# Using Delta Debugging to Minimize Stress Tests for Concurrent Data Structures

Jing Xu*, Yu Lei*, Richard Carver

*Dept. of Computer Science and Engineering, University of Texas at Arlington, Arlington, TX 76019, USA*

***Abstract*—Concurrent data structures are often tested under stress to detect bugs that can only be exposed by some rare interleavings of instructions. A typical stress test for a concurrent data structure creates a number of threads that repeatedly invoke methods of the target data structure. After a failure is detected by a stress test, developers need to localize the fault causing the failure. However, the execution trace of a failed stress test may be very long, making it time-consuming to replay the failure and localize the fault.**

**In this paper, we present an approach to minimizing stress tests for concurrent data structures. Our approach is to create a smaller test that still produces the same failure by removing some of the threads and/or method invocations in the original stress test.. We apply delta debugging to identify the threads and method invocations that are essential for causing the failure. Other threads and method invocations are removed to create a smaller stress test. To increase the chance of triggering the original failure during the execution of the new stress test, we force the new execution to replay the original failed execution trace when possible, and try to guide the execution back to the failed trace when the execution diverges. We describe a tool called *TestMinimizer* and report the results of an empirical study in which *TestMinimizer* was applied to 16 real-life concurrent data structures. The results of our evaluation showed that *TestMinimizer* can effectively and efficiently minimize the stress tests for these concurrent data structures.**

***Keywords*—stress testing, minimization, concurrent data structures, delta debugging, execution replay.**

## I. Introduction

A concurrent data structure stores and organizes data that is accessed by multiple computing threads (or processes) [3]. As multi-core processors become the dominant computing platform, it is important to ensure the correctness of concurrent data structures, which play a critical role in the behaviour of concurrent threads. Some bugs in concurrent data structures, however, are hard to expose, since they can be exposed only by certain, rare interleavings of instructions [1].

Stress testing is often employed to test a concurrent data structure so that rare interleavings can be exercised. A stress test driver, or simply a stress test, involves multiple threads that repeatedly invoke methods of the target data structure. After a failure is detected by a stress test, developers need to localize the fault. However, the execution trace of a failed stress test may contain a large number of execution events. This makes it time consuming to replay the failure and localize the fault. If the size of failed execution traces can be reduced, then faults can be localized faster and more easily.

A straightforward approach to minimizing a stress test for a concurrent data structure is to remove one or more threads and/or method invocations from the original, failed stress test and let the new test execute non-deterministically. We refer to the set of threads and/or method invocations that are removed from the original test as the *removal set*. If the new test execution results in the same failure (due to the same failed assertion or thrown exception) as the original execution, then the new, smaller test execution can be used to localize the fault. Otherwise, a different removal set can be identified and used to derive a new test that results in the same failure.

There are two major challenges to be addressed in order to make the above approach effective and efficient. The first challenge is how to identify the removal set in a systematic manner such that a smaller stress test that reproduces the original failure can be quickly obtained. Our approach uses delta debugging, which employs a binary search procedure to systematically identify the threads and/or method invocations that can be removed from the original test [2].

The second challenge is how to deal with the non-deterministic nature of concurrently executing threads. That is, when we execute the new, reduced test, which involves the execution of concurrent threads, it may not execute an interleaving that repeats the original failure, even though such an interleaving is possible. To address this problem, we use the original, failed execution trace to guide the execution of the reduced test towards repeating the failure. Since the reduced test will not execute the events in the removal set, we remove them from the original failed execution trace. We refer to the resulting new execution trace as the retained trace. A guided execution follows the retained trace as closely as possible.

It is possible, however, that some event in the retained execution trace cannot be executed when this event is reached during the reduced test execution, either because this event cannot be exercised at all in the execution or because this event can only be exercised at some point later in the execution. When this happens, our guided execution analyzes the retained execution trace to determine whether some of the events in the retained trace should be skipped, or whether some extra events should be added to the execution. The modifications made to the retained trace increase the chance that the guided execution repeats the failure of the original execution. We refer to the trace of the guided execution as the guided execution trace. Note that we do not explore all of the possible interleavings for a reduced test *r*; we only consider the one interleaving generated by the guided execution. The guided execution of *r* may not repeat the original failure. If this happens, a larger stress test may be generated. However, it is possible that a different interleaving for *r* repeats the failure. In this case, the reduced stress test that is ultimately generated may not be minimal.

