Synchronization in Shared Memory Computers

Outline

1. Review basic issues in concurrency
2. Basic locks: SW, atomic memory ops
3. A better HW mechanism: LL/SC
4. Other types of locks
5. Barriers

History and Perspectives

- Much debate over hardware primitives over the years
- Conclusions depend on technology and machine style
  - speed vs. flexibility
- Most modern methods use a form of atomic read-modify-write
  - IBM 370: included atomic compare&swap for multiprogramming
  - x86: any instruction can be prefixed with a lock modifier
    - XCHG is commonly used. Default is LOCKED. Can have two memory operands.
  - SPARC: atomic register-memory ops (swap, compare&swap)
  - MIPS, IBM Power: no atomic operations but pair of instructions
    - **load-locked (a.k.a. load-linked), store-conditional**
    - later used by PowerPC and DEC Alpha too
- Rich set of tradeoffs
  - High-level language advocates want hardware locks/barriers
    - but it's goes against the "RISC" flow, and has other problems

Synchronization Basics

Components of a Sync Event

(recall from A&S)

- **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
  - Generally ***busy-wait*** or ***block***
- **Release** method
  - Enable other processors to acquire right to the synch

Role of System and User

- **User** wants to use high-level synchronization operations
  - Locks, barriers...
  - Doesn't care about implementation
- **System** designer: how much hardware support in implementation?
  - Speed versus cost and flexibility
  - Waiting algorithm difficult in hardware, so provide support for others
- Popular trend:
  - System provides simple hardware primitives (atomic operations)
  - Software libraries implement lock, barrier algorithms using these
  - But some propose and implement full-hardware synchronization

Challenges

- **Same synchronization may have different needs at different times**
  - Lock accessed with low or high contention
  - Different performance requirements: low latency or high throughput

Different algorithms best for each case, and need different primitives

- **Rich area of software-hardware interactions**
  - Which primitives available affects what algorithms can be used
  - Which algorithms are effective affects what primitives to provide
- **Need to evaluate using workloads**

First Attempt at Simple Software Lock

```
lock:    ld     register, location  /* copy location to register */
         cmp    register, #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 */
```

Problem:

Violates "Mutual Exclusion"

lock needs atomicity in its own implementation

– Read (test) and write (set) of lock variable by a process not atomic

– Recall SW-only solution … but preferred has HW support.

Solution:

atomic ***read-modify-write*** or ***exchange*** instructions

- atomically test value of location and set it to another value, return success or failure somehow

Atomic Exchange Instruction: test&set

- Specifies a location and register. In 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

Can be used to build locks

Simple Test&Set Lock

```
lock:  t&s    register, location
       bnz    register, 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 can be used too
  - Swap, Fetch&op, Compare&swap – for C&S:
    - Three operands: location, register to compare with, register to swap with
    - Not commonly supported by RISC instruction sets
- Can be cacheable or uncacheable (we assume cacheable)

- **Problem:**
  - spinning on **t&s *can*** cause large numbers of writes. Two cases:

    One process spinning: lock is in M state (exclusive) so OK

    Multiple processes spinning: continuous bus traffic from **BusUpgrd**s

Test&Test&Set Lock

Enhancement to simple lock algorithm …

- Busy-wait with **read (test)** operations only 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 attempter will succeed; others will fail and start testing again

```
Label:  ld   R1,lock          /* is lock free? */
        bnz  R1,label         /* if not zero, then try again */
        t&s  register,lock    /* it's free, so grab! */
        bnz  label            /* if not 0, then try again */
        ret                   /* return control to caller */
```

- Let N processes contend for a lock being held by another process
- Spinning solves part of problem: Lock is cached locally for ALL processors spinning on lock.

Test&Test&Set Lock

```
Label: ld    R1,lock          /* is lock free? */
       bnz   R1,label         /* if not zero, then try again */
       t&s   register,lock    /* it's free, so grab! */
       bnz   label            /* if not 0, then try again */
       ret                    /* return control to caller */
```

- Let N processes contend for a lock being held by another process
- Spinning solves part of problem: Lock is cached locally for ALL processors spinning on lock.

- BUT, what happens when lock is released? Holder sets Lock to 0
  - All (N) spinning processes have Lock invalidated simultaneously (V= FALSE)
    - All processes do ld and since V = FALSE for all, all ld's result in bus transactions
    - All processes' ld's return 0
  - All (N) spinning processes now fall through bnz and do test&set
    - One process gets there first, does test&set (invalidate), enters critical section. Now Lock = 0 and V = FALSE for Lock (would require bus x-action anyway)
    - Other N-1 processes do test&set (invalidate), do not get lock and loop back to label
  - All processes but winner (N-1) now do ld on LockLocation with V = FALSE
    - N-1 bus transactions to do new ld to get copy of Lock = 1 into cache with V = TRUE

- Result: 4N – 1 bus transactions for each release.
- For N releases ~2N**2 bus transactions! N=20 and 50 cycles = 40,000 cycles!

Test&Set Lock with Exponential Back-off

Another enhancement to simple lock:

- Reduce frequency of issuing test&sets while waiting
  - *Test&set lock* ***with back-off***
  - Don't back off too much or will be backed off when lock becomes free
  - Exponential back-off works quite well empirically: i^{th} time = $k*c^i$

