6.033 Critique 2: Eraser

Jisoo Min

1 Introduction

Eraser is a debugging tool that dynamically enforces the locking discipline. It uses a binary modification system to implement the Lockset algorithm to the input program. [3]

The system designers aim to catch the races as thoroughly as possible while minimizing false alarms. To meet this objective, the design goals of simplicity and fault-tolerance (i.e., resilience to internal algorithmic fault) are emphasized over performance and scalability. In particular, simple procedure call mechanism and omission of details (e.g., inline monitoring code), lead to time and space overheads. [3.2] Security is less of a concern for Eraser because the results are explicitly returned for debugging purposes.

2 Background

Multi-threaded programming is difficult to manage, but is prevalent in modern applications. [1] Despite previous attempts to enforce the locking discipline, dynamic race detection for complex programs was difficult. For instance, Lamport’s happens-before algorithm approach was highly scheduler-dependent, and thus missed races in subtle cases. [1] Similarly, Hoare’s monitor concept only handled static globals, and Sun’s locklint required static reasoning. [1.2]

We shall focus this critique on simplicity and fault-tolerance of internal designs, and on performance and scalability analysis. Then security shall be briefly discussed.

3 System Design

3.1 Simplicity

The system relies heavily on simplicity of its internal design. It attempts to keep the logic, representations, and implementations simple.

3.1.1 Logic: Lockset Algorithm

With its definition of four new, simple states, Virgin (after new variable allocation), Exclusive (after data access), Shared (after read access), and Shared-Modified (after write access), Eraser only requires a single update per access for data race detection. The Lockset algorithm updates the candidate lock set $C(v)$ to the intersection of the set of the all previously held locks and the set of locks held by the current thread. [2] Eraser reports empty $C(v)$ of Shared-Modified state only. [2.2] This refined algorithm is not only maintains simplicity, but also catches data races and avoids some major false reports.

3.1.2 Representations

First, lockset index module is a notable, easy component that is associated with each variable in the program. It is maintained in a table module to keep track of sorted vectors, which represent unique sets of locks. [3.1] The set intersection $C(v)$ provided by the Lockset algorithm module is cached in the table. Both cache lookup and search in sorted vectors are simple. [3.1]
Second, program memory and shadow memory that use 32-bit word unit are critical representations needed in Eraser. Communication between program memory and shadow memory modules lead to the discovery of proper lockset index and lock vectors. [3.1] With the addresses, offset, and lockset index, several modules can communicate easily with one another.

3.1.3 Implementations

Eraser does not trade its simplicity for edge case handling. Although system designers acknowledge that Eraser may report false negatives on programs with multiple-lock system, they choose not to implement sets of sets of locks which is overly difficult. [5] It is more important for Eraser to detect races thoroughly, than to remove a few false reports with complicated implementations. Moreover, Eraser does not focus much on improving its performance for complex applications with deadlocks. Although some adjustments for deadlock accommodation is acknowledged to be useful, system designers are not concerned much about deadlock cases. [5]

3.2 Fault-Tolerance

The second major design goal of Eraser is fault-tolerance. Failure can be defined as the potential false reports. Therefore, all designs for false report minimization shall be categorized efforts for achieving the fault-tolerance goal.

Eraser defines three major types of false reports: memory reuse, private locks, and benign races. [3.3] For each cases, Eraser developed program annotations so that common false alarms are suppressed. [3.3]

Annotations are as following: [3.3]
- \texttt{EraserReuse(address, size)}
- \texttt{EraserReadLock(lock)}
- \texttt{EraserReadUnlock(lock)}
- \texttt{EraserWriteLock(lock)}
- \texttt{EraserWriteUnlock(lock)}
- \texttt{EraserIgnoreOn( )}
- \texttt{EraserIgnoreOff( )}

In the lockset index table, all vectors are put in sorted order. [3.2] This implementation detail can also be seen as an attempt to recover from failure in the case when cache fails.

3.3 Performance

Eraser has less emphasis on performance than on simplicity. As long as it can debug programs by detecting data races, speeding up is not much of a concern. [4.4] In other words, the design goal of performance is mainly on accuracy.

3.3.1 Accuracy

Accuracy aspect of performance is a strength of the system. Eraser was initially designed to perform better than previous models. It achieved its goal of more dynamic detection, greater thread-independence, and minimization of false negatives. [3.2]

3.3.2 Speed and Space (Effectiveness)

Eraser uses multiple space-consuming modules and slows down applications by a factor of 10 to 30. [3.2] Because Eraser focuses on catching the data races, procedure calls is necessary at every load and store instructions. [3.2] The slowdown is not an issue if Eraser is at least as fast as manual debugging. In the case of \texttt{Ni2} libraries, manual debugging took months, while Eraser spotted the data race in a few minutes. [4.5] Therefore, as long as the input programs are not crucially time sensitive, speed and space issues are not detrimental. Because Eraser is a debugging tool, slow monitoring technique is permissible.

3.4 Scalability

The system designers confirmed that Eraser performs well on programs of multiple scales. For instance, 5,000 lines of code of a lightweight HTTP server of AltaVista \texttt{mhtpd}, 20,000 lines of code of AltaVista indexing engine \texttt{Ni2} [4.1], 25,000 lines of code of distributed storage system \texttt{Petal} [4], and 30,000 lines of code of cache.
server **Vesta** [4.2] all proved that Eraser doesn’t fail under larger programs. In particular, testing on **Petal** [4] showed that Eraser can react to random lock accesses, and testing on undergraduate students’ coursework showed that Eraser is valuable for programs in the working stage. [4.4] As discussed in the performance section, speed is not of the interest.

### 3.5 Security

Eraser is a race detector. Its implementation details or detected data races are non-sensitive contents. Eraser primarily focuses on observing the lock and variable behaviors. Exposure of the representation to the clients is not a problem. Detected data races are meant to be revealed for debugging purposes.

### 4 Conclusion

Eraser dynamically detects data race in lock-based concurrency programs. Centered around its own four state definitions based on whether read or write has occurred, Eraser keeps track of candidate lock set and determines data races. It suppresses major false reports although miscellaneous false reports are left for humans to check manually. Because the focus of Eraser is to avoid missing any data races, its designs around simplicity and fault-tolerance is sufficient. Performance and scalability are not perfect by any means, but Eraser can still accurately perform its job within a reasonable amount of time compared to manual work.
References