CS184b: Computer Architecture (Abstractions and Optimizations)

Day 12: May 3, 2003

Shared Memory

Today

- Shared Memory
  - Model
  - Bus-based Snooping
  - Cache Coherence
- Synchronization
  - Primitives
  - Algorithms
  - Performance

Shared Memory Model

- Same model as multithreaded uniprocessor
  - Single, shared, global address space
  - Multiple threads (PCs)
  - Run in same address space
  - Communicate through memory
    - Memory appear identical between threads
    - Hidden from users (looks like memory op)

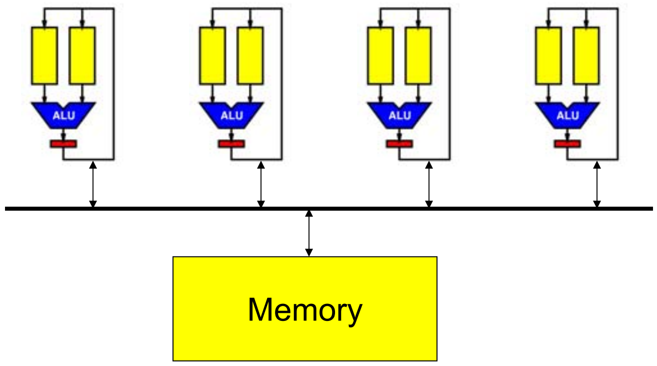
Synchronization

- For correctness have to worry about synchronization
  - Otherwise non-deterministic behavior
  - Threads run asynchronously
  - Without additional/synchronization discipline
    - Cannot say anything about relative timing

Models

- **Conceptual model**:
  - Processor per thread
  - Single shared memory
- **Programming Model**:
  - Sequential language
  - Thread Package
  - Synchronization primitives
- **Architecture Model**: Multithreaded uniprocessor

Conceptual Model



Architecture Model Implications

- Coherent view of memory
  - Any processor reading at time X will see **same** value
  - All writes eventually effect memory
    - Until overwritten
  - Writes to memory seen in same order by all processors
- Sequentially Consistent Memory View

Sequential Consistency

- Memory must reflect some valid sequential interleaving of the threads

Sequential Consistency

- P1: A = 0
-
- A = 1
- L1: if (B==0)

- P2: B = 0
- B = 1
- L2: if (A==0)

Can both conditionals be true?

Sequential Consistency

- P1: A = 0
-
- A = 1
- L1: if (B==0)

- P2: B = 0
- B = 1
- L2: if (A==0)

Both can be false

Sequential Consistency

- P1: A = 0
-
- A = 1
- L1: if (B==0)

- P2: B = 0
- B = 1
- L2: if (A==0)

If enter L1, then A must be 1
$\rightarrow$ not enter L2

Sequential Consistency

- P1: A = 0
-
- A = 1
- L1: if (B==0)

- P2: B = 0
- B = 1
- L2: if (A==0)

If enter L2, then B must be 1
$\rightarrow$ not enter L1

Coherence Alone

- Coherent view of memory
  - Any processor reading at time X will see same value
  - All writes eventually effect memory
    - Until overwritten
  - Writes to memory seen in same order by all processors
- Coherence alone does not guarantee sequential consistency

Sequential Consistency

```
• P1:  A = 0                • P2:  B = 0
•
•        A = 1              •        B = 1
• L1:   if (B==0)           • L2:  if (A==0)
```

If not force visible changes of variable, (assignments of A, B), could end up inside both.

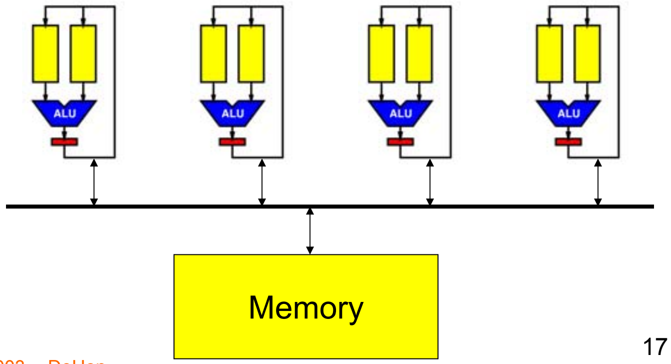
Consistency

- Deals with when written value must be seen by readers
- **Coherence** – w/ respect to same memory location
- **Consistency** – w/ respect to other memory locations
- …there are less strict consistency models…

Implementation

Naïve

- What's wrong with naïve model?



What's Wrong?