```
lock:  li     r3, 1          /* r3 has delay. Initially 1 */
test:  t&s    r1, location   /* try to get lock */
       bez    done           /* if 0, then done */
       sll    r3             /* double pause time */
       pause  r3             /* delay by value in r3 */
       j      test           /* try to get lock again */
done:  ret                   /* return control to caller */

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

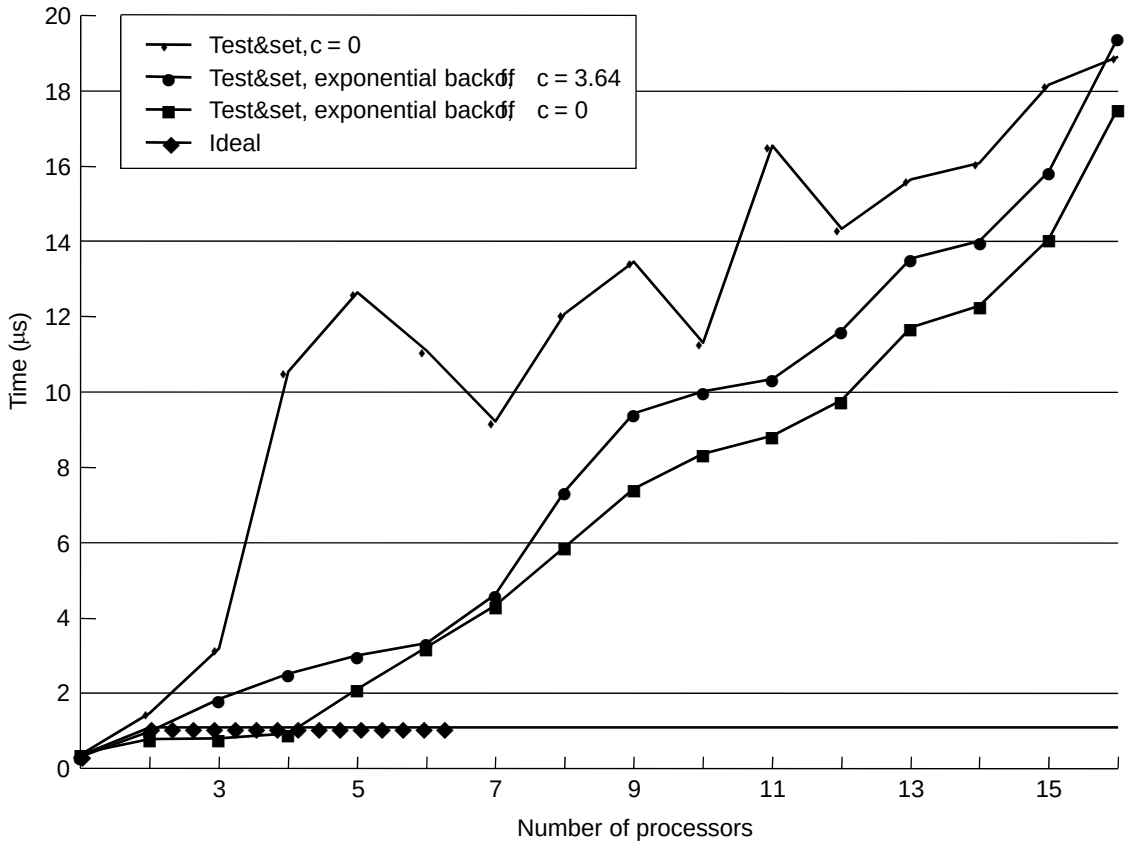
Pause should start with time ~= the expected length of the critical section

T&S Lock Micro-benchmark Performance

- On SGI Challenge. Code:

  lock;
  critical section with delay (c);
  unlock;
- Question: does delay(c) cause bus transactions? Probably not.
- Same total number of lock calls as number of processes increases
- Time spent in delay(c) is subtracted out for comparison
- SGI Challenge does not have atomic test&set. New code:
  - Every time a SC fails, a write is performed to another variable in the same cache block causing the invalidations that would have been caused by a test&set.
- SGI Challenge parameters:
  - Cache block = 128 bytes
  - Bus width = 256 bits = 32 bytes
  - Bus is “split transaction,” i.e. separate read requests and read responses
  - Data returns from memory require 5 bus cycles, 1 overhead and 4 data
  - Bus frequency is 48MHz, i.e. 20 ns cycle time
  - Memory access takes 2-3 cycles, but begins before Read completes
  - Read transaction takes 5 cycles
  - Read takes ~250ns
- Complications: split data/address buses, write buffers, “piggy-backing,” etc.

T&S Lock Micro-benchmark Performance



- Recall time per read is 250ns
- Performance degrades because unsuccessful test&sets generate traffic
- Does each lock take 40,000 cycles? (40K*20ns = 800us)!

Performance Criteria (T&S Lock)

- Uncontended Latency
  - Very low if repeatedly accessed by same processor; independent. of *p*
- Traffic
  - Lots if many processors compete; poor scaling with *p*
  - Each t&s generates invalidations, and all rush out again to t&s
- Storage
  - Very small (single variable); independent of *p*
- Fairness
  - Poor, can cause starvation
- Test&set with back-off: similar, but less traffic
- Test-and-test&set: slightly higher latency, much less traffic
- But still all rush out to read miss and test&set on release
  - Traffic for *p* processors to access once each: $O(p^2)$
- Luckily, better hardware primitives as well as algorithms exist
- Another problem with atomic Test&Set: modern memory buses have complex split transaction protocols and memories have long latencies w.r.t. CPUs.

