Eraser: Dynamic Data Race Detection

15-712

Topics overview

- Concurrency and race detection
  - Framework: dynamic, static
  - Sound vs. unsound
- Tools, generally:
  - Binary rewriting (ATOM, Etch, ...) and analysis (BAP, etc) - see also use in JITs and VMMs.
  - Techniques used for state space reduction (c.f. model checking)
- What are the eval criteria they used? How did they argue?

Debugging Concurrency is Hard

- Few tools beyond per-proc gdb
  - Debugging multi-process systems is harder than single process, even without...
- Threads come with unique problems
  - Deadlock, data races, non-determinism
- High performance typically implies lots of locks
  - For max parallelism, vs. one “big” lock
  - Shows up in modern kernel evolution as SMP grows
  - As #cores grow, parallelism will have to extend further and further into apps/libs to keep getting faster

Eraser: Lint for multi-threading

- Multiple threads in single address space
- Shared memory, one CPU
- Assumes:
  - pthreads lock() is sync, not monitors
  - libc memory allocation
- Doesn’t work out causality
  - Costly bookkeeping; requires observing right interleaving
- Instead looks for idiom/style
  - Must hold lock to access shared variable
  - Data race: simultaneous access w/1+ writer
  - Limitation: must be consistent locking, not “either of two” locks
    - e.g., DB pages or multi-page locks
Binary Rewriting

- “Metaprogramming” tool for executables
  - Read & partially understand binary
  - Could also be done in compiler...
  - Write new extended code after adding function
- (Harder on x86, but it’s been done - variable length binary instruction set vs. more RISC-like systems)
- DEC ATOM tool for Alpha
  - Insert counters -- “super gprof”
  - Idea: Add memory restriction for safer code (dynamic bounds checking)
  - Add lock analysis to “lint” the sync code.
- compare to: modifying the source, interpreting the instruction stream, trapping system calls in the kernel

See Savage Slides

- See Savage Slides
- Next bunch of slides taken in very large part, mostly verbatim, from the SOSP talk.
  - Some annotations. Blame dga, not SOSP talk, for bugs

Lockset algorithm

- Dynamic analysis
- Require programmer to adhere to convention
  - Locking Discipline:
    - Consistently hold lock when using resource
    - Automatically infers which lock(s) protect resource
- Finds more bugs than “happens-before”
- But can generate many false positives!

How happens-before misses races

Thread 1

```
y := y + 1;
lock(mu);
v := v + 1;
unlock(mu);
```

Thread 2

```
lock(mu);
v := v + 1;
unlock(mu);
y := y + 1;
```

Not detected as a race by happens-before
Checking simple discipline

C(v): locks that might protect variable v
Initialize C(v) := set of all locks

On each access to v by thread t,
C(v) := C(v) \cap \text{locks\_held}(t)

If C(v) is empty, then issue a warning

Refining the candidate set

Program            New value of C(v)
lock(mu1);
\[ v := v + 1; \]
unlock(mu1);
\[ \cdots \]
lock(mu2);
\[ v := v + 1; \]
unlock(mu2);

False Positives

- FPs are the defining problem with this type of approach
- We'll see later some other examples...
- Really hurt usability. 1000 FPs + 1 bug isn't useful.
- Much of the rest of paper is about avoiding FPs
- Ideally without reducing true positives (does it?)

Limitations of simple algorithm

- Initialization
  - Don't need locks until data is shared
- Read-shared data
  - Don't need locks if all accesses are reads
- Reader/writer locks
  - Read locks can't protect writes
Modified algorithm

- Assume first thread is initializer
  - Only update C(v) after two threads touch v
- Only report races after data is known to be write-shared
- Track read and write locks separately
  - Remove read locks from C(v) on a write

Mapping variables to sets of locks

Performance

- Fast enough to be useful
  - 10-30x user-time slowdown
- Lots of opportunities for optimization
  - Half of overhead due to ATOM

