Eraser: A dynamic data race detector for multithreaded programs

By Stefan Savage et al.
MOTIVATION

- Only parallel programs can benefit from today’s multi-core processors
- Multithreading has become a common (important) programming technique
- Synchronization between threads is a challenge
  - Synchronization errors are easy to cause and hard to debug
    - DATA RACE
  - Tracking errors take weeks and months
Data race: Non-Determinism

When two concurrent threads access a shared variable
1. at least one is a write and
2. the threads use no explicit synchronization to prevent simultaneous access
then the execution will depend on interleaving
Data race : Non-Determinism

'a'-----???????
Data race : Non-Determinism

Locks are used to avoid data race; Let’s see a tool that detects a race.
Complexity in Data Race Detection

- Data Race detection is a NP complete problem.
- For \( t \) threads of \( n \) instructions, the number of possible orders is about \( t^{n*t} \).
- A thorough detection will involve examining all the possible order to make sure there exist only one order.
- Practical race detection tools are based on heuristics - so that they can detect maximum number of races, within limited computation.
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

SOSP’ 1997
Cited by 876
Objective of the work

The work presents the theory, implementation and experience of a testing tool that detect dynamic data race in multi-threaded programs.

Author's Claims

Does not identify all races in the program, but for programs using lock-based synchronization ensures better results than Lamport's (previous) work.
Outline

- Previous Work
- The Lockset algorithm
- Eraser implementation
- Experiences
Lamport's Happenes-before

The happens-before order is a partial order on all events of all threads in a concurrent execution,

- **Single Thread** - events are ordered by their occurrence.
- **Between threads** - events are ordered by the synchronization objects they access.

