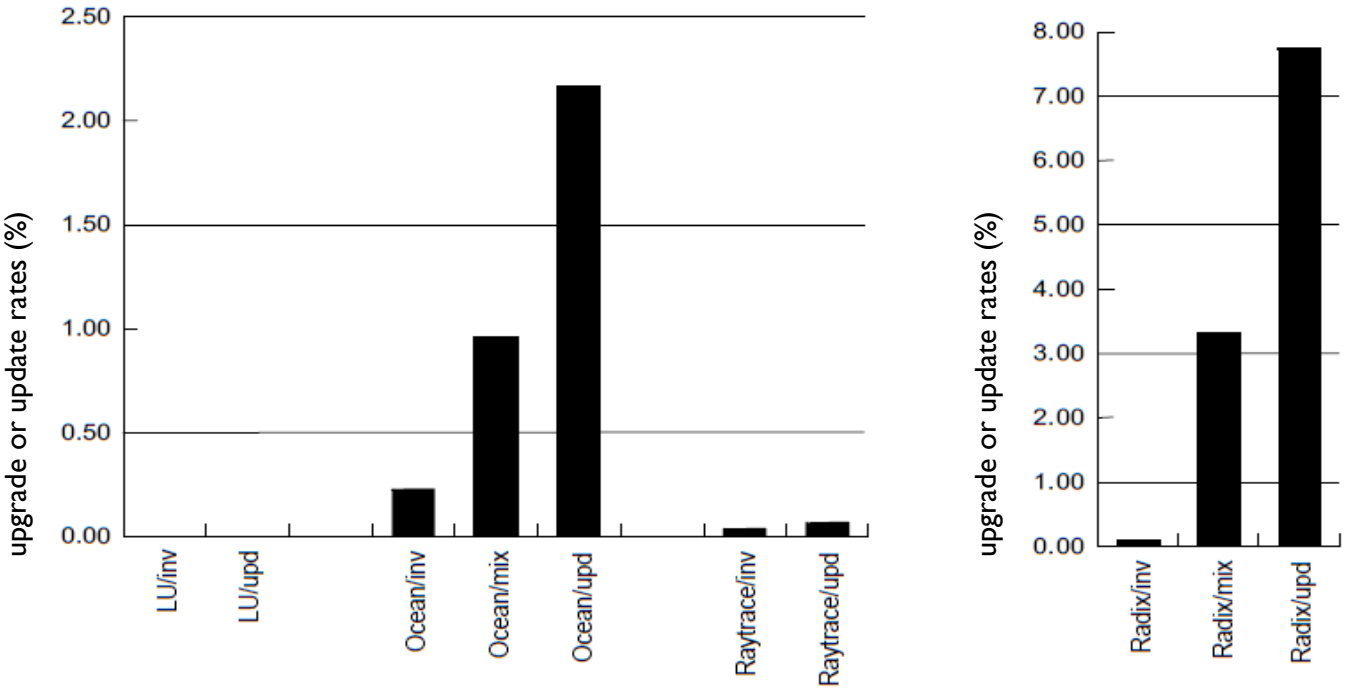
Update vs. Invalidate : Miss Rates

- Depends on patterns
  - Lots of coherence misses: updates help
  - Lots of capacity misses: updates may hurt (keep useless data in cache)
  - Updates seem to help, but this ignores upgrade and update traffic



Upgrade and Update Rates (Traffic)

- Update traffic is substantial
  - Main cause is multiple writes by a processor before a read by other
    - Many bus transactions could delay updates or use merging
  - Overall, trend is away from update based protocol as default
    - Updates potentially have greater problems for scalable systems

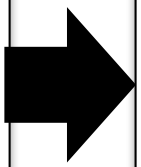


(Rates relative to total memory references)

False sharing (1)

- Unnecessary cache coherence packets (miss + invalidate) due to unintended cache line sharing

```
int i, j, m, n;
double y[m];
…
for ( i=0; i<m; i++)
   for ( j=0; j<n; j++ )
       y[i] += f(i, j);
```



```
int i, j, m, n;
double y[m];
…
for ( i=0; i<m/2; i++)
   for ( j=0; j<n; j++ )
       y[i] += f(i, j);

for ( i=0; i<m/2; i++)
   for ( j=0; j<n; j++ )
       y[i] += f(i, j);
```

