CS315A/EE382B: Lecture 8

Symmetric Multiprocessors I

Kunle Olukotun
Stanford University

http://eeclass.stanford.edu/cs315a

Today's Outline

- Motivation for shared memory
- Cache coherence protocols
- SMP performance

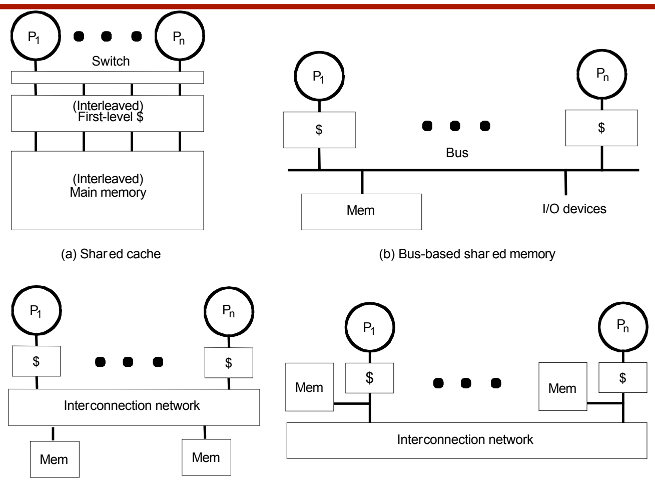
What is (Hardware) Shared Memory?

- Take multiple (micro-)processors
- Implement a memory system with
  a single global physical address space
- Allow caching of shared and private data
  - Minimize memory latency
  - Maximize memory bandwidth

Why Shared Memory?

- Pluses
  - To applications looks like multitasking uniprocessor
  - Programmers can worry about correctness first then performance
  - Easy to do communication without OS
  - For OS only evolutionary extensions required
- Minuses
  - Proper synchronization can be difficult
  - Communication is implicit so harder to optimize
  - Hardware support can be complex
- Result
  - Symmetric Multiprocessors (SMPs) are
    the most success parallel machines ever
  - And the first with multi-billion-dollar markets
    - 90% commercial (TPC, DSS, web)
    - 10% high-performance computing (eng, bio, financial)

Some Shared Memory System Options



(a) Shared cache

(b) Bus-based shared memory

(c) Dancehall

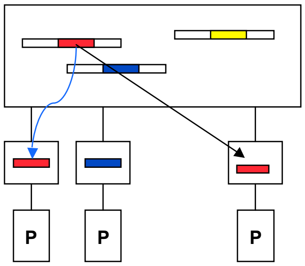
(d) Distributed-memory

Symmetric Multiprocessors (SMP)

- Multiple (micro-)processors
- Each has cache (a cache hierarchy)
- Connect with logical bus
  - broadcast
  - totally-ordered

Caches are Critical for Performance

- Reduce average latency
  - automatic replication and migration closer to processor
- Reduce average bandwidth
- Data is logically transferred from producer to consumer to memory
  - store reg --> mem
  - load reg <-- mem



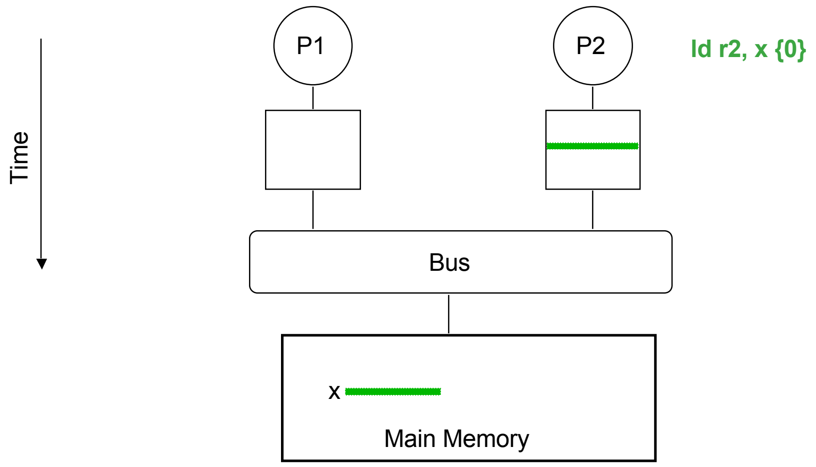
- **Many processor can share data efficiently**

