ADVANCED OPERATING SYSTEMS

Eraser, Capriccio

Eraser: A Dynamic Data Race Detector for Multi-threaded Programs (1997)

- What kind of paper is this?
  - New big idea?
  - Measurement paper?
  - Experiences/lessons learnt paper?
  - A system description?
  - Performance study?
  - Refute-conventional wisdom?
  - Survey paper?
  - Something else?

Synchronization background

- What happens if you run this function in two threads concurrently?

```
void f () {
      x++;
}
```

Synchronization background

- What happens if you run this function in two threads concurrently?

```
void f () {
    x++;
}
```

- Undefined – e.g., could increment x by 1 or 2 depending on interleaving
- Note even one instruction not atomic on multiprocessor

Synchronization background

- What might p2 return if run concurrently with p1?

```
int data=0, ready=0;
void p1()  {
    data = 2000;
    ready = 1;
}

int p2() {
           while (!ready);
          return data;
}
```

Synchronization background

- What might p2 return if run concurrently with p1?

```
int data=0, ready=0;
void p1()  {
    data = 2000;
    ready = 1;
}

int p2() {
         while (!ready);
         return data;
}
```

- Undefined – e.g., could be 0 or 2000
- Depends on memory model of the machine
- If hardware has *sequential consistency,* always returns 2000

What is sequential consistency?

- Definition: sequential consistency means the result of execution is as if all operations were executed in some sequential order, and the operations of each processor occurred in the order specified by the program [Lamport]
- Boils down to two requirements:
  - 1. maintaining program order on individual processors
  - 2. ensuring write atomicity

Bottom line

- Need synchronization primitives to write correct concurrent code
  - Monitor: exclusive code and associated data
    - Shared variables in monitor an only be accessed by its methods
    - System ensures only one method of a given monitor executes at a time
    - Accesses to variables ordered across method invocations
  - Mutex (lock): bracket access to shared variable with lock(m) & unlock(m)
    - System ensures lock(m) returns before next unlock(m)
    - Memory ops after lock(m) appear to happen after ones before lock(m)
    - Memory ops before unlock(m) happen before ones after it

Bottom line

- Need synchronization primitives to write correct concurrent code
  - Semaphore: two functions P(S) [a.k.a wait] and V(S) [a.k.a signal]. Only N more waits will return than signals (for initialization par N) I N==1, will act like a mutex
  - Interrupt masking:
    - Bracket code with x=spl(high, ..) (i.e., block out all interrupts)

Eraser

- What is the goal of this work?

Eraser

- What is the goal of this work?
  - Find data races by checking whether program follows locking discipline
- What is a data race?
  - Two accesses to same memory location in different threads
  - At least one is a write
  - No synchronization mechanisms to prevent simultaneous execution
- Why are races a hard problem?

Races are a hard problem

- Highly timing dependent
- Often has to do with interaction of two different modules
- Manifestation of bug may be far away from or long after buggy code was initially deployed
  - E.g., linked list corrupted, discover next time you traverse

Happens before model [Lamport]

- Processes are an ordered sequence of events, including send & receive events
- A happens before B (A-> B) if:
  - 1) A&B are events in same process P, and A precedes B, or
  - 2) A is the send of a message m by P1, and B is receive of same m by P2
- A and B concurrent if neither happened before the other
  - Idea: neither events could have influenced the other

What does happens-before have to do with race conditions?

What does happens-before have to do with race conditions?

- Idea: say that action A happens before B if:
  - 1. A&B are events in same thread T, and A preceded B, or
  - 2. A&B are accesses to same synchronization object, and A preceded B
- If A and B access same variable and neither happens before other, it's a race
  - So run program over test cases, and see if you get any such races
    - Example: look at figure 1
    - What are deficiencies of happens before approach?

Happens before approach

- Deficiencies:
  - Hard to implement efficiently (how would you detect races?)
  - Depends on particular interleaving of events (see Figure 2)
- Eraser "enforces a locking discipline"; which one?

Happens before approach

- Deficiencies:
  - Hard to implement efficiently (how would you detect races?)
  - Depends on particular interleaving of events (see Figure 2)
- Eraser "enforces a locking discipline"
  - Every shared variable must be protected by some lock
  - How can se detect this?
  - Use ATOM to implement lockset algorithm
    - ATOM: a binary instrumentation tool

What is the lockset algorithm?

