Shared Memory Cache Architecture

Outline

1. Review and generalize uniprocessor cache
2. Coherent memory systems: basic issues
3. Design space of shared memory protocols
4. Performance implications

Part 1: Review Uniprocessor Cache

1. Basic Picture -- Cache controller at the center of things
2. Standard unit of interaction among levels is the cache block
3. Write through versus write back
4. What cache looks like -- tag, data, status bits: v, d
5. State transition diagrams
6. Write allocate versus write no-allocate
7. Memory coherence w.r.t. cache and I/O

Mostly on the board …

7. DMA Problem – Data Coherence

DMA ⇔ Direct memory access. DMA refers to transfers of blocks of data (often pages) between memory and I/O devices *with no CPU intervention* (other than set-up).

- CPU can continue processing using cache.

Recall –

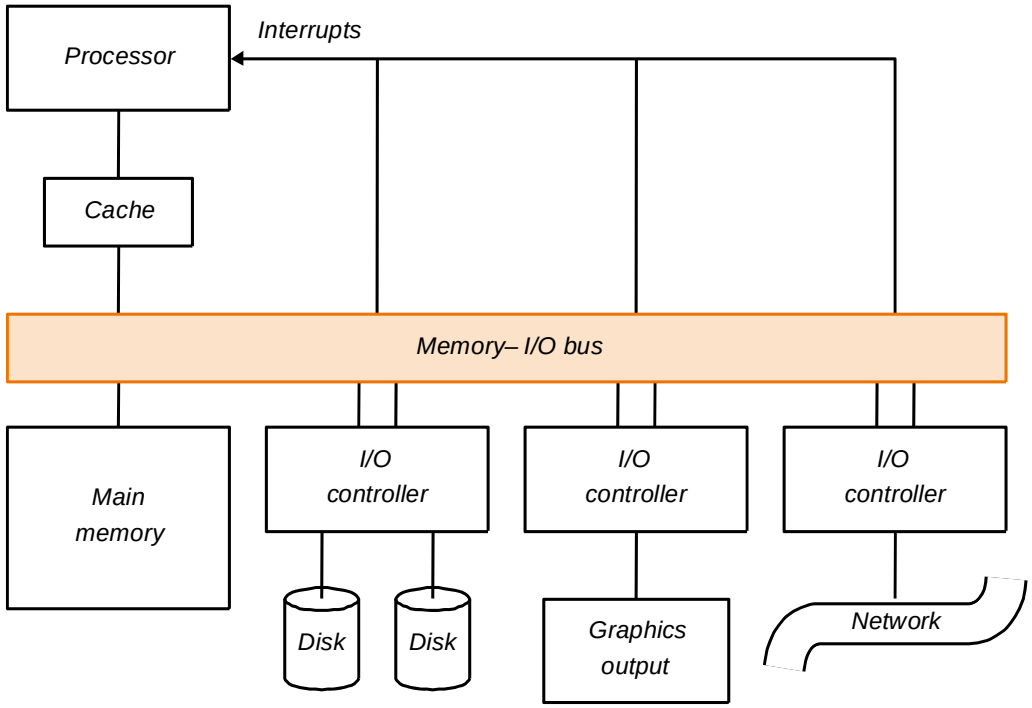
- in cached systems, there are multiple copies of some data (one in memory, one in cache)
- In write-back systems, one copy of the data can be *stale*
- Normally, this is not a problem: stale copy is updated as needed

Case 1 – Memory to I/O DMA transfer

Problem: Before transfer, memory probably stale.

Case 2 – I/O to Memory DMA transfer

Problem: After transfer, memory has latest copy, now cache is stale!



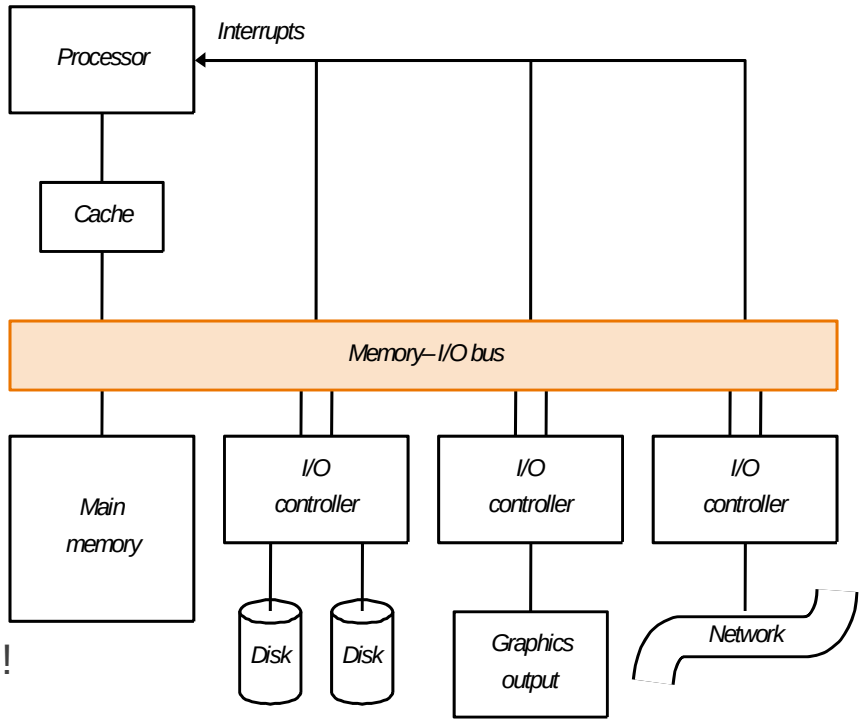
7. Data Coherence Solutions

Case 1: Memory to I/O *(memory stale)*

- Use write-through cache *>> then memory never stale*
- For write-back cache,
  - Before transfer: O.S. forces write-back of dirty cache blocks (corresponding to source addresses)
  - During transfer: Make pages non-cacheable, lock pages (no write)

Case 2: I/O to Memory *(cache stale)*

- For write-back cache,
  - Before transfer, O.S. invalidates cached-copies corresponding to destination addresses
    - Note: the memory data to which the cached data corresponds no longer exists!
- For write-back and write-through caches
  - Make pages non-cacheable, lock pages (no read or write)



Part 2: Coherent Memory Systems

Outline

1. Intuition and motivating example

2. Cache Coherence in Snooping Protocols

3. Write-through cache protocols

A Coherent Memory System: Intuition

Read of a location should return latest value written

Easy in uniprocessors

- Except for I/O -- coherence is an issue between I/O devices and processors
  - But infrequent so software solutions work: uncacheable memory, uncacheable operations, flush pages, pass I/O data through caches