- **What happens when store & load are executed on different processors?**

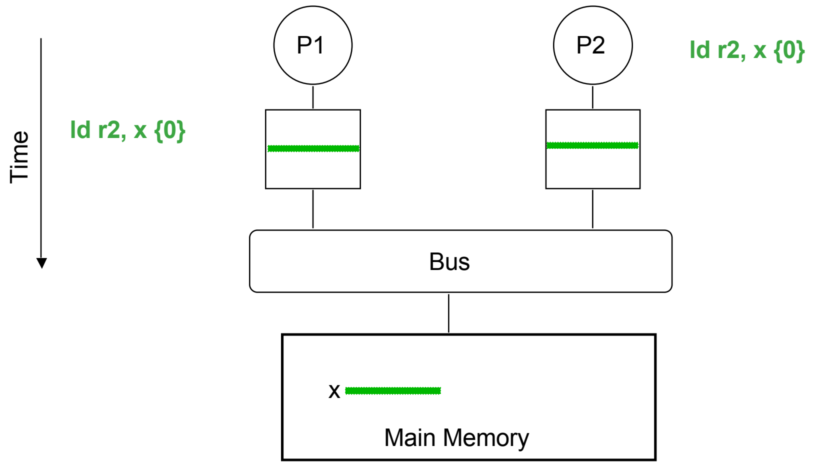
Coherence and Consistency

- Intuition says loads should return the most recent value
  - what is most recent?
- Coherence concerns only a single memory location
- Consistency concerns apparent ordering for multiple locations
- A Memory System is Coherent if:
  - can serialize all operations to that location such that,
  - operations performed by any processor appear in program order
    - program order = order defined by program text or assembly code
  - value returned by a read on one processor is value written by last store to that location by another processor
  - Writes to a location are seen in same order by all processors
    - Write serialization

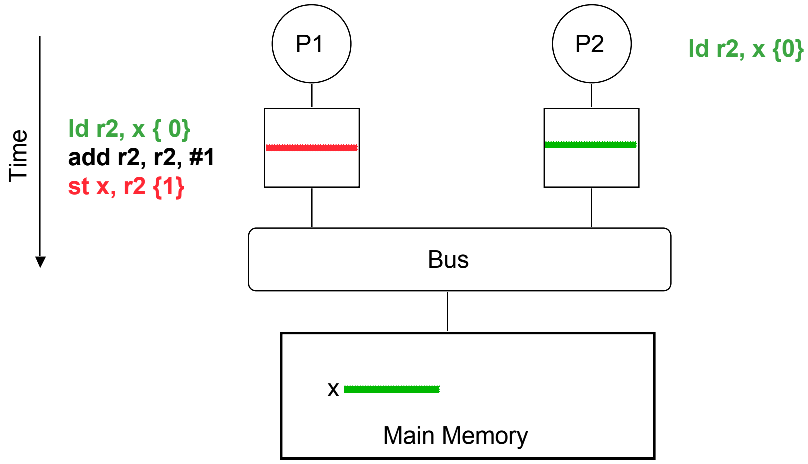
Cache Coherence Problem (Step 1)



Cache Coherence Problem (Step 2)



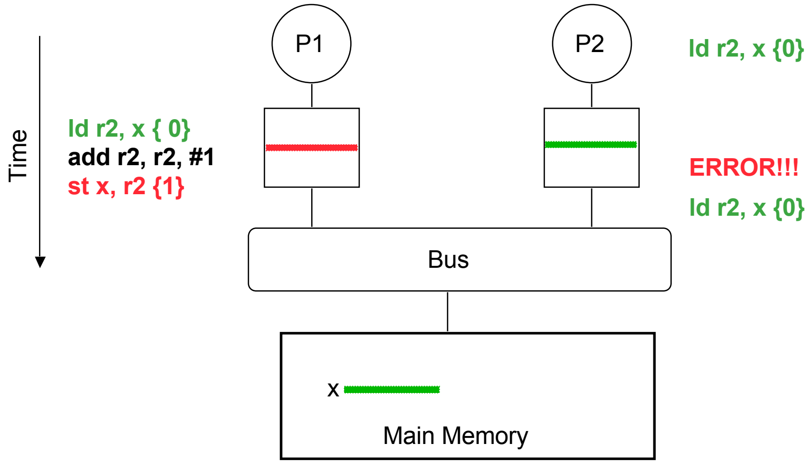
Cache Coherence Problem (Step 3)



ld r2, x { 0}
add r2, r2, #1
st x, r2 {1}

ld r2, x {0}

Cache Coherence Problem (Step 4)



ld r2, x { 0}
add r2, r2, #1
st x, r2 {1}

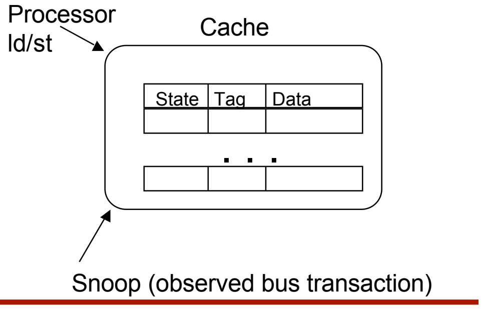
ld r2, x {0}

ERROR!!!