What is the lockset algorithm?

- For each variable, keep track of set of candidate locks
- If set becomes empty, no lock protecting data, so flag error
- Example: figure 3
- Why is this better than happens before?

What is the lockset algorithm?

- For each variable, keep track of set of candidate locks
- If set becomes empty, no lock protecting data, so flag error
- Example: figure 3
- Why is this better than happens before?
  - Because Eraser detects violations of locking discipline, not races;
  - It can detect possible races even if they never occur during testing!

What does Eraser instrument with ATOM?

What does Eraser instrument with ATOM?

- Every load and store (except stack-relative)
- Every mutex operation (so you know what locks a thread has)
- Thread initialization and finalization code
- Malloc and free operations (so you know when memory reused)

Observations

- Locking discipline only necessary for data accessed by more than one thread
- What memory locations should not be subject to lockset algorithm?

Observations

- Locking discipline only necessary for data accessed by more than one thread
- What memory locations should not be subject to lockset algorithm?
  - Initialization – data only accessed by one thread during initialization
  - Read-shared data – no one writes, so no lock
    - Examples: version number of program, global configuration parameter, constant string entered into hash table, etc
  - Read-write locks
    - Allow multiple readers to access a shared variable but only singler write to do so

How to avoid generating false positive for these?

- Keep track of state of each variable (figure 4)
- When accessed by only 1 thread, no updates made to C(v)
- When accessed by reads only, C(v) update but races not reported when C(v) becomes empty
- A write access from new thread switches to Shared-Modified sate where C(v) updated and races are reported

Supporting Read-write locks

- Modify lockset algorithm slightly for read-write locks
  - On read of v by thread t, C(v) gets intersected with locks_held(t)
  - On write of v by thread t, C(v) gets intersected with write_locks_held(t)

Discussion

- P. 4 bottom: "Our support for initialization makes Eraser's checking more dependent on the scheduler than we would like." Why?

Discussion

- P. 4 bottom: "Our support for initialization makes Eraser's checking more dependent on the scheduler than we would like." Why?
- If variable made available before initialization complete, might not detect

What does Eraser report? How to find the bugs? (Section 3)

- Reports line of code (backtrace, regs) where lockset becomes empty set
- Is that enough?
  - Say in 100 places var accessed with correct lock, in 1 it's not
  - Likely lockset becomes empty on offending access, unless it was first access
  - But what it unlucky?
    - Ask Eraser to log all access that change a variable's lockset
    - One of them will have to be incorrect line of code

Implementation details

- How much memory?

Implementation details

- How much memory?
  - Slightly more than double the heap
  - For each 32 bit word, keep extra 32 bits of state:
    - 2 bits for state – Initialization, Exclusive, Shared, Shared-Modified
    - 30 bits for thread ID (Initialization) or lockset index number – what is the lockset index number?

Lockset index number

- Number of locksets much smaller than maximum possible
- Only keep one, sorted copy of lockset in a table
- Sorting helps speed up comparison/intersection; intersection also cached
- Hash list of locks to look up possible index number in hash table
- Cache intersection of different locks
- So… small amount of memory needed for lockset and hash tables

Why does this work?

- Greatest number of locksets seen 10,000
- Could have been exponential in number of locks – why not?
  - Lock usage very stylized – same patterns, sets of locks, etc.
  - E.g., each instant of object foo might have an internal lock but foo objects don't call each other's methods so will only lock one foo at a time (#lock sets = #locks)

Evaluation

- Performance slows down by factor of 10-30
- What possible causes of false positive are there?

Possible causes of false positive are there?

- Memory reuse – private allocators
  - Example: Vesta Log flush routine makes all entries private & empties list
- Private locks – why would you do this?
  - Pthreads does not include shared/exclusive locks
- Benign races
  - True data races that do not affect correctness of the program
  - Example: set kill_queries =1, then when threads notice, they exit
- How do they eliminate false positives?

Annotations

- EraserIgnoreOn(), EraserIgnoreOff()
  - Don't report any races in the bracketed code
- EraserReuse (address, size)
  - Tells Eraser to reset shadow memory in the memory range indicated to Virgin state
- To indicated private lock usage
  - EraserReadLock(lock), EraserReadUnlock(lock)
  - EraserWriteLock(lock), EraserWriteUnlock(lock)
- "We found that a handful of annotations usually suffices to eliminate all false alarms."