Would like same to hold when processes run on different processors

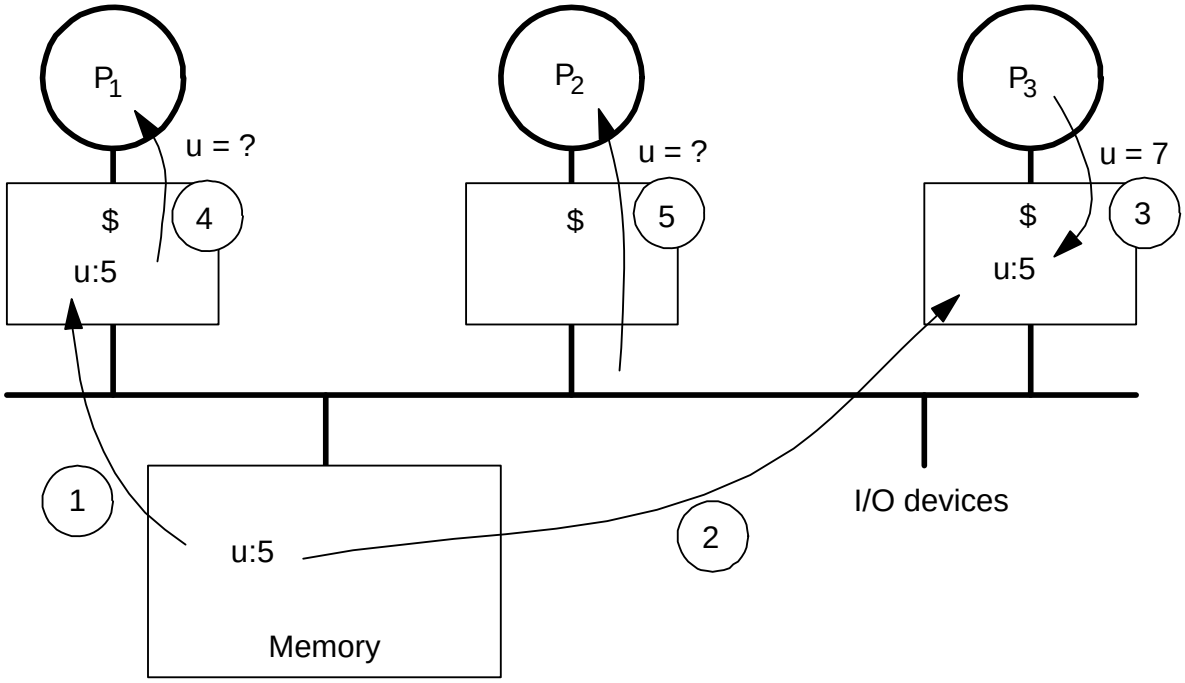
- E.g. → as if the processes were interleaved on a uniprocessor

But coherence problem much more critical in multiprocessors

- Pervasive
- Performance-critical
- Must be treated as a basic hardware design issue

Example: Cache Coherence Problem

P1: LD U
P3: LD U
P3: ST U
P1: LD U
P2: LD U



WT? WB?

- Processors see different values for ***u*** after event 3
- With writeback caches, value written back to memory depends on happenstance of which cache flushes or writes back value when

  → Processes accessing main memory may see very stale value
- Unacceptable to programs, and frequent!

Problems with the Intuition

- Recall: Value returned by read should be last value written
- But “last” is not well-defined
- Even in seq. case, last defined in terms of program order, not time:
  - Order of operations in the machine language presented to processor
  - “Subsequent” defined in analogous way, and well defined
- In parallel case, program order defined within a process, but need to make sense of orders across processes

Some Basic Definitions

Extend from definitions in uniprocessors to those in multiprocessors

Memory operation ⇔ a single read, write, or read-modify-write

- Assume execute atomically w.r.t each other *(for now, reality is more complex)*

Issue ⇔ a memory operation ***issues*** when it leaves processor's internal environment and is presented to memory system (cache, buffer …)

Perform ⇔ operation appears to have taken place, as far as the processor can tell from other memory operations it ***issues***

- A **write** ***performs*** w.r.t. the processor when a subsequent read by the processor returns the value of that write or a later write
- A **read** ***performs*** w.r.t the processor when subsequent writes issued by the processor cannot affect the value returned by the read

In multiprocessors, same -- but replace "the" by "a" processor

Complete ⇔ ***perform*** with respect to all processors

Still need to make sense of order in operations from different processes

Sharpening the Intuition

Imagine a single shared memory and no caches

- Every read and write to a location accesses the same physical location
- Operation completes when it does so

Memory imposes a ***serial*** or ***total order*** on operations to the location

- Operations to the location from a given processor are ***in program order***
- The order of operations to the location from different processors ***is some interleaving that preserves the individual program orders***

"Last" now means most recent in a hypothetical serial order that maintains these properties

For the ***serial order*** to be ***consistent***, all processors must see writes to the location in the same order (if they bother to look, i.e., to read)

Note that the ***total order*** is never really constructed in real systems

- Don't even want memory, or any hardware, to see all operations

But ***program should behave as if some serial order is enforced***

- Order in which things ***appear*** to happen, not actually happen

Formal Definition of Coherence

Results of a program ⇔ values returned by its read operations

Coherent ⇔ A memory system is ***coherent*** if the results of any execution of a program are such that, for each location, it is possible to construct a hypothetical serial order of all operations to the location that is consistent with the results of the execution and in which:

1. operations issued by any particular process occur in the order issued by that process, and

2. the value returned by a read is the value written by the last write to that location in the serial order

Two necessary features:

- ***Write propagation:*** value written must become ***visible*** to others
- ***Write serialization*****:** writes to location are seen in same order by all
  - if I see w1 after w2, you should not see w2 before w1
  - no need for analogous read serialization since reads not visible to others

Snooping-based Coherence

Basic Idea

Transactions on bus are visible to all processors

Processors or their representatives (CCs) can snoop (monitor) bus and take action on relevant events (e.g., change state)

Implementing a Protocol

Cache controller now receives inputs from both sides:

- Requests from processor, bus requests/responses from bus snooper

In either case, takes zero or more actions:

- Updates state, responds with data, generates new bus transactions

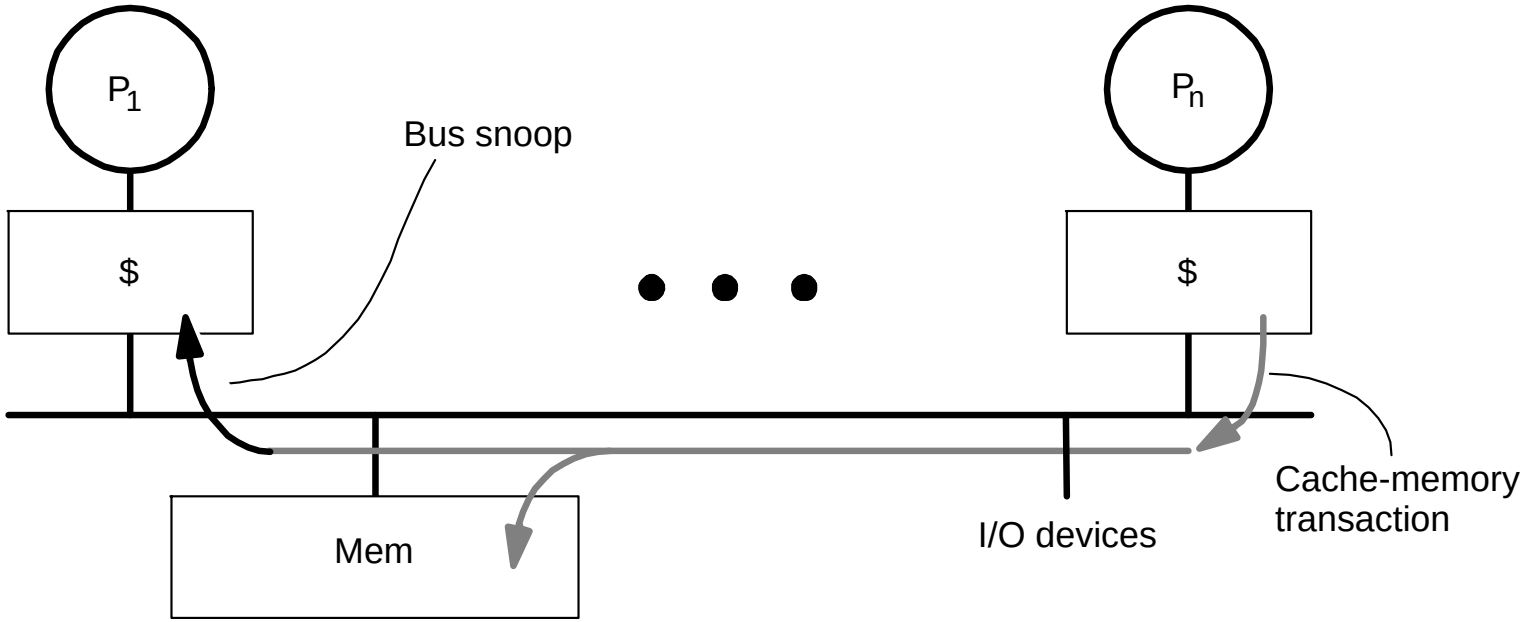
Protocol is distributed algorithm: cooperating state machines