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

Thread 2
lock(mu);
↓
v := v+1;
↓
unlock(mu);
```
**Happens-Before Fails**

Thread 1

\[ y := y+1; \]
\[ \downarrow \]
\[ \textbf{lock}(\text{mu}); \]
\[ \downarrow \]
\[ v := v+1; \]
\[ \downarrow \]
\[ \textbf{unlock}(\text{mu}); \]

\[ \textbf{lock}(\text{mu}); \]
\[ \downarrow \]
\[ v := v+1; \]
\[ \downarrow \]
\[ \textbf{unlock}(\text{mu}); \]
\[ y := y+1; \]

Thread 2

\[ \textbf{lock}(\text{mu}); \]
\[ \downarrow \]
\[ v := v+1; \]
\[ \downarrow \]
\[ \textbf{unlock}(\text{mu}); \]

\[ \textbf{lock}(\text{mu}); \]
\[ \downarrow \]
\[ v := v+1; \]
\[ \downarrow \]
\[ \textbf{unlock}(\text{mu}); \]
\[ y := y+1; \]
Other Issues

- Difficult to implement efficiently - need per-thread information about ordering to all shared memory locations.
- Highly dependent on scheduler - needs large number of test cases.
Eraser's approach

Eraser uses binary rewriting techniques to monitor every shared memory reference and verify that consistent locking behavior is observed.

Heuristics – Consistent Locking Behavior
Eraser's approach

Eraser uses **binary rewriting techniques** to monitor every shared memory reference and verify that **consistent locking behavior** is observed.

'binary rewriting' – Observe all the Load and Store instructions.

'consistent locking behavior' - All instance of a shared variable \( v \) is locked by same set of locks \( L(v) \)
**Lockset Algorithm**

- \( v \) = shared variable
- \( C(v) \) = candidate locks for \( v \)
- \( \text{locks\_held}(t) = \text{set of locks held by thread } t \)
- A lock \( l \) is in \( C(v) \) if all threads hold \( l \) while accessing \( v \)
- A new variable at initialization is supposed to have all possible locks in \( C(v) \)
**Traversing Lockset Algorithm**

<table>
<thead>
<tr>
<th>Program</th>
<th>locks_held</th>
<th>C(v)</th>
</tr>
</thead>
<tbody>
<tr>
<td>int v; v := 1024;</td>
<td>{}</td>
<td>{mu1, mu2}</td>
</tr>
<tr>
<td>lock(mu1); v := v + 1;</td>
<td>{mu1}</td>
<td>{}</td>
</tr>
<tr>
<td>unlock(mu1);</td>
<td></td>
<td>{mu1}</td>
</tr>
<tr>
<td>lock(mu2); v := v + 1;</td>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>unlock(mu2);</td>
<td>{}</td>
<td>{}</td>
</tr>
</tbody>
</table>
Increasing the comfort zone

Initialization - Initialization can be done without holding a lock.

Read-shared data - Multiple reads safely accessed without locks.

Read-write locks - Multiple readers, but allow only a single writer.
Initialization and Read-Sharing

- A variable is considered initialized when it is accessed by a second thread.
- Simultaneous reads of shared variable are not races.
- Report races only after an initialized variable becomes write-shared.
Per-Location State

- **virgin**: wr, 1st thread
- **exclusive**: rd, wr, 1st thread, rd, new thread
- **shared**: C(v) updated, no race reporting
- **Shared modified**: C(v) updated, races reported

Arrows indicate transitions between states.
Extending Lockset Algorithm

- Read / Write locking modes
- Locks held in read mode are removed from $C(v)$ when a write occurs.
- On each read of $v$ by thread $t$,
  - set $C(v) := C(v) \cap \text{locks\_held}(t)$;
  - if $C(v) = \emptyset$, then issue a warning.
- On each write of $v$ by thread $t$,
  - set $C(v) := C(v) \cap \text{write\_locks\_held}(t)$;
  - if $C(v) = \emptyset$, then issue a warning.
### Traversing Lockset Algorithm

<table>
<thead>
<tr>
<th>Program</th>
<th>locks_held</th>
<th>C(v)</th>
<th>State(v)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int v; v := 1024;</code></td>
<td><code>{}</code></td>
<td><code>{mu1, mu2}</code></td>
<td>Virgin</td>
</tr>
<tr>
<td><code>lock(mu1);</code></td>
<td><code>{mu1}</code></td>
<td></td>
<td>Exclusive</td>
</tr>
<tr>
<td><code>v := v + 1;</code></td>
<td><code>{}</code></td>
<td><code>{mu1}</code></td>
<td>Shared</td>
</tr>
<tr>
<td><code>unlock(mu1);</code></td>
<td></td>
<td></td>
<td>Shared-Modified</td>
</tr>
<tr>
<td><code>lock(mu2);</code></td>
<td><code>{mu2}</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>v := v + 1;</code></td>
<td><code>{}</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>unlock(mu2);</code></td>
<td><code>{}</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Race detected correctly
Implementation

Eraser's binary modification involves

1. Calls to storage allocator initializes $C(v)$
2. Each load and store updates $C(v)$
3. Each acquire or release call updates $\text{locks\_held}(t)$
Data Structure,

1. Maintains hash table of sets of locks.
2. Represents each set of locks with an index.
3. Every shared memory location has shadow memory containing lockset index and state.
4. Shadow memory is located by adding offset to shared memory location address.
Implementation

Program memory

\&v + shadow offset

Shadow memory

3

Lockset index table

mu1

mu2

Lock vector
False Positives

- Memory reuse
  Caused by memory reset, with out resetting the shadow memory.
- Private locks
  When Locks that are not part of standard pthread interface are used.
- Benign races
  True Data race that did not affect the execution of the program.

“I have a positive attitude… but it might be a false positive.”
Annotations to Avoid False Positives

1. **For memory reuse**
   1. EraserReuse(address, size)

2. **For private locks**
   1. EraserReadLock(lock)
   2. EraserReadUnlock(lock)
   3. EraserWriteLock(lock)
   4. EraserWriteUnlock(lock)

3. **For benign races**
   1. EraserIgnoreOn()
   2. EraserIgnoreOff()
Experience

1. AltaVista
   1. *Mhttpd* http server - 5,000 lines of C source code, 100 distinct locks, 9 annotations.
   2. *Ni2* indexing engine - 20,000 lines of C source code, 900 distinct locks, 10 annotations.

2. Vesta Cache Server - 30,000 lines of C++ source code, 10 threads, 26 distinct locks, 10 annotations.

3. Petal - distributed disk server - 30,000 lines of C++ source code, 64 threads

4. Undergrad coursework – 100 multi threaded programs
Experience

- Deliberately introduced race conditions were detected.
- Other data races were also detected.
- False alarms were raised, but use of annotations resolved sizable number of them.
## Experience

<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>
Performance

- NP-Complete – Computationally hard problem.
- Implementing Eraser slows down the application by a factor of 10 to 30
- Overhead of making a procedure call at every load and store instruction
- Performance was never a major goal
RECAP: Objective of the Work

The work presents the theory, implementation and experience of a testing tool that detect dynamic data race in multi-threaded programs.
Conclusion

- Uses the locking principles as a heuristics to identify data races
- Successful implementation identifies potential races in enterprise software
- Implementing Eraser slows down the application by a factor of 10 to 30
- Diverse case study was given to support the tool
Let’s Discuss