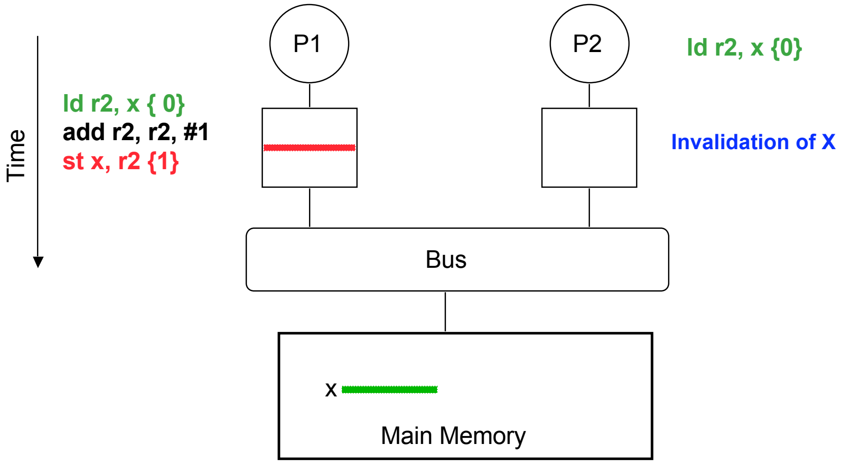
ld r2, x {0}

Snooping Cache-Coherence Protocols

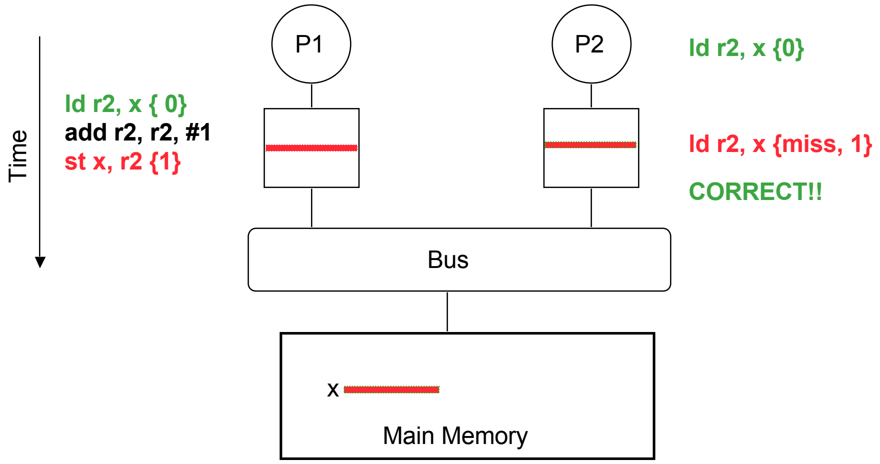
- Bus provides serialization point
  - Broadcast, totally ordered
- Each cache controller “snoops” all bus transactions
  - Controller updates state of blocks in response to processor and snoop events and generates bus transactions
- Snoopy protocol (FSM)
  - set of states
  - state-transition diagram
  - actions
- Basic Choices
  - write-through vs. write-back
  - invalidate vs. update



Cache Coherence Invalidate Protocol (Step 3)



Cache Coherence Invalidate Protocol (Step 4)



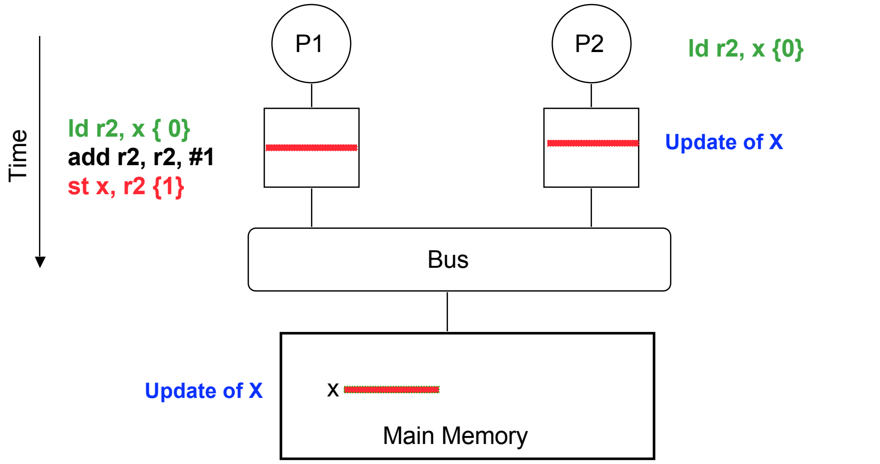
ld r2, x { 0}
add r2, r2, #1
st x, r2 {1}

ld r2, x {0}

ld r2, x {miss, 1}

CORRECT!!