- Set of states, state transition diagram, actions

Granularity of coherence is typically cache block

- Like that of allocation in cache and transfer to/from cache

Coherence with Write-through Caches



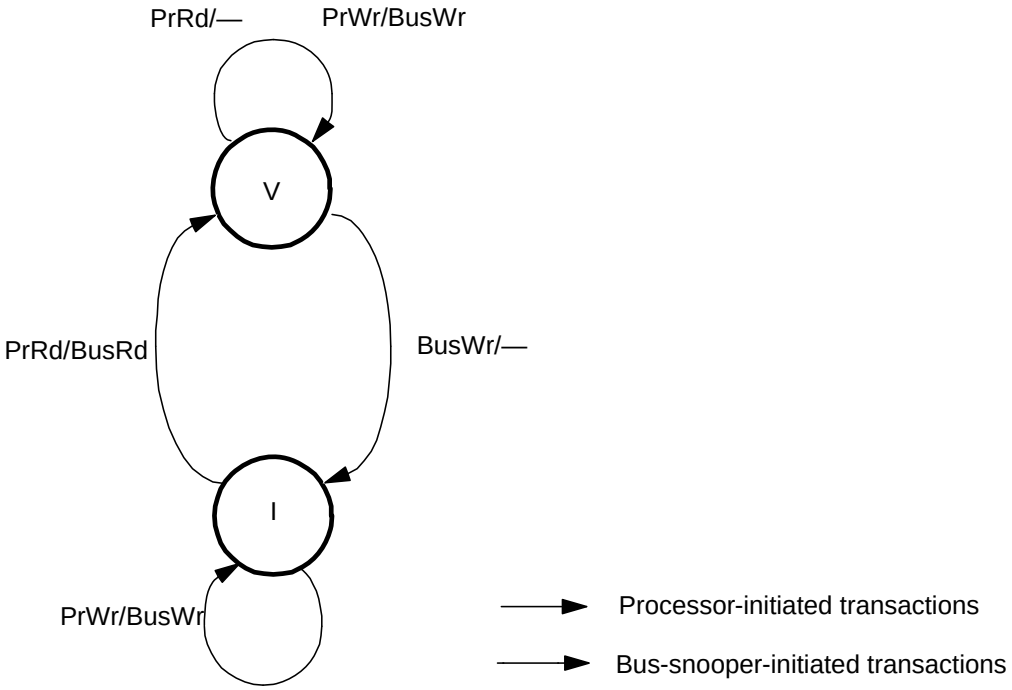
- Key extensions to uniprocessor: snooping, invalidating/updating other caches
  - no new states or bus transactions in this case
  - invalidation- versus update-based protocols

- Write propagation: even in invalidate case, later reads will see new value
  - **inval** causes miss on later access, and memory up-to-date via write-through

Write-Through Basics

- STATES: Two states per block in each cache, as in uniprocessor
  - V = valid (V = TRUE and match = TRUE)
  - I = invalid (V = FALSE or match = FALSE)
- CPU ACTIONS: PrRd/PrWr
- BUS ACTIONS: BusRd, BusWr
- CACHE CONTROLLER ACTIONS:
  - “snoop” bus for BusWr. Invalidate if address matches cache contents

Write-through State Transition Diagram



- Two states per block in each cache, as in uniprocessor
  - state of a block can be seen as *p*-vector
- Hardware state bits associated with only blocks that are in the cache
  - other blocks can be seen as being in invalid (not-present) state in that cache
- Write will invalidate all other caches (no local change of state)
  - can have multiple simultaneous readers of block,but write invalidates them

Example

Processor Action	P1	P2	P3	Bus Action	Data Supplied By	Memory Status
1. P1 reads u	V	-	-	BusRd	Memory	<always fresh>
2. P3 reads u	V	-	V	BusRd	Memory	
3. P3 writes u	I	-	V	BusWr	P3 cache	
4. P1 reads u	V	-	V	BusRd	Memory	
5. P2 reads u	V	V	V	BusRd	Memory	
6. P3 reads u	V	V	V	--	P3 cache	
7. P2 writes u	I	V	I	BusWr	P2 cache	
8. P1 writes u	V	I	I	BusRd/BusWr	Memory/P1 cache	

Problem with Write-Through

High bandwidth requirements

- Every write from every processor goes to shared bus and memory
- Consider 200MHz, 1CPI processor, and 15% instrs. are 8-byte stores
- Each processor generates 30M stores or 240MB data per second
- 1GB/s bus can support only about 4 processors without saturating
- Write-through especially unpopular for SMPs

Write-back caches absorb most writes as cache hits

- Write hits don’t go on bus
- But now how do we ensure write propagation and serialization?
- Need more sophisticated protocols: large design space

Part 3: Design Space of Snooping Protocols

Outline

1. Invalidation versus Update protocols
2. Invalidation 1: MSI and variations
3. Invalidation 2: MESI and variations
4. Update: Dragon

Design Space for Snooping Protocols

- No need to change processor, main memory, cache …
  - Extend cache controller and exploit bus (provides serialization)
- Focus on protocols for ***write-back caches***
- Dirty state now also indicates exclusive ownership
  - Exclusive: only cache with a valid copy (main memory may be too)
  - Owner: responsible for supplying block upon a request for it
    - Ownership an issue w/ clean blocks or w/ update protocols
- Design space
  - Invalidation versus Update-based protocols
  - Set of states

Invalidation-based Protocols

Invalidation-Based Protocol ⇔ A write to clean data generates a notification to other caches

Exclusive means: can modify (again) without notifying anyone else

- i.e. without bus transaction
- Must first get block in exclusive state before writing into it
  - With a notification bus transaction
- Even if already in valid state, need a transaction, e.g., a ***write miss***
  - *A miss on a hit? Huh?*

- Store to non-dirty data generates a *read-exclusive* bus transaction
  - Tells others about impending write, obtains exclusive ownership
    - makes the write ***visible***, i.e. write is ***performed***
    - may be actually observed (by a read miss) only later
    - write hit made ***visible (performed)*** 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 bus transactions drive coherence actions
  - Writeback transactions also, but not caused by memory operation and quite incidental to coherence protocol

Update-based Protocols

Update Protocol ⇔ A write operation updates values in other caches

- New bus transaction: ***Update***

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 same processor cause multiple update transactions
  - In invalidation, first write gets exclusive ownership, others local

Detailed tradeoffs more complex

Invalidate versus Update

- Basic question of program behavior:

  Is a block written by one processor read by others before it is rewritten?

