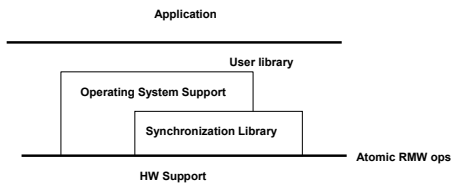
Hardware-Software Trade-offs in Synchronization

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 */
```

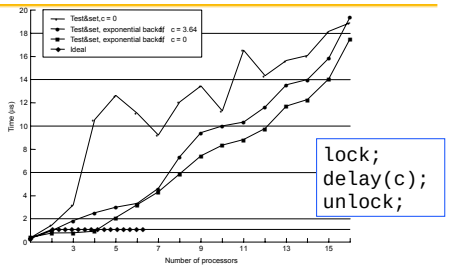
- Other read-modify-write primitives
  - Swap, Exch
  - Fetch&op
  - Compare&swap
    - »Three operands: location, register to compare with, register to swap with
    - »Not commonly supported by RISC instruction sets
- cacheable or uncacheable

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 attemptor 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
```

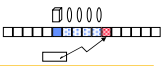
- Can do more fancy atomic ops by changing what's between LL & SC
  - But keep it small so SC likely to succeed
  - Don't include instructions that would need to be undone (e.g. stores)
- SC can fail (without putting transaction on bus) if:
  - Detects intervening write even before trying to get bus
  - 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

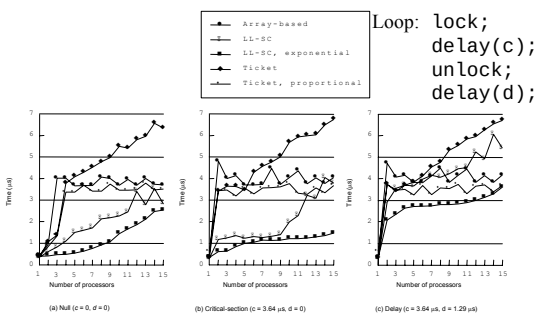


- Only one r-m-w per acquire
- Two counters per lock (next_ticket, now_serving)
  - Acquire: fetch&inc next_ticket;
    wait for now_serving == next_ticket
    - » atomic op when arrive at lock, not when it's free (so less contention)
  - Release: increment now-serving
- Performance
  - low latency for 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



Fairness

- Unfair locks look good in contention tests because same processor reacquires lock without miss.
- Fair locks take a miss between each pair of acquires

