Module 11: "Synchronization"

Lecture 22: "Scalable Locking Primitives"

- Traffic of test & set
- Backoff test & set
- Test & test & set
- TTS traffic analysis
- Goals of a lock algorithm
- Ticket lock
- Array-based lock
- RISC processors
- LL/SC
- Locks with LL/SC
- Fetch & op with LL/SC
- Store conditional & OOO
- Speculative SC?
- Point-to-point synch.

Previous Next

Module 11: "Synchronization"
Lecture 22: "Scalable Locking Primitives"

Traffic of test & set

- In some machines (e.g., SGI Origin 2000) uncached fetch & op is supported
  - every such instruction will generate a transaction (may be good or bad depending on the support in memory controller; will discuss later)
- Let us assume that the lock location is cacheable and is kept coherent
  - Every invocation of test & set must generate a bus transaction; Why? What is the transaction? What are the possible states of the cache line holding lock_addr?
  - Therefore all lock contenders repeatedly generate bus transactions even if someone is still in the critical section and is holding the lock
- Can we improve this?
  - Test & set with backoff

Backoff test & set

- Instead of retrying immediately wait for a while
  - How long to wait?
  - Waiting for too long may lead to long latency and lost opportunity
  - Constant and variable backoff
  - Special kind of variable backoff: exponential backoff (after the i th attempt the delay is k*ci where k and c are constants)
  - Test & set with exponential backoff works pretty well

```
        delay = k
Lock:   ts register, lock_addr
        bez register, Enter_CS
        pause (delay)          /* Can be simulated as a timed loop */
        delay = delay*c
        j Lock
```

Test & test & set

- Reduce traffic further
  - Before trying test & set make sure that the lock is free

```
Lock:  ts register, lock_addr
       bez register, Enter_CS
Test:  lw register, lock_addr
       bnez register, Test
       j Lock
```

- How good is it?
  - In a cacheable lock environment the Test loop will execute from cache until it receives an invalidation (due to store in unlock); at this point the load may return a zero value after fetching the cache line
  - If the location is zero then only everyone will try test & set

TTS traffic analysis

- Recall that unlock is always a simple store

- In the worst case everyone will try to enter the CS at the same time
  - First time P transactions for ts and one succeeds; every other processor suffers a miss on the load in Test loop; then loops from cache
  - The lock-holder when unlocking generates an upgrade (why?) and invalidates all others
  - All other processors suffer read miss and get value zero now; so they break Test loop and try ts and the process continues until everyone has visited the CS

    (P+(P-1)+1+(P-1))+((P-1)+(P-2)+1+(P-2))+… = (3P-1) + (3P-4) + (3P-7) + … ~ 1.5P2 asymptotically

  - For distributed shared memory the situation is worse because each invalidation becomes a separate message (more later)

Previous Next

Module 11: "Synchronization"
Lecture 22: "Scalable Locking Primitives"

Goals of a lock algorithm

- Low latency: if no contender the lock should be acquired fast
- Low traffic: worst case lock acquire traffic should be low; otherwise it may affect unrelated transactions
- Scalability: Traffic and latency should scale slowly with the number of processors
- Low storage cost: Maintaining lock states should not impose unrealistic memory overhead
- Fairness: Ideally processors should enter CS according to the order of lock request (TS or TTS does not guarantee this)

Ticket lock

- Similar to Bakery algorithm but simpler
- A nice application of fetch & inc
- Basic idea is to come and hold a unique ticket and wait until your turn comes
  - Bakery algorithm failed to offer this uniqueness thereby increasing complexity

```
Shared:   ticket = 0, release_count = 0;
Lock:     fetch & inc reg1, ticket_addr
Wait:     lw reg2, release_count_addr        /* while (release_count != ticket); */
          sub reg3, reg2, reg1
          bnez reg3, Wait

Unlock:  addi reg2, reg2, 0x1  /* release_count++ */
         sw reg2, release_count_addr
```

- Initial fetch & inc generates O(P) traffic on bus-based machines (may be worse in DSM depending on implementation of fetch & inc)
- But the waiting algorithm still suffers from 0.5P2 messages asymptotically
  - Researchers have proposed proportional backoff i.e. in the wait loop put a delay proportional to the difference between ticket value and last read release_count
- Latency and storage-wise better than Bakery
- Traffic-wise better than TTS and Bakery (I leave it to you to analyze the traffic of Bakery)
- Guaranteed fairness: the ticket value induces a FIFO queue

Array-based lock

- Solves the $O(P^2)$ traffic problem
- The idea is to have a bit vector (essentially a character array if boolean type is not supported)
- Each processor comes and takes the next free index into the array via fetch & inc
- Then each processor loops on its index location until it becomes set
- On unlock a processor is responsible to set the next index location if someone is waiting
- Initial fetch & inc still needs O(P) traffic, but the wait loop now needs O(1) traffic
- Disadvantage: storage overhead is O(P)
- Performance concerns
  - Avoid false sharing: allocate each array location on a different cache line
  - Assume a cache line size of 128 bytes and a character array: allocate an array of size 128P bytes and use every 128th position in the array
  - For distributed shared memory the location a processor loops on may not be in its local

memory: on acquire it must take a remote miss; allocate P pages and let each processor loop on one bit in a page? Too much wastage; better solution: MCS lock (Mellor-Crummey & Scott)

