Lecture Topics

- catching errors in the field
- [the next two topics are in the notes only]
  - detecting data races with fork-join parallelism
  - offset-span labeling
- data race propagation
- synchronization classes (theory)

Administrivia

- groups?
- should have been contacted about CRHC/corezilla accounts
- Lab 2 due on Friday 4 May

- comments on previous lecture + Matt’s talk
  - discuss lock sets & Eraser vs. vector clocks with regard to false positive rates (some can be eliminated by identifying happens-before relations)
  - example: many-to-many queue with mutex and reuse of elements creates false positives for Eraser
Catching Errors in the Field

- Can we detect these errors in production code running in the field?

- to answer
  - return to idea of read and write sets for blocks of code
  - all of this work traces back to
    - database work in mid-1970s
      (Eswaran et al., 1974; Stearns et al., 1976)
    - H.T. Kung’s optimistic concurrency, 1980

- Tom Knight in 1986 suggested
  - track read and write sets using caches
    [last lecture covered to about here]
  - for “mostly functional” programs (one or a few shared stores)
  - at end of code fragment (fragments ordered by sequential semantics)
    - write set broadcast to other processors
    - later code fragments that had used value optimistically restarted
  - paper also introduced idea of value prediction

- Transactional Memory (Herlihy, 1993)
  - same idea: use caches to track read and write sets
  - but apply to parallel execution, so no ordering between transactions
  - software control for check completion/restart

- TM has caught on again since ~2001
  - lots of interesting work
  - we may return to it later
• Speculative Lock Elision (SLE) (Rajwar & Goodman, 2001)
  
  – conservative synchronization is attractive
    • one big lock in Linux 2.0
    • if contention is low enough, it’s faster, simpler, & safer
  
  – real conflicts depend on read and write sets
    • two methods may be serialized
    • but can execute in parallel
    • and still appear atomic w.r.t. one another
    • iff
      - \( W_B \cap (R_C \cup W_C) = \emptyset \) AND
      - \( W_C \cap (R_B \cup W_B) = \emptyset \)
  
  – caches can track read and write sets
    • while processor speculates
    • detect a conflict \( \rightarrow \) discard and start over
  
  – lock cache line bouncing is expensive
    • speculate without changing lock to 1
    • cache will detect read/write set conflicts
    • lock cache line is then read shared amongst threads
    • on conflict, start over and grab lock
    • may also run out of space for speculative changes
• Detecting Synchronization Errors with SLE (Lumetta, 2002)
  – instead of executing non-speculatively after conflict
    • advertise lock acquisition (allow change to escape cache)
    • blocks all properly synchronized accesses
    • trace read and write sets speculatively
    • if another conflict is detected, must be an unsynchronized access!
      – throw exceptions on both sides
      – report same info as Eraser on both sides
      – instead of 10-30× slower
        » may be faster than original because of SLE improvement
        » can use in the field
  – false positives are an issue
    • example: safely testing condition before acquiring lock when preparing to sleep in Linux kernel
    • test condition
    • lock
    • test condition again to ensure that it has not changed
    • mark thread as sleeping, release lock, schedule another…
  – and no TM hardware exists in commercial processors (yet)
    [Sun’s chip never quite made it]
• Conflict Exceptions (Lucia et al., ISCA 2010)
  – implements a similar idea more thoroughly and evaluates it
  – uses explicit synchronization free region markers (instructions)
    • these generalize elided locks
    • mark regions by wrapping synchronization inversely
    • for example: end region, acquire lock, start region
  – false positives
    • claims that they don’t occur
    • but claim is based on assertion
      – that programmer does not write such code
      – as prescribed by Posix standard
  – hardware overhead probably still too high (debatable)

Detecting Data Races with Fork-Join Parallelism [NOTES ONLY]

• What is fork-join parallelism?

• fork-join (e.g., Cilk)
  – mostly academic approach for expressing parallelism
  – shared memory systems, with some efforts to apply more broadly
  – dynamic number of threads
    • fork new set of threads, join to wait for completion
    • forked threads may also fork new threads (hierarchical/nested)
  – explicit parallelism (threads usually based on functions)
  – asynchronous sharing, but sometimes cast into functional language
Offset-Span Labeling (Mellor-Crummey, 1991) [NOTES ONLY]

- a general technique for detecting data races with fork-join parallelism
- labels are sequences of integer pairs
- label root of fork-join graph as [0,1]
- for fork node with label L and out degree N
  - label children L[i,N] for i from 0 to N-1
  - the first number is the offset within the fork
  - the second number is the span of the fork
- for join node from children L[o,s][x,y], label it L[o+s,s]
• example graph

