CS 258
Parallel Computer Architecture
Lecture 23

Hardware-Software Trade-offs in Synchronization and Data Layout

April 21, 2008

Prof John D. Kubiatowicz
http://www.cs.berkeley.edu/~kubitron/cs258

Role of Synchronization

- "A parallel computer is a collection of processing elements that cooperate and communicate to solve large problems fast."
- Types of Synchronization
  - *Mutual Exclusion*
  - Event synchronization
    - » *point-to-point*
    - » group
    - » *global (barriers)*
- How much hardware support?
  - high-level operations?
  - atomic instructions?
  - specialized interconnect?

Components of a Synchronization Event

- Acquire method
  - Acquire right to the synch
    - » enter critical section, go past event
- Waiting algorithm
  - Wait for synch to become available when it isn't
  - busy-waiting, blocking, or hybrid
- Release method
  - Enable other processors to acquire right to the synch
- Waiting algorithm is independent of type of synchronization
  - makes no sense to put in hardware

Strawman Lock

Busy-Wait

```
lock:   ld   register, location   /* copy location to register */
        cmp  location, #0         /* compare with 0 */
        bnz  lock                 /* if not 0, try again */
        st   location, #1         /* store 1 to mark it locked */
        ret                       /* return control to caller */

unlock: st   location, #0         /* write 0 to location */
        ret                       /* return control to caller */
```

Why doesn't the acquire method work?

Release method?

What to do if only load and store?

- Here is a possible two-thread solution:

Thread A	Thread B
`Set A=1;`	`Set B=1;`
`while (B) {//X`	`if (!A) {//Y`
`  do nothing;`	`  Critical Section;`
`}`	`}`
`Critical Section;`	`Set B=0;`
`Set A=0;`	

- Does this work? Yes. Both can guarantee that:
  - Only one will enter critical section at a time.
- At X:
  - if B=0, safe for A to perform critical section,
  - otherwise wait to find out what will happen
- At Y:
  - if A=0, safe for B to perform critical section.
  - Otherwise, A is in critical section or waiting for B to quit
- But:
  - Really messy
  - Generalization gets worse

Atomic Instructions

- Specifies a location, register, & atomic operation
  - Value in location read into a register
  - Another value (function of value read or not) stored into location
- Many variants
  - Varying degrees of flexibility in second part
- Simple example: test&set
  - Value in location read into a specified register
  - Constant 1 stored into location
  - Successful if value loaded into register is 0
  - Other constants could be used instead of 1 and 0

Zoo of hardware primitives

-
```
test&set (&address) {     /* most architectures */
    result = M[address];
    M[address] = 1;
    return result;
}
```
-
```
swap (&address, register) { /* x86 */
    temp = M[address];
    M[address] = register;
    register = temp;
}
```
-
```
compare&swap (&address, reg1, reg2) { /* 68000 */
    if (reg1 == M[address]) {
        M[address] = reg2;
        return success;
    } else {
        return failure;
    }
}
```
-
```
load-linked&store conditional(&address) {
    /* R4000, alpha */
    loop:
        ll r1, M[address];
        movi r2, 1;           /* Can do arbitrary comp */
        sc r2, M[address];
        beqz r2, loop;
}
```


Mini-Instruction Set debate

- atomic read-modify-write instructions
  - IBM 370: included atomic compare&swap for multiprogramming
  - x86: any instruction can be prefixed with a lock modifier
  - High-level language advocates want hardware locks/barriers
    - » but it's goes against the "RISC" flow,and has other problems
  - SPARC: atomic register-memory ops (swap, compare&swap)
  - MIPS, IBM Power: no atomic operations but pair of instructions
    - » load-locked, store-conditional
    - » later used by PowerPC and DEC Alpha too
  - 68000: CCS: Compare and compare and swap
    - » No-one does this any more
- Rich set of tradeoffs

Other forms of hardware support

- Separate lock lines on the bus
- Lock locations in memory
- Lock registers (Cray Xmp)
- Hardware full/empty bits (Tera)
- QOLB (machines supporting SCI protocol)
- Bus support for interrupt dispatch

