Eraser: A Dynamic Data Race Detector for Multithreaded Programs (Savage et al., 1997)

Presented by Scott Fleming

A Paper Presentation for
CSE 891 - Formal Methods in Software Development: Reliable Computing With Threads
Fall Semester, 2008
What Does Eraser Do?

Function: *Dynamically detects* the occurrence of *data races* in multithreaded programs

Data race: Occurs when

- two or more threads concurrently access the same memory location,
- at least one of the threads is writing, and
- no explicit mechanism is used to prevent the accesses from being simultaneous

Dynamic detection: Detects as program executes

- Occurrence of data race depends on what code is executed
  - Therefore, may depend on thread schedule
- May miss *potential* data races
Producer-Consumer with Data Race

```java
q.enqueue(item);
```
Producer-Consumer with General Race

```
for (;;) {
    if (!buf->full())
        buf->push(produce());
}
```

```
q.enqueue(item);
```

```
<<monitor>>
BoundedBuffer

q : Queue<T>
empty() : Boolean
full() : Boolean
push(item : T)
pull() : T
```

```
Producer

Consumer
```
How Does Eraser Do It?

Assumes: Lock-based synchronization

Approach: Checks that all shared-memory accesses follow a consistent *locking discipline*

Lock: Primitive synchronization object used for mutual exclusion (also called a *mutex*)

- States: available or owned (by a thread)
- Operations: (atomic) lock and unlock

Locking discipline: Protocol of lock acquisition and release to prevent data races
Definition: $C(v)$ is the set of candidate locks for variable $v$
Definition: $locks\_held(t)$ is the set of locks held by thread $t$

Algorithm:
For each variable $v$, initialize $C(v)$ to the set of all locks

On each access to $v$ by thread $t$,
set $C(v) := C(v) \cap locks\_held(t)$;
if $C(v) = \{\}$, then issue a warning
Example Application of Lockset Algorithm

<table>
<thead>
<tr>
<th>t1</th>
<th>t2</th>
<th>locks_held(t1)</th>
<th>locks_held(t2)</th>
<th>C(v)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>{}</td>
<td>{}</td>
<td>{ l1, l2 }</td>
</tr>
<tr>
<td>lock(l1)</td>
<td></td>
<td>{ l1 }</td>
<td>{}</td>
<td>{ l1 }</td>
</tr>
<tr>
<td>++v</td>
<td></td>
<td></td>
<td>{}</td>
<td></td>
</tr>
<tr>
<td>unlock(l1)</td>
<td></td>
<td></td>
<td>{}</td>
<td></td>
</tr>
<tr>
<td>lock(l2)</td>
<td></td>
<td></td>
<td>{ l2 }</td>
<td></td>
</tr>
<tr>
<td>++v</td>
<td></td>
<td></td>
<td>{}</td>
<td></td>
</tr>
<tr>
<td>unlock(l2)</td>
<td></td>
<td></td>
<td>{}</td>
<td></td>
</tr>
</tbody>
</table>
Three Problematic Practices

**Initialization:** Shared variables initialized without locking

**Read-shared data:** Shared constant variables

**Read-write locks:** Read-write locks allow multiple readers to access a shared variable, but allow only one writer
Solution to Initialization Problem

Only refine lockset after initialization

**Problem:** When is initialization complete?

**Solution:** Initialization ends when a second thread first accesses the variable
Solution to Read-Shared Data Problem

Only report races after an *initialized* variable has been written.
State Transitions of a Shared Variable

Virgin

Exclusive

Shared

Shared-Modified

- Virgin
  - write
  - read/write, 1st thread

- Exclusive
  - write
  - write, new thread

- Shared
  - read
  - write

- Shared-Modified
  - read/write
  - write
  - read, new thread
  - write, new thread
Possible Problem with Solutions

Scenario:

- Thread t1 allocates variable v and begins initializing v
- Thread t2 gets a reference to v before initialization is complete

Possible problem: Eraser will only detect a race if t2 is scheduled to write to v before t1 finishes initializing v

- Detection is scheduler dependent
- Scheduling is nondeterministic
### Example of R/W Lock Problem

<table>
<thead>
<tr>
<th>t1</th>
<th>t2</th>
<th>locks_held(t1)</th>
<th>locks_held(t2)</th>
<th>C(v)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>{}</td>
<td>{}</td>
<td>{ l1</td>
</tr>
<tr>
<td>rdlock(l1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>rdlock(l1)</td>
<td>{}</td>
<td>l1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>++v</td>
<td>l1</td>
<td></td>
<td>l1</td>
</tr>
<tr>
<td></td>
<td>unlock(l1)</td>
<td></td>
<td>{}</td>
<td></td>
</tr>
<tr>
<td>read(v)</td>
<td>unlock(l1)</td>
<td></td>
<td>{}</td>
<td>l1</td>
</tr>
</tbody>
</table>
Solution to Read-Write Locks Problem

Require that for each variable \( v \),
- some lock \( l \) protects \( v \), and
- \( l \) is held in some mode for every read of \( v \)

Definition: \( l \) protects \( v \) if \( l \) is held in write mode for every write of \( v \)
**Lockset Algorithm, Final Version**

**Definition:** \( C(v) \) is the set of candidate locks for variable \( v \)

**Definition:** \( locks\_held(t) \) is the set of locks held by thread \( t \) in any mode

**Definition:** \( write\_locks\_held(t) \) is the set of locks held in write mode by thread \( t \)

**Algorithm:**
For each variable \( v \), initialize \( C(v) \) to the set of all locks

On each read to \( v \) by thread \( t \),
set \( C(v) := C(v) \cap locks\_held(t) \);
if \( C(v) = \{ \} \), then issue a warning

On each write of \( v \) by thread \( t \),
set \( C(v) := C(v) \cap write\_locks\_held(t) \);
if \( C(v) = \{ \} \), then issue a warning
## Example of R/W Lock Solution

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>l_held(t1)</th>
<th>wl_held(t1)</th>
<th>l_held(t2)</th>
<th>wl_held(t2)</th>
<th>C(v)</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>t2</td>
<td>{}</td>
<td>{}</td>
<td>{}</td>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{ l1 }</td>
<td></td>
<td>{}</td>
<td></td>
<td>{}</td>
</tr>
<tr>
<td>rdlock(l1)</td>
<td></td>
<td>{ l1 }</td>
<td></td>
<td>{}</td>
<td></td>
<td>{}</td>
</tr>
<tr>
<td>rdlock(l1)</td>
<td>++v</td>
<td>{}</td>
<td></td>
<td>{ l1 }</td>
<td></td>
<td>{}</td>
</tr>
<tr>
<td></td>
<td>unlock(l1)</td>
<td></td>
<td>{}</td>
<td>{}</td>
<td></td>
<td>{}</td>
</tr>
<tr>
<td>read(v)</td>
<td>unlock(l1)</td>
<td></td>
<td>{}</td>
<td>{}</td>
<td></td>
<td>{}</td>
</tr>
</tbody>
</table>
Implementation Details

**Approach:** Automatically instruments program binary with calls to Eraser runtime

- memory loads and stores
- lock acquires and releases
- storage allocators

**Warning messages:**

- Source file and line number where race discovered
- Backtrace listing all active stack frames
- Thread ID
- Memory address
- Type of memory access
- Register values such as program counter and stack pointer
Evaluation

Performance: Applications slow down by a factor of 10 to 30

Precision: False alarms common; 3 main causes:
  - Memory reuse
  - Private locks
  - Benign races

Recall: ????
  - “differences in thread scheduling have little effect on Eraser's results”

Usability of output: Information is usually sufficient to locate origin of race