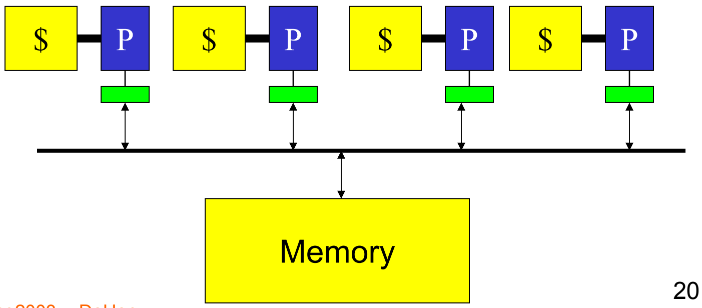
- Memory bandwidth
  - 1 instruction reference per instruction
  - 0.3 memory references per instruction
  - 333ps cycle
  - N*5 Gwords/s ?
- Interconnect
- Memory access latency

Optimizing

- How do we improve?

Naïve Caching

- What happens when add caches to processors?



Naïve Caching

- Cached answers may be stale
- Shadow the correct value

How have both?

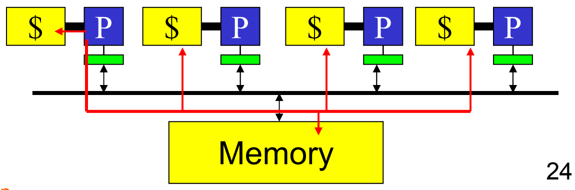
- Keep caching
  - Reduces main memory bandwidth
  - Reduces access latency
- Satisfy Model

Cache Coherence

- Make sure everyone sees **same** values
- Avoid having stale values in caches
- At end of write, all cached values should be the same

Idea

- Make sure everyone sees the new value
- Broadcast new value to everyone who needs it
  - Use bus in shared-bus system



Effects

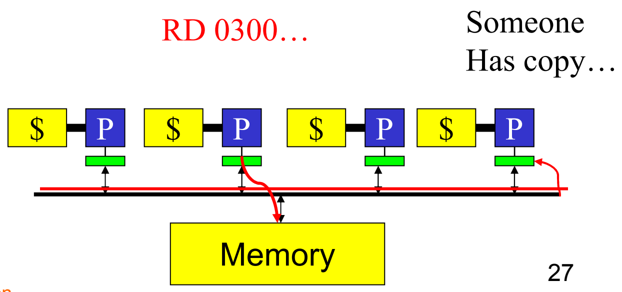
- Memory traffic is now just:
  - Cache misses
  - All writes

Additional Structure?

- Only necessary to write/broadcast a value if someone else has it cached
- Can write locally if know sole owner
  - Reduces main memory traffic
  - Reduces write latency

Idea

- Track usage in cache state
- “Snoop” on shared bus to detect changes in state



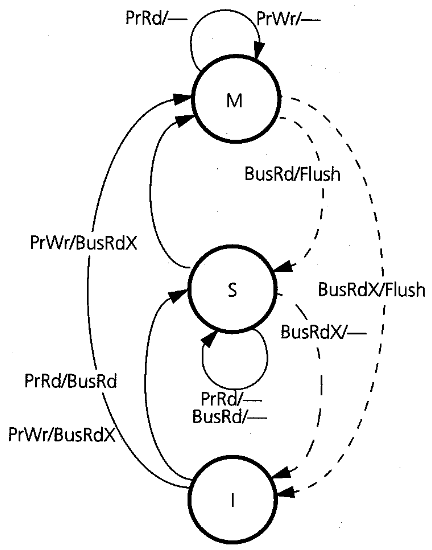
Cache State

- Data in cache can be in one of several states
  - Not cached (not present)
  - Exclusive (not shared)
    - Safe to write to
  - Shared
    - Must share writes with others
- Update state with each memory op

Cache Protocol

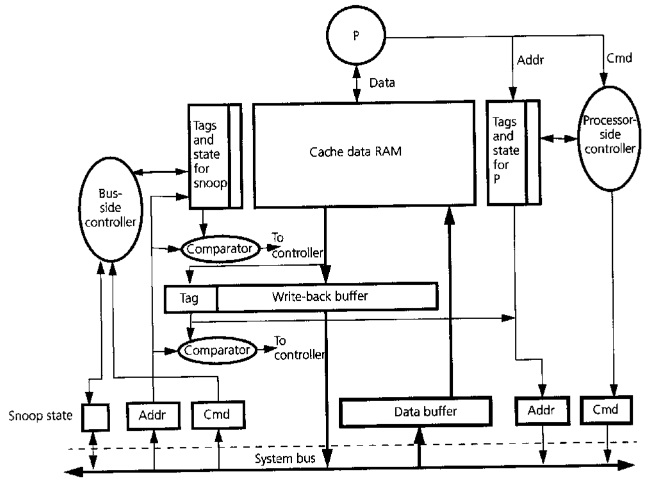
RdX = Read Exclusive

Perform Write by:
•Reading exclusive
•Writing locally



[Culler/Singh/Gupta 5.13]

