Peer review - Eraser: A dynamic data race detector for multithreaded programs

María Fernanda Mora Alba - Maestría en Ciencias en Computación, ITAM

I. Introduction

Multithreaded programs are error prone but debugging is hard and takes time. Simple errors in synchronization can lead to data races that can take even months to track down. This has taken many programmers away from threads [1].

There exist two main approaches for data race detection: *static* or *dynamic*. The *static* approach is effective if all shared variables are static globals, while the dynamic allows for dynamically allocated shared variables. The first work in dynamic race detection was based on Lamport's *happens-before* relationship which allows a partial order of events on the concurrent threads. The problem with these detectors is that they require too much information making them inefficient and also its efficacy depends on the interleaving given by the scheduler. The second work is based on ensuring consistent locking discipline. *Eraser* falls under this category and promises to detect more races than detectors based on *happens-before* relationship [1].

II. The lockset algorithm

Before describing the algorithm behind *Eraser* we provide two key definitions provided by [1]:

- Lock: binary semaphore used for mutual exclusion. It has two operations: **lock** (owned by a thread) and **unlock** (only the owner can do this and it becomes available).
- Data race: when two concurrent threads access a shared variable v with at least one write and there is no mechanism to prevent simultaneous accesses.

The basic version of the algorithm seeks to ensure that a lock is held by any thread that tries to access any shared variable, thus "protecting" the variable. *Eraser* will infer the protection relationship during program execution as follows. For each shared variable v *Eraser* keeps a set of candidate locks $C(v)$ that have protected v so far in the computation (in the beginning $C(v)$ contains all locks). When a shared variable v is accessed $C(v)$ is updated with the intersection of $C(v)$ and the set of locks held by the current thread, thus refining the set. Intuitively, if $C(v) = \emptyset$ there is no lock that consistently protects v and a warning is issued.

However, there exist three common situations in which $C(v) = \emptyset$ but data races are nonexistent: 1. variable initialization (commonly done without a lock), 2. read-shared data (e.g. constants or final variables in which reads can always be accessed safely) and 3. read-write locks (multiple readers and a single writer). To address the first situation *Eraser* delays the refinement of $C(v)$ -and hence of warnings- until the shared variable v is initialized, which is completed when v is first accessed by a second thread. To address the second, races are only reported after a variable becomes write-shared by more than one thread. The third is resolved by removing locks exclusively in read mode -as they don't protect against a race-each time a write occurs.

III. Implementation

Eraser takes as input the machine code of a program and adds the necessary mechanisms to call the *Lockset algorithm* when needed. Before each *load* and *store* instruction it checks if there exist valid locks for the variable.

There is a table where the set of locks are kept with an associated index (indexes run small). Its entries are never modified and new indexes are created thorough the acquisition/release of locks or the intersection operation. As of the data, every 32-bit word in the heap or global data is considered a shared variable. Each has associated a *shadow word* with 30-bit for lockset index and 2-bits for state condition.

IV. Experience

Eraser was successfully tested on programs with common synchronization errors: HTTP server and indexing engine from AltaVista (the most notable result: it took only 30 minutes to detect the data races), Vesta cache server, Petal distributed disk system and programs written by undergraduate programmers.

V. Critic

- To obtain $C(v)$ at first, a swept of the entire program has to be done **before** the execution, at compile time. This is against *Eraser's* main claim: to run dynamically.
- *Eraser* causes a slowdown in programs of 10-30 but authors claim that this is low enough to leave programs runnable. However, a this reduced rate it is possible that some subtle race conditions don't show (because some threads don't execute at these new time conditions, for example) or in the worst case some programs may not even suffer from race conditions, making *Eraser* pointless.
- For false positives *Eraser* proposes annotations. But if the source code/compiler is unavailable and the false positives rate is high this could represent a problem.
- The paper doesn't discuss how *Eraser* handles multiple locks, which would increase *Eraser's* practical use.

References

[1] Savage, Stefan, et al. "Eraser: A dynamic data race detector for multi-threaded programs." ACM Transactions on Computer Systems (TOCS) 15.4 (1997): 391-411.