Cache Coherence Update Protocol (Step 3)

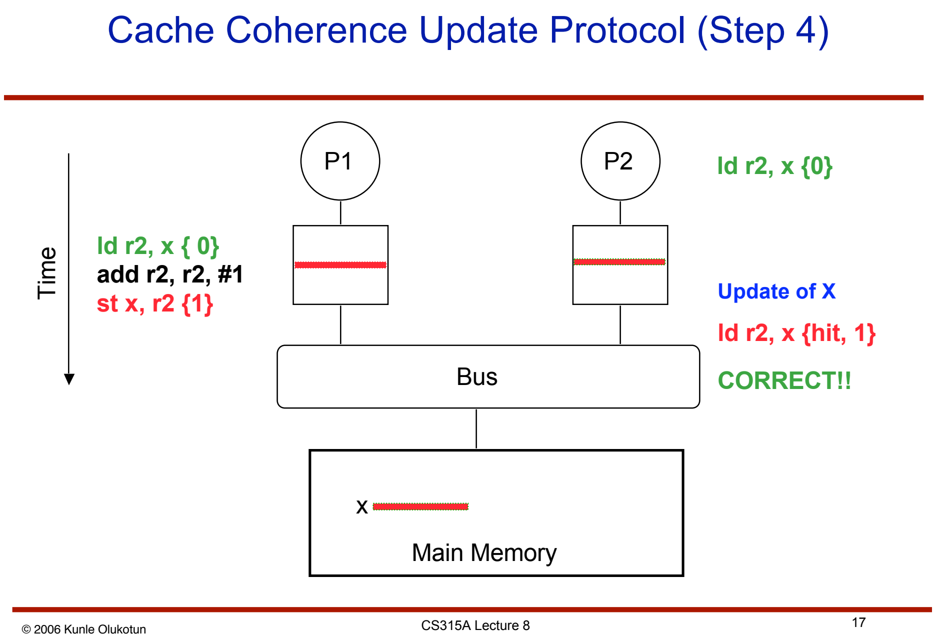


ld r2, x { 0}
add r2, r2, #1
st x, r2 {1}

ld r2, x {0}

Update of X

Update of X



Cache Coherence Update Protocol (Step 4)

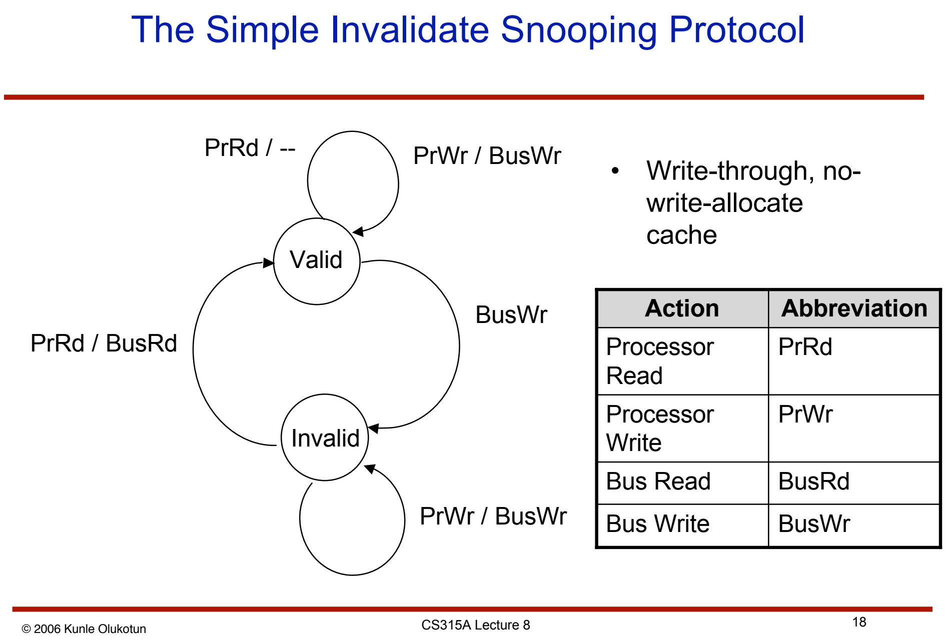
ld r2, x { 0}
add r2, r2, #1
st x, r2 {1}

ld r2, x {0}

Update of X

ld r2, x {hit, 1}

CORRECT!!



The Simple Invalidate Snooping Protocol

- Write-through, no-write-allocate cache

Action	Abbreviation
Processor Read	PrRd
Processor Write	PrWr
Bus Read	BusRd
Bus Write	BusWr

Is 2-state Protocol Coherent?

