18-742
Lecture 7

Synchronization

Locks & Barriers

Spring 2005
Prof. Babak Falsafi
http://www.ece.cmu.edu/~ece742

Slides developed in part by Profs. Adve, Falsafi, Hill, Lebeck, Reinhardt, Smith, and Singh of University of Illinois, Carnegie Mellon University, University of Wisconsin, Duke University, University of Michigan, and Princeton University.

Announcements

Project posted on the web
Homework 3 posted due on Friday

Seminar @ Pitt:

Single Chip Multiprocessors: The Next Wave of Computer Architecture

Guri Sohi
University of Wisconsin
Friday, February 4, 2005
10:30am SENSQ 5317

Invalidate vs. Update

- **Pattern 1:**

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


Invalidate vs. Update, cont.

- **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; 6 cntrl, 8 data)
  - Total Update Traffic = 16*70 + 9*14 = 1246 bytes
    - » book assumes 10 updates instead of 9…
  - 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*17 = 10,624 bytes
- **Pattern 2 (many writes before reads)**
  - Update = 1400 bytes
  - Invalidate = 824 bytes

Invalidate vs. Update, cont.

- What about real workloads?
  - Update can generate too much traffic
  - Must limit (e.g., competitive snooping)
- Current Assessment
  - Update very hard to implement correctly (c.f., consistency discussion coming next)
  - 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?

Qualitative Sharing Patterns

- [Weber & Gupta, ASPLOS3]
- Read-Only
- Migratory Objects
  - Manipulated by one processor at a time
  - Often protected by a lock
  - Usually a write causes only a single invalidation
- Synchronization Objects
  - Often more processors imply more invalidations
- Mostly Read
  - More processors imply more invalidations, but writes are rare
- Frequently Read/Written
  - More processors imply more invalidations

Coherence vs. Consistency

- Intuition says loads should return latest value
  - what is latest?
- Coherence concerns only one memory location
- Consistency concerns apparent ordering for all 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 a read is value written by last store to that location

Why Coherence != Consistency

```
/* initial A = B = flag = 0 */
 P1                    P2
A = 1;                 while (flag == 0); /* spin */
B = 1;                 print A;
flag = 1;              print B;
```

Intuition says printed A = B = 1

Coherence doesn't say anything, why?

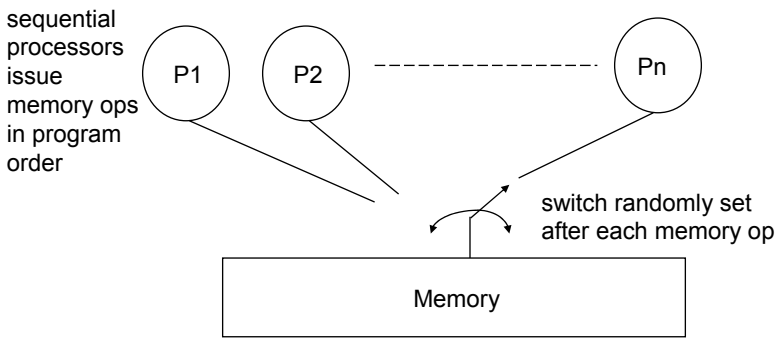
Consider coalescing write buffer

Sequential Consistency

- **Lamport 1979**

"A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program"

The Memory Model



Definitions and Sufficient Conditions

- **Sequentially Consistent Execution**
  - **result is same as one of the possible interleavings on uniprocessor**
- **Sequentially Consistent System**
  - **any possible execution corresponds to some possible total order**

Definitions

- **Memory operation**
  - exection of load, store, atomic read-modify-write access to mem location
- **Issue**
  - operation is issued when it leaves processor and is presented to memory system (cache, write-buffer, local and remote memories)
- **Perform**
  - store is performed wrt to a processor p when a load by p returns value produced by that store or a later store
  - A load is performed wrt to a processor when subsequent stores cannot affect value returned by that load
- **Complete**
  - memory operation is performed wrt all processors.
- **Program Execution**
  - Memory operations for specific run only (ignore non memory-referencing instructions)

Sufficient Conditions for Sequential Consistency

- Every processor issues memory ops in program order
- Processor must wait for store to complete before issuing next memory operation
- After load, issuing proc waits for load to complete, and store that produced value to complete before issuing next op
- Easily implemented with shared bus.

Synchronization

- Mutual Exclusion (critical sections)
  - Lock & Unlock
- Event Notification
  - point-to-point (producer-consumer, flags)
  - global (barrier)
- LOCK, BARRIER
  - How are these implemented?

