Module 11: "Synchronization"

Lecture 21: "Introduction to Synchronization"

Cache Coherence & OOO Execution

[From Chapter 5 of Culler, Singh, Gupta]
[Speculative synchronization material taken from ASPLOS 2002 proceedings]

Previous Next

Complication with stores

- In OOO execution instructions issue out of program order
    - A store may issue out of program order
    - But it cannot write its value to cache until it retires i.e. comes to the head of ROB; Why? (assume 1p)
    - So its value is kept in a store buffer (this is normally part of the store queue entry occupied by the store)
    - If it hits in the cache (i.e. a write hit), nothing happens
    - If it misses in the cache, either a ReadX or an Upgrade request is issued on the bus depending on the state of the requested cache line
    - Until the store retires subsequent loads from the same processor to the same address can steal the value from store buffer (why not the old value?)

What about others?

- Take the following example (assume invalidation-based protocol)
    - P0 writes x, P1 reads x
    - P0 issues store, assume that it hits in cache, but it commits much later (any simple reason?)
    - P1 issues BusRd (Can it hit in P1's cache?)
    - Snoop logic in P0's cache controller finds that it is responsible for sourcing the cache line (M state)
    - What value of x does the launched cache line contain? New value or the old value?
    - After this BusRd what is the state of P0's line?
    - After this BusRd can the loads from P0 still continue to use the value written by the store?
    - What happens when P0 ultimately commits the store?
- Take the following example (assume invalidation-based protocol)
    - P0 writes x, P1 reads x
    - P0 issues store, assume that it hits in cache, but it commits much later (any simple reason?)
    - P1 issues BusRd (Can it hit in P1's cache?)
    - Snoop logic in P0's cache controller finds that it is responsible for sourcing the cache line (M state)
    - What value of x does the launched cache line contain? New value or the old value? OLD VALUE
    - After this BusRd what is the state of P0's line? S
    - After this BusRd can the matching loads from P0 still continue to use the value written by the store? YES
    - What happens when P0 ultimately commits the store? UPGRADE MISS

More example

- In the previous example same situation may arise even if P0 misses in the cache; the timing of P1's read decides whether the race happens or not
- Another example
    - P0 writes x, P1 writes x

- Suppose the race does happen i.e. P1 launches BusRdX before P0's store commits (Can P1 launch upgrade?)
- Surely the launched cache line will have old value of x as before
- Is it safe for the matching loads from P0 to use the new value of x from store buffer?
- What happens when P0's store ultimately commits?

- In the previous example same situation may arise even if P0 misses in the cache; the timing of P1's read decides whether the race happens or not
- Another example
  - P0 writes x, P1 writes x
  - Suppose the race does happen i.e. P1 launches BusRdX before P0's store commits (Can P1 launch upgrade?)
  - Surely the launched cache line will have old value of x as before
  - Is it safe for the matching loads from P0 to use the new value of x from store buffer? YES
  - What happens when P0 's store ultimately commits? READ-EXCLUSIVE MISS

Yet another example

- Another example
  - P0 reads x, P0 writes x, P1 writes x
  - Suppose the race does happen i.e. P1 launches BusRdX before P0's store commits
  - Surely the launched cache line will have old value of x as before
  - What value does P0's load commit?

Synchronization Types

- Mutual exclusion
  - Synchronize entry into critical sections
  - Normally done with locks
- Point-to-point synchronization
  - Tell a set of processors (normally set cardinality is one) that they can proceed
  - Normally done with flags
- Global synchronization
  - Bring every processor to sync
  - Wait at a point until everyone is there
  - Normally done with barriers

Synchronization

- Normally a two-part process: acquire and release; acquire can be broken into two parts: intent and wait
  - Intent: express intent to synchronize (i.e. contend for the lock, arrive at a barrier)
  - Wait: wait for your turn to synchronization (i.e. wait until you get the lock)
  - Release: proceed past synchronization and enable other contenders to synchronize
  - Waiting algorithms do not depend on the type of synchronization