- Assume bus transactions and memory operations are atomic, one-level cache
  - processor waits for memory operation to finish before issuing next
  - with one-level cache, assume invalidations applied during bus xaction
- All writes go to bus + atomicity
  - Writes serialized by order in which they appear on bus (bus order)
    - invalidations applied to caches in bus order
- How to insert reads in this order?
  - Read misses
    - appear on bus, and will "see" last write in bus order
  - Read hits: do not appear on bus
    - But value read was placed in cache by either
      - most recent write by this processor, or most recent read miss
      - Both these transactions appeared on the bus
    - So reads hits also see values as produced bus order

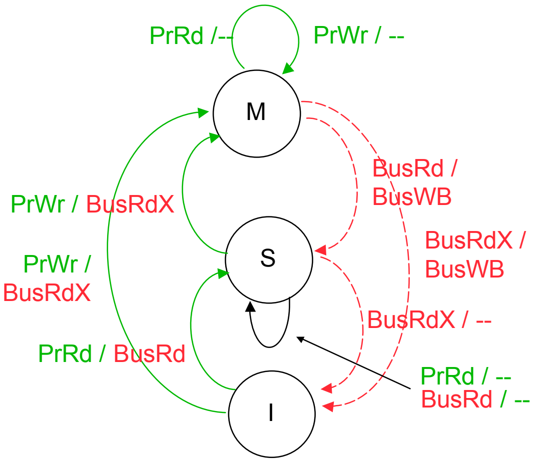
A 3-State Write-Back Invalidation Protocol

- 2-State Protocol
  - + Simple hardware and protocol
  - – Bandwidth (every write goes on bus!)
- 3-State Protocol (MSI)
  - Modified (H&P calls Exclusive)
    - one cache has valid/latest copy
    - memory is stale
  - Shared
    - one or more caches (and memory) have valid copy
  - Invalid
- Must invalidate all other copies before entering modified state
- Requires bus transaction (order and invalidate)

MSI Processor and Bus Actions

- Processor:
  - PrRd
  - PrWr
  - Writeback on replacement of modified block
- Bus
  - Bus Read (BusRd) Read without intent to modify, data could come from memory or another cache
  - Bus Read-Exclusive (BusRdX) Read with intent to modify, must invalidate all other caches copies
  - Writeback (BusWB) cache controller puts contents on bus and memory is updated
  - Definition: cache-to-cache transfer occurs when another cache satisfies BusRd or BusRdX request
- Let's draw it!

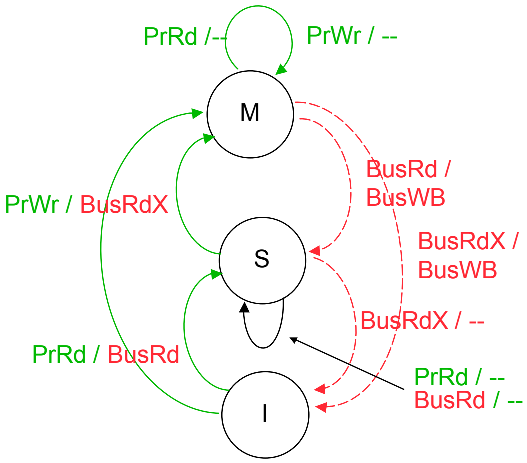
MSI State Diagram



Abbreviation	Action
PrRd	Processor Read
PrWr	Processor Write
BusRd	Bus Read
BusRdX	Bus Read Exclusive
BusWB	Bus Writeback

MSI Invalidate Protocol

- Read obtains block in "shared"
  - even if only cache copy
- Obtain exclusive ownership before writing
  - BusRdX causes others to invalidate
  - If M in another cache, will cause writeback
  - BusRdX even if hit in S
    - promote to M (upgrade)



A Cache Coherence Example

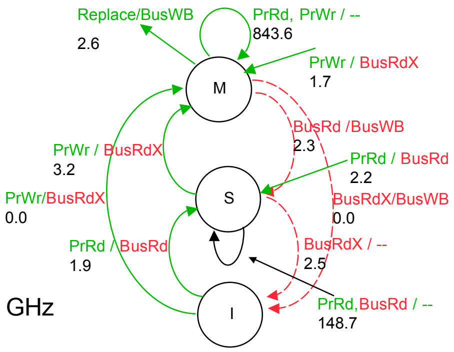
Proc Action	P1 State	P2 state	P3 state	Bus Act	Data from
1. P1 read u	S	--	--	BusRd	Memory
2. P3 read u	S	--	S	BusRd	Memory
3. P3 write u	I	--	M	BusRdX	Memory *or* P3
4. P1 read u	S	--	S	BusRd	P3's cache
5. P2 read u	S	S	S	BusRd	Memory
6. P2 write u	I	M	I	BusRdX	P2's cache

- Single writer, multiple reader protocol
- Why do you need Modified to Shared?