Anatomy of A Synchronization Operation

- **Acquire Method**
  - **method for trying to obtain the lock, or proceed past barrier**
- **Waiting Algorithm**
  - **Spin or busy wait**
  - **Block (suspend)**
- **Release Method**
  - **method to allow other processes to proceed past synchronization event**

HW/SW Implementation Tradeoffs

- **User wants high level (ease of programming)**
  - **LOCK(lock_variable), UNLOCK(lock_variable)**
  - **BARRIER(barrier_variable, Num_Procs)**
- **Hardware**
  - **The Need for Speed (it's fast)**
- **Software**
  - **Flexible**
- **Want**
  - **low latency**
  - **low traffic**
  - **Scalability**
  - **low storage overhead**
  - **fairness**

How Not To Implement Locks

- **LOCK**
  while(lock_variable **== 1);**
  lock_variable **= 1;**

- **UNLOCK**
  lock_variable **= 0;**

- **Implementation requires Mutual Exclusion!**
  - **Can have two processes successfully acquire the lock**

Atomic Read-Modify-Write Operations

- **Test&Set(r,x)**
  r = m[x]
  m[x] = 1
- **Swap(r,x)**
  r = m[x], m[x] = r
- **Compare&Swap(r1,r2,x)**
  if (r1 == m[x]) then
  r2 = m[x], m[x] = r2
- **Fetch&Op(r,x,op)**
  r = m[x], m[x] = op(m[x])

- r is register
- m[x] is memory location x

Test & Set in SPARC

```
test_and_set()
        ldstub      [%l1], %l0    # lock = 256 (set)

unset()
        st          %g0, [%l1]    # lock = 0
```


Better Lock Implementations

- **Two choices:**
  - Don't execute test&set so much
  - Spin without generating bus traffic
- **Test&Set with Backoff**
  - Insert delay between test&set operations (not too long)
  - Exponential seems good ($k*c^i$)
  - Not fair
- **Test-and-Test&Set**
  - Spin (test) on local cached copy until it gets invalidated, then issue test&set
  - Intuition: No point in trying to set the location until we know that it's not set, which we can detect when it get invalidated...
  - Still contention after invalidate
  - Still not fair

Test & Test & Set in SPARC

```
test_and_test_and_set()
L1:     ld          [%l1], %l0      # check first
        bne         L1              # if (z) not set, loop
        ldstub      [%l1], %l0      # lock = 256 (set)

unset()
        st          %g0, [%l1]      # lock = 0
```


Load-Locked Store-Conditional

- Pair of Instructions
- Load-Locked sets flag and address
  - results in local spinning for free (no need for test&test&set)
- Store-Conditional fails if flag clear
- Flag is cleared on
  - invalidation
  - replacement
  - context switch

Test & Set in Alpha (with Local Spinning)

```
test_and_set()
L1:     ldl_l   t0, 0(t1)       # load locked, t0 = lock
        bne     t0, L1          # if not free, loop
        lda     t0, 1(0)        # t0 = 1
        stl_c   t0, 0(t1)       # conditional store,
                                # lock = 1
        beq     t0, L1          # if failed, loop

unset()
        stl     0, 0(t1)
```


Performance of Test & Set

LOCK

```
while (test&set(x) == 1);
```

UNLOCK

```
x = 0;
```

- High contention (many processes want lock)
- Remember the CACHE!
- Each test&set is a read miss and a write miss
  - Not fair
- Problem is?
- Waiting Algorithm!

Fetch&Inc-Based Locks

- **Ticket Lock**

LOCK

  - **Obtain number via fetch&inc**
  - **Spin on now-serving counter**

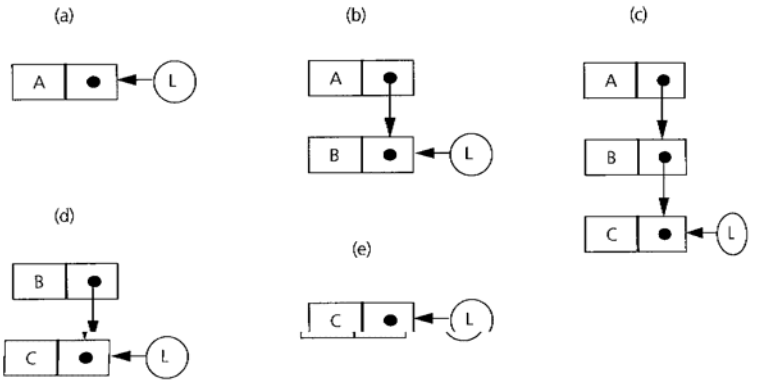
Unlock

  - **Increment now-serving counter**

- **Array based Lock**
  - **Obtain location to spin on rather than value**
  - **Fair**
  - **Slight increase in storage**
  - **Put locations in separate cache blocks, else same traffic as t&t&s**

Queue-Based Locks

- **Linked list points to next in line**
  - **QOLB in hardware**
  - **MCS in software**



What are the Performance Issues?