Our approach has been implemented in a tool called *TestMinimizer*. This tool was used to perform an empirical study of stress tests for 16 Java concurrent data structures. The first nine are faulty concurrent data structures that were used in [4]. These nine concurrent data structures were developed by students in a programming course. Note that these data structures do not come with stress tests. We added stress tests for them following the common stress testing design for current data structure. We found additional data structures in Github by searching for "concurrent data structure", "Java", and "stress test". Our query was matched in 20 projects, seven of which had stress tests. For these seven matching projects, we selected the first matching concurrent data structure. The results of our study show that our approach can significantly reduce the number of threads and method invocations in failed stress tests for concurrent data structures. In particular, for 14 out of these 16 concurrent data structures, our approach was able to reduce the size of the stress tests so that they contained no more than four threads and no more than five methods. The total time used to perform the minimization process was less than 10 minutes.

The rest of the paper is organized as follows. Section 2 describes a common design for stress testing concurrent data structures and shows an example program. Section 3 presents our execution model, which is used to perform deterministic test executions. Section 4 describes the delta debugging-based framework for identifying the removal set. Section 5 presents our guided execution control technique. Section 6 presents the design and implementation of *TestMinimizer*. Section 7 reports the results of our empirical study on the 16 Java concurrent data structures. Section 8 reviews related work. Section 9 provides concluding remarks and discusses our plan for future work.

## II. STRESS TESTS FOR CONCURRENT DATA STRUCTURE

To determine how developers write stress tests for concurrent data structures, we examined stress tests in the Github project [5]. A search for "concurrent data structure", "Java", and "stress test" produced 20 project results, seven of which had stress tests. All of the stress tests had the following design ─ multiple threads are created and each uses the target data structure by repeatedly invoking public methods of the data structure. This is the common design for stress testing the concurrent data structure. It was also used in the Java concurrency utility package (i.e., *java.util.concurrency*) [6]. Note that, typically, all of the threads access a single instance of the target data structure. When some public methods need to use more than one instance of the target data structure, e.g., as method parameters, multiple instances may be created.

Figure 1 shows an example stress test from the *ConTest* benchmark programs [7]. Class *Account* has one field *amount*, which is the current balance in the account, and three *synchronized* methods *deposit*, *withdraw,* and *transfer*. When one thread is executing a *synchronized* method for an *Account* object, all other threads that invoke any *synchronized* method on the same *Account* object block (suspend execution) until the first thread is done executing its method.

```
class Account {
    int amount;

    public Account(int amnt) {
        amount = amnt;
    }

    synchronized void deposit(int money) {
        amount += money;
    }

    synchronized void withdraw(int money) {
        if (amount >= money) {
            amount -= money;
        }
    }

    synchronized void transfer(
      Account ac, int money) {
        if (amount >= money) {
            amount -= money;
            ac.amount += money;
        }
    }
}

class AccountStressTest {
    Account[] accounts;

    public static void main(String[] args) {
      accounts = new Account[3];
      for (int i = 0; i < 3; i++)
        accounts[i] = new Account(0);
      Thread[] threads = new Thread[3];
      for (int i = 0; i < 3; i++) {
        threads[i] = new Thread() {
          public void run() {
            for (int j = 0; j < 3; j++) {
              int methodID = Random.nextInt(3);
              Account account =
                accounts[Random.nextInt(3)];
              Account dest_account =
                accounts[Random.nextInt(3)];
              if (methodID == 0)
                account.deposit(400);
              else if (methodID == 1)
                account.withdraw(100);
              else if (methodID == 2)
                account .transfer(200, dest_account));
            }
          }
        };
      }
      for (int i = 0; i < 3; i++)
        threads[i].start();
      for (int i = 0; i < 3; i++)
        threads[i].join();
    }
}
```

Figure 1. A Motivating Example

Method *deposit* adds *money* to the account; method *withdraw* withdraws *money* from the account if *money* is less than or equal to *amount*; and method *transfer* transfers *money* from a source account to a destination account if *money* is less than or equal to the *amount* in the first account. In the stress test, three threads and three *Account* instances are created. Each thread randomly calls methods *deposit*, *withdraw*, and *transfer* on accounts that are randomly chosen from the three Account instances. Note that since this example also serves as a running example for our approach, the test does not stress the data structure heavily for simplifying the explanation in the following sections.

Assume that during the execution of the stress test, each thread threads[*i*] deposits $400 into and then withdraws $100 from *Account accounts[i]*, and then transfers $200 to the next *Account accounts[(i+1)%3]*. The balance in each account is expected to be $300 at the end of the test. However, the test may fail with a final balance of $100 in *accounts[i],* instead of $300. This is because method *transfer* makes an unsynchronized, direct access to *ac.amount*, where *ac* is the destination *Account* object passed to method *transfer*.