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
  - Don't hog the bus
- *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 if and only if no one else has written to the variable since this processor's LL
  - If SC succeeds, means all three steps happened atomically
  - If fails, doesn't write or generate invalidations (need to retry LL)
  - Success indicated by condition codes; implementation later

- Load Linked looks just like Load:

  EX: ll r2, 234(r7)
- Store Conditional returns a value in a source register

  EX: sc r3, 234(r7) /* r3 ← 0 for failure and r3 ← 1 for success */
- Failure ➔ If, since the ll has read address X:

  IF (An interrupt has occurred) **OR** (X has been written)

  THEN

  (SC does not complete the write [no bus transaction!]) **AND**

  Returns a 0 to the source register

  ELSE

  (SC completes the write [bus transaction]) **AND**

  Returns a 1 to the source register

LL-SC Hardware Implementation

- Special register inside processor called the LINK REGISTER
- LINK REGISTER keeps track of the last address used by ll. If
  - Interrupt, or
  - Invalidate of contents of location at address in LINK REGISTER
  - Then LINK REGISTER ← 0
- SC checks address against Link Register, which also guarantees that no other ll's have been done in between

Atomic Swap Using LL-SC

Exchange `r4` with `0(r1)` – start with r4 with key and 0(r1) with lock

```
exch: mov   r3,r4       ; r3 ← r4
      ll    r2,0(r1)    ; r2 ← mem
      sc    r3,0(r1)    ; mem ← r3
      beqz  r3,exch     ; loop on failure
      mov   r4,r2       ; finish swap with r4 ← r2
                        ; r4 has lock and 0(r1) has key
```

Note: don't let swap complete until you know it was atomic.

- *That is, that it occurred at all! SC fails altogether if an intervening write has occurred.*

Atomic Fetch and Increment Using LL-SC

```
try:  ll    r2,0(r1)  ; r2 ← mem
      addi  r2,r2,1   ; r2 += 1
      sc    r2,0(r1)  ; mem ← r3
                      ; note:  no store (0) if failure
      beqz  r2,try    ; loop on failure
```

Note: ops between ll and sc:

- Should be few
- Can only involve registers and immediates or risk deadlock

Try Another Spin Lock with EXCH Operation

```
test: ld    r2,0(r1)    ; get copy of lock
      bnez  r2,test     ; if still locked, spin on
                        ;  cached copy
      li    r2,1        ; set key to 1
      exch  r2,0(r1)    ; swap lock with key
      bnez  r2, test    ; if too late and failed,
                        ;  go back to test
```

Almost the same as previous, just using exch instead of test&set

A Spin Lock With inlined EXCH operation

Now, substitute **exch** function (inline) for HW primitive

```
test:  ld     r2,0(r1)   ; get copy of lock
       bnez   r2,test    ; if still locked, spin on cached copy
       li     r2,1       ; set key to 1
                         ; exchange r2 and 0(r1)
exch:  mov    r2,r4      ; r2 → r4 to make copy
       ll     r3,0(r1)   ; mem → r3 to get lock
       sc     r4,0(r1)   ; r2 → mem to store key
       beqz   r2,exch    ; loop on failure to store key
       mov    r3,r2      ; finish swap with r3 → r2
                         ; r2 has lock and 0(r1) has key
       bnez   r2, test   ; if too late & failed, go back to test
       ret
```

This works but has substantial redundancies, see next for simpler code

Simple Lock with LL-SC

```
start: ll     r1,lock       /* LL location to reg1 */
       bnez   r1,start      /* spin on cached lock */
       li     r2,1
       sc     r2,lock       /* SC reg2 into location*/
       beqz   r2,start      /* 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

Analyze Simple Lock with LL-SC

