Eraser: A dynamic data race detector for multithreaded programs

S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson
TOCS Nov. 97
Overview

• Dynamic data race detection tool – testing paradigm instead of static analysis.
• Checks that each shared memory access follows a consistent *locking discipline*
• *Data race* – when 2 concurrent threads access a shared variable and at least one is a write and the threads use no explicit synchronization to prevent simultaneous access.
  – Effect will depend on interleaving
Previous Approaches: Lamport’s Happened-Before

Previous work

- If 2 threads access a shared variable and the accesses are not ordered by happens-before, then potential race.
Drawbacks of Happened-Before

• Difficult to implement efficiently – need per-thread information about access ordering to all shared memory locations.

• Highly dependent on scheduler – needs large number of test cases.
Previous work

- If 2 threads access a shared variable and the accesses are not ordered by happens-before then potential race.
- Depends on scheduler
Previous work

- If 2 threads access a shared variable and the accesses are not ordered by happens-before then potential race.
- Depends on scheduler

```plaintext
lock(mutex)
v = v+1;
unlock(mutex)
y=y+1;
```

```plaintext
lock(mutex)
\downarrow
v = v+1;
\downarrow
unlock(mutex)
\downarrow
y=y+1;
```
Idea in Eraser

• Checks that locking discipline is observed.
  – That the same lock(s) is held whenever the shared data object is accessed.
  – Infer which locks protect which data items
Lockset Algorithm

- $C(v)$ – candidate locks for $v$
- $\text{locks-held}(t)$ – set of locks held by thread $t$
- Lock refinement
  - for each $v$, init $C(v)$ to set of all locks
  - On each access to $v$ by thread $t$:
    - $C(v) = C(v) \cap \text{locks-held}(t)$
    - If $C(v) = \emptyset$ issue warning
Example

locks-held
{}

C(v)
{mu1, mu2}

lock(mu1)

{mu1}

v=v+1

unlock(mu1)

{}

lock(mu2)

{mu2}

v=v+1

unlock(mu2)

{mu1}

{mu2}

{}}
More Sophistication

- Initialization without locks
- Read-shared data (written only during init, read-only afterwards)
- Reader-writer locking (multiple readers)
- False Alarms still possible

- Don’t start until see a second thread
- Report only after it becomes write shared
- Change algorithm to reflect lock types
  - On read of v by t:
    \[ C(v) = C(v) \cap \text{locks-held}(t) \]
  - On write of v by t:
    \[ C(v) = C(v) \cap \text{write-locks-held}(t) \]

False Alarms still possible
Per-Location State

- **virgin**
  - wr, 1st thread

- **exclusive**
  - rd, wr, 1st thread
  - wr, new thread

- **shared**
  - rd, new thread
  - wr
  - C(v) updated, no race reporting

- **Shared modified**
  - wr, new thread
  - C(v) updated, races reported
Implementation

• Binary rewriting used
  – Add instrumentation to call Eraser runtime
  – Each load and store updates C(v)
  – Each Acquire and Release call updates locks-held(t)
  – Calls to storage allocator initializes C(v)

• Storage explosion handled by table lookup and use of indexes to represent sets
  – Shadow word holds index number

• Slowdown by factor of 10 to 30x
  – Will change interleavings
Shadow Memory and Lockset Indexes

Program memory \( \rightarrow \) \&v + shadow offset

Shadow memory \( \rightarrow \) Lockset index table

Lockset index table \( \rightarrow \) mu1, mu2

mu1 \( \rightarrow \) Lock vector
Common False Alarms - Annotations

- Memory reuse
- Private locks
- Benign races

```c
if (some_condition) {
    LOCK m DO
    if (some_condition)
        {stuff}
    END
}
```

- EraseReuse – resets shadow word to virgin state
- Lock annotations
- EraserIgnoreOn()
- EraserIgnoreOff()
Races inside OS

• Using interrupt system to provide mutual exclusion – this implicitly locks everything affected (by interrupt level specified)
  – Explicitly associate a lock with interrupt level – disabling interrupt is like acquiring that lock

• Signal and wait kind of synchronization
  – V to signal for P which waits -- semaphore not “held” by thread.
An OK Race in AltaVista

```c
if (p->ip_fp == (NI2_XFILE *) 0) {
    NI2_LOCKS_LOCK (&p->ip_lock);
    if (p->ip_fp == (NI2_XFILE *) 0) {
        p->ip_fp = ni2_xfopen (p->ip_name, "rb");
    }
    NI2_LOCKS_UNLOCK (&p->ip_lock);
}
```

// has file pointer been set?
// no? take lock for update
// was file pointer set
// since we last checked?
// no? set file pointer

```
    ...

    // no locking overhead if file
    // pointer is already set
```
Bad Race in Vesta

Combine::XorFP::FPVal( ) {
    if (!this->validFP) {
        NamesFP(fps, bv, this->fp, imap);
        this->validFP = true;
    }
    return this->fp;
}

This is a serious data race, since in the absence of memory barriers the Alpha semantics does not guarantee that the contents of the validFP field are consistent with the fp field.
Core Loop of Lock-Coupling

// ptr->lock.Acquire(); has been done before loop

while (next != null & key > next->data)
    {
    next->lock.Acquire();
    ptr->lock.Release();
    ptr=next;
    next=ptr->next;
    }

ptr = 2
next = 4
2 → 4 → 6 → 8 → null