Invalidation:

- Yes ➔ readers will take a miss
- No ➔ multiple writes without additional traffic
  - and clears out copies that won't be used again

Update:

- Yes ➔ readers will *not* miss if they had a copy previously
  - single bus transaction to update all copies
- No ➔ multiple useless updates, even to dead copies

Need to look at program behavior and hardware complexity

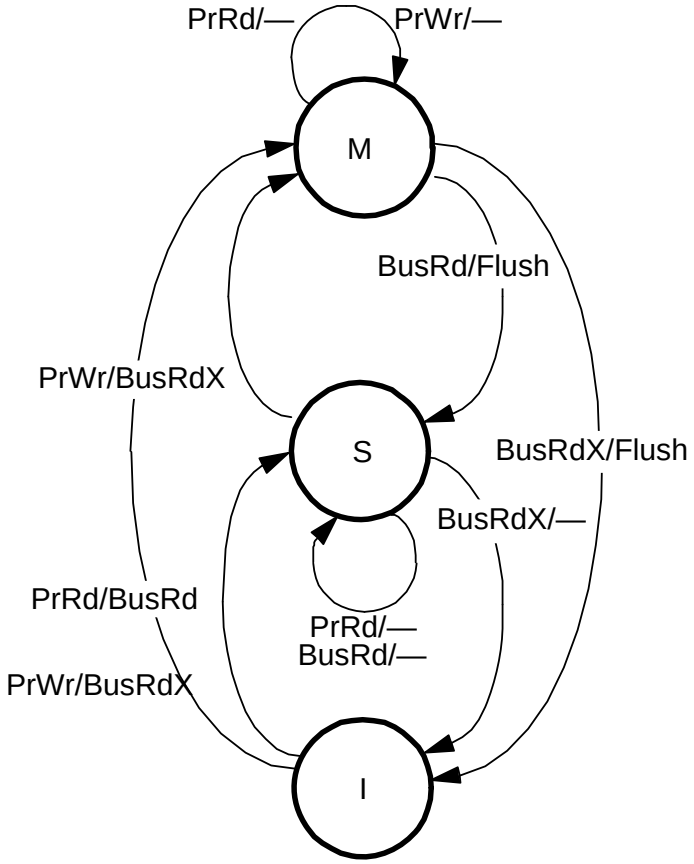
Invalidation protocols much more popular (more later)

- *Some systems provide both, or even hybrid*

Basic MSI Writeback Invalidate Protocol

- States
  - Invalid (I) → V == FALSE **AND** D == d.c.
  - Shared (S): one or more → V == TRUE **AND** D == FALSE
  - Dirty or Modified (M): one only → V == TRUE **AND** D == TRUE
- Processor Events:
  - PrRd (read)
  - PrWr (write)
- Bus Transactions
  - BusRd: asks for copy with no intent to modify
  - BusRdX: asks for copy with intent to modify
  - BusWB: updates memory – only on replacement or explicit flush!
  - *(BusUpgr: tells bus that a shared copy is being written)*
- Actions
  - Update state
  - perform bus transaction
  - flush value onto bus:
    - stop transaction
    - write to bus
    - memory and reading processor cache get data.

State Transition Diagram



Note: PrWr in S causes BusRdX which is a real read!

Note: Flush =
-Stop transaction
-Flush contents onto bus
-Mem and other cache get data

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

Example

Processor Action	P1	P2	P3	Bus Action	Data Supplied By	Memory Status
1. P1 reads u	S	-	-	BusRd	Memory	Fresh
2. P3 reads u	S	-	S	BusRd	Memory	Fresh
3. P3 writes u	I	-	M	BusRdX	Memory	Stale
4. P1 reads u	S	-	S	BusRd	P3 Cache	Fresh
5. P2 reads u	S	S	S	BusRd	Memory	Fresh
6. P3 reads u	S	S	S	--	P3 Cache	Fresh
7. P2 writes u	I	M	I	BusRdX	Memory	Stale
8. P1 writes u	M	I	I	BusRdX	P2 cache	Stale

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, so ordered in same way even w.r.t. writer
  - Reads that appear on the bus ordered wrt these
  - Writes that don't appear on the bus:
    - sequence of such writes between two bus xactions for the block must come from same processor, say P
    - in serialization, the sequence appears between these two bus xactions
    - reads by P will see them in this order w.r.t. other bus transactions
    - reads by other processors separated from sequence by a bus xaction, which places them in the serialized order w.r.t the writes
    - so reads by all processors see writes in same order

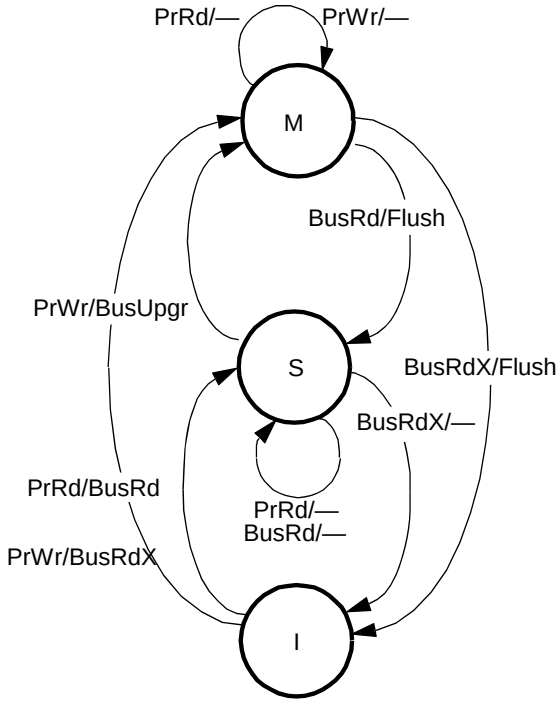
Lower-level Protocol Choice

- BusRd observed in M state: what transition to make?
  - go to S
  - go to I
- Depends on expectations of access patterns
  - S: assumption that I'll read again soon, rather than other will write
    - good for mostly read data
    - what about "migratory" data
      - I read and write, then you read and write, then X reads and writes...
      - better to go to I state, so I don't have to be invalidated on your write
- Choice can affect performance of memory system (later)

Other Lower-level Protocol Choices

- Use BusUpgr instead of BusRdX – invalidates other copies
  - Does not require a memory read
  - Another bus transaction type

- Flush: Update memory (BusWr) or not
  - If not, then Flush requires another bus transaction to write only to another cache.



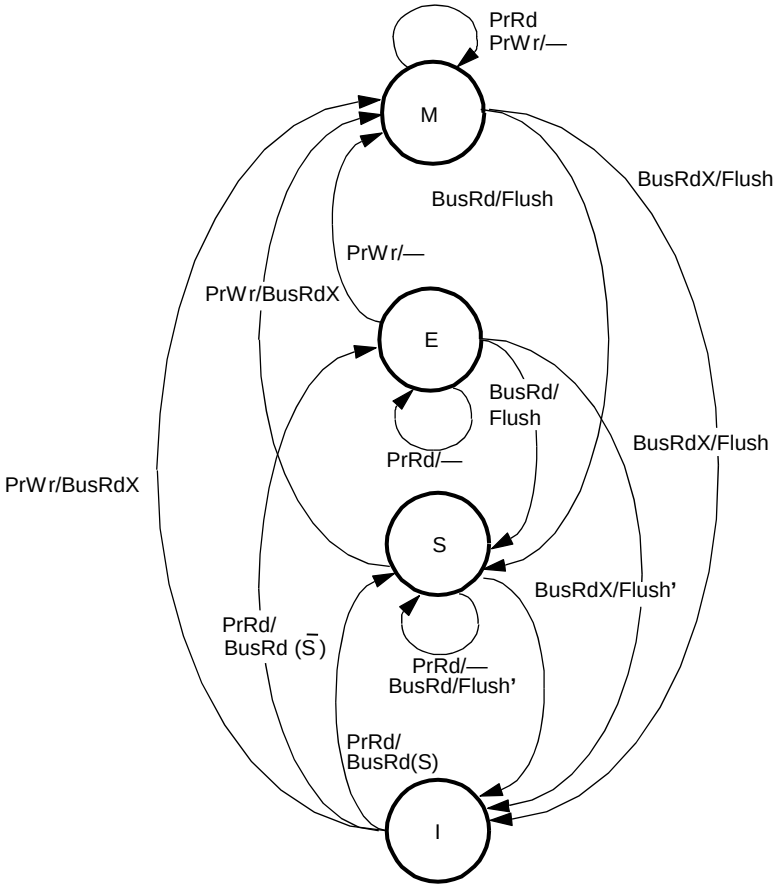
MESI (4-state) Invalidation Protocol

- Problem with MSI protocol
  - Reading and modifying data is 2 bus xactions, ***even if no one sharing***
    - e.g. even in sequential program
    - BusRd (I->S) followed by BusRdX or BusUpgr (S->M)
- Add *exclusive* state: write locally without xaction, but not modified
  - Main memory is up to date, so cache not necessarily owner
  - 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 one else has copy
    - needs “shared” signal on bus: wired-or line asserted in response to BusRd