Simple Test&Set Lock

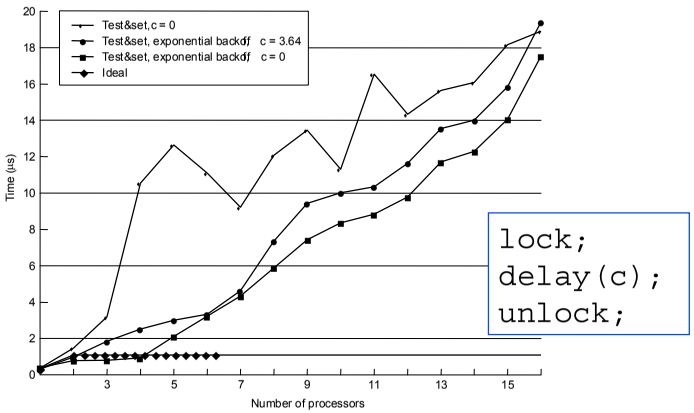
```
lock:   t&s   register, location
        bnz   lock                  /* if not 0, try again */
        ret                         /* return control to caller */
unlock: st    location, #0          /* write 0 to location */
        ret                         /* return control to caller */
```

- Other read-modify-write primitives
  - Swap
  - Fetch&op
  - Compare&swap
    - »Three operands: location, register to compare with, register to swap with
    - »Not commonly supported by RISC instruction sets
- cacheable or uncacheable

Performance Criteria for Synch. Ops

- Latency (time per op)
  - especially when light contention
- Bandwidth (ops per sec)
  - especially under high contention
- Traffic
  - load on critical resources
  - especially on failures under contention
- Storage
- Fairness

T&S Lock Microbenchmark: SGI Chal.



- Why does performance degrade?
- Bus Transactions on T&S?
- Hardware support in CC protocol?

Enhancements to Simple Lock

- Reduce frequency of issuing test&sets while waiting
  - *Test&set lock with backoff*
  - Don't back off too much or will be backed off when lock becomes free
  - Exponential backoff works quite well empirically: i^{th} time = $k*c^i$
- Busy-wait with read operations rather than test&set
  - *Test-and-test&set lock*
  - Keep testing with ordinary load
    - » cached lock variable will be invalidated when release occurs
  - When value changes (to 0), try to obtain lock with test&set
    - » only one attemptor will succeed; others will fail and start testing again

Busy-wait vs Blocking

- Busy-wait: I.e. spin lock
  - Keep trying to acquire lock until read
  - Very low latency/processor overhead!
  - Very high system overhead!
    - » Causing stress on network while spinning
    - » Processor is not doing anything else useful
- Blocking:
  - If can't acquire lock, deschedule process (I.e. unload state)
  - Higher latency/processor overhead (1000s of cycles?)
    - » Takes time to unload/restart task
    - » Notification mechanism needed
  - Low system overheadd
    - » No stress on network
    - » Processor does something useful
- Hybrid:
  - Spin for a while, then block
  - 2-competitive: spin until have waited blocking time

Improved Hardware Primitives: LL-SC

- Goals:
  - Test with reads
  - Failed read-modify-write attempts don't generate invalidations
  - Nice if single primitive can implement range of r-m-w operations
- *Load-Locked* (or -linked), *Store-Conditional*
  - LL reads variable into register
  - Follow with arbitrary instructions to manipulate its value
  - SC tries to store back to location
  - succeed if and only if no other write to the variable since this processor's LL
    - » indicated by condition codes;
- If SC succeeds, all three steps happened atomically
- If fails, doesn't write or generate invalidations
  - must retry aquire

Simple Lock with LL-SC

```
lock:   ll     reg1, location     /* LL location to reg1 */
        sc     location, reg2     /* SC reg2 into location*/
        beqz   reg2, lock         /* if failed, start again */
        ret
unlock: st     location, #0       /* write 0 to location */
        ret
```

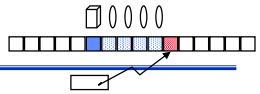
- Can do more fancy atomic ops by changing what's between LL & SC
  - But keep it small so SC likely to succeed
  - Don't include instructions that would need to be undone (e.g. stores)
- SC can fail (without putting transaction on bus) if:
  - Detects intervening write even before trying to get bus
  - Tries to get bus but another processor's SC gets bus first
- LL, SC are not lock, unlock respectively
  - Only guarantee no conflicting write to lock variable between them
  - But can use directly to implement simple operations on shared variables

Trade-offs So Far

- Latency?
- Bandwidth?
- Traffic?
- Storage?
- Fairness?

- What happens when several processors spinning on lock and it is released?
  - traffic per P lock operations?

