Thread-Safety Violation Detection for Java Applications

Shiven Mian Prashanth Duggirala Nikhil Wadhwa
Alfred Liu
{smian,pduggirala,nwadhwa,asliu}@ucdavis.edu

February 7, 2020

1 Introduction

With the introduction of multi-core and other parallel hardware, there is an increased prevalence of concurrent programs which utilize such architectures. Unfortunately, writing correct concurrent programs is tricky - it's difficult for humans to think about all the vast number of potential interactions between processes, and this makes concurrent programs hard to get right in the first place [11]. Due to this, it is essential to identify concurrency bugs, which occur during parallel and concurrent execution of programs.

Concurrency bugs and research focusing on detecting them can be divided into two areas. Data race bugs [20, 14, 4, 5, 16] occur when a program concurrently executes two conflicting accesses to one shared object/variable without proper synchronization and lock protection. Deadlock bugs [19, 18] occur when two or more programs circularly wait for each other to release particular resources due to faulty implementation of resource locks. In this project, we particularly focus on a class of concurrency bugs called Thread-Safety Violations (TSVs), which are a generalization of data races to coarser-grained objects like libraries and classes.

Every library or class specifies an informal Thread Safety Contract (TSC), which specifies which methods of the library / class can and cannot be called concurrently. A Thread-Safety Violation (TSV) occurs when such a contract is violated, i.e two threads call conflicting methods on the same object at the same time. For example, in the following Java snippet:

```
// ArrayList list
// Thread 1
list.add(value1);
// Thread 2
list.add(value2);
```

since ArrayList is not synchronized by default, the 'add' function cannot be called concurrently on the same list, thus resulting in a TSV.

Finding concurrency bugs efficiently is an open research area, with several challenges:

1. **Detection:** It is not so intuitive to find these bugs, and even if found, the non-deterministic nature of concurrent programs makes it harder to reproduce them for further analysis [17]. Many bugs also occur in thread interleavings that do not show up in testing but do so in production, leading to large-scale outages [9].

2. **False Positives:** Even though it is important for such bugs to be detected, it is equally important to minimise False Positives. A small loss of time and productivity per user in chasing false bugs can multiply into huge productivity losses across an organization.

3. **Integration:** Modern codebases are maintained in build-and-run frameworks [1, 15], where thousands of machines build and run tests on software maintained by thousands of groups in an organisation. Tools for concurrency detection need to be agnostic to the underlying concurrency structure being used (be it multithreading, event-based or message passing) and the various different forms of synchronizations (locks, monitors, semaphores etc) so that they can be used across modules and teams with little configuration (like a push-button).

4. **Scalability:** Due to the volume of code and the hardware maintaining it, testing comes at a premium. It is infeasible for concurrency bug detection tools to run multiple consecutive tests to detect bugs and prune false positives, since it incurs a lot of overhead. Thus, for scalability it is important for such tools to have low resource overhead [10] i.e detect bugs with a small number of test runs.

A promising approach being researched for detecting data-race bugs with low false positives is active *delay injection*, utilised by tools like DataCollider[7], CTrigger [17] and RaceFuzzer[20]. This approach works by strategically delaying threads at certain locations during run time, in order to make them do conflicting accesses. However, such an approach requires a careful trade-off between the number of delays, and the cost of identifying optimal locations for delay injection. For example: RaceFuzzer and CTrigger identify buggy locations by analysing memory accesses and program synchronization and insert delays at

those locations, which may lead to small, optimum delay locations but potentially a high cost of deriving them. On the other hand, DataCollider conducts minimal analysis but requires a large number of delay injections and multiple runs of the same test to detect bugs. All these tools come with huge overhead and thus don't scale.

In this regard, Microsoft Research at SOSP'19 proposed TSVD [13], a Thread-Safety Violation Detector, which uses dynamic analysis and lightweight instrumentation to detect TSVs. The initial motivation behind this work was to try balancing the aforementioned trade off between the number of delay locations and cost of identifying them and exploit a middle ground, and also to specifically detect TSV concurrency bugs, which Microsoft found were increasingly prevalent in their production codebase, and were due to fairly common mistakes made by developers. TSVD is designed for scalability (especially for build-and-run frameworks), is independent of the program's concurrency or synchronisation mechanism, provides **no** false positives, and is able to detect more bugs than prior work, often in a **single** test run.

In brief, TSVD tries to monitor calling behaviors of thread-unsafe methods to identify 'likely' conflicting accesses at run-time, and then injects delays to drive the program towards thread-unsafe behavior and learns from its ability or inability to do so [13]. TSVD dynamically maintains a set, called trap set, containing 'dangerous' pairs of program locations that are likely to contribute to TSVs, and injects delays at these locations. These locations are derived using *near-miss tracking* of threads that perform access calls to the same object close to each other in real time. TSVD then uses Happens-Before (HB) inference (as opposed to HB analysis) to discard accesses ordered by the HB relation [12]. Since a delay in one such access would lead to a transitive delay in the other access(es) in the relation, TSVD avoids inserting delays at these locations. Finally, the trap set is kept persistent, and is directly used from one TSVD test run to another.

The original TSVD paper was written in the context of .NET applications (C#). For our project, we propose an implementation of TSVD for **Java** programs. This was a convenient language choice, given the widespread use of Java in various applications involving concurrent operations (web servlets, Android apps etc). We aim to implement the two parts of TSVD as described in the paper:

1. a runtime library, that implements the core algorithm.
2. an instrumentation agent for Java bytecode, to incorporate the runtime library and observe various events. The original TSVD implementation used Mono.Cecil [8] for doing the instrumentation; for now, we aim to use Java's built-in instrumentation API.

Along with this, we aim to benchmark our TSVD implementation on various Java multithreaded programs and compare TSVD's performance with the previous approaches discussed above. For now, we aim to use some of the benchmarks used in RaceFuzzer [20] (elaborated in Section 3), but if need be, we will also try to run TSVD on other multithreaded programs on GitHub, Android applications etc (which was also done by the original TSVD implementation [13]). Though our anticipated results depend on what metrics and benchmarks we use, we aim to deliver a Java implementation of TSVD which provides equivalent or more true-positive bug reports, and is more lightweight (i.e lesser number and cost of identifying delay locations, and less overhead) than previous tools for concurrency bug detection.

2 Timeline

Based on the previous section, which describes our proposed work and the motivations behind it, we propose the following tentative timeline for completing the different aspects of the project:

1. **Feb 10th - 16th:** We will start by understanding Java's multithreading model, and developing our Thread-safety Contracts and our instrumentation agent on top of Java's built-in instrumentation API. A few simple toy examples that replace the original functions with print statements could suffice to validate the idea. We will also focus on thread-unsafe classes (e.g., ArrayList, HashMap) in Java and manually identify conflicting write and read-APIs for building our thread-safety contracts.

2. **Feb 17th - 23rd:** We will start implementing the core runtime library. We would need to write functions which determine when and where to inject delays. We will develop the trap-set functionality. With that we can implement an algorithm which uses the thread safety contract to decide whether the trap functions should be called for a specific object and access call pair.

3. **Feb 24th - March 1st:** The next function will be the proxy method, which is essentially a wrapper around the access call. We will write unit tests for each function we develop. We will then start working on integrating our core functionality with the Java instrumentation agent, which we would have completed by this time.

4. **March 2nd - 8th:** We will carry out our evaluation plan, as outlined in Section 3 to analyze the functioning of our implementation and determine whether we achieve desired results. The first step in testing is to run programs written by us with known bugs and try to trigger the bugs by running our TSVD tool. Once

we see it works properly, we shall benchmark TSVD on the benchmarks described in Section 3. Our observations will be incorporated into our final presentation.

3 Evaluation Plan

The evaluation plan for our implementation of TSVD will be following the procedures outlined in the original paper [13] in terms of bug validation, comparison with existing techniques and evaluating different parameters.

- Since the original TSVD implementation by Microsoft was benchmarked on proprietary Microsoft code, we shall be using publicly available benchmarks and codebases for testing and evaluating TSVD. Initially, we aimed to explore the Multi-threaded benchmark from the Java Grande Benchmark [6] for testing, since it has been used in previous studies [20], however it is available only on request. Thus, for now we plan on using open libraries like cache4j [3], sor and hedc [21] for benchmarking TSVD. Since all these benchmarks are merely multithreaded Java programs, we will test TSVD on other Java codebases on GitHub if required.

- We will contrast TSVD with previous work on concurrency bug detection in Java (some of which were also used in [13]). For now, we plan on comparing TSVD with RaceFuzzer [20], RacerD [2] and DynamicRandom (a simpler variant of TSVD where every TSVD point is taken as an eligible delay location, and delays are injected at random times). If time permits, we will also compare with TSVD_{HB}, which uses TSVD with the costlier HB analysis instead of inference. For metrics, we first aim to compare basic statistics on bugs (i.e # of bugs found per run, unique bugs, bugs per class type etc), false positive reports, overhead and CPU consumption; after this, we plan on utilising the other metrics regarding HB-inference, near-miss tracking as described in [13] if time permits.

4 Anticipated Results

Our anticipated results may vary depending on what and how many of the benchmarks we end up using, how many classes we include in our thread-safety contract, and whether the benchmarks (especially the Java Grande Suite) contain bug reports if any. The bug reports are required for measuring False Positive rate; otherwise we can also self-examine and annotate the bug reports we get to see if there are false positives, similar to [13].

Overall, we aim to deliver an implementation of Microsoft's Thread-Safety Violation Detector for Java programs. At the very least, our framework should be able to detect equivalent or more true positive bug reports than previous works have managed for Java programs. Additionally, our framework should be lightweight, which necessitates that it should have:

1. Minimal cost of identifying locations to insert delays
2. Minimal number of delay locations
3. Small overhead during testing
4. Few or no false positive bug reports (if sufficient annotation is possible)

References

[1] Bazel. Bazel: a fast, scalable, multi-language and extensible build system: `https://bazel.build`.

[2] S. Blackshear, N. Gorogiannis, P. W. O'Hearn, and I. Sergey. Racerd: compositional static race detection. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–28, 2018.

[3] Cache4j. Cache4j: `http://cache4j.sourceforge.net/`.

[4] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 258–269, 2002.

[5] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. *ACM SIGOPS operating systems review*, 37(5):237–252, 2003.

[6] EPCC. Java grande benchmark suite: `https://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/java-grande-benchmark-suite`.

[7] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective data-race detection for the kernel. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, page 151–162, USA, 2010. USENIX Association.

[8] J. Evain. Mono.cecil: `https://github.com/jbevain/cecil`.

[9] S. Focus. Software bug contributed to blackout: http://www.securityfocus.com/news/8016, 2004.

[10] D. Gupta, K. V. Vishwanath, M. McNett, A. Vahdat, K. Yocum, A. Snoeren, and G. M. Voelker. Diecast: Testing distributed systems with an accurate scale model. *ACM Trans. Comput. Syst.*, 29(2), May 2011.

[11] D. Hovemeyer and W. Pugh. Finding concurrency bugs in javaio. *Papers Presented at the 2004 Workshop on Concurrency and Synchronization in Java Programs*, page 80, 2004.

[12] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[13] G. Li, S. Lu, M. Musuvathi, S. Nath, and R. Padhye. Efficient scalable thread-safety-violation detection: finding thousands of concurrency bugs during testing. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 162–180, 2019.

[14] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, page 329–339, New York, NY, USA, 2008. Association for Computing Machinery.

[15] Maven. Apache maven: https://maven.apache.org/.

[16] R. H. Netzer and B. P. Miller. Improving the accuracy of data race detection. In *Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 133–144, 1991.

[17] S. Park, S. Lu, and Y. Zhou. Ctrigger: exposing atomicity violation bugs from their hiding places. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, pages 25–36, 2009.

[18] M. Prvulovic and J. Torrellas. Reenact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.*, pages 110–121. IEEE, 2003.

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

[20] K. Sen. Race directed random testing of concurrent programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 11–21, 2008.

[21] C. Von Praun and T. R. Gross. Object race detection. *Acm Sigplan Notices*, 36(11):70–82, 2001.