```
start:  ll     r1, lock        /* LL location to reg1 */
        bnez   r1, start       /* still spinning on cached 1 */
        li     r2, 1
        sc     r2, lock        /* SC reg2 into location*/
        beqz   r2, start       /* if failed, start again */
        ret
```

- What happens NOW when lock is released? Holder sets Lock to 0
  - All (N) spinning processes have Lock invalidated simultaneously (V= FALSE)
    - All processes do ll and since V = FALSE for all, all N ll's result in bus transactions
    - All processes' ll's return 0
  - All (N) processes now attempt sc
    - One process gets there first and succeeds and enters critical section
    - Result is 1 bus transaction for store
    - The successful sc invalidates Lock in all the other processor caches by setting it to 1
    - All other processes' sc's fail and do not cause a bus transaction
  - All processes but winner (N-1) now do ll on Lock with V = FALSE
    - N-1 bus transactions to do new ll to get copy of Lock = 1 into cache with V = TRUE

- Performance:
  - 2N+2 bus transactions when N processes are waiting
  - Total of ~N**2 bus transactions for all N processes to get lock

More Efficient SW Locking Algorithms

- Problem with Simple LL-SC lock
  - No INVALIDATEs on failure to get lock, but read misses by all waiters after both release and successful SC by winner
  - No test-and-test&set analog, but can use back-off to reduce burstyness
  - Doesn't reduce traffic to minimum, and not a fair lock

- Better SW algorithms for bus (for r-m-w instructions or LL-SC)
  1. Only one process to try to get lock upon release
     - valuable when using test&set instructions; LL-SC does it already
  2. Only one process to have read miss upon release
     - valuable with LL-SC too

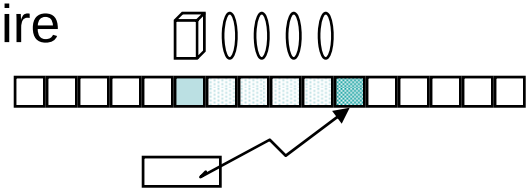
Ticket lock achieves first

Array-based queuing lock achieves both

Both are fair (FIFO) locks as well

Ticket Lock

- Only one r-m-w (from only one processor) per acquire
- Works like waiting line at deli or bank
  - Two counters per lock (*next_ticket*, *now_serving*)
  - Acquire: fetch&inc *next_ticket;* wait for *now_serving* to equal it
    - atomic op when arrive at lock, not when it's free (so less contention)
  - Release: increment *now-serving*
  - FIFO order, low latency for low-contention if fetch&inc cacheable
  - Still *O(p)* read misses at release, since all spin on same variable
    - like simple LL-SC lock, but no inval when SC succeeds, and fair
  - Can be difficult to find a good amount to delay on backoff
    - exponential backoff not a good idea due to FIFO order
    - backoff proportional to *now-serving* - *next-ticket* may work well



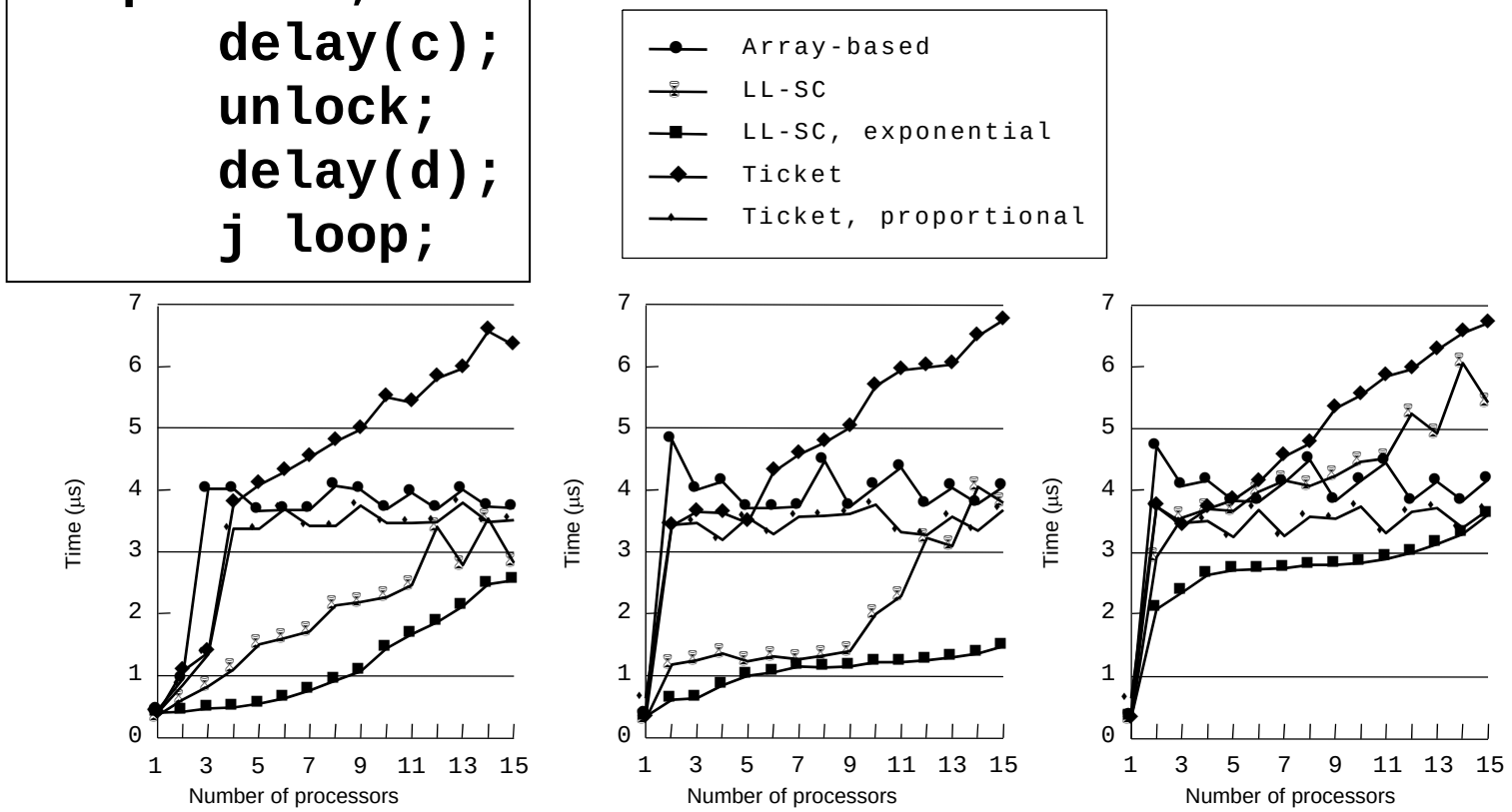
Wouldn't it be nice to poll different locations ...

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
  - Good performance for bus-based machines
  - 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