Discussion

- p. 7 (Sec 4.1): "… it might seem safe to access the p->ip_fp field in the rest of the procedure… But in fact this would be a mistake."
  - Why? P->ip_fp might point to unitialized data because alpha architecture does not offer sequential consistency
- p. 8 (Sec 4.2): why is Combine::XorFPTag::FPVal incorrect?
  - Again no sequential consistency; need some kind of locking before setting this->validFP=true

What to do about deadlock?

- Add partial order checking to enforced lockset discipline
- Seemed to find one deadlock easily this way

Results

- Performance slowdown by factor of 10-30
- Utility?
  - No serious bugs in AltaVista
    - But might have found bugs that earlier took several months to fix
  - One bug in Vesta
  - No serious bugs in Petal

Results

- How about claim that less sensitive to schedule than happens before?
  - Found same bugs in AltaVista & Vesta using 2 or 10 threads – good sign
- Undergraduate programs?
  - 10% of seemingly working programs had bugs

Capriccio: Scalable Threads for Internet Services (2003)

- What kind of paper is this?
  - New big idea?
  - Measurement paper?
  - Experiences/lessons learnt paper?
  - A system description?
  - Performance study?
  - Refute-conventional wisdom?
  - Survey paper?
  - Something else?

What is goal of Capriccio?

- Gain benefits of event-drive systems
  - Cooperative (non-preemptive) threads package
  - Scalability to 100,000 threads – O(1) efficient ops, minimize stack space
  - “Resource-aware” scheduling
- Look at Figure 1. Why do NPTL/LinuxThreads tank?
  - Capriccio non-preemptive, so no lock contention!

What are the real scalability bottlenecks?

- Virtual and physical memory for stacks
  - Why do we care about VM?

What are the real scalability bottlenecks?

- Virtual and physical memory for stacks
  - Why do we care about VM?
    - 32-bit machines, max stack might be 2MB
    - Note: newer 64-bit machines typically have 48-bit virtual address space
  - Why do we care about PM?
    - OS might not be clever enough to reclaim thread stack space
- Algorithms?
  - Generally we want O(1), not O(# threads)
  - Examples: prioritizing run threads and stride scheduling in Sec 4.2

What is linked stack management?

What is linked stack management?

- Break stack into pieces
- Can code deal with non-contiguous stack?
  - Yes, if discontinuity across procedure boundary;
  - Return usually restores stack pointer based on frame pointer can just restore to different place
  - But we have to know max space used within function
    - What about alloca(size) – oops, probably can't allow

Compute graph

- Place checkpoints where might overflow
- What is internal wasted space?
- What is external wasted space?
- How do these get tuned?

Compute graph

- Place checkpoints where might overflow
- What is internal wasted space?
  - Skipped rest of chunk, added bookkeeping to get new chunk
- What is external wasted space?
  - Bottom of current chunk is unused
  - This is what traditional threads packages have with one big stack
- How do these get tuned?
  - MaxPath and MinChunk parameters

Discussion on compute graph

- How to deal with library code?
- How to handle function pointers?
  - Guess based on type signature
- Sec 3.5 compares to guard pages – what are these?
  - If have enough VM, use page faults to allocate physical memory as needed
  - Idea: overflowing stack always gives page fault on guard page
    - Adv: common case is cheap (no extra code)
    - Drawbacks: overhead for handling page faults probably higher than check point; might use more physical memory?

What is resource-aware scheduling?

- What resources?

What is resource-aware scheduling?

- What resources?
  - CPU, memory, file descriptors
  - What about disk? Hard because request rate depends on locality
  - Compute blocking graph (Fig 8)
    - How do they annotate this graph?

Annotating blocking graph

- Avg running time on each edge
- Node annotated with how long next edge will take on avg
- Annotate graph change in resource usage
- Track resource utilization
- Maintain run Q for each node
- Promote nodes (and thus threads) that release a resource that is becoming scarce and demote those that acquire the resource
  - “Natural admission control” on p. 9

Discussion

- What about tasks that don't yield?
  - P. 10 suggest "use multiple kernel threads"
  - Think this would just work?
- Extending to multiprocessor machines
  - "we can no longer rely on the cooperative threading model to provide atomicity"

Evaluation

- Convincing?
- Improvement worth the effort?