Basic MESI Writeback Inval Protocol

- States
  - Invalid (I) → V == FALSE **AND** D == d.c.
  - Shared (S): → V == TRUE **AND** D == FALSE **AND** S == TRUE
  - Dirty or Modified (M): → V == TRUE **AND** D == TRUE **AND** S == FALSE
  - Exclusive (E): → V == TRUE **AND** D == FALSE **AND** S == FALSE
- Processor Events:
  - PrRd (read)
  - PrWr (write)
- Bus Transactions
  - BusRd: asks for copy with no intent to modify
  - BusRdX: asks for copy with intent to modify
  - BusWB: updates memory – only on replacement or explicit flush!
- Actions
  - Update state
  - Perform bus transaction
  - Recognize a BusRd to own valid data and assert SHARED line
  - Flush as previous
  - Flush' – if cache-to-cache sharing is in effect, supplies data but does not update memory

MESI State Transition Diagram



- BusRd(S) means shared line asserted on BusRd transaction
- Flush': if cache-to-cache sharing (see next), only one cache flushes data
- MOESI protocol: Owned state: exclusive but memory not valid

Example

Processor Action	P1	P2	P3	Bus Action	Data Supplied By	Memory Status
1. P1 reads u	E	-	-	BusRd(~S)	Memory	Fresh
2. P3 reads u	S	-	S	BusRd(S)	Memory or P1 Cache	Fresh
3. P3 writes u	I	-	M	BusRdX	Memory	Stale
4. P1 reads u	S	-	S	BusRd(S)/Flush	P3 Cache	Fresh
5. P2 reads u	S	S	S	BusRd(S)	Memory	Fresh
6. P3 reads u	S	S	S	--	P3 Cache	Fresh
7. P2 writes u	I	M	I	BusRdX	Memory	Stale
8. P1 writes u	M	I	I	BusRdX	P2 cache	Stale
9. P1 write-back	I	I	I	BusWr	P1 cache	Fresh
10. P1 reads u	E	-	-	BusRd(~S)	Memory	Fresh
11. P1 writes u	M	-	-	--	P1 cache	Stale

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*
- In modern systems: ***Memory*** can be faster
  - Intervening in another cache more expensive than getting from memory

More on Cache-to-Cache Sharing:

- *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
- *Cache-to-cache sharing* may be 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

- New bus transaction: BusUpd
  - Broadcasts single word written on bus; updates other relevant caches
  - Note: not same as MSI *BusUpgr* where only invalidated
- 4 states
  - Exclusive-clean or exclusive (E): I and memory have it
    - SHARED == FALSE **AND** DIRTY == FALSE
  - Shared clean (Sc): I, others, and maybe memory, but I'm not owner
    - SHARED == TRUE **AND** DIRTY == FALSE
  - Shared modified (Sm): I and others but not memory, and I'm the owner
    - SHARED == TRUE **AND** DIRTY == TRUE
    - Sm and Sc can coexist in different caches, with only one Sm
  - Modified or dirty (D): I and, no one else
    - SHARED == FALSE **AND** DIRTY == TRUE
- No invalid state
  - If in cache, cannot be invalid
  - If not present in cache, can view as being in not-present or invalid state

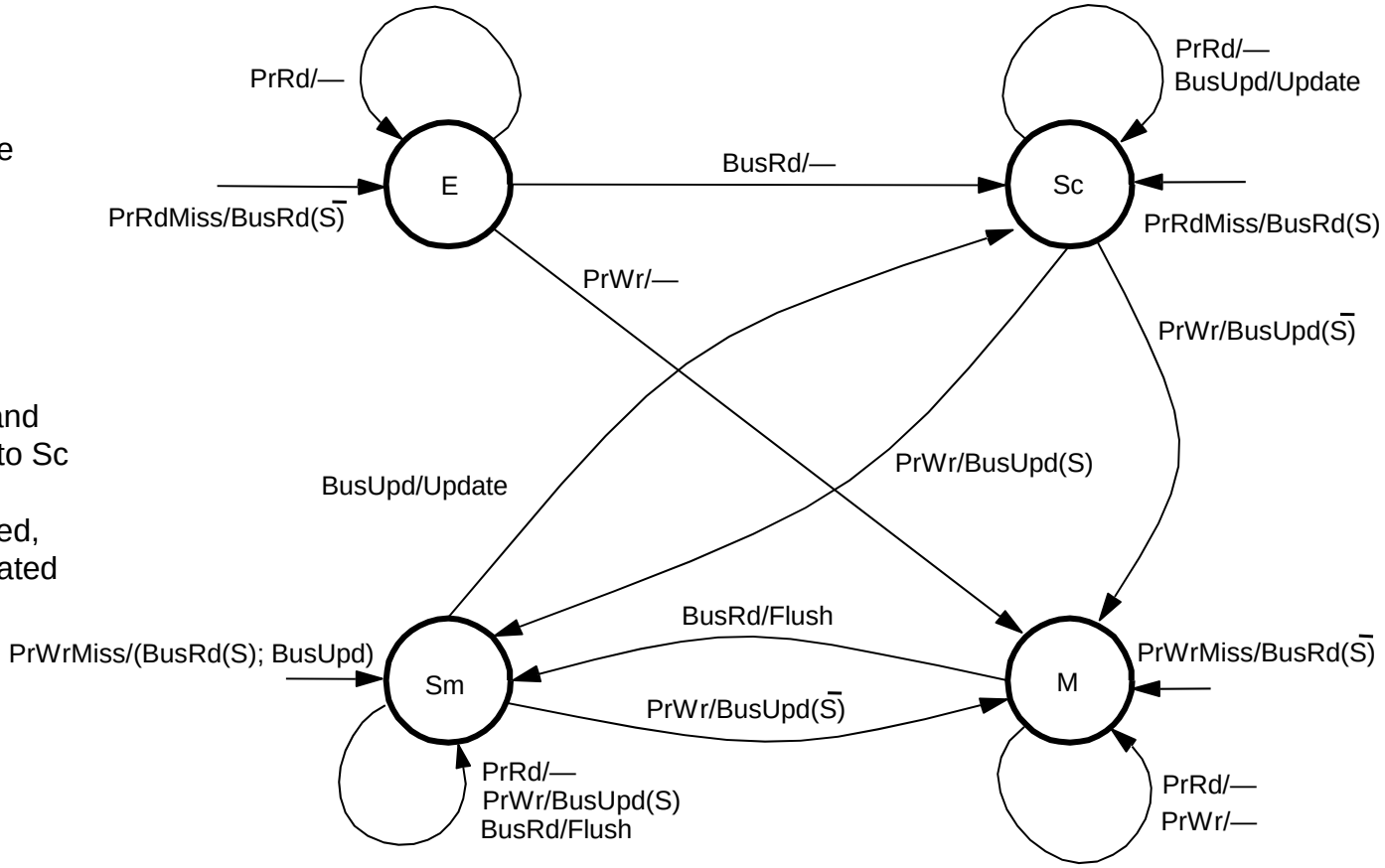
Dragon Write-back Update Protocol, cont.

- New processor events: PrRdMiss, PrWrMiss
  - Introduced to specify actions when block not present in cache
  - [IS THIS REALLY DIFFERENT? HOW DOES PROCESSOR KNOW??]

- Sm?? How can one cache have modified and the others not, simultaneously?
  - Sm means memory is stale and I'm responsible for supplying data to other caches
  - Contents are the same among caches!