Core 0

Core 1

False sharing (2)

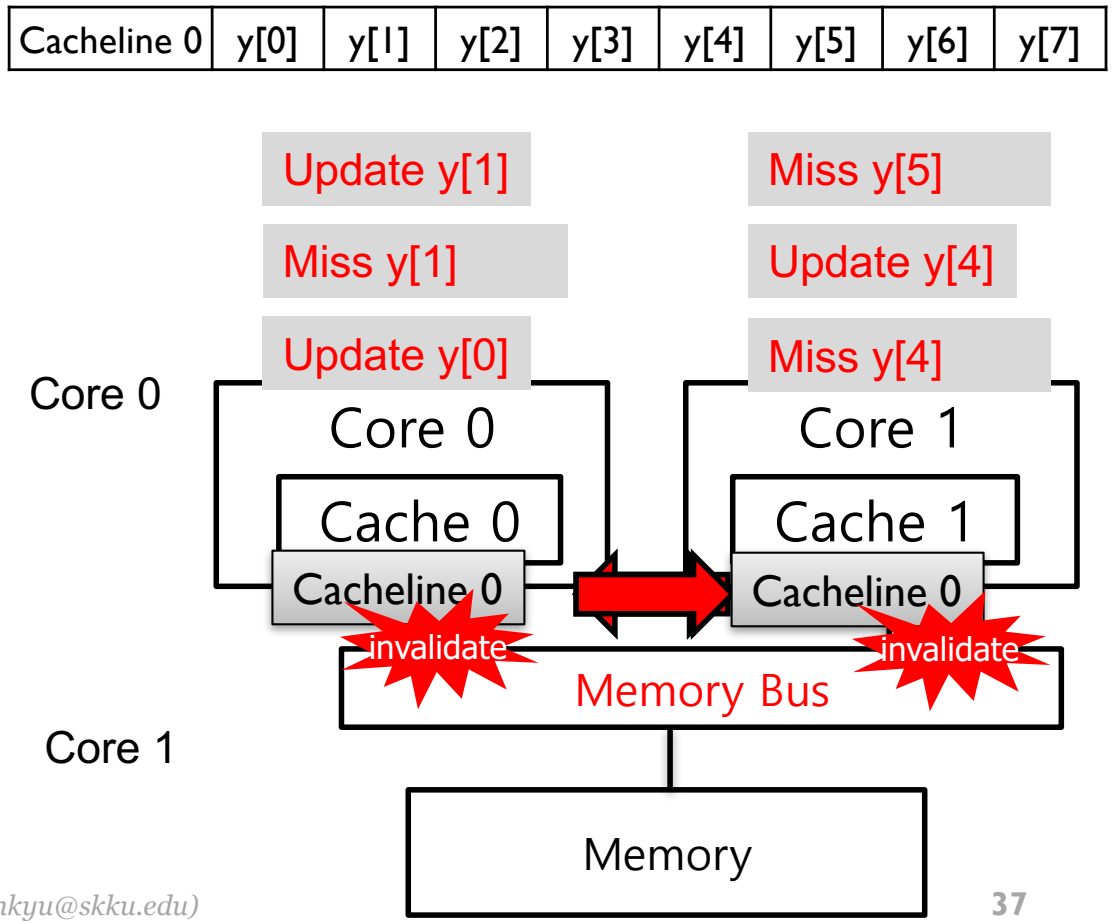
- Unnecessary cache coherence packets (miss + invalidate) due to unintended cache line sharing

```
int i, j, m, n;
double y[m];
…
for ( i=0; i<m/2; i++)
    for ( j=0; j<n; j++ )
        y[i] += f(i, j);

for ( i=m/2; i<m; i++)
    for ( j=0; j<n; j++ )
        y[i] += f(i, j);
```

Core 0

Core 1



References

- Chapter 5.1~5.5 in Parallel Computer Architecture: A Hardware/Software Approach, D. E. Culler, J. P. Singh, A. Gupta, Morgan Kaufmann, 1999