Eraser:
A Dynamic Race Detector for Multi-Threaded Programs

Stefan Savage
University of Washington

Michael Burrows, Greg Nelson, Patrick Sobalvarro
Digital System Research Center

Thomas Anderson
University of Washington
Why data races are a problem

- Multi-threading has become common
  - e.g. Oracle, Powerpoint, Notes/Domino, Netscape, Explorer, Informix, Excel, Apache, SQL Server, Java...

- Hard to debug concurrent programs
  - Non-deterministic race conditions
  - Difficult to diagnose or isolate

- Very few tools to help
Design goals for Eraser

- Simple to use
  - Rewrites program binary to check for races on each memory access
  - Modified program prints warnings as it runs
- Effective with *real world* programs
  - Used to find races in several programs (e.g. AltaVista)
Outline

- Background
- The *Lockset* algorithm
- Eraser implementation
- Experiences
What is a data race?

When two or more threads access a shared variable, and:

- At least one access is a write
- There is no mechanism used to prevent the accesses from being simultaneous

The result is non-deterministic
Race detection algorithms

- Static analysis (proof)
  - Intractable in general
  - Heuristics require programmer intervention

- Dynamic analysis (testing)
  - Can’t prove correctness
  - *Happens-before* relation [Lamport]
    - Very general, but misses potential races
**Lockset algorithm**

- Form of dynamic analysis
- Specialized for lock-based programs
  - Based on checking *locking disciplines*
  - Detects more races than *happens-before*
  - Can generate more *false alarms*
- Infers locking behavior automatically
Locking disciplines

- Some programming policy used to prevent data races
- Simple discipline:
  
  *Every shared variable v must be protected by some lock m*
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) ∩ locks_held(t)

If C(v) is empty, then issue a warning
Refining the candidate set

Program

```
lock (mu1);
v := v + 1;
unlock (mu1);
...
lock (mu2);
v := v + 1;
unlock (mu2);
```

New value of $C(v)$

$C(v) = \{\text{mu1, mu2}\}$

$C(v) = \{\text{mu1}\}$

$C(v) = \{\}$
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
Life of a write-shared variable

Virgin

Exclusive

Shared

Update C(v)

rd, new thread

wr

rd/wr, first thread

wr, new thread

rd/wr

Shared-Modified

Update C(V) and check for races

wr
Implementing Eraser

- Uses ATOM binary rewriter
  - Instrument loads, stores, and API’s for threads, locks and memory allocation
  - Modified program prints race warnings

- Many possible implementations
  - Compiler support
  - Virtual machines (e.g. Java, SimOS)
  - MMU traps
Mapping variables to sets of locks

\[ v \]

Program memory

\[ &v + \text{offset} \]

Shadow memory

2

Lockset index table

\[ \mu_1 \]

Lock vector

\[ \mu_4 \]
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
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
Benign race example

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

- Data races make multi-threaded programs difficult to debug
- Even experienced coders need help
- We designed, built and validated a tool that detects and locates data races
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)
How \textit{happens-before} misses races

Thread 1

\begin{verbatim}
y := y + 1;
lock(mu);
v := v + 1;
unlock(mu);
\end{verbatim}

Thread 2

\begin{verbatim}
lock(mu);
v := v + 1;
unlock(mu);
y := y + 1;
\end{verbatim}

Not detected as a race by \textit{happens-before}
Time dilation

- Changes thread execution order
  - Eraser generally insensitive to order
- Perturbs program observations of time
  - Not an issue in our tests
  - Options for real-time software
    - More efficient instrumentation technique
    - Simulate slower clock
## 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 Ni2</td>
<td>20,000</td>
<td>10</td>
<td>900</td>
<td>3600</td>
</tr>
<tr>
<td>AltaVista mhttpd</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>
Checking R/W discipline

Initialize $C(v) := \text{set of all locks}$

On each read from $v$ by thread $t$,

$C(v) := C(v) \cap \text{locks\_held}(t)$

On each write to $v$ by thread $t$,

$C(v) := C(v) \cap \text{write\_locks\_held}(t)$

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