Snoopy Cache Organization

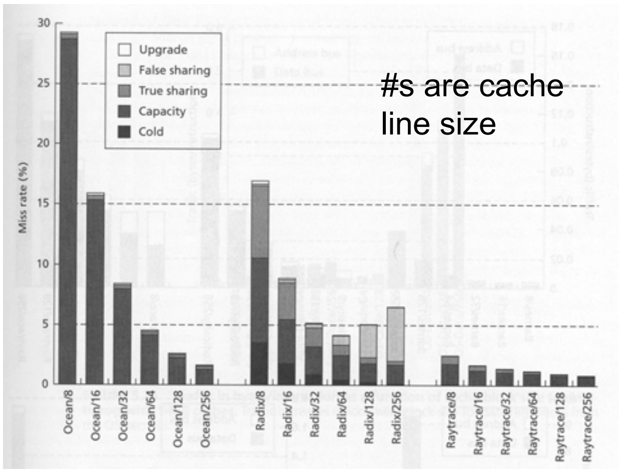


[Culler/Singh/Gupta 6.4]

Cache States

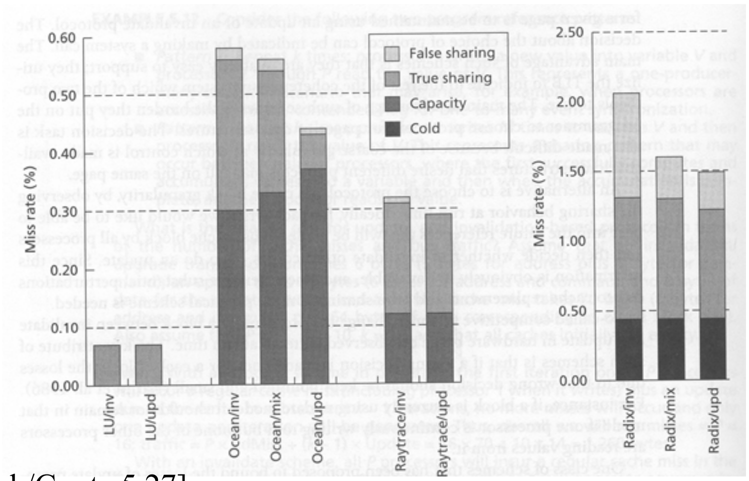
- Extra bits in cache
  - Like valid, dirty

Misses



[Culler/Singh/Gupta 5.23]

Misses



[Culler/Singh/Gupta 5.27]

Synchronization

Problem

- If correctness requires an ordering between threads,
  - have to enforce it
- Was not a problem we had in the single-thread case
  - does occur in the multiple threads on single processor case

Desired Guarantees

- Precedence
  - barrier synchronization
    - Everything before barrier completes before anything after begins
  - producer-consumer
    - Consumer reads value produced by producer
- Atomic Operation Set
- Mutual exclusion

Read/Write Locks?

- Try implement lock with r/w:

```
if (~A.lock)
     A.lock=true
    do stuff
    A.lock=false
```

Problem with R/W locks?

- Consider context switch between test (~A.lock=true?) and assignment (A.lock=true)

```
if (~A.lock)
     A.lock=true
    do stuff
    A.lock=false
```

Primitive Need

- Need Indivisible primitive to enabled atomic operations

Original Examples

- Test-and-set
  - combine test of A.lock and set into single atomic operation
  - once have lock
    - can guarantee mutual exclusion at higher level
- Read-Modify-Write
  - atomic read…write sequence
- Exchange

Examples (cont.)

- Exchange
  - Exchange **true** with A.lock
  - if value retrieved was **false**
    - this process got the lock
  - if value retrieved was **true**
    - already locked
    - (didn't change value)
    - keep trying
  - key is, only single exchanger get the **false** value

Implementing...

- What required to implement?
  - Uniprocessor
  - Bus-based

Implement: Uniprocessor

- Prevent Interrupt/context switch
- Primitives use single address
  - so page fault at beginning
  - then ok, to computation (defer faults…)
- SMT?

Implement: Snoop Bus

- Need to reserve for Write
  - write-through
    - hold the bus between read and write
    - Guarantee no operation can intervene
  - write-back
    - need exclusive read
    - and way to defer other writes until written

Performance Concerns?