## III. Execution model

In this section, we present our execution model that is adapted from the execution model in Lei's and Carver's work [10]. This execution model provides sufficient information for replaying an execution. We focus on monitor-based programs, since monitors are the main synchronization construct provided in Java. Replay techniques for monitor-programs have already been developed [11][12]. Our execution model contains all the information required by these techniques and some additional information that is required by our stress testing technique.

A monitor is a high-level synchronization construct that supports data encapsulation and information hiding. The data members of a monitor represent shared data. Threads communicate by calling public monitor methods that read and write the shared data

At most one thread is allowed to execute inside a monitor at any time. A monitor has an *entry* queue that holds the calling threads that are waiting to enter the monitor. Conditional synchronization is achieved using operations a*wait()* and *signal*() on *condition* variables. In a Java-style monitor, a thread that executes *a signal* operation continues to execute inside the monitor. The signaled thread joins the entry queue and thus competes with other threads trying to (re)enter the monitor.

As described in Section II, executions of a stress test involve multiple threads. Each thread manipulates the target data structure by invoking, in a loop, the data structure's public monitor methods. During execution, several types of execution events are captured and recorded.

The format of an event is (*event type*, *thread ID*, *name*, *iteration ID* (*method ID*)), where *event type* is the type of event, *thread ID* is the ID of the thread that executed the event, *name* is the name of the method executed, or the shared variable accessed, depending on the event type, and *iteration ID* indicates which loop iteration was being performed by the executing thread when it executed the event. In each iteration, one public method is invoked. Thus, the *iteration ID* also serves as the *method ID*. *Event type*, *thread ID*, and *name* are required by the replay technique [11], which tracks all of the synchronization actions and operations on shared variables that are exercised during an execution.

The valid event types are: *enterMonitor, exitMonitor, enterMethod*, *exitMethod*, *wait*, *notify*, *reenterMonitor*, *read*, and *write*. Events *enterMonitor* and *exitMonitor* occur when a thread enters or exits a synchronized monitor method or a synchronized block. Events *enterMethod* and *exitMethod* occur when a thread enters and exits any public method, including monitor methods. Event *reenterMonitor* occurs when a thread reenters a monitor after being notified. Events *enterMethod* and *exitMethod* are needed for grouping all of the events that are exercised by a specific method, when we try to remove methods. All the other event types are required by the replay technique [11] to trace and replay all of the synchronization actions and operations on shared variables during an execution.

For *enterMonitor*, *exitMonitor, enterMethod* and *exitMethod*, *name* is the name of the monitor object and the name of the method invoked on the object, e.g., "accounts[0].deposit" in Fig. 1. For the other event types, *name* includes the name of the monitor object, the name of the condition variable or shared variable, and the name of the operation (*wait*, *notify*, read, or write) performed on the variable, e.g., from Fig. 1: "accounts[1].amount:read".

For the example program in Fig. 1, the events that are exercised when Thread 1 executes *accounts[0].deposit()* on its first iteration are:

(entermethod,1,accounts[0]:deposit,1)
(entermonitor,1,accounts[0]:deposit,1)
(read,1,accounts[0].amount:read,1)
(write,1,accounts[0].amount:write,1)
(exitmonitor,1,accounts[0]:deposit,1)
(exitmethod,1,accounts[0]:deposit,1).

## IV. The Delta Debugging-based Framework

Assume that the execution of a stress test *s* detects a failure in a concurrent data structure. Our objective is to use delta debugging [2] to remove threads and/or method invocations from *s* to create a stress test that is smaller, but that still detects the same failure.

In this section, we first give an overview of the delta debugging approach. Then we present a framework that applies delta debugging to identify failure inducing threads and methods more efficiently than randomly choosing which threads and method invocations to remove.

### *4.1 Overview of delta debugging*

Delta debugging is an automated debugging approach based on systematic testing. With delta debugging, we can automatically find failure-inducing factors, such as program inputs, changes to the program code, or program events.

The basic idea is to identify the failure-inducing factors from a set of possible factors *c* using a binary search. If *c* contains only one factor, this factor is failure-inducing. Otherwise, we try to narrow down which of the factors in *c* are failure-inducing by partitioning *c* into two subsets *c*1 and *c*2, and generating one test with the factors in *c1* and another test with the factors in *c2*. If either the test for *c1* or *c2* results in a failure, then we continue the narrowing process on one that results in a failure. If neither of the tests results in a failure, it means that neither test contains enough of the failure-inducing factors to cause a failure. If neither of the tests results in a failure, which means the failure-inducing factors are in both halves, we must search in both halves—with all changes in the other half remaining applied, respectively. Thus, the new tests that are generated will contain factors from both halves. Let *n* be the size of *c*, i.e., the number of possible factors, the worst case complexity of this technique is $O(n^2)$ and the best case complexity is $O(\log n)$ [2].