- label \( X \) precedes label \( Y \) \((X \rightarrow Y)\) iff any of the following holds
  1. \( X = Y \)
  2. \( \exists \) non-null sequences \( P \) and \( S \)
      such that \( X = P \) and \( Y = P \cdot S \)
  3. \( \exists \) sequences \( P, S_X, \) and \( S_Y \)
      such that \( X = P \cdot [o_X, s \cdot S_X] \) \( S_X \) and \( X = P \cdot [o_Y, s \cdot S_Y] \) \( S_Y \) and
      \( o_X < o_Y \) and \( (o_X = o_Y) \mod s \)

- label \( X \) is to the left of label \( Y \) \((X < Y)\) iff
  \( \exists \) sequences \( P, S_X, \) and \( S_Y \)
  such that \( X = P \cdot [o_X, s \cdot S_X] \) \( S_X \) and \( X = P \cdot [o_Y, s \cdot S_Y] \) \( S_Y \) and
  \( (o_X \mod s) < (o_Y \mod s) \)
for each variable, algorithm tracks
- \( W_V \equiv \text{label of last write to variable} \)
- \( L_V \equiv \text{label of lowest, leftmost read to variable} \)
- \( R_V \equiv \text{label of lowest, leftmost write to variable} \)

when a thread with label \( X \) reads variable \( V \)
- if \( W_V \to X \) does not hold, report a write-read data race
- if \( (X < L_V \text{ OR } L_V \to X) \), \( L_V \leftarrow X \)
- if \( (R_V < X \text{ OR } R_V \to X) \), \( R_V \leftarrow X \)

when a thread with label \( X \) writes variable \( V \)
- if \( W_V \to X \) does not hold, report a write-write data race
- if \( L_V \to X \) does not hold, report a write-read data race
- if \( R_V \to X \) does not hold, report a write-read data race
- \( W_V \leftarrow X \)

space and time requirements
- \( V \equiv \text{number of shared variables} \)
- \( N \equiv \text{maximum level of nesting (label length)} \)
- \( B \equiv \text{total number of threads} \)

- space dominated by access history
  - store labels directly = \( O(VN) \)
  - store pointers to thread labels = \( O(V + BN) \)

- time
  - thread creation = \( O(N) \) (per thread)
  - access cost = \( O(N) \) (per access)
Data Race Propagation

- if only things were so simple…
  - real programs can have control that depends on data
  - as well as access patterns that depend on data
- need to try to identify the “first” data race
  - others may be spurious
  - can try to identify dependences, too
- we’ll just look at an example (from Netzer and Miller, 1991)
- two threads working on array
  - queue of work starts with elements (0 to 9), (30 to 39)
  - fork to two threads; each…
    - works on private region
    - dequeues more work
    - does the extra work
  - dequeue is broken, so one thread reads (10 to 39) instead...
  - BLUE line is real race
  - RED line is feasible, but an artifact of real race’s data corruption
  - GREEN line is not feasible, since removes must overlap to fail
Synchronization Classes

- most of you should be familiar with lock-based synchronization
  - mutual exclusion primitive (a lock)
  - for atomicity (with respect to all other operations executed with lock)
    - obtain lock
    - execute operation
    - release lock
  - for dependence
    - set state
    - consumer waits for state to change (perhaps go to sleep)
    - producer changes state under protection of lock
    - locks can sometimes be omitted
      - for certain simple state changes
      - usually only if consumer is unique / change is a broadcast
• pre-emption safe locking implies OS support for locking
  – the problem
    • thread idles waiting for lock
      – wastes processor cycles
      – potentially causes deadlock
    • also leads to priority inversion
      – low-priority job holds lock, but is sleeping
      – high-priority job also sleeps until lock released
      – meanwhile, medium-priority job runs!
  – some solutions
    • priority inheritance
      – lock-holder scheduled at highest priority of waiters and self
      – used with Posix mutex
    • process holding a lock cannot be preempted
      – used for spin locks in Linux 2.4-
      – still used in certain cases in Linux 2.6+
      – also useful at user level in some cases
• lock-free algorithms/data structures do not make use of locks
  – strongest class: wait-free
    • what you expect for instruction execution, for example
    • each (executing) thread makes progress
    • in a bounded number of its own time steps
  – weaker class: non-blocking
    • some thread makes progress
    • in a bounded number of “system” time steps
  – lock-free with no guarantees
    • can starve or livelock
    • may be unlikely/impossible or an unstable equilibrium
    • progress in a bounded expected number of time steps