Dragon State Transition Diagram

What happens on:
-- Read Miss?
  * BusRd is generated
  * check S signal for new state
  * If M or Sm elsewhere, then that cache supplies data, else memory supplies it.
-- Write?
  * If M, then no action
  * If E, then M and no action
  * If Sc or Sm, then BusUpd and other caches change state to Sc
  * Memory not updated
  * If NC, then BusRd. If Shared, then BusUpd is also generated
-- Replacement?
  * Written back to memory only if in M or Sm states



Example

Processor Action	P1	P2	P3	Bus Action	Data Supplied By	Memory Status
1. P1 reads u	E	-	-	BusRd(~S)	Memory	Fresh
2. P3 reads u	Sc	-	Sc	BusRd(S)	Memory	Fresh
3. P3 writes u	Sc	-	Sm	BusUpd	P3 Cache	Stale
4. P1 reads u	Sc	-	Sm	--	P1 Cache	Stale
5. P2 reads u	Sc	Sc	Sm	BusRd(S)	P3 Cache	Stale
6. P3 reads u	Sc	Sc	Sm	--	P3 Cache	Stale
7. P2 writes u	Sc	Sm	Sc	BusUpd	P2 Cache	Stale
8. P1 writes u	Sm	Sc	Sc	BusUpd	P1 cache	Stale
9. P1 write-back	NC	Sc	Sc	BusWr	P1 cache	Fresh
10. P1 reads u	Sc	Sc	Sc	BusRd(S)	Memory	Fresh
11. P1 writes u	Sm	Sc	Sc	BusUpd	P1 cache	Stale

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
- Shouldn't update local copy on write hit before controller gets bus
  - Can mess up serialization
- Coherence, consistency considerations much like write-through case

- In general, many subtle race conditions in protocols
- But first, let's illustrate quantitative assessment at logical level

Part 4: Assessing Protocol Tradeoffs

Outline

1. Basic results – invalidate protocols

2. One of the big three – block size

   for cache size and associativity see text

3. Update versus invalidate

Methodology:

- Use simulator; choose parameters per earlier methodology (default 1MB, 4-way cache, 64-byte block, 16 processors; 64K cache for some)
- Focus on frequencies, not end performance for now
  - transcends architectural details, but not what we're really after
- Use idealized memory performance model to avoid changes of reference interleaving across processors with machine parameters
  - Cheap simulation: no need to model contention

State Transitions per 1000 Data Memory References Issued by Application

Application	From	To: NP	I	E	S	M
Barnes-Hut	NP	0	0	0.0011	0.0362	0.0035
	I	0.0201	0	0.0001	0.1856	0.0010
	E	0.0000	0.0000	0.0153	0.0002	0.0010
	S	0.0029	0.2130	0	97.1712	0.1253
	M	0.0013	0.0010	0	0.1277	902.782
LU	NP	0	0	0.0000	0.6593	0.0011
	I	0.0000	0	0	0.0002	0.0003
	E	0.0000	0	0.4454	0.0004	0.2164
	S	0.0339	0.0001	0	302.702	0.0000
	M	0.0001	0.0007	0	0.2164	697.129
Ocean	NP	0	0	1.2484	0.9565	1.6787
	I	0.6362	0	0	1.8676	0.0015
	E	0.2040	0	14.0040	0.0240	0.9955
	S	0.4175	2.4994	0	134.716	2.2392
	M	2.6259	0.0015	0	2.2996	843.565
Radiosity	NP	0	0	0.0068	0.2581	0.0354
	I	0.0262	0	0	0.5766	0.0324
	E	0	0.0003	0.0241	0.0001	0.0060
	S	0.0092	0.7264	0	162.569	0.2768
	M	0.0219	0.0305	0	0.3125	839.507
Radix	NP	0	0	0.0030	1.3153	5.4051
	I	0.0485	0	0	0.4119	1.7050
	E	0.0006	0.0008	0.0284	0.0001	0
	S	0.0109	0.4156	0	84.6671	0.3051
	M	0.0173	4.2886	0	1.4982	906.945

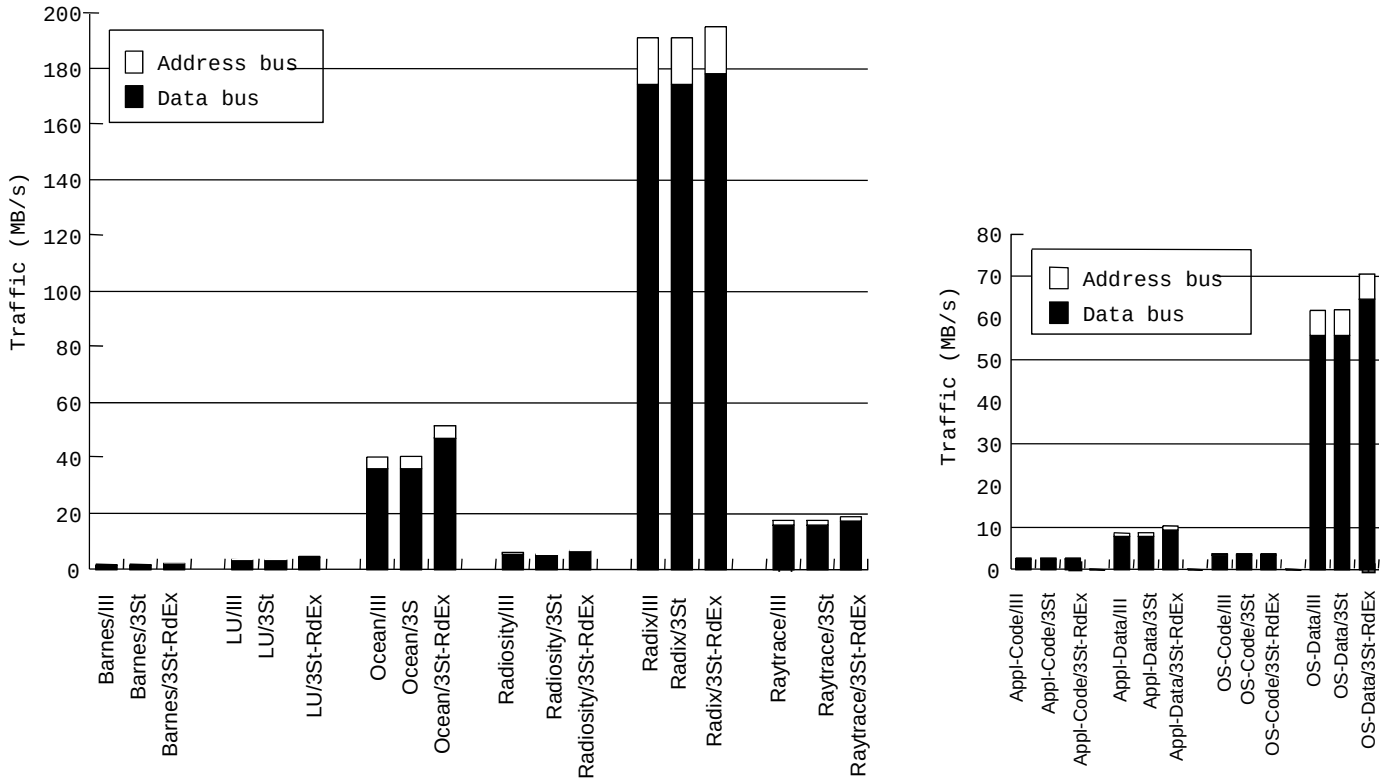
Application	From	To: NP	I	E	S	M
Raytrace	NP	0	0	1.3358	0.15486	0.0026
	I	0.0242	0	0.0000	0.3403	0.0000
	E	0.8663	0	29.0187	0.3639	0.0175
	S	1.1181	0.3740	0	310.949	0.2898
	M	0.0559	0.0001	0	0.2970	661.011
Multiprog User Data References	NP	0	0	0.1675	0.5253	0.1843
	I	0.2619	0	0.0007	0.0072	0.0013
	E	0.0729	0.0008	11.6629	0.0221	0.0680
	S	0.3062	0.2787	0	214.6523	0.2570
	M	0.2134	0.1196	0	0.3732	772.7819
Multiprog User Instruction References	NP	0	0	3.2709	15.7722	0
	I	0	0	0	0	0
	E	1.3029	0	46.7898	1.8961	0
	S	16.9032	0	0	981.2618	0
	M	0	0	0	0	0
Multiprog Kernel Data References	NP	0	0	1.0241	1.7209	4.0793
	I	1.3950	0	0.0079	1.1495	0.1153
	E	0.5511	0.0063	55.7680	0.0999	0.3352
	S	1.2740	2.0514	0	393.5066	1.7800
	M	3.1827	0.3551	0	2.0732	542.4318
Multiprog Kernel Instruction References	NP	0	0	2.1799	26.5124	0
	I	0	0	0	0	0
	E	0.8829	0	5.2156	1.2223	0
	S	24.6963	0	0	1,075.2158	0
	M	0	0	0	0	0