Waiting algorithms

- Busy wait (common in multiprocessors)
  - Waiting processes repeatedly poll a location (implemented as a load in a loop)
  - Releasing process sets the location appropriately
  - May cause network or bus transactions
- Block
  - Waiting processes are de-scheduled
  - Frees up processor cycles for doing something else
- Busy waiting is better if
  - De-scheduling and re-scheduling take longer than busy waiting
  - No other active process
  - Does not work for single processor
- Hybrid policies: busy wait for some time and then block

Implementation

- Popular trend
  - Architects offer some simple atomic primitives
  - Library writers use these primitives to implement synchronization algorithms
  - Normally hardware primitives for acquire and possibly release are provided
  - Hard to offer hardware solutions for waiting
  - Also hardwired waiting may not offer that much of flexibility

Previous Next

Hardwired locks

- Not popular today
  - Less flexible
  - Cannot support large number of locks
- Possible designs
  - Dedicated lock line in bus so that the lock holder keeps it asserted and waiters snoop the lock line in hardware
  - Set of lock registers shared among processors and lock holder gets a lock register (Cray Xmp)

Software locks

- Bakery algorithm
  Shared: choosing[P] = FALSE, ticket[P] = 0;
  Acquire: choosing[i] = TRUE; ticket[i] = max(ticket[0],…,ticket[P-1]) + 1; choosing[i] = FALSE;
   for j = 0 to P-1
   while (choosing[j]);
   while (ticket[j] && ((ticket[j], j) < (ticket[i], i)));
   endfor
  Release: ticket[i] = 0;
- Does it work for multiprocessors?
  - Assume sequential consistency
  - Performance issues related to coherence?
- Too much overhead: need faster and simpler lock algorithms
  - Need some hardware support

Hardware support

- Start with a simple software lock
  Shared: lock = 0;
  Acquire: while (lock); lock = 1;
  Release or Unlock: lock = 0;
- Assembly translation
  Lock: lw register, lock_addr /* register is any processor register */
        bnez register, Lock
        addi register, register, 0x1
        sw register, lock_addr
  Unlock: xor register, register, register
          sw register, lock_addr
- Does it work?
  - What went wrong?
  - We wanted the read-modify-write sequence to be atomic

Previous Next

Atomic exchange

- We can fix this if we have an atomic exchange instruction

```
       addi register, r0, 0x1          /* r0 is hardwired to 0 */
Lock:  xchg register, lock_addr        /* An atomic load and store */
       bnez register, Lock
Unlock remains unchanged
```

- Various processors support this type of instruction
  - Intel x86 has xchg, Sun UltraSPARC has ldstub (load-store-unsigned byte), UltraSPARC also has swap
  - Normally easy to implement for bus-based systems: whoever wins the bus for xchg can lock the bus
  - Difficult to support in distributed memory systems

Test & set

- Less general compared to exchange

```
Lock:  ts register, lock_addr
       bnez register, Lock
Unlock remains unchanged
```

- Loads current lock value in a register and sets location always with 1
  - Exchange allows to swap any value
- A similar type of instruction is fetch & op
  - Fetch memory location in a register and apply op on the memory location
  - Op can be a set of supported operations e.g. add, increment, decrement, store etc.
  - In Test & set op=set

Fetch & op

- Possible to implement a lock with fetch & clear then add (used to be supported in BBN Butterfly 1)

```
       addi reg1, r0, 0x1
Lock:  fetch & clr then add reg1, reg2, lock_addr        /* fetch in reg2, clear, add reg1 */
       bnez reg2, Lock
```

- Butterfly 1 also supports fetch & clear then xor
- Sequent Symmetry supports fetch & store
- More sophisticated: compare & swap
  - Takes three operands: reg1, reg2, memory address
  - Compares the value in reg1 with address and if they are equal swaps the contents of reg2 and address
  - Not in line with RISC philosophy (same goes for fetch & add)

Compare & swap

```
       addi reg1, r0, 0x0          /* reg1 has 0x0 */
```

```
        addi reg2, r0, 0x1          /* reg2 has 0x1 */
Lock:   compare & swap reg1, reg2, lock_addr
        bnez reg2, Lock
```