- Correctness concerns
  - Make sure to handle corner cases such as determining if someone is waiting on the next location (this must be an atomic operation) while unlocking
  - Remember to reset your index location to zero while unlocking

Previous Next

Module 11: "Synchronization"
Lecture 22: "Scalable Locking Primitives"

RISC processors

- All these atomic instructions deviate from the RISC line
  - Instruction needs a load as well as a store
- Also, it would be great if we can offer a few simple instructions with which we can build most of the atomic primitives
  - Note that it is impossible to build atomic fetch & inc with xchg instruction
- MIPS, Alpha and IBM processors support a pair of instructions: LL and SC
  - Load linked and store conditional

LL/SC

- Load linked behaves just like a normal load with some extra tricks
  - Puts the loaded value in destination register as usual
  - Sets a load_linked bit residing in cache controller to 1
  - Puts the address in a special lock_address register residing in the cache controller
- Store conditional is a special store
  - sc reg, addr stores value in reg to addr only if load_linked bit is set; also it copies the value in load_linked bit to reg and resets load_linked bit
- Any intervening “operation” (e.g., bus transaction or cache replacement) to the cache line containing the address in lock_address register clears the load_linked bit so that subsequent sc fails

Locks with LL/SC

- Test & set

```
Lock:  LL r1, lock_addr          /* Normal read miss/BusRead */
       addi r2, r0, 0x1
       SC r2, lock_addr          /* Possibly upgrade miss */
       beqz r2, Lock             /* Check if SC succeeded */
       bnez r1, Lock             /* Check if someone is in CS */
```

- LL/SC is best-suited for test & test & set locks

```
Lock:  LL r1, lock_addr
       bnez r1, Lock
       addi r1, r0, 0x1
       SC r1, lock_addr
       beqz r1, Lock
```

Fetch & op with LL/SC

- Fetch & inc

```
Try:   LL r1, addr
       addi r1, r1, 0x1
       SC r1, addr
       beqz r1, Try
```

Compare & swap: Compare with r1, swap r2 and memory location (here we keep on trying until comparison passes)

```
Try:  LL r3, addr
      sub r4, r3, r1
      bnez r4, Try
      add r4, r2, r0
      SC r4, addr
      beqz r4, Try
      add r2, r3, r0
```

Previous Next

Module 11: "Synchronization"
Lecture 22: "Scalable Locking Primitives"

Store conditional & OOO

- Execution of SC in an OOO pipeline
  - Rather subtle
  - For now assume that SC issues only when it comes to the head of ROB i.e. non-speculative execution of SC
  - It first checks the load_linked bit; if reset doesn't even access cache (saves cache bandwidth and unnecessary bus transactions) and returns zero in register
  - If load_linked bit is set, it accesses cache and issues bus transaction if needed (BusReadX if cache line in I state and BusUpgr if in S state)
  - Checks load_linked bit again before writing to cache (note that cache line goes to M state in any case)
  - Can wake up dependents only when SC graduates (a case where a store initiates a dependence chain)

Speculative SC?

- What happens if SC is issued speculatively?
  - Actual store happens only when it graduates and issuing a store early only starts the write permission process
  - Suppose two processors are contending for a lock
  - Both do LL and succeed because nobody is in CS
  - Both issue SC speculatively and due to some reason the graduation of SC in both of them gets delayed
  - So although initially both may get the line one after another in M state in their caches, the load_linked bit will get reset in both by the time SC tries to graduate
  - They go back and start over with LL and may issue SC again speculatively leading to a livelock (probability of this type of livelock increases with more processors)
  - Speculative issue of SC with hardwired backoff may help
  - Better to turn off speculation for SC
- What about the branch following SC?
  - Can we speculate past that branch?
  - Assume that the branch predictor tells you that the branch is not taken i.e. fall through: we speculatively venture into the critical section
  - We speculatively execute the critical section
  - This may be good and bad
  - If the branch prediction was correct we did great
  - If the predictor went wrong, we might have interfered with the execution of the processor that is actually in CS: may cause unnecessary invalidations and extra traffic
  - Any correctness issues?

Point-to-point synch.

- Normally done in software with flags

  P0: A = 1; flag = 1;
  P1: while (!flag); print A;

- Some old machines supported full/empty bits in memory

- Each memory location is augmented with a full/empty bit
- Producer writes the location only if bit is reset
- Consumer reads location if bit is set and resets it
- Lot less flexible: one producer-one consumer sharing only (one producer-many consumers is very popular); all accesses to a memory location become synchronized (unless compiler flags some accesses as special)

- Possible optimization for shared memory
  - Allocate flag and data structures (if small) guarded by flag in same cache line e.g., flag and A in above example

Previous Next