Bus Actions Corresponding to State Transitions in the MESI Protocol

From	To: NP	I	E	S	M
NP	—	—	BusRd	BusRd	BusRdX
I	—	—	BusRd	BusRd	BusRdX
E	—	—	—	—	—
S	—	—	Not possible	—	BusUpgr
M	BusWB	BusWB	Not possible	BusWB	—

Impact of Protocol Optimizations

(Computing traffic from state transitions discussed in book)
Effect of E state, and of BusUpgr instead of BusRdX



- MSI versus MESI doesn't seem to matter for bw for these workloads
- Upgrades instead of read-exclusive helps
- Same story when working sets don't fit for Ocean, Radix, Raytrace

Impact of Cache Block Size

- Reducing misses architecturally in invalidation protocol
  - Capacity: enlarge cache; increase block size (if spatial locality)
  - Conflict: increase associativity
  - Cold and Coherence: only block size

- Multiprocessors add new kind of miss to cold, capacity, conflict
  - Coherence misses: true sharing and false sharing
    - latter due to granularity of coherence being larger than a word
  - Both miss rate and traffic matter

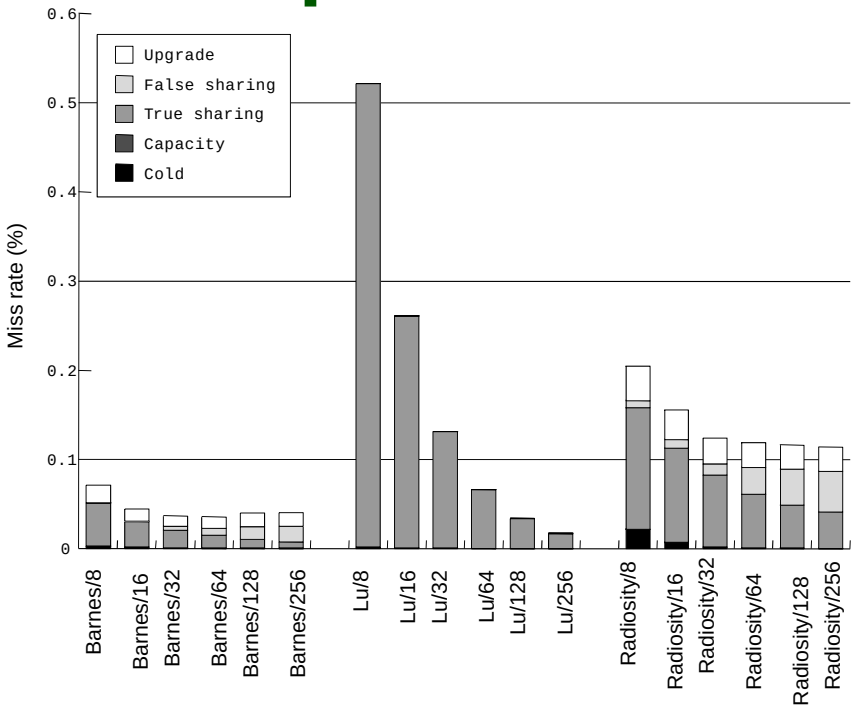
- Increasing block size has advantages and disadvantages
  - Can reduce misses if spatial locality is good
  - Can hurt too
    - increase misses due to false sharing if spatial locality not good
    - increase misses due to conflicts in fixed-size cache
    - increase traffic due to fetching unnecessary data and due to false sharing
    - can increase miss penalty and perhaps hit cost

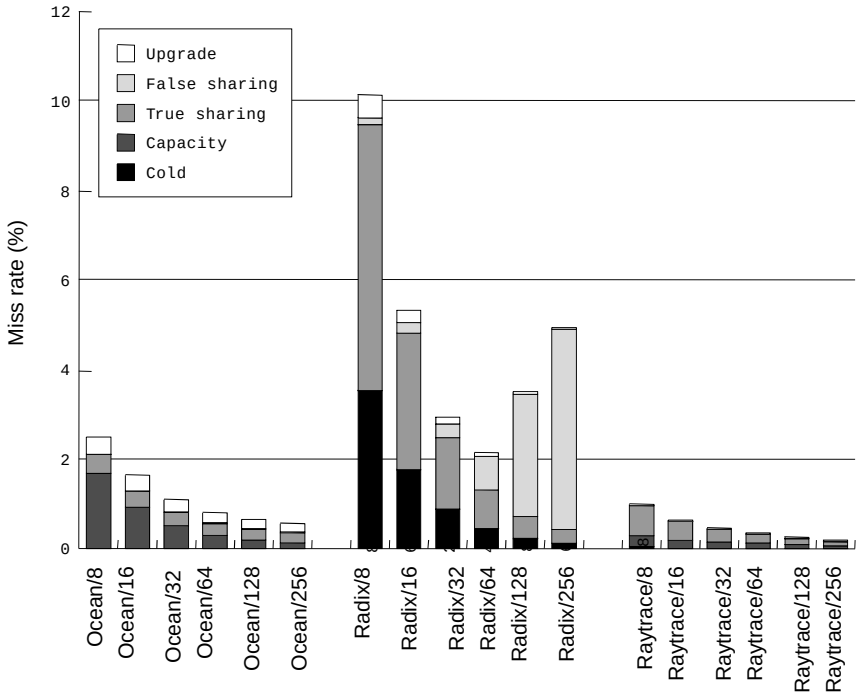
False Sharing

Assume block X with two words, x1 and x2:

Time	P1 Action	P2 Action	State	Comment
T0	Read x1		S|I	Compulsory miss
T1		Read x1	S|S	Compulsory miss
T2	Write x1		M|I	Promotion/Invalidate
T3		Read x2	S|S	False sharing miss since x2 was invalidated by x1 write
T4	Write x1		M|I	False sharing miss since x1 is shared because of x2 read
T5		Write x2	I|M	False sharing miss since x2 was invalidated because of x1 write
T6	Read x2		S|S	True sharing miss since x2 write caused invalidation – P1 is reading data written by p2

