Hardware-Software Trade-offs in Synchronization and Data Layout

CS 258, Spring 99  
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?

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 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
  - 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

```plaintext
lock: ld register, location  /* copy location to register */
     cmp location, #0  /* compare with 0 */
     bne 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

- 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 acquire
Simple Lock with LL-SC

```c
lock:  ll  reg1, location  /* LL location to reg1 */
      sc  location, reg2  /* SC reg2 into location*/
beqz  reg2, lock
      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

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 lval when SC succeeds, and fail?
- Backoff?
- Wouldn't it be nice to poll different locations ...

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?

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(p) 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;}
struct 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 (mycount == p)
    UNLOCK(bar_name.lock);
  bar_name.flag = local_sense;
  /* release waiters */
  else
    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 (2)j 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
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

Bag of Tricks for Spatial Locality

• Assign tasks to reduce spatial interleaving of accesses from procs
  – Contiguous rather than interleaved assignment of array elements
• Structure data to reduce spatial interleaving of accesses
  – Higher-dimensional arrays to keep partitions contiguous
  – Reduce false sharing and fragmentation as well as conflict misses

Conflict Misses in a 2-D Array Grid

• Assign tasks to reduce spatial interleaving of accesses from procs
  – Contiguous rather than interleaved assignment of array elements
• Structure data to reduce spatial interleaving of accesses
  – Higher-dimensional arrays to keep partitions contiguous
  – Reduce false sharing and fragmentation as well as conflict misses

Bag of Tricks (contd.)

• Beware conflict misses more generally
  – Allocate non-power-of-2 even if application needs power-of-2
  – Conflict misses across data structures: ad-hoc padding/alignment
  – Conflict misses on small, seemingly harmless data
• Use per-processor heaps for dynamic memory allocation
• Copy data to increase locality
  – If noncontiguous data are to be reused
  – Must trade off against cost of copying
• Pad and align arrays: can have false sharing v. fragmentation tradeoff
• Organize arrays of records for spatial locality
  – E.g. particles with fields: organize by particle or by field
  – In vector programs by field for unit-stride, in parallel often by particle
  – Phases of program may have different access patterns and needs

Implications for Software

• Processor caches do well with temporal locality
• Synchron. algorithms reduce inherent communication
• Large cache lines (spatial locality) less effective

Bag of Tricks (contd.)

• Beware conflict misses more generally
  – Allocate non-power-of-2 even if application needs power-of-2
  – Conflict misses across data structures: ad-hoc padding/alignment
  – Conflict misses on small, seemingly harmless data
• Use per-processor heaps for dynamic memory allocation
• Copy data to increase locality
  – If noncontiguous data are to be reused
  – Must trade off against cost of copying
• Pad and align arrays: can have false sharing v. fragmentation tradeoff
• Organize arrays of records for spatial locality
  – E.g. particles with fields: organize by particle or by field
  – In vector programs by field for unit-stride, in parallel often by particle
  – Phases of program may have different access patterns and needs