Experiences

- Tested real programs
  - AltaVista web server and index library
  - Vesta cache server
  - Petal distributed disk server
  - Undergrad coursework from intro OS class
- Most programs found to contain races
- False alarms easy to manage
### Program characterization

<table>
<thead>
<tr>
<th>Program</th>
<th>Lines of code</th>
<th>Threads</th>
<th>Locks</th>
<th>Lock sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>AltaVista</td>
<td>20,000</td>
<td>10</td>
<td>900</td>
<td>3600</td>
</tr>
<tr>
<td>mhtpd</td>
<td>5,000</td>
<td>10</td>
<td>100</td>
<td>250</td>
</tr>
<tr>
<td>Vesta</td>
<td>30,000</td>
<td>10</td>
<td>26</td>
<td>70</td>
</tr>
<tr>
<td>Petal</td>
<td>25,000</td>
<td>64</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Case study: AltaVista

- Double blind experiment
  - Two old races reintroduced
  - Previously undetected for several months
  - Found and fixed in 30 minutes
- Several additional (minor) races found
- Several benign races
  - Tricky optimizations
- Re-introducing bugs is a useful and now common technique

### Benign race example

```c
if (p->fp == 0) {
    lock(p->lock);
    if (p->fp == 0) {
        p->fp = open_file();
    }
    unlock(p->lock);
}
```

- Advanced programmers make automatic inference hard...
- Subtle optimization, hard to reason about its correctness

### Serious race (subtle)

```c
if (p->fp == 0) {
    lock(p->lock);
    if (p->fp == 0) {
        p->fp = open_file();
    }
    unlock(p->lock);
    pos = p->fp->pos;
```
Case study: Undergraduate OS

- Four simple synchronization problems
  - e.g. producer consumer
- ~180 homeworks tested
- Found data races in more than 10%

Overall races detected

<table>
<thead>
<tr>
<th>Program</th>
<th>Serious races</th>
<th>Minor races</th>
<th>Benign races</th>
</tr>
</thead>
<tbody>
<tr>
<td>AltaVista</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Vesta</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Petal</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Undergrad assignments</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Kinds of false alarms

- Private memory allocators
  - e.g. free list
  - Need to reinitialize C(v)
- Private lock implementations
  - e.g. reader/writer locks
  - Need to know when locks are held
- Benign races

Removing false alarms

- Simple program annotations
- Number of annotations needed to remove all false alarms:
  - AltaVista (19)
  - Vesta (10)
  - Petal (4)
end SOSP talk slides

Eval

- Modern systems eval
  - Real impl (distributable; recently rediscovered value for research code)
  - Injected faults by sticking old bugs back into code
    - This technique used in many subsequent evals
    - cvs/svn/git/etc. history
  - Applied to large, real applications and multiple mostly independent tests (students)

- Would have been nice to see more quantitative comparison between systems, but that’s another paper... :)

Eval 2

- Not perfect tool:
  - 10x slowdown
  - Processor specific for dead processor
  - No guarantee to catch all races
    - In particular: Engler papers suggest many bugs lurk in infrequently hit code -- error handling, etc.
    - Dynamic detection has hard time catching those
      - Just like testing does.
  - But useful for an otherwise hard problem

Design Space

- Static vs. Dynamic analysis
  - Dynamic: Depends on execution order
  - Static: Intractable (but can work well, today)
- Sound vs. Unsound
  - Note tension between pragmatists and correct-ists

- Existing languages (C...) vs. “Better” languages
  - Actual code? Annotations? Model?
  - Type systems; correct by construction?
  - Code generation: prove correct, then generate code.

- Eval questions:
  - False positives? False negatives? Coverage?
  - Running time?
  - Run in tests vs. in production system?
Follow-on

- Want more?
    - SOSP 2001. Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf
  - “The Daikon system for dynamic detection of likely invariants”
  - Both examine more general invariants
    - Daikon examines more invariants;
    - Engler et al. is static
    - Both use machine-learning/statistical techniques to infer invariants