```
Loop: lock;
      delay(c);
      unlock;
      delay(d);
      j loop;
```



(a) Null (*c* = 0, *d* = 0)

(b) Critical-section (c = 3.64 µs, d = 0)

(c) Delay (c = 3.64 µs, d = 1.29 µs)

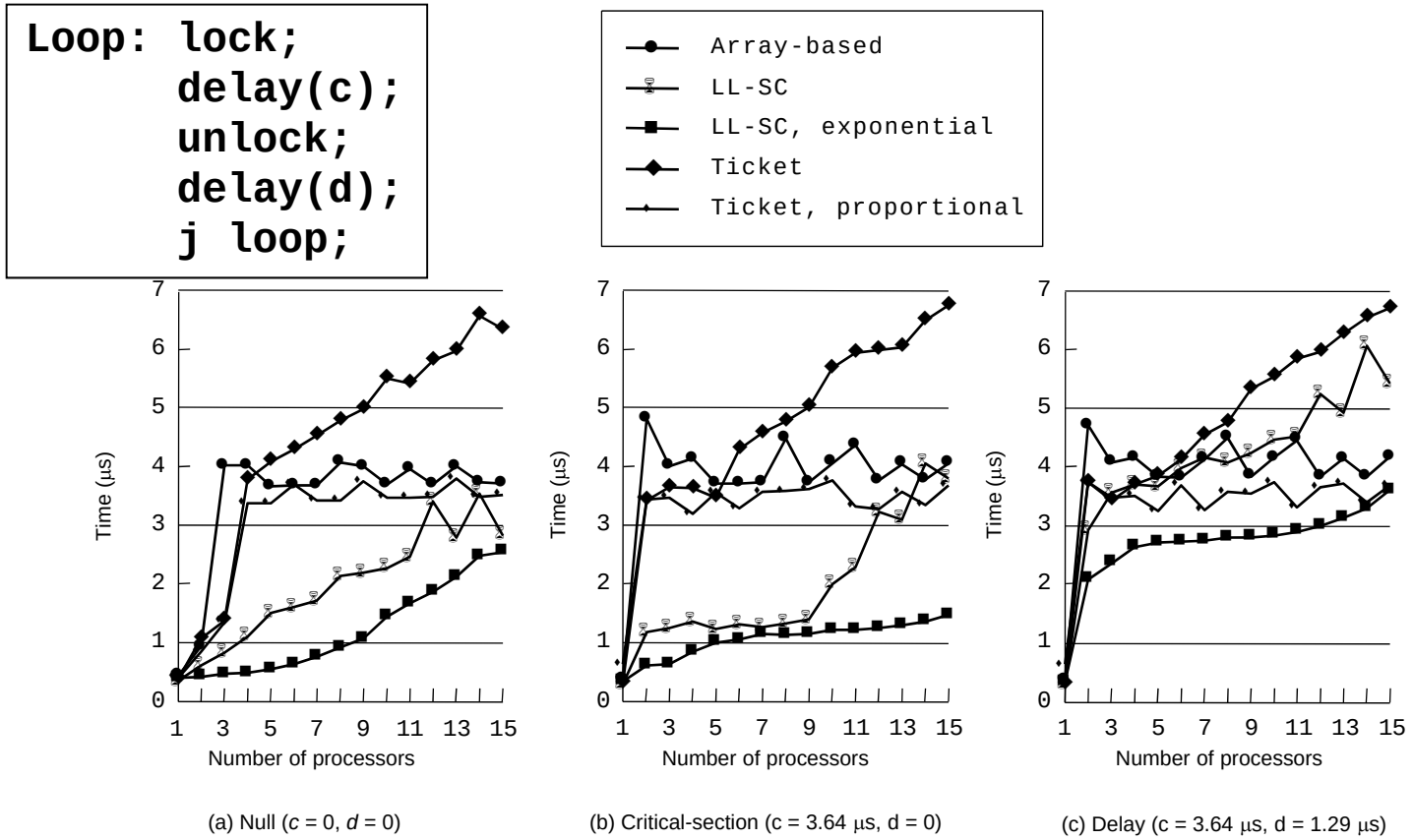
– Compare with atomic test&set: all do much better for P > 8

– Simple LL-SC lock does best! Why? Look at transactions between release and acquire

- Processor releases lock and immediately LL's and SC's. These are likely to be cache hits
- In fact LL is likely to get data from the write buffer
- Unfairness is exploited by architecture to improve performance

Lock Performance on SGI Challenge

