6.033 Critique: Eraser

Word Count: 899

March 16th 2017

1 Introduction

Eraser is a tool for dynamic detection of race conditions in multithreaded programs. It ensures the presence of a consistent locking discipline by checking that shared-memory references are correctly protected by locks. While the system is augmented for certain scenarios, at a high level Eraser maintains a state of whether each variable is accessed by a single thread, multiple read-only threads, or multiple threads where at least one is read-write. When a variable is accessed, Eraser uses a variation of the *Lockset* algorithm, depending on the variable's state, to update the set of protecting locks to only contain those that were both previously in the protecting set and currently held on access. This data is stored by assigning each variable a shadow address which contains an index into a table that holds the sorted set of locks. If this set is empty, and the address has been accessed by multiple threads including a write, Eraser reports a potential race (2.2). Eraser, as a system, is algorithmically simple but difficult to utilize, efficient in terms of space but not time, and a reliable source of information regarding potential races. However, it has significant scalability and failure handling concerns. Security is not a relevant major design goal because Eraser runs on an isolated machine.

1.1 Simplicity

Much of Eraser's simplicity stems from its difference in approach compared to previous systems. The system works dynamically, which eliminates the need for static reasoning about the semantics of a program like an earlier approach called $lock-lint$. Additionally, it is simpler to check for data races with Eraser, which only needs to be run once, than the traditional $happens-before$ approach where the ordering of the scheduler matters, so each test must be run multiple times. Eraser is also simple in that it reasons about memory addresses instead of threads individually (1.2). In practice, however, using Eraser is not simple because the potential race conditions flagged by the system need to be evaluated by the user since the system often returns false positives. The algorithmic simplicity of the system comes at the expensive of user simplicity by shifting the handling of edge cases to the user.

1.2 Efficiency

Eraser's space efficiency stems from its storage implementation. When the protecting lock set of a variable is updated, instead of changing the set held at its index in the table, which would require storing identical sets multiple times, the index itself is updated. The system also caches the result of performing the *Lockset* algorithm to increase efficiency if the same update happens again and new calculations take limited space because the set of locks is already sorted making comparison easy (3.2). Furthermore, the size of the table is limited since the number of locks is usually small; for example, the Vesta Cache server is 30,000 lines of code but uses only 26 distinct locks (4.2). However, Eraser was not designed for time efficiency and the many procedure calls result in significant slow down of up to thirty times (3.3). This design trade-off was made because in 1997, when Eraser was developed, space was a very limited resource, so it was more essential to optimize for space than time.

1.3 Thoroughness

Eraser was designed to be especially thorough at detecting race conditions. While Eraser reports orders of magnitude more false positives than actual bugs, it can help point users in the direction of bugs they might have otherwise missed (4). Additionally, when race conditions are specifically introduced into a code base Eraser detects them well, even when it took engineers months to discover these bugs on their own (4.5). Limited false negatives, even at the expense of false positives, is in line with the original goal of reliable testing. Judging the false negative rate of Eraser beyond these experiments is difficult, but since Eraser is a testing tool, not a guarantee of correctness, it is a reliable indicator of potential errors (1.2).

1.4 Scalability

The Eraser system is scalable in that it works well on both small code bases produced by students and large code bases designed by engineers (4). However, Eraser does not scale well with respect to computation time, making it infeasible for systems that take significant time to run (3.2). Furthermore, Eraser was designed for a particular operating system and is not able to easily scale to users who use different operating systems (3). It's important to note that this scalability concern is a direct consequence of the designers' efforts to create a simple system.

1.5 Failure Handling

Another key downfall of Eraser is that there is no way to know when a race condition has gone undetected. On the other hand, the system is susceptible to false positives, often due to memory reuse (3.3). These present a challenge because users are able to manually suppress warnings, which could erroneously

be used in situations where race conditions exist, creating false negatives. Again, this drawback is a result of desired algorithmic simplicity.

1.6 Conclusion

Overall, Eraser, at the time it was created, was simple compared to previous methods, space efficient, and a thorough generator of potential race conditions. However, it also had scalability and failure handling shortcomings that existed as consequences of prioritizing simplicity. These concerns made Eraser better suited for sparing use by systems experts, after they had done their own careful analysis of potential race conditions, than a general-purpose debugging tool used by a wider audience. In modern times, since memory is no longer a precious resource, Eraser is obsolete compared to alternatives such as Valgrind, which capitalize on the advances Eraser made while reducing its limitations.

1.7 Works Cited

Savage, Stefan, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. "Eraser: A Dynamic Data Race Detector for Multithreaded Programs." ACM Transactions on Computer Systems 15.4 (1997): 391-411. Web.