- **Low latency:**
  - **should be able to get a free lock quickly**
- **Scalability:**
  - **should perform well beyond a small number of procs (< 64)**
- **Low storage overhead**
- **Fairness**
- **Blocking/Non-blocking**
- **Are spin locks fair?**

Implementation Details

- **To Cache or Not to Cache, that is the question.**

Uncached

- Latency for one operation increases

+ Fast hand-off between processes

Cached

- Might generate a lot of traffic if lock moves around

+ Might reuse lock a lot (locality), then traffic would be reduced by caching

- **Must keep ownership for entire read-modify-write cycle**
  - **synchronization operation is visible to the memory system**
  - **we'll exploit this fact later in the semester**

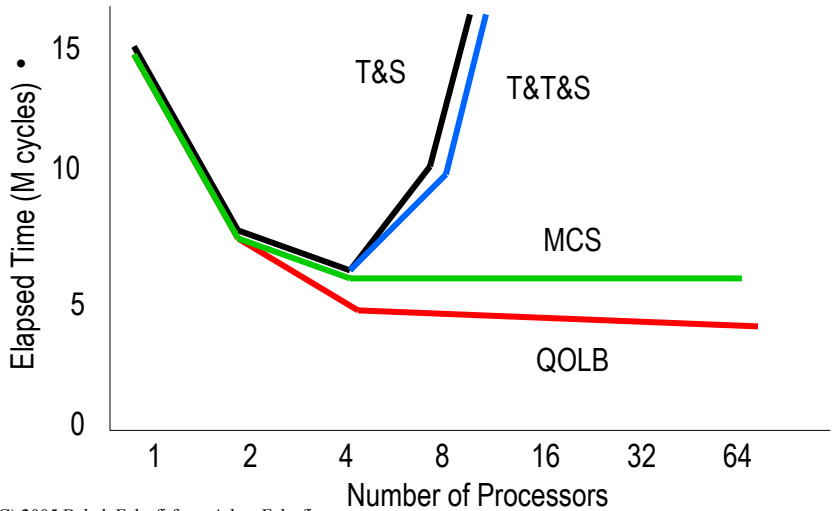
Kaegi et al.'s Paper: Fundamental Mechanisms to Reduce Overhead

- **Basic mechanisms used by locks:**
  - **Local spinning**
  - **Queue-based locking**
  - **Collocation**
  - **Synchronous prefetch**

	L-spin	Queue	Collocation	S-fetch
T&S	no	no	optional	no
T&T&S	yes	no	optional	no
MCS	yes	yes	partial	no
QOLB	yes	yes	optional	yes

Microbenchmark Analysis

Lock performance with increase in processors



Performance of Locks

- Contested vs. Uncontested
- Test&set is good with no contention
- Array based (Queue) is best with high contention
- *Reactive Synchronization* by Lim & Agarwal
  - Choose lock implementation based on contention

Point-to-Point Event Synchronization

- Often use normal variables as flags

```
a = f(x);              while (flag == 0);
flag = 1;              b = g(a);
```

- If we know a before hand

```
a = f(x)               while (a == 0);
                       b = g(a);
```

- Assumes Sequential Consistency!!
- Full/Empty Bits
  - Set on Write
  - Cleared on Read
  - Can't write if set, can't read if clear

Implementing a Centralized Barrier

```
BARRIER(bar_name, p) {
    LOCK(bar_name.lock);
    if (bar_name.counter = 0)
      bar_name.flag = 0;
    bar_name.counter++;
    UNLOCK(bar_name.lock);
    if (bar_name.counter == p) {
      bar_name.counter = 0;
      bar_name.flag = 1;
    }
    else
      while(bar_name.flag = 0) {};        /* busy wait */
  }
```

- **Does this work?**

Barrier With Sense Reversal

```
BARRIER(bar_name, p) {
    local_sense = !(local_sense);              /* toggle private state */
    LOCK(bar_name.lock);
    bar_name.counter++;
    UNLOCK(bar_name.lock);
    if (bar_name.counter == p) {
      bar_name.counter = 0;
      bar_name.flag = local_sense;
    }
    else
      while(bar_name.flag != local_sense) {};    /* busy wait */
  }
```

Synchronization Algorithms

- **Tournament Barriers, SW Combining Tree**