Ticket Lock

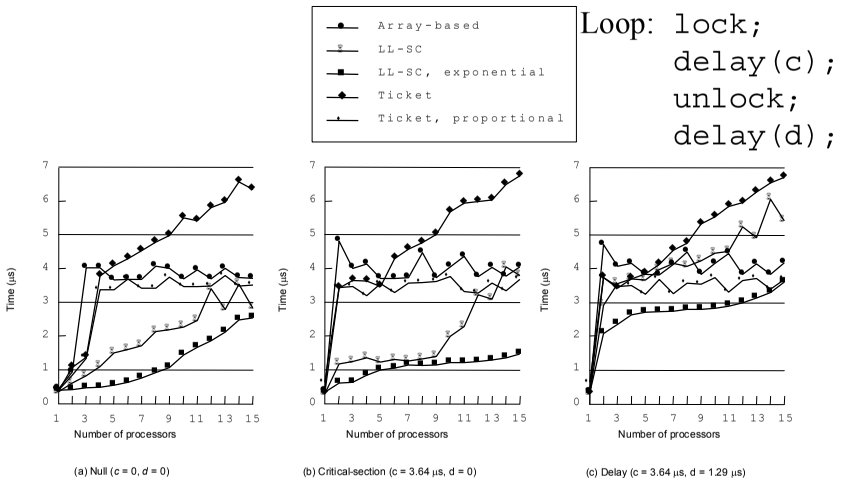


- Only one r-m-w per acquire
- Two counters per lock (next_ticket, now_serving)
  - Acquire: fetch&inc next_ticket;
    wait for now_serving == next_ticket
    - » atomic op when arrive at lock, not when it's free (so less contention)
  - Release: increment now-serving
- Performance
  - low latency for low-contention - if fetch&inc cacheable
  - O(p) read misses at release, since all spin on same variable
  - FIFO order
    - » like simple LL-SC lock, but no inval when SC succeeds, and fair
  - Backoff?
- Wouldn't it be nice to poll different locations ...

Array-based Queuing Locks

- Waiting processes poll on different locations in an array of size *p*
  - Acquire
    - » fetch&inc to obtain address on which to spin (next array element)
    - » ensure that these addresses are in different cache lines or memories
  - Release
    - » set next location in array, thus waking up process spinning on it
  - *O(1)* traffic per acquire with coherent caches
  - FIFO ordering, as in ticket lock, but, *O(p)* space per lock
  - Not so great for non-cache-coherent machines with distributed memory
    - » array location I spin on not necessarily in my local memory (solution later)

Lock Performance on SGI Challenge



Point to Point Event Synchronization

- Software methods:
  - Interrupts
  - Busy-waiting: use ordinary variables as flags
  - Blocking: use semaphores
- Full hardware support: *full-empty bit* with each word in memory
  - Set when word is "full" with newly produced data (i.e. when written)
  - Unset when word is "empty" due to being consumed (i.e. when read)
  - Natural for word-level producer-consumer synchronization
    - » producer: write if empty, set to full; consumer: read if full; set to empty
  - Hardware preserves atomicity of bit manipulation with read or write
  - Problem: flexibility
    - » multiple consumers, or multiple writes before consumer reads?
    - » needs language support to specify when to use
    - » composite data structures?

Barriers

- Software algorithms implemented using locks, flags, counters
- Hardware barriers
  - Wired-AND line separate from address/data bus
    - » Set input high when arrive, wait for output to be high to leave
  - In practice, multiple wires to allow reuse
  - Useful when barriers are global and very frequent
  - Difficult to support arbitrary subset of processors
    - » even harder with multiple processes per processor
  - Difficult to dynamically change number and identity of participants
    - » e.g. latter due to process migration
  - Not common today on bus-based machines

A Simple Centralized Barrier

- Shared counter maintains number of processes that have arrived
  - increment when arrive (lock), check until reaches numprocs
  - Problem?

```
struct bar_type {int counter; struct lock_type lock;
                              int flag = 0;} bar_name;
BARRIER (bar_name, p) {
    LOCK(bar_name.lock);
    if (bar_name.counter == 0)
        bar_name.flag = 0;              /* reset flag if first to reach*/
    mycount = bar_name.counter++;       /* mycount is private */
    UNLOCK(bar_name.lock);
    if (mycount == p) {                 /* last to arrive */
        bar_name.counter = 0;           /* reset for next barrier */
        bar_name.flag = 1;              /* release waiters */
    }
    else while (bar_name.flag == 0) {};   /* busy wait for release */
}
```


A Working Centralized Barrier

- Consecutively entering the same barrier doesn't work
  - Must prevent process from entering until all have left previous instance
  - Could use another counter, but increases latency and contention
- Sense reversal: wait for flag to take different value consecutive times
  - Toggle this value only when all processes reach

```
BARRIER (bar_name, p) {
    local_sense = !(local_sense);  /* toggle private sense variable */
    LOCK(bar_name.lock);
    mycount = bar_name.counter++;          /* mycount is private */
    if (bar_name.counter == p)
      UNLOCK(bar_name.lock);
      bar_name.flag = local_sense;         /* release waiters*/
    else
       {      UNLOCK(bar_name.lock);
      while (bar_name.flag != local_sense) {}; }
}
```