- Locking resources reduce parallelism
- Bus (network) traffic
- Processor utilization
- Latency of operation

Basic Synch. Components

- Acquisition
- Waiting
- Release

Possible Problems

- Spin wait generates considerable memory traffic
- Release traffic
- Bottleneck on resources
- Invalidation
  - can't cache locally…
- Fairness

Test-and-Set

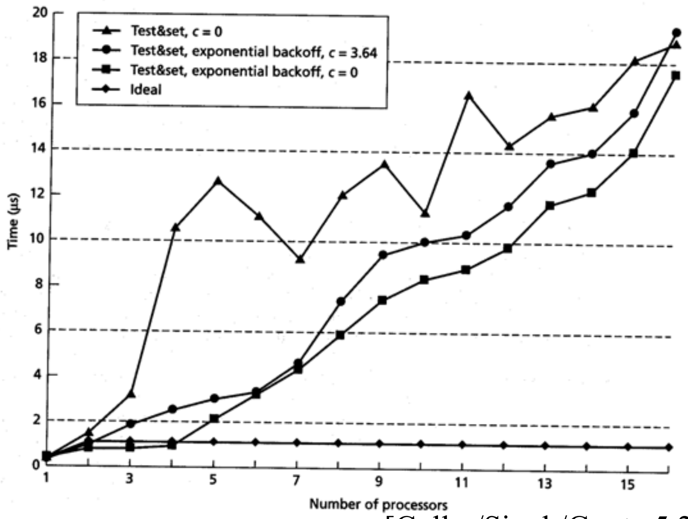
```
Try: t&s R1, A.lock
     bnz R1, Try
     return
```

- Simple algorithm generate considerable traffic
- p contenders
  - p try first, 1 wins
  - for o(1) time p-1 spin
  - …then p-2…
  - c*(p+p-1+p-2,,,)
  - $O(p^2)$

Backoff

- Instead of immediately retrying
  - wait some time before retry
  - reduces contention
  - may increase latency
    - (what if I'm only contender and is about to be released?)

Primitive Bus Performance



[Culler/Singh/Gupta 5.29]

Bad Effects

- Performance Decreases with users
  - From growing traffic already noted

Test-test-and-Set

```
Try:  ld R1, A.lock
      bnz R1, Try
      t&s R1, A.lock
      bnz R1, Try
      return
```

- Read can be to local cache
- Not generate bus traffic
- Generates less contention traffic

Detecting atomicity sufficient

- Fine to detect if operation will appear atomic
- Pair of instructions
  - ll -- load locked
    - load value and mark in cache as locked
  - sc -- store conditional
    - stores value iff no intervening write to address
    - *e.g.* cache-line never invalidated by write

LL/SC operation

```
Try: LL R1 A.lock
     BNZ R1, Try
     SC   R2, A.lock
     BEQZ Try
     return from lock
```

LL/SC

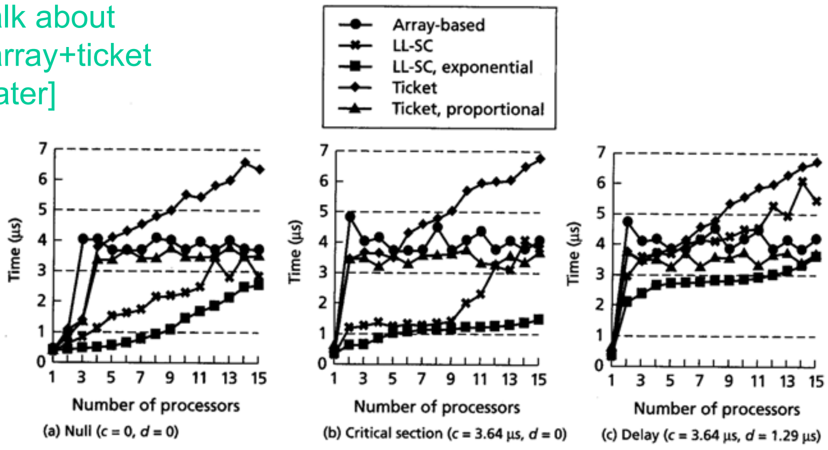
- Pair doesn't really lock value
- Just detects if result would appear that way
- Ok to have arbitrary interleaving between LL and SC
- Ok to have capacity eviction between LL and SC
  - will just fail and retry

LL/SC and MP Traffic

- Address can be cached
- Spin on LL not generate global traffic (everyone have their own copy)
- After write (*e.g.* unlock)
  - everyone miss -- O(p) message traffic
- No need to lock down bus during operation

Performance Bus

[talk about
array+ticket
later]



[Culler/Singh/Gupta 5.30]

Big Ideas

- Simple Model
  - Preserve model
  - While optimizing implementation
- Exploit Locality
  - Reduce bandwidth and latency

Big Ideas

- Simple primitives
  - Must have primitives to support atomic operations
  - don't have to implement atomicly
    - just detect non-atomicity
- Make fast case common
  - optimize for locality
  - minimize contention