```
Loop: lock;
      delay(c);
      unlock;
      delay(d);
      j loop;
```



(a) Null (*c* = 0, *d* = 0)

(b) Critical-section (c = 3.64 µs, d = 0)

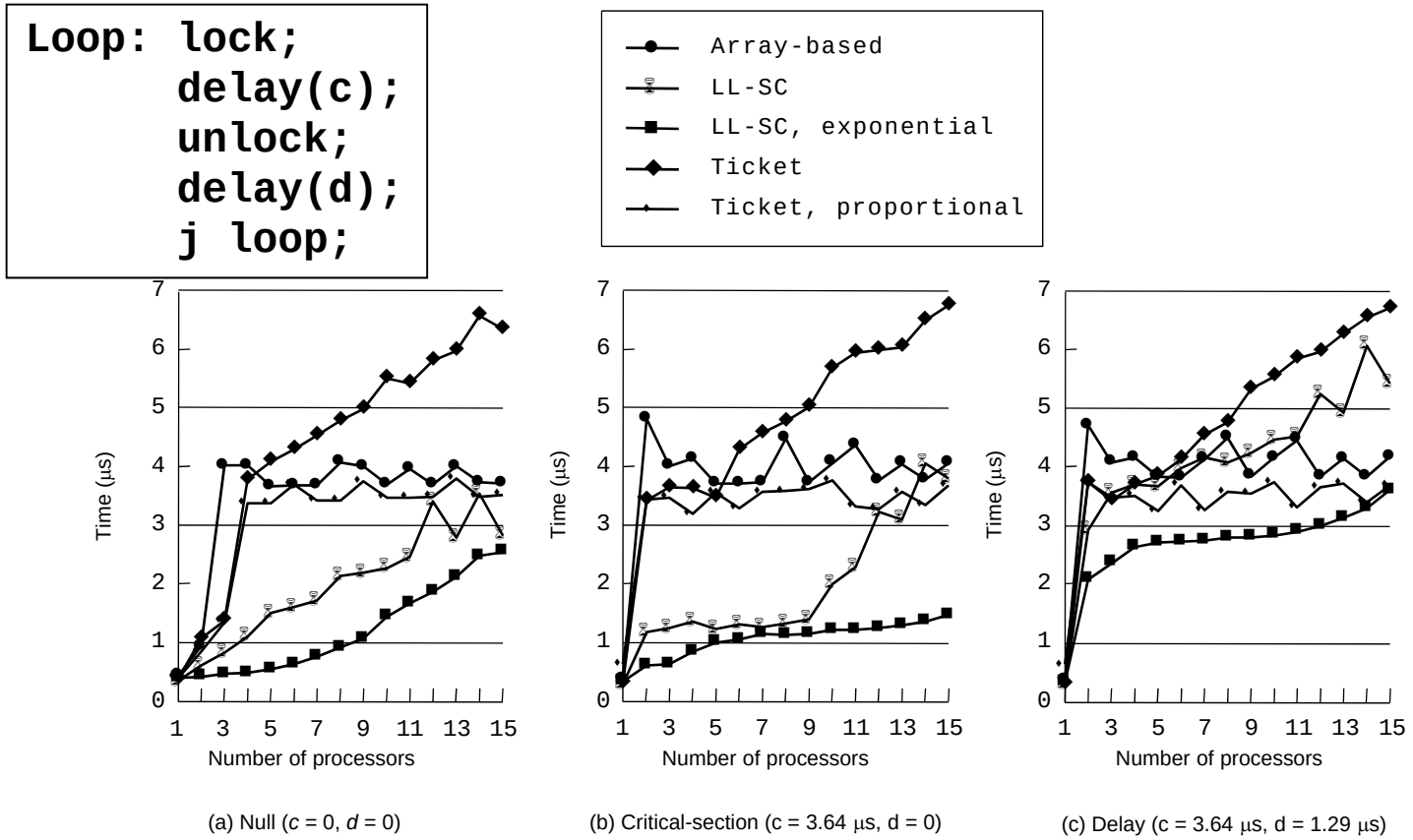
(c) Delay (c = 3.64 µs, d = 1.29 µs)

– Simple LL-SC lock does best at small p due to unfairness
  - However, even with no back-off, increased traffic reduces likelihood of self-transfer
  - With delay between unlock and next lock, likelihood of self-transfer decreases further
  - Need to be careful with back-off: can be ALL waiting (see (c), p = 2) in back-off

– Ticket and Array Locks always require a bus transaction to transfer lock. For P > 1, 3 transactions per critical section are required (need to set number, transfer it, etc.)

Lock Performance on SGI Challenge

```
Loop: lock;
      delay(c);
      unlock;
      delay(d);
      j loop;
```



(a) Null (*c* = 0, *d* = 0)

(b) Critical-section (c = 3.64 µs, d = 0)

(c) Delay (c = 3.64 µs, d = 1.29 µs)

- Ordinary Ticket lock scales poorly: all processors try to read the "now-serving counter" and expected number of transactions between release and read is P/2. Linear degradation of critical path
- Ticket lock with proportional back-off scales well. Note back-off is ~expected delay
- Array lock also does well: only one read per acquire
- However, LL-SC with exponential back-off still does the best!
- Methodologically challenging, and need to look at real workloads

Barrier Synchronization

- Forces all processes to wait until all processes have reached barrier. Then releases all processes.

- Example
  - All processors do a 1D FFT on their rows, then wait at barrier
  - All processors participate in matrix transpose
  - All processors do a 1D FFT on their columns

  Barrier prevents transpose of incomplete data

- Barrier Implementation: 2 locks and 1 counter
  - 1 lock holds processes
  - Counter sees whether everybody is there
  - What's the 2nd lock for???

A Centralized Barrier -- 1st attempt

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

```
struct bar_type {int counter = 0;        /* keep track of arrivals */
                 struct lock_type lock;  /* internal lock */
                 int flag = 0;} bar_name;  /* flag holds threads */

  BARRIER (bar_name, p) {          /* p is # of threads participating */
  LOCK(bar_name.lock);              /* lock to manipulate common vars */
  if (bar_name.counter == 0)        /* reset arrivals if first to reach */
       bar_name.flag = 0;           /* reset flag if first to reach*/
  mycount = ++bar_name.counter;     /* my arrival place - private */
  UNLOCK(bar_name.lock);
  if (mycount == p) {               /* If I'm last to arrive */
       bar_name.counter = 0;        /* reset for next barrier */
       bar_name.flag = 1;           /* release waiters */
  }
  else while (bar_name.flag == 0) {};  /* Else busy wait for release */
}
```

Problem?

Complication with Simple Barrier

Scenario:

