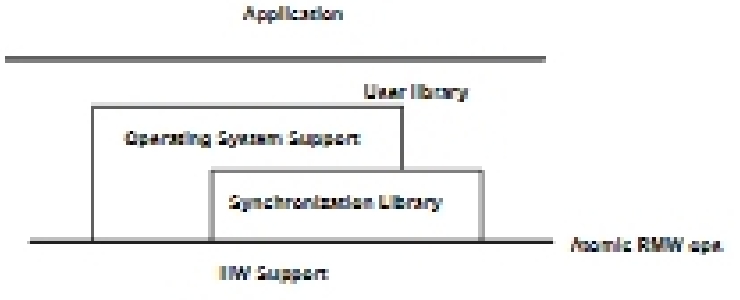
Hardware-Software Trade-offs in Synchronization

CS 252, Spring 05
David E. Culler
Computer Science Division
U.C. Berkeley

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?

Layers of synch support



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
- 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)
- Bus support for interrupt dispatch

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?

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

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 */
```

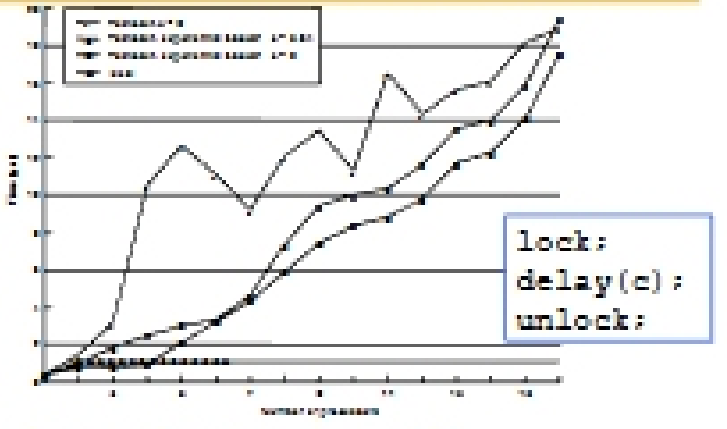
- Other read-modify-write primitives
  - Swap, Exch
  - Fetch&op
  - Compare&swap
    - Three operands: location, register to compare with, register to swap with
    - Not commonly supported by RISC instruction sets
- cacheable or uncacheable

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

Under what conditions do you measure synchronization performance? Contention? Scale? Duration?

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 attempt will succeed; others will fail and start testing again

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