System Critique: Eraser

Jason Paulos

Introduction

Eraser is a runtime-analysis tool used to detect data races in multithreaded programs. It was designed in the 1990s as a joint effort between researchers from prominent universities and scientists from Digital Equipment Corporation [1]. As a result of this collaboration, Eraser demonstrates its utility in both academic environments and in large-scale production software [4]. It is therefore no surprise that Eraser was designed to be more correct and more scalable to a variety of applications than prior static-analysis attempts, such as Sun's lock_lint and the monitor and happens-before patterns [1.2].

The core of Eraser's data race detection is the examination of locks and shared variables. To Eraser, a shared variable is one that can be accessed from multiple threads and a data race occurs when such a variable is simultaneously read and modified from multiple threads in the absence of exclusive locks [1.1]. Eraser instruments binary programs at runtime to determine if such data races occur under normal operation of the program [1]. Since one of Eraser's primary design goals is correctness, its insights are powerful and have identified previously unknown race conditions in professional software [4.2]. However, Eraser's second design goal, scalability to many environments, falls short when one considers the huge variety of programming environments that have emerged since the release of Eraser in 1997.

Correctness

The designers of Eraser prioritized the correctness of Eraser above all. They decided it is better for Eraser to detect all possible data races, even those that do not affect the correctness of the program (benign races), than to allow the possibility of undetected races. This is evident in their decision to sacrifice the ability to support the multiple lock pattern, as it allowed the possibility of certain races to go undetected [5], as well as their decision to allow benign races to be reported by default [3.3].

Of course benign races can be forcibly ignored through the use of source code level annotations [3.3]. This idea that every race is cause for concern, unless it is specifically found to be benign and thus documented in the code through annotations, is a powerful one, since it enhances the credibility of Eraser's warnings. Yet there may be cases in which annotating a codebase for use with Eraser poses a significant barrier of entry, thus exemplifying a tradeoff between correctness and scalability.

Scalability

Here I interpret scalability as the ability for Eraser to be used on many different types of programs. As mentioned in the introduction, scalability is the second design goal of the project, yet Eraser does not scale to all programs and developer workflows equally. Since correctness is Eraser's top priority, it is a very powerful tool when it can be used effectively by developers [4]. Consider the following scenarios where the utility of Eraser is small or nonexistent due to its inability to scale to the situation's needs.

1. A program written in an interpreted language: All of the codebases Eraser was tested on were written in C or C++ and compiled directly to binary programs, so it is unclear if Eraser can work with interpreted languages, such as Java or Python [4]. While it may be assumed in 1997 that most multithreaded programs were probably written in C or C++, that is not an assumption that one can make today.

2. A binary program already injected with debugging instrumentation: Consider the deadlock-checking tool described in the paper [5]. If this existed as a separate tool that augmented a binary programming similar to Eraser, could Eraser also be used on the resulting binary[1]? If not, then Eraser proves to be more difficult to use than a static checking tool, since multiple such tools can be run in series easily. And if so, does this affect the quality of warnings produced by Eraser?

Theoretically Eraser could be applied to both of the above scenarios, but as it stands the current version of Eraser sacrificed scalability for simplicity, namely the requirement that Eraser instruments a binary program in order to analyze it at runtime [3]. In many cases it will be simple to apply Eraser to a program in this way since there are no changes to the middle of a program's compilation process, but as I have shown this runtime instrumentation may not always be possible.

Simplicity

Eraser employs its own algorithm, the Lockset algorithm, to detect races on shared variables. This algorithm is quite straightforward and works by determining which program locks are associated with each shared variable, issuing a warning if no lock is consistently used to control access to a variable [2]. Since the Lockset algorithm does not do any complex reasoning about the scheduling of threads and only considers

[1] Interestingly, if Eraser could properly instrument a binary already augmented by another tool, then could Eraser be applied twice to a binary as a way to test that Eraser does not produce its own races? Sadly the authors do not address this use case.

the locks held locally, a programmer can understand the cause of errors predicted by the Lockset algorithm almost immediately. This is evident by one of the authors of Eraser using the tool to identify and fix bugs in the Ni2 codebase in 30 minutes that took the original program author months to track down and fix [4.5].

In order to maintain such a simple algorithm, Eraser again sacrifices easy integration with many programs. This is because the Lockset algorithm relies on being able to identify and instrument all occurrences of locking, variable usage, and variable allocation in a program. While Eraser does instrument instances of standard library functions to trigger events in the Lockset algorithm, these were not sufficient in **any** application of Eraser mentioned in the paper [4]. Indeed, every application required the use of additional source code level annotations to help Eraser notice benign data races, custom memory reuse, or custom locking implementations [3.3]. Annotating a codebase adds another step before Eraser can be useful for program analysis.

Conclusion

The authors of Eraser have shown that their tool can be very powerful in detecting data races and can help programmers quickly improve their code when Eraser is used effectively. The simplicity of its Lockset algorithm and its overwhelming correctness have been proven to catch bugs in the production software of Digital Equipment Corporation as well as in undergraduate programming projects [4]. As such, the main flaw with Eraser is not the validity of its race detection, but rather its inability to scale for use in programs that vary significantly from those examined in the paper. The requirement that Eraser be applied to program binaries greatly hinders its applicability to interpreted programming languages, and the additional work needed to annotate a codebase before Eraser's warnings can be most useful creates a barrier of entry for most programs.

Works Cited

Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., & Anderson, T. (1997). Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems, 15*(4), 391-411.