```
DO 1000 Times {
  Lottsastuff
  Barrier(L)
}
```

- Complication
  - Process P is spinning on release
  - Process P gets swapped out
  - As last process Z arrives at barrier, RELEASE FLAG is set to 1, COUNT to 0
  - Process Q executes *Lottsastuff*, re-enters barrier and reinitializes RELEASE FLAG to 0
  - Process P gets swapped in, continues spinning
  - Processes A,B,C, … all reach barrier
  - Count = Total -1
  - ????

- Problem when same barrier is used consecutively and processes are interruptible
  - Must prevent process from entering until all have left previous instance
  - Could use another counter, but increases latency and contention
  - Better solution is “sense reversing”

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 that would increase latency and contention
- Sense reversal: wait for flag to take different value consecutive times
  - Toggle this value only when all processes reach barrier

```
// initialize local_sense (local) and bar_name.flag (global) to FOO
// Note: !FOO = FOOBAR,  !FOOBAR = FOOBARBAR = FOO
struct bar_type {int counter = 0;        /* keep track of arrivals */
                 struct lock_type lock;  /* internal lock */
                 int flag = 0;} bar_name;  /* flag holds threads */
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)          /* last */
     UNLOCK(bar_name.lock);
     bar_name_counter = 0;            /* reset for next barrier */
     bar_name.flag = local_sense;     /* release waiters*/
  else {                             /* NOT last */
    UNLOCK(bar_name.lock);
    while (bar_name.flag != local_sense) {};
  } } /* end else & BARRIER */
```

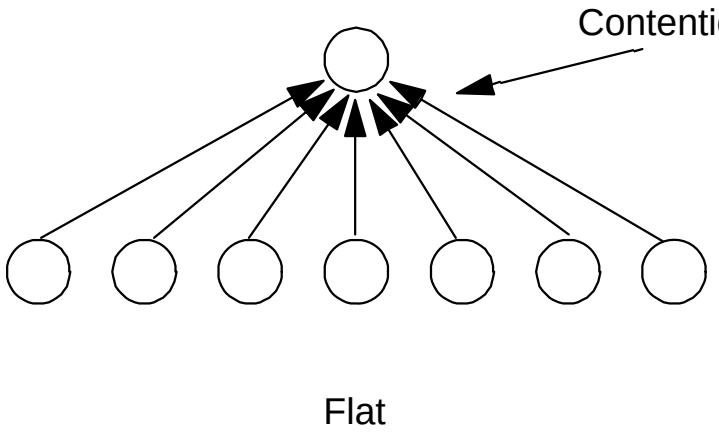
Centralized Barrier Performance

- Latency
  - Want short critical path in barrier
  - Centralized has critical path length at least proportional to *p*
  - *this is the lower bound for bus-based shared memory systems*
  - *Can be higher if contention in arriving at barrier (up to O(p^2)), although high-end is not likely since critical section very short (inc)*
- Traffic
  - Barriers likely to be highly contended, so want traffic to scale well
  - Minimum 3*p (5p)* bus transactions in centralized → *what are they?*
- 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
  - Especially the ***shared bus***, if threads arrive at lock at same time, the O(P) traffic and delay goes to O(P^2) as we saw previously. For barriers, however, the critical section is very short (or even integrated into the lock). So perhaps O(P^2) not as likely here.
  - In both cases, there are powerful methods to solve the problem – see next …

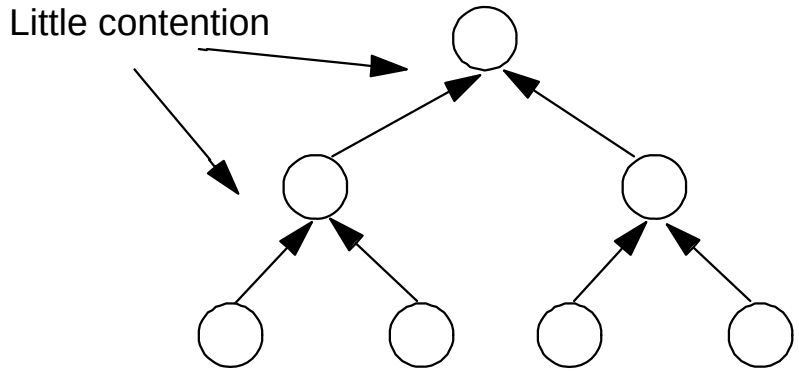
Improved Barrier Algorithms for a Bus

Software combining tree

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



Flat



Tree structured

- ***Separate arrival and exit trees***, and use sense reversal
- Valuable in distributed network: communicate along different paths
- ***On bus, all traffic on same bus, so no less total traffic (if no contention)***
- 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

Combining Tree Barrier

Idea: Tree of locks with one leaf per thread pair. Threads arrive at lock, if 1st then wait for release, if 2nd then proceed to parent. Only one thread reaches root and releases all threads.