Protocol Analysis

- Ocean
  - Transitions per 1000 data refs
  - 22% loads, 5% stores, 73% other
    - 0.27 data refs/instr
- Compute traffic per CPU (1 GHz)
  - I→S : 0.19%
  - S→M : 0.32%
  - M→S : 0.23%
  - M→NP : 0.26%
  - NP→S : 0.22%
  - NP→M : 0.17%
  - Total : 1.22% of data refs
  - 1.22% x 0.27 refs/instr x 70 bytes x 1 GHz
  - = 230 Mbytes/sec

Bus trans = 70 bytes (6 addr/cmd + 64 data)



A Cache Coherence: Optimizations

Proc Action	P1 State	P2 state	P3 state	Bus Act	Data from
1. P1 read u	S	--	--	BusRd	Memory
2. P3 read u	S	--	S	BusRd	Memory
3. P3 write u	I	--	M	BusRdX	Memory *or* P3
4. P1 read u	S	--	S	BusRd	P3's cache
5. P2 read u	S	S	S	BusRd	Memory
6. P2 write u	I	M	I	BusInv	P2's cache

- Upgrade (ownership) misses (S→M)
  - Use invalidate instead of BusRdX
- What if not in any cache (sequential application)?
  - Read, Write produces 2 bus transactions!

4-State (MESI) Invalidation Protocol

- Often called the ***Illinois*** protocol
- Modified (dirty)
- Exclusive (clean unshared) only this cache has copy, but not dirty
- Shared
- Invalid
- Requires a shared signal to detect if other caches have a copy of block (S)
- Bus writeback for cache-to-cache transfers
  - Only one can do it though
- What does state diagram look like?

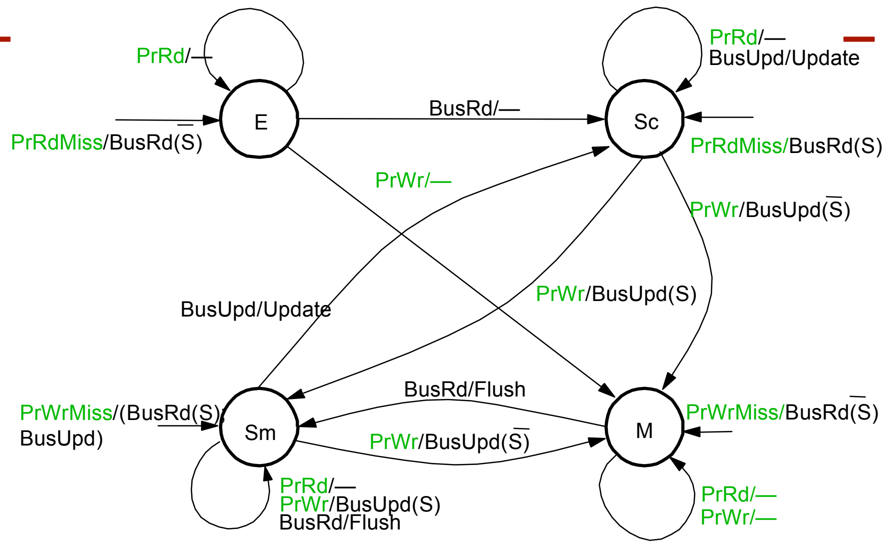
Update Protocols

- If data is to be communicated between processors, invalidate protocols seem inefficient
- Consider a 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

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



SMP Performance

- Cache coherence protocol
  - Update vs. invalidate
  - Bus bandwidth
- Memory hierarchy performance
  - Miss rate
  - Number of processors
  - Cache size
  - Block size
- Highly application dependent
  - Commercial
  - Scientific

Update versus Invalidate

- Much debate over the years: tradeoff depends on sharing patterns
- Intuition:
  - If reads and writes are interleaved, 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
    - particularly bad under process migration
    - useless updates where only last one will be used
- Can construct scenarios where one or other is much better
- Can combine them in hybrid schemes
  - E.g. competitive: observe patterns at runtime and change protocol

Bus Traffic for Invalidate vs. Update

- Pattern 1:

```
for i = 1 to k
  P1(write, x);     // one write before reads
  P2-PN(read, x);
end for i
```

- Pattern 2:

```
for i = 1 to k
  for j = 1 to m
   P1(write, x);   // many writes before reads
  end for j
  P2(read, x);
end for i
```

Assume:

1. **Invalidation/upgrade = 6 bytes (5 addr, 1 cmd)**
2. **Update = 14 bytes (6 addr/cmd + 8 data)**
3. **Cache miss = 70 bytes (6 addr/cmd + 64 data)**

Bus Traffic for Invalidate vs. Update, cont.