Impact of Block Size on Miss Rate





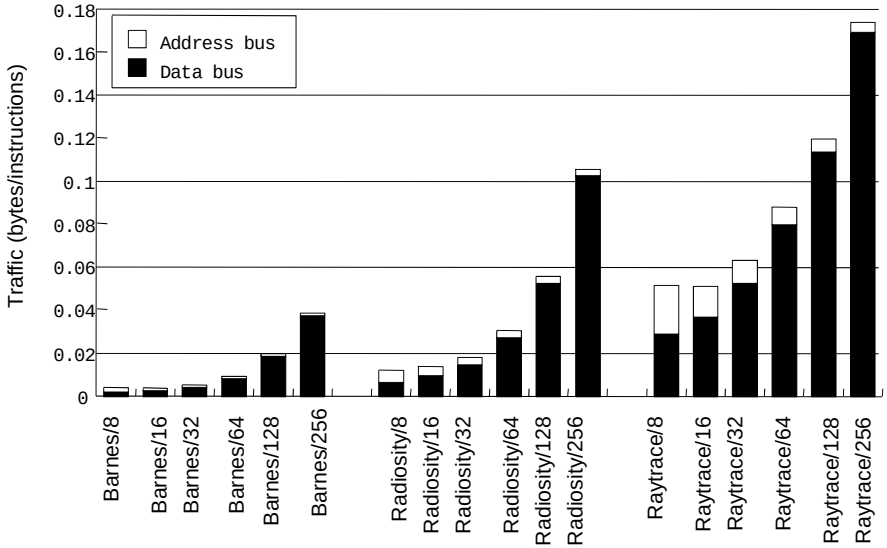
- Cold, capacity, and true sharing misses decrease with increasing block size
- False sharing and conflict misses increase with block size
- Cannot get rid of true sharing misses

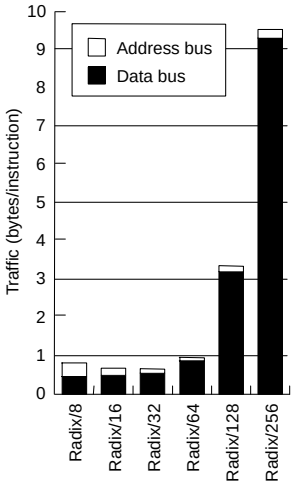
<u>Application specific observations:</u>

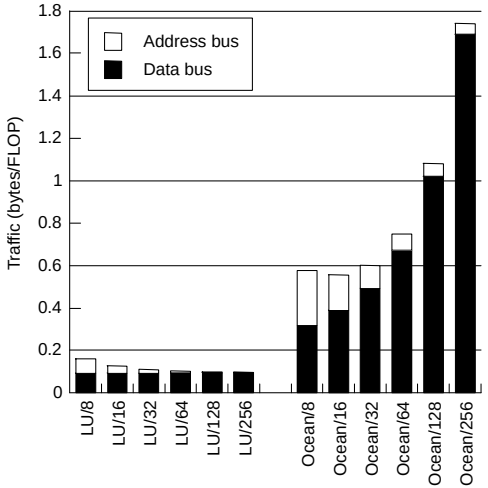
- LU and Ocean have good spatial locality and little false sharing – good data structures
- Note capacity misses in Ocean – they behave like cold misses
- True sharing in Ocean is boundary elements – do not decrease much as block size increases
- Raytrace has worse spatial locality but because mostly read-only, no false sharing
- True sharing in Raytrace is reduced by increasing block size
- Barnes-Hut has moderate spatial locality and false sharing

Impact of Block Size on Traffic

Traffic affects performance indirectly through contention







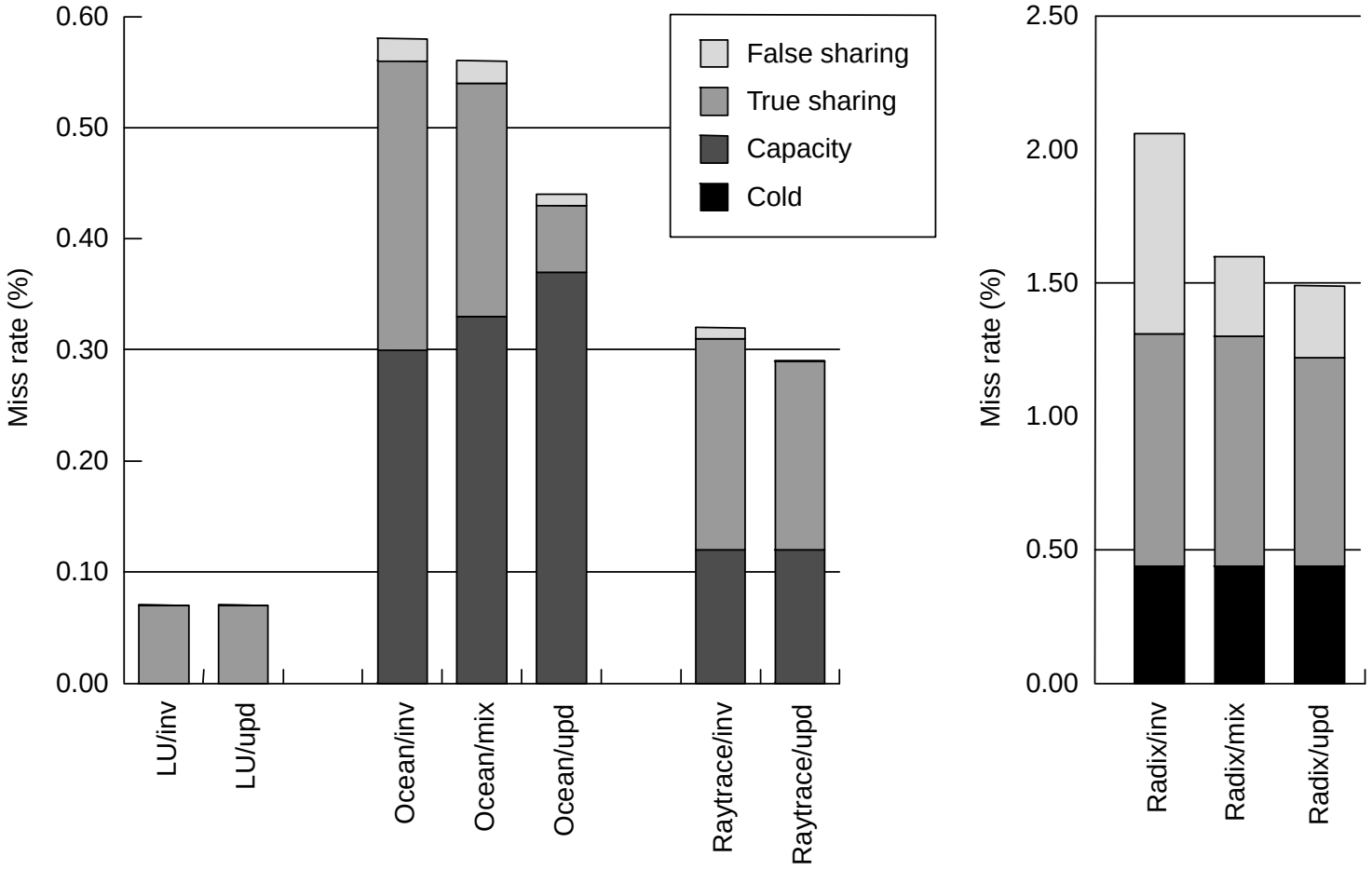
- Results different than for miss rate: traffic almost always increases
- When working sets fit, overall traffic still small, except for Radix
- Fixed overhead is significant component
  - So total traffic often minimized at 16-32 byte block, not smaller
- Working set doesn't fit: even 128-byte good for Ocean due to capacity

Update versus Invalidate

- Much debate over the years: tradeoff depends on sharing patterns
- Intuition:
  - If those that used continue to use, and writes between use are few, update should do better
    - e.g. producer-consumer pattern
  - If those that use unlikely to use again, or many writes between reads, updates not good
    - “pack rat” phenomenon particularly bad under process migration
    - useless updates where only last one will be used
- Can construct scenarios where one or other is much better

Let’s look at real workloads

Update vs Invalidate: Miss Rates



- Lots of coherence misses: updates help
- Lots of capacity misses: updates hurt (keep data in cache uselessly)
- 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 versus one in invalidation case
  - could delay updates or use merging
  -
- Overall, trend is away from update based protocols as default
  - bandwidth, complexity, large blocks trend, pack rat for process migration
- Will see later that updates have greater problems for scalable systems