```
public class CBarrier {
 AtomicInteger count; int size;  // count down to 1
 CBarrier parent;                // pointer to parent

 public void await(boolean mySense) {   // odd or even?
  if (this.count.getAndDecrement()==1){ // am I last?
   if (this.parent != null) // if not root, go to parent
    this.parent.await();    // recurse
   this.count.set(this.size); // prepare for next phase
   this.sense = mySense     // RELEASE
  } else {                  // Not last, …
   while (this.sense != mySense) {} // … so wait
 }}}}
```

- No sequential bottleneck
  - Parallel getAndDecrement() calls
- Low memory contention (same reason)
- Cache behavior
  - Local spinning on bus-based architecture
  - Not so good for NUMA

Tournament Tree Barrier

- If tree nodes have fan-in 2
  - Don't need to call getAndDecrement()
  - Winner chosen statically (0 = W, 1 = L)
  - 0/1 of ith bit for ith level
- If L, set neighbor (W) flag and wait
- If W, wait for flag. When set, move up
- On way down, release L

- No need for read-modify-write calls
- Each thread spins on fixed location
  - Good for bus-based architectures
  - Good for NUMA architectures

```
class TBarrier {
  boolean flag;   // notifications
  TBarrier partner; // pointer
  TBarrier parent; // pointer/null
  boolean top;    // root?
  …
}
```

```
void await(boolean mySense) {
 if (this.top) {                    // I'm the root
  return;
 } else if (this.parent != null) { // Winner
  while (this.flag != mySense) {}; // wait for partner
  this.parent.await(mySense);      // partner here, go up
  this.partner.flag = mySense; // return, release partner
 } else {                           // loser
  this.partner.flag = mySense;     // tell partner
  while (this.flag != mySense) {}; // wait for partner
}}}
```

Dissemination Barrier

Idea – Every thread has it's own tree

- Not as bad as it sounds – share nodes in the tree to form a log(n)-depth interconnection network, e.g., a *butterfly*

- At round i
  - Thread A notifies thread A+2^i (mod n)
- Requires log n rounds

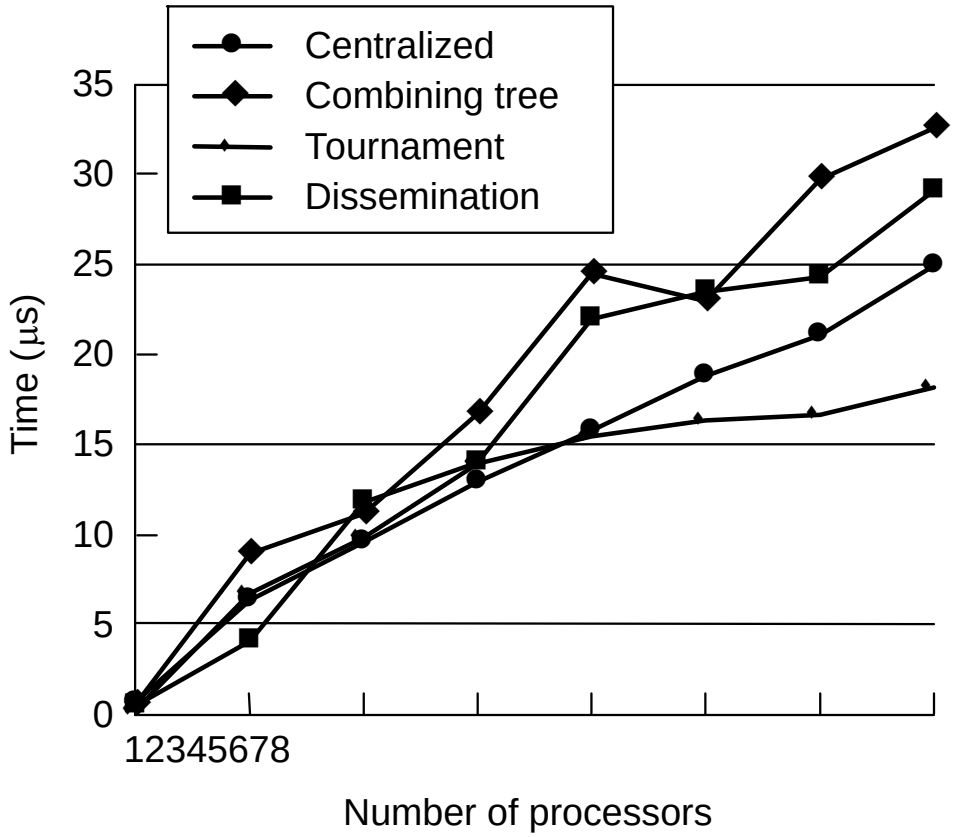

```
public DisBarrier(int n) {
 nodes = new Node[n][logn];
 int d = 1;
 for (int i = 0; i < n; i++) {
  for (int j = 0; j < logn; j++)
   nodes[i][j].partner
    = nodes[(i+d) % n][j];
   d = 2 * d;
 }}
```

```
public class DisBarrier {
 private class Node {
  boolean[] flag = {false, false};
  Node      partner = null;
 }
  private Node[][] nodes;
  …
}
```

```
void await(int parity,
           boolean mySense) {
 int i = Thread.myIndex();
 for (int r = 0; r < logn; r++) {
  nodes[i][r].partner.flag[parity]
   = mySense;
  while (nodes[i][r].flag[parity]
         != mySense) {}
   }}
```

- Every thread spins in the same place
  - Good for NUMA implementations
- Works even if n not a power of 2
- Not very space efficient

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