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: flexiblity
    - » 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) {}; }
}
```

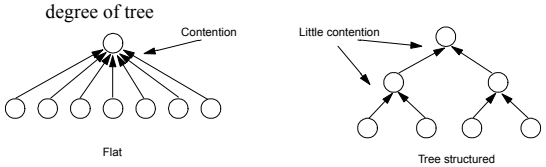

Centralized Barrier Performance

- Latency
  - Centralized has critical path length at least proportional to *p*
- Traffic
  - About 3*p* 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

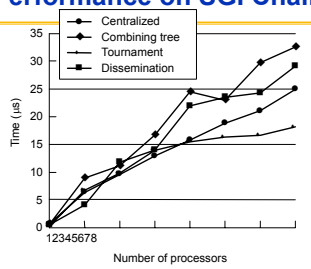
Software combining tree

- 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

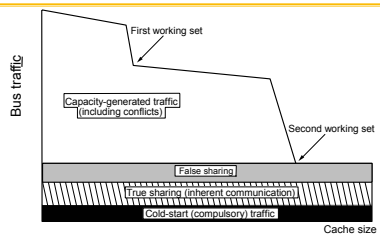


- Centralized does quite well
  - » fancier barrier algorithms for distributed machines
- Helpful hardware support: piggybacking of reads misses on bus
  - » Also for spinning on highly contended locks

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

Implications for Software



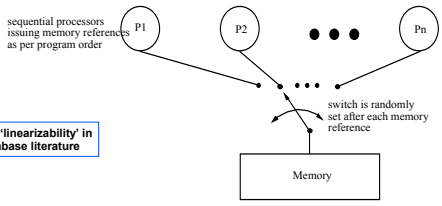
- Processor caches do well with temporal locality
- Synch. algorithms reduce inherent communication
- Large cache lines (spatial locality) less effective

Memory Consistency Model

- for a SAS specifies constraints on the order in which memory operations (to the same or different locations) can appear to execute with respect to one another,
- enabling programmers to reason about the behavior and correctness of their programs.
- fewer possible reorderings => more intuitive
- more possible reorderings => allows for more performance optimization
  - 'fast but wrong' ?

Multiprogrammed Uniprocessor Mem. Model

- A MP system 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, and the operations of each individual processor appear in this sequence in the order specified by its program (Lamport)



like 'linearizability' in database literature

Reasoning with Sequential Consistency

initial: A, flag, x, y == 0

p1	p2
(a) A := 1;	(c) x := flag;
(b) flag := 1;	(d) y := A

- program order: (a) → (b) and (c) → (d) "preceeds"
- claim: (x,y) == (1,0) cannot occur
  - x == 1 => (b) → (c)
  - y == 0 => (d) → (a)
  - thus, (a) → (b) → (c) → (d) → (a)
  - so (a) → (a)

Then again, . . .

initial: A, flag, x, y == 0

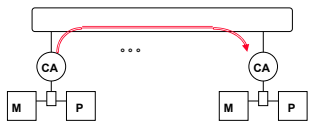
p1	p2
(a) A := 1;	(c) x := flag;
B := 3.1415	
C := 2.78	
(b) flag := 1;	(d) y := A+B+C

- Many variables are not used to effect the flow of control, but only to shared data
  - synchronizing variables
  - non-synchronizing variables

Requirements for SC (Dubois & Scheurich)

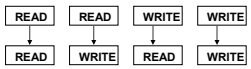
- Each processor issues memory requests in the order specified by the program.
- After a store operation is issued, the issuing processor should wait for the store to complete before issuing its next operation.
- After a load operation is issued, the issuing processor should wait for the load to complete, and for the store whose value is being returned by the load to complete, before issuing its next operation.
- the last point ensures that stores appear atomic to loads
  - note, in an invalidation-based protocol, if a processor has a copy of a block in the dirty state, then a store to the block can complete immediately, since no other processor could access an older value

Architecture Implications



- need write completion for atomicity and access ordering
  - w/o caches, ack writes
  - w/ caches, ack all invalidates
- atomicity
  - delay access to new value till all inv. are acked
- access ordering
  - delay each access till previous completes

Summary of Sequential Consistency



- Maintain order between shared access in each thread
  - reads or writes wait for previous reads or writes to complete

Do we really need SC?

- Programmer needs a model to reason with
  - not a different model for each machine

=> Define "correct" as same results as sequential consistency

- Many programs execute correctly even without "strong" ordering

initial: A, flag, x, y == 0

p1	p2
A := 1;	
B := 3.1415	
unlock (L)	lock (L)
	... = A;
	... = B;

- explicit synch operations order key accesses

Does SC eliminate synchronization?

- No, still need critical sections, barriers, events
  - insert element into a doubly-linked list
  - generation of independent portions of an array
- only ensures interleaving semantics of individual memory operations

Is SC hardware enough?

- No, Compiler can violate ordering constraints
  - Register allocation to eliminate memory accesses
  - Common subexpression elimination
  - Instruction reordering
  - Software Pipelining

P1	P2	P1	P2
B=0	A=0	r1=0	r2=0
A=1	B=1	A=1	B=1
u=B	v=A	u=r1	v=r2
		B=r1	A=r2
(u,v)=(0,0) disallowed under SC		may occur here	

- Unfortunately, programming languages and compilers are largely oblivious to memory consistency models
  - languages that take a clear stand, such as HPF too restrictive

What orderings are essential?

initial: A, flag, x, y == 0

