CalFuzzer: An Extensible Active Testing Framework for Concurrent Programs

Pallavi Joshi[1], Mayur Naik[2], Chang-Seo Park[1], and Koushik Sen[1]

[1] University of California, Berkeley, USA
{pallavi,parkcs,ksen}@eecs.berkeley.edu
[2] Intel Research mayur.naik@intel.com

Abstract. Active testing has recently been introduced to effectively test concurrent programs. Active testing works in two phases. It first uses predictive off-the-shelf static or dynamic program analyses to identify potential concurrency bugs, such as data races, deadlocks, and atomicity violations. In the second phase, active testing uses the reports from these predictive analyses to explicitly control the underlying scheduler of the concurrent program to accurately and quickly discover real concurrency bugs, if any, with very high probability and little overhead. In this paper, we present an extensible framework for active testing of Java programs. The framework currently implements three active testers based on data races, atomic blocks, and deadlocks.

1 Introduction

Multi-threaded programs often exhibit incorrect behavior due to unintended interference between threads. These concurrency bugs—such as data races and deadlocks—are often difficult to find using conventional testing techniques because they typically happen under very specific thread interleavings. Testing done in a particular environment often fails to come up with interleavings that could happen in other environments, such as under different system loads. Moreover, testing depends on the underlying operating system or the virtual machine for thread scheduling—it does not try to explicitly control the thread schedules; thus, it often ends up executing the same interleaving repeatedly.

Numerous program analysis techniques [2, 4, 1] have been developed to *predict* concurrency bugs in multi-threaded programs by detecting violations of commonly used synchronization idioms. For instance, accesses to a memory location without holding a common lock are used to predict data races on the location, and cycles in the program's lock order graph are used to predict deadlocks. However, these techniques often report many false warnings because violations of commonly used synchronization idioms do not necessarily indicate concurrency bugs. Manually inspecting these warnings is often tedious and error-prone.

Recently, we have proposed a new technique for finding *real* bugs in concurrent programs, called *active testing* [7, 5, 3]. Active testing uses a randomized thread scheduler to verify if warnings reported by a predictive program analysis are real bugs. The technique works as follows. Active testing first uses an existing predictive off-the-shelf static or dynamic analysis, such as Lockset [6, 4], Atomizer [1], or Goodlock [2], to compute potential concurrency bugs. Each such potential bug is identified by a set of program statements. For example, in the case of a data race, the set contains two program statements that could potentially race with each other in some execution. For each potential concurrency bug, active testing runs the given concurrent program under random schedules. Further, active testing biases the random scheduling by pausing the execution of any thread when the thread reaches a statement involved in the potential concurrency bug. After pausing a thread, active testing also checks if a set of paused threads could exhibit a real concurrency bug. For example, in the case of a data race, active testing checks if two paused threads are about to access the same memory location and at least one of them is a write. Thus, active testing attempts to force the program to take a schedule in which the concurrency bug actually occurs. In previous work, we have developed active testing algorithms for detecting real data races, atomicity violations, and deadlocks.

In this paper, we describe an extensible tool for active testing of concurrent Java programs, called CalFuzzer. CalFuzzer provides a framework for implementing both predictive dynamic analyses and custom schedulers for active testing. We elaborate upon each of these next.

CalFuzzer provides a framework for implementing various predictive dynamic analyses to obtain a set of program statements involved in a potential concurrency bug. We have implemented three such techniques in CalFuzzer: a hybrid race detector [4], the Atomizer algorithm [1] for finding potential atomicity violations, and iGoodlock [3] which is a more informative variant of the Goodlock algorithm [2] for detecting potential deadlocks. More generally, CalFuzzer provides an interface and utility classes to enable users to implement additional such techniques.

CalFuzzer also provides a framework for implementing custom schedulers for active testing. We call these custom schedulers *active checkers*. We have implemented three active checkers in CalFuzzer for detecting real data races, atomicity violations, and deadlocks. More generally, CalFuzzer allows users to specify an arbitrary set of program statements in the concurrent program under test where an active checker should pause. Such statements may be thought of as “concurrent breakpoints”.

We have applied CalFuzzer to several real-world concurrent Java programs comprising a total of 600K lines of code and have detected both previously known and unknown data races, atomicity violations, and deadlocks. CalFuzzer could easily be extended to detect other kinds of concurrency bugs, such as missed notifications and atomic set violations.

2 The Active Testing Framework

In this section, we give a high-level description of our active testing framework. We consider a concurrent system composed of a finite set of threads. Given a concurrent state s, let $\texttt{Enabled}(s)$ denote the set of transitions that are enabled in the state s. Each thread executes a sequence of transitions and communicates

Algorithm 1 CalFuzzer with user defined `analyze` and `check` methods

```
 1: Inputs: the initial state $s_0$ and a set of transitions breakpoints
 2: paused := $\emptyset$
 3: $s := s_0$
 4: while Enabled($s$) $\neq \emptyset$ do
 5:    $t$ := a random transition in Enabled($s$) $\setminus$ paused
 6:    analyze($t$)
 7:    if $t \in$ breakpoints then
 8:       paused := check($t$, paused)
 9:    end if
10:    if $t \notin$ paused then
11:       $s$ := Execute($s, t$)
12:    end if
13:    if paused = Enabled($s$) then
14:       remove a random element from paused
15:    end if
16: end while
17: if Alive($s$) $\neq \emptyset$ then
18:    print "ERROR: system stall"
19: end if
```

with other threads through shared objects. We assume that each thread terminates after the execution of a finite number of transitions. A concurrent system evolves by transitioning from one state to another state. If s is a concurrent state and t is a transition, then `Execute`(s, t) executes the transition t in state s and returns the updated state.

The pseudo-code in Algorithm 1 describes the CalFuzzer algorithm. The algorithm takes an initial state s_0 and a set of transitions (denoting a potential concurrency bug), called *breakpoints*, as input. The set of transitions *paused* is initialized to the empty set. Starting from the initial state s_0, at every state, CalFuzzer randomly picks a transition enabled at the state and not present in the set *paused*. It then calls the user defined method `analyze` to perform a user defined dynamic analysis, such as Lockset, Atomizer, or Goodlock. The `analyze` method can maintain its own local state; for example, the local state could maintain locksets and vector clocks in the case of hybrid race detection. If transition t is in the set *breakpoints*, CalFuzzer invokes the user defined method `check`, which takes t and the *paused* set as input and returns an updated *paused* set. The `check` method could be used to implement various active checkers. A typical implementation of the `check` method could add t to the *paused* set and remove some transitions from the *paused* set. After the invocation of `check`, CalFuzzer executes the transition t if it has not been added to the *paused* set by the `check` method. At the end of each iteration, CalFuzzer removes a random transition from the *paused* set if all the enabled transitions have been paused. The algorithm terminates when the system reaches a state that has no enabled transitions. At termination, if there is at least one thread that is alive, the algorithm reports a system stall.

Algorithm 2 The `check` method for active testing of data races

```
1:  Inputs: transition $t$ and a set of transitions paused
2:  if $\exists t' \in$ paused s.t. $t$ and $t'$ access same location and one is a write then
3:      print "Real data race between $t$ and $t'$" (* next resolve race randomly
        to check if something could go wrong due to the race *)
4:      if random() then
5:          add $t$ to paused and remove $t'$ from paused
6:      end if
7:  else
8:      add $t$ to paused
9:  end if
10: return paused
```

3 An Example Instantiation of the Framework

CalFuzzer takes two user defined methods: `analyze` and `check`. In order to implement an active testing technique, one needs to define these two methods. For example, an active testing technique for data races [7] would require us to implement the hybrid race detection algorithm [4] in the `analyze` method and a `check` method as shown in Algorithm 2. Recall that the `check` method takes an enabled transition t in *breakpoints* and the set of paused transitions, *paused*, as input. If there exists a transition t' in *paused* such that both t and t' access the same memory location, and one of them is a write to that location, then we have exhibited a thread schedule which has a real race, namely the race between transitions t and t'. In principle, we could stop at this point, but we go further and determine if this race is benign or harmful (e.g. causes an exception). For this purpose, we randomly decide whether we want t' to execute before t, or vice versa. If `random()` returns true, then we let t' to execute before t, by adding t to *paused* and removing t' from it. Otherwise, we let t to execute before t'. Since t' is already paused, we keep it paused, and let t execute.

4 Implementation Details

We have implemented the CalFuzzer active testing framework for Java. CalFuzzer (available at `http://srl.cs.berkeley.edu/~ksen/calfuzzer/`) instruments Java bytecode using the SOOT compiler framework [8] to insert callback functions before or after various synchronization operations and shared memory accesses.[3] These callback functions are used to implement various predictive dynamic analyses and active checkers. Each predictive dynamic analysis

[3] We decided to instrument bytecode instead of changing the Java virtual machine or instrumenting Java source code because Java bytecode changes less frequently than JVM and Java source may not be available for libraries.

implements an interface called `Analysis`. The methods of these interface implements the `analyze` function in Algorithm 1. Likewise, each active checker is implemented by extending a class called `ActiveChecker` which implements the `check` functions in Algorithm 1. The methods of these two classes are called by the callback functions.

The framework provides various utility classes, such as `VectorClockTracker` and `LocksetTracker` to compute vector clocks and locksets at runtime. Methods of these classes are invoked in the various callback functions described above. These utility classes are used in the hybrid race detection [4] and iGoodlock [3] algorithms; other user defined dynamic analyses could also use these classes.

The instrumentor of CalFuzzer modifies all bytecode associated with a Java program including the libraries it uses, except for the classes that are used to implement CalFuzzer. This is because CalFuzzer runs in the same memory space as the program under analysis. CalFuzzer cannot track lock acquires and releases by native code and can therefore go into a deadlock if there are synchronization operations inside uninstrumented classes or native code. To avoid such scenarios, CalFuzzer runs a low-priority monitor thread that periodically polls to check if there is any deadlock. If the monitor discovers a deadlock, then it removes one random transition from the *paused* set.

CalFuzzer can also go into livelocks. Livelocks happen when all threads of the program end up in the *paused* set, except for one thread that does something in a loop without synchronizing with other threads. We observed such livelocks in a couple of our benchmarks including `moldyn`. In the presence of livelocks, these benchmarks work correctly because the correctness of these benchmarks assumes that the underlying Java thread scheduler is fair. In order to avoid livelocks, CalFuzzer creates a monitor thread that periodically removes those transitions from the *paused* set that are waiting for a long time.

5 Results

Table 1 summarizes some of the results of running active testing on several real-world concurrency Java programs comprising a total of 600K lines of code. Further details are available in [7, 5, 3]. Note that the bugs reported by the active checkers (RaceFuzzer, AtomFuzzer, and DeadlockFuzzer) are real, whereas those reported by the dynamic analyses (hybrid race detector, Atomizer, and iGoodlock) could be false warnings. Although active testing may not be able reproduce some real bugs, all previously known real bugs were reproduced, with the exception of AtomFuzzer (see [5] for a discussion on its limitations).

The runtime overhead of CalFuzzer is from 1.1x to 20x. Normally, the slowdown is low since only the synchronization points and memory accesses of interest are instrumented. However, in some cases the slowdown is significant—this is caused when CalFuzzer pauses redundantly at an event. We use precise abstractions [3] to distinguish relevant events, which lessens redundant pauses.

6 Conclusion

CalFuzzer provides a framework for implementing predictive dynamic analyses to find potential concurrency bugs and custom randomized schedulers, called active checkers, to automatically verify if they are real bugs. We have implemented

		Avg. runtime(s)				Number of reported bugs								
Benchmark	LoC	Norm	RF	DF	AF	HRD	RF	KR	iG	DF	KR	Az	AF	KR
moldyn	1,352	2.07	42.4	-	-	5	2	0	0	-	-	-	-	-
jspider	10,252	4.62	4.81	-	51	29	0	-	0	0	-	28	4	0
sor	17,718	0.163	0.23	-	1.0	8	0	0	0	0	-	0	0	0
hedc	25,024	0.99	1.11	-	1.8	9	1	1	0	0	-	3	0	1
DBCP	27,194	0.60	-	1.4	-	-	-	-	2	2	2	-	-	-
jigsaw	160,388	-	-	-	-	547	36	-	283	29	-	60	2	1
Java Swing	337,291	4.69	-	28.1	-	-	-	-	1	1	1	-	-	-

Table 1. Average execution time and number of bugs reported for each checker implemented with the CalFuzzer framework (LoC: Lines of Code, Norm: Uninstrumented code, RF: RaceFuzzer, DF: DeadlockFuzzer, AF: AtomFuzzer, HRD: Hybrid Race Detection, KR: Previously known real bugs, iG: iGoodlock, Az: Atomizer)

three active checkers in this framework for detecting data races, atomicity violations, and deadlocks. We have shown the effectiveness of these checkers on several real-world concurrent Java programs comprising a total of 600K lines of code. We believe CalFuzzer provides a simple extensible framework to implement other predictive dynamic analyses and active checkers.

Acknowledgements. This research was supported in part by a generous gift from Intel, Microsoft and Intel funding (award #20080469), matching funding by U.C. Discovery (award #DIG07-10227), and NSF Grant CNS-0720906.

References

1. C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 256–267, 2004.
2. K. Havelund. Using runtime analysis to guide model checking of java programs. In *7th International SPIN Workshop on Model Checking and Software Verification*, pages 245–264, 2000.
3. P. Joshi, C.-S. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09) (to appear)*, 2009.
4. R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 167–178. ACM, 2003.
5. C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 135–145. ACM, 2008.
6. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.
7. K. Sen. Race directed random testing of concurrent programs. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 11–21, New York, NY, USA, 2008. ACM.
8. R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *CASCON 1999*, pages 125–135, 1999.