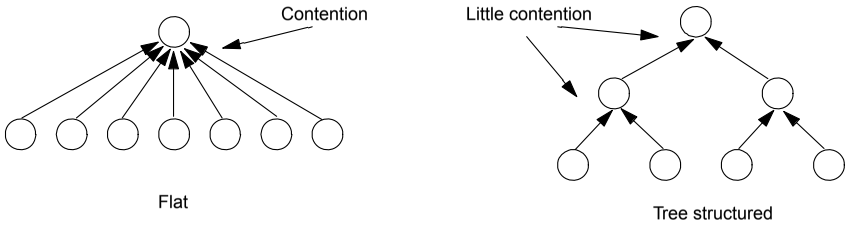
Centralized Barrier Performance

- Latency
  - Centralized has critical path length at least proportional to p
- Traffic
  - About $3p$ bus transactions
- Storage Cost
  - Very low: centralized counter and flag
- Fairness
  - Same processor should not always be last to exit barrier
  - No such bias in centralized
- Key problems for centralized barrier are latency and traffic
  - Especially with distributed memory, traffic goes to same node

Improved Barrier Algorithms for a Bus

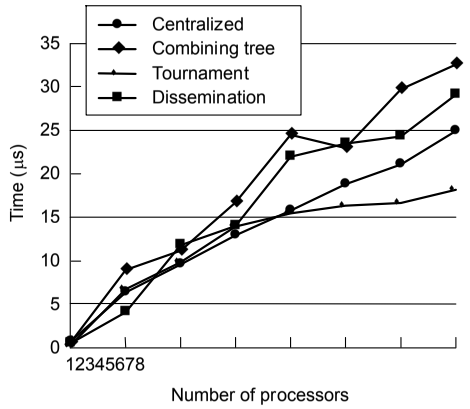
Software combining tree

- Only k processors access the same location, where k is degree of tree



- Separate arrival and exit trees, and use sense reversal
- Valuable in distributed network: communicate along different paths
- On bus, all traffic goes on same bus, and no less total traffic
- Higher latency (*log p* steps of work, and *O(p)* serialized bus xactions)
- Advantage on bus is use of ordinary reads/writes instead of locks

Barrier Performance on SGI Challenge



- Centralized does quite well
  - » Will discuss fancier barrier algorithms for distributed machines
- Helpful hardware support: piggybacking of reads misses on bus
  - » Also for spinning on highly contended locks

Lock-Free Synchronization

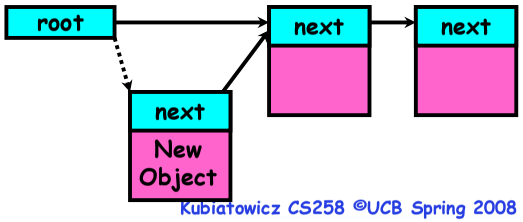
- What happens if process grabs lock, then goes to sleep???
  - Page fault
  - Processor scheduling
  - Etc
- Lock-free synchronization:
  - Operations do not require mutual exclusion of multiple insts
- Nonblocking:
  - Some process will complete in a finite amount of time even if other processors halt
- Wait-Free (Herlihy):
  - Every (nonfaulting) process will complete in a finite amount of time
- Systems based on LL&SC can implement these

Using of Compare&Swap for queues

```
• compare&swap (&address, reg1, reg2) { /* 68000 */
     if (reg1 == M[address]) {
        M[address] = reg2;
        return success;
     } else {
        return failure;
     }
  }
```

Here is an atomic add to linked-list function:

```
addToQueue(&object) {
    do {                    // repeat until no conflict
       ld r1, M[root]       // Get ptr to current head
       st r1, M[object]     // Save link in new object
    } until (compare&swap(&root,r1,object));
}
```



Synchronization Summary

- Rich interaction of hardware-software tradeoffs
- Must evaluate hardware primitives and software algorithms together
  - primitives determine which algorithms perform well
- Evaluation methodology is challenging
  - Use of delays, microbenchmarks
  - Should use both microbenchmarks and real workloads
- Simple software algorithms with common hardware primitives do well on bus
  - Will see more sophisticated techniques for distributed machines
  - Hardware support still subject of debate
- Theoretical research argues for swap or compare&swap, not fetch&op
  - Algorithms that ensure constant-time access, but complex