```
p1                  p2
 A    := 1;
 B    := 3.1415
 unlock (L)         lock (L)
                    ... = A;
                    ... = B;
```

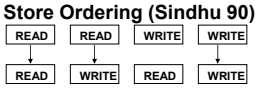
- Stores to A and B must complete before unlock
- Loads to A and B must be performed after lock

How do we exploit this?

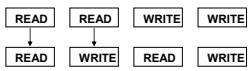
- Difficult to automatically determine orders that are not necessary
- Relaxed Models:
  - hardware centric: specify orders maintained (or not) by hardware
  - software centric: specify methodology for writing "safe" programs
- All reasonable consistency models retain program order as seen from each processor
  - i.e., dependence order
  - purely sequential code should not break!

Hardware Centric Models

- Processor Consistency (Goodman 89)
- Total Store Ordering (Sindhu 90)



- Partial Store Ordering (Sindhu 90)



- Causal Memory (Hutto 90)
- Weak Ordering (Dubois 86)

Properly Synchronized Programs

- All synchronization operations explicitly identified
- All data accesses ordered though synchronizations
  - no data races!

=> Compiler generated programs from structured high-level parallel languages

=> Structured programming in explicit thread code

Complete Relaxed Consistency Model

- System specification
  - what program orders among mem operations are preserved
  - what mechanisms are provided to enforce order explicitly, when desired
- Programmer's interface
  - what program annotations are available
  - what 'rules' must be followed to maintain the illusion of SC
- Translation mechanism

Relaxing write-to-read (PC, TSO)

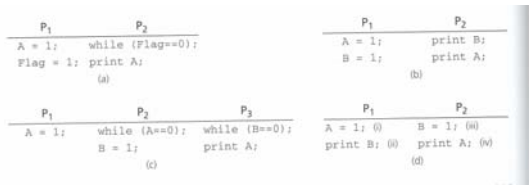
- Why?
  - write-miss in write buffer, later reads hit, maybe even bypass write
- Many common idioms still work

initial: A, flag, x, y == 0

p1	p2
(a) A := 1;	(c) while (flag ==0) {}
(b) flag := 1;	(d) y := A

  - write to flag not visible till previous writes visible
- Ex: Sequent Balance, Encore Multimax, vax 8800, SparcCenter, SGI Challenge, Pentium-Pro

Detecting weakness wrt SC



- Different results
  - a, b: same for SC, TSO, PC
  - c: PC allows A=0 --- no write atomicity
  - d: TSO and PC allow A=B=0
- Mechanism
  - Sparc V9 provides MEMBAR

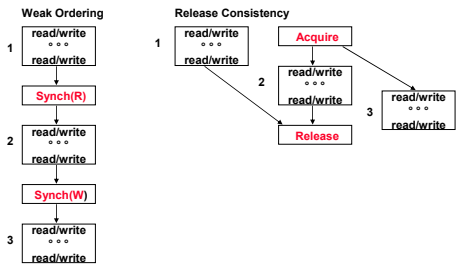
Relaxing write-to-read and write-to-write (PSO)

- Why?
  - write-buffer merging
  - multiple overlapping writes
  - retire out of completion order
- But, even simple use of flags breaks
- Sparc V9 allows write-write membar
- Sparc V8 stbar

Relaxing all orders

- Retain control and data dependences within each thread
- Why?
  - allow multiple, overlapping read operations
  - it is what most sequential compilers give you on multithreaded code!
- Weak ordering
  - synchronization operations wait for all previous mem ops to complete
  - arbitrary completion ordering between
- Release Consistency
  - acquire: read operation to gain access to set of operations or variables
  - release: write operation to grant access to others
  - acquire must occur before following accesses
  - release must wait for preceding accesses to complete

Preserved Orderings



Examples

```
P1, P2, ..., Pn
. . .
Lock(TaskQ)
newTask->next = Head;
if (Head != NULL)
  Head->prev = newTask;
Head = newTask;
UnLock(TaskQ)
. . .
```

```
P1                          P2
TOP: while(flag2==0);       TOP: while(flag1==0);
     A = 1;                       x = A;
     u = B;                       y = D;
     v = C;                       B = 3;
     D = B * C;                   C = D / B;
     flag2 = 0;                   flag1 = 0;
     flag1 = 1;                   flag2 = 1;
     goto TOP;                    goto TOP;
```

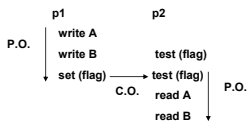

Programmer's Interface

- weak ordering allows programmer to reason in terms of SC, as long as programs are 'data race free'
- release consistency allows programmer to reason in terms of SC for "properly labeled programs"
  - lock is acquire
  - unlock is release
  - barrier is both
  - ok if no synchronization conveyed through ordinary variables

Identifying Synch events

- two memory operation in different threads conflict is they access same location and one is write
- two conflicting operations compete if one may follow the other in a SC execution with no intervening memory operations on shared data
- a parallel program is synchronized if all competing memory operations have been labeled as synchronization operations
  - perhaps differentiated into acquire and release
- allows programmer to reason in terms of SC, rather than underlying potential reorderings

Example



- Accesses to flag are competing
  - they constitute a Data Race
  - two conflicting accesses in different threads not ordered by intervening accesses
- Accesses to A (or B) conflict, but do not compete
  - as long as accesses to flag are labeled as synchronizing

How should programs be labeled?

- Data parallel statements ala HPF
- Library routines
- Variable attributes
- Operators

Summary of Programmer Model

- Contract between programmer and system:
  - programmer provides synchronized programs
  - system provides effective "sequential consistency" with more room for optimization
- Allows portability over a range of implementations
- Research on similar frameworks:
  - Properly-labeled (PL) programs - Gharachorloo 90
  - Data-race-free (DRF) - Adve 90
  - Unifying framework (PLpc) - Gharachorloo,Adve 92

Interplay of Micro and multi processor design

- Multiprocessors tend to inherit consistency model from their microprocessor
  - MIPS R10000 -> SGI Origin: SC
  - PPro -> NUMA-Q: PC
  - Sparc: TSO, PSO, RMO
- Can weaken model or strengthen it
- As micros get better at speculation and reordering it is easier to provide SC without as severe performance penalties
  - speculative execution
  - speculative loads
  - write-completion (precise interrupts)

Questions

- What about larger units of coherence?
  - page-based shared virtual memory
- What happens as latency increases? BW?
- What happens as processors become more sophisticated? Multiple processors on a chip?
- What path should programming languages follow?
  - Java has threads, what's the consistency model?
- How is SC different from transactions?