- Pattern 1:

```
for i = 1 to k
  P1(write, x);
  P2-PN(read, x);
end for i
```

- Pattern 2:

```
for i = 1 to k
  for j = 1 to m
   P1(write, x);
  end for j
  P2(read, x);
end for i
```

- Pattern 1 (one write before reads)
  - N = 16, m = 10, k = 10
  - Update
    - Iteration 1: N regular cache misses (70 bytes)
    - Remaining iterations: update per iteration (14 bytes)
    - Total Update Traffic = 16*70 + 9*14 = 1246 bytes
  - Invalidate
    - Iteration 1: N regular cache misses (70 bytes)
    - Remaining: P1 generates upgrade (6), 15 others Read miss (70)
    - Total Invalidate Traffic = 16*70 + 9*6 + 15*9*70 = 10,624 bytes
- Pattern 2 (many writes before reads)
  - Update = 2*70 + 10*9*14 = 1400 bytes
  - Invalidate = 11*70 + 9*6 = 824 bytes

Invalidate vs. Update Reality

- What about real workloads?
  - Update can generate too much traffic
  - Must limit (e.g., competitive snooping)

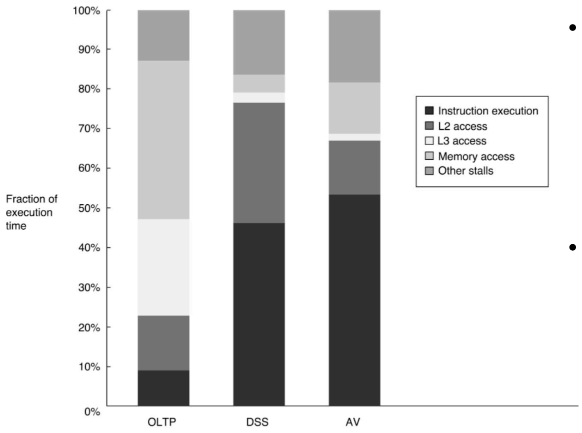
- Current Assessment
  - Update very hard to implement correctly (consistency discussion coming next week)
  - Rarely done

- Future Assessment
  - May be same as current or
  - Chip multiprocessors may revive update protocols
    - More intra-chip bandwidth
    - Easier to have predictable timing paths?

Memory Hierarchy Performance

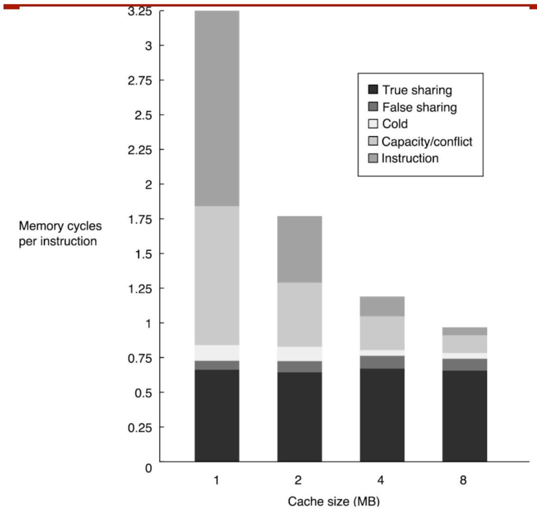
- Uniprocessor 3C's
  - (Compulsory, Capacity, Conflict)
- SM adds Coherence Miss Type (communication)
  - True Sharing miss fetches data written by another processor
  - False Sharing miss results from independent data in same coherence block
- Increasing cache size
  - Usually fewer capacity/conflict misses
  - No effect on true/false "coherence" misses (so may dominate)
- Block size is unit of transfer and of coherence
  - Doesn't have to be, could make coherence smaller
- Increasing block size
  - Usually fewer 3C misses but more bandwidth
  - Usually more false sharing misses

Commercial Application Performance on a 4-Proc AlphaServer



- Alphaserver 4100
  - L1: 8KB D.M.
  - L2: 96KB 3-way S.A.
  - L3: 2MB D.M.
  - 4 processors
- Performance
  - OLTP: 7.0 CPI
  - DSS: 1.6 CPI
  - AltaVista: 1.3 CPI

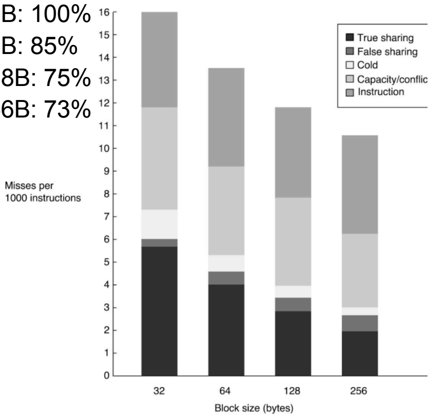
OLTP Memory Performance

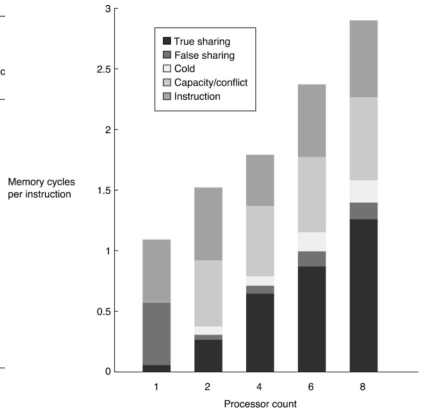


- 2-way S.A.
- Dominant sources
  - Instruction
  - Capacity/conflict
- Increasing cache size
  - Reduces 1-proc misses
  - Sharing misses remain

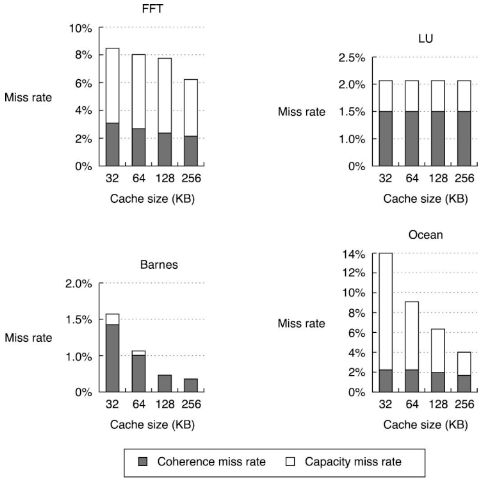
Block Size and Processor Count Effect on OLTP Memory Performance

- Miss rate reduction in 2 MB, 2-way S.A.
  - 32B: 100%
  - 64B: 85%
  - 128B: 75%
  - 256B: 73%

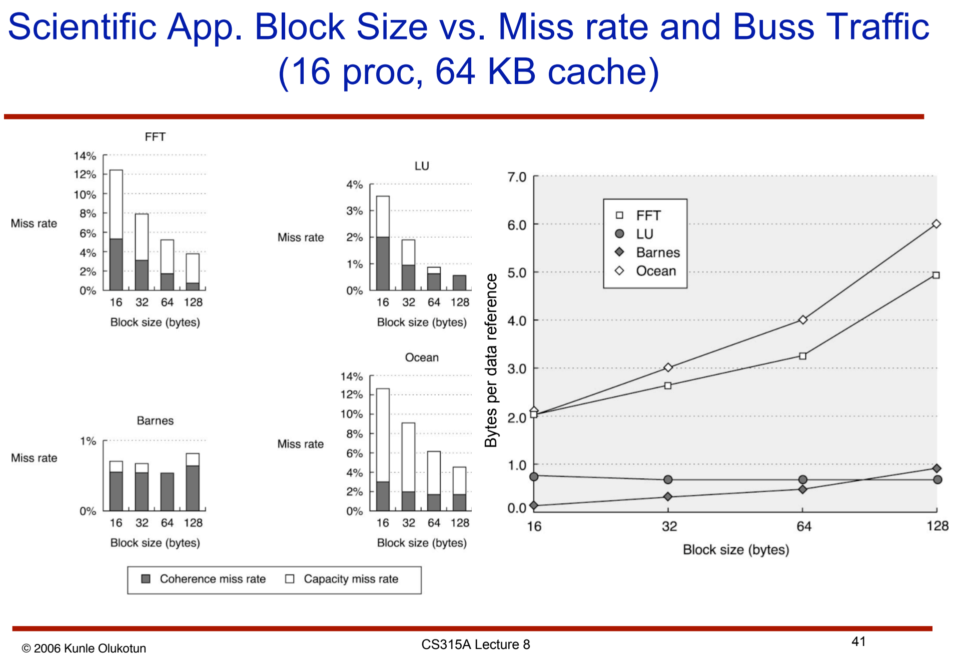




Scientific App. Cache Size vs. Miss rate



- Arch parameters
  - 16 processors
  - 32B block size
- FFT
  - Benefit limited due to . . .
  - Communication for transpose
  - Large working set
- LU
  - Next WS is both matrices
- Barnes
  - Comm *and* capacity reduced
- Ocean
  - Large impact on capacity



Scientific App. Block Size vs. Miss rate and Buss Traffic (16 proc, 64 KB cache)

© 2006 Kunle Olukotun

CS315A Lecture 8

41