Figure 2 shows an example execution of delta debugging. Assume that eight events are exercised in a failed execution. In the first step, only the first 4 events are included and the system passes. This indicates that the failure cannot be reproduced if the last 4 events are all removed. In the second step, only the last 4 events are included and the system passes once again. This indicates that we cannot remove all of the first 4 events. Since both halves could not be removed, both halves should be processed recursively by delta debugging. In the third step, we keep the first half of the event set in step 1, i.e., E1 and E2. Since the execution in step 3 passes, we continue to try to keep the second half of the set of step 1, i.e., E3 and E4. Since the execution in step 4 fails, E1 and E2 can be removed. Then we continue to test whether E3 or E4 could be removed. Since the execution in step 5 fails, E4 could be removed. The recursive call for the first half of the original event subsequence is done. Next the second half of it is done by step 6-8 in the same manner. At the end of this process, we find that the set consisting of E3 and E6 also triggers the failure.

| Step | Event subsequence | | | | | | | | Test |
|---|---|---|---|---|---|---|---|---|---|
| 1 | E1 | E2 | E3 | E4 | | | | | P |
| 2 | | | | | E5 | E6 | E7 | E8 | P |
| 3 | E1 | E2 | | | E5 | E6 | E7 | E8 | P |
| 4 | | | E3 | E4 | E5 | E6 | E7 | E8 | F |
| 5 | | | E3 | | E5 | E6 | E7 | E8 | F |
| 6 | | | E3 | | E5 | E6 | | | F |
| 7 | | | E3 | | E5 | | | | P |
| 8 | | | E3 | | | E6 | | | F |
| result | | | E3 | | | E6 | | | |

Figure 2. An Example Execution of Delta Debugging

### *4.2 The Framework*

The failure-inducing factors for the stress test of a concurrent data structure are the threads and the method invocations performed by each thread. Using delta debugging, we can first find failure-inducing threads and then find failure-inducing method invocations in each failure-inducing thread.

The first problem we need to address is how to remove threads or method invocations from the original stress test (driver). Threads and method invocations are removed by instrumenting the original stress test so that attempts to start a thread or invoke a method are controlled by a special *RemoveController* object. The *RemoveController* reads the IDs of the threads and methods in the removal set. Statements in the program that start a thread are preceded by a call to the *RemoveController* method *isRemovedThread(int threadID)*, which returns true if the thread should be removed and hence should not be started. Statements that invoke a method are preceded by a call to *isRemovedMethod(int threadID, int methodID)*, which returns true if the method invocation should not be made.

The following shows the instrumentation for the program in Fig. 1. The grey code is the added instrumentation code. An execution of the instrumented program will not start the threads or invoke the methods in the removal set. Note that the randomly generated values, e.g., the id of the *Account* instance and the id of the method to be called in Fig. 1, are traced and read from a file which records the randomly generated values.

```
    ...
    public void run() {
      for (int j = 0; j < 3; j++) {
        if (removingControl.isRemovedMethod(i, j))
          continue;
        ...
      }
    }
    for (int i = 0; i < 3; i++) {
      if (!removingControl.isRemovedThread(i)) {
        threads[i].start();
      }
    }
    for (int i = 0; i < 3; i++) {
      if (!removingControl.isRemovedThread(i)) {
        threads[i].join();
      }
    }
  }
}
```

For each reduced stress test, we need to determine whether the reduced stress test can repeat the original failure. A simple technique to determine whether a reduced stress test *s* can repeat a failure is to let the new test execute without control. If this execution reproduces the original failure, we can use the execution trace of the smaller stress test to localize the fault; otherwise, the threads/method invocations in the removal set must be restored and another removal set of the same size must be selected.

However, executing the new stress test without any control may not repeat the original failure due to the non-deterministic execution behaviour of concurrent programs. In the next section, we introduce a more advanced technique, which utilizes the original failed execution trace to guide the execution of the reduced stress test.

## V. Guided Execution

In this section, we present our approach to guided executions, which control the execution of a reduced stress test so that it follows the original failed execution trace whenever possible. Guided executions are more likely to reproduce the original failure than non-deterministic, uncontrolled executions.

### *5.1 The Problem*

It is possible that the execution of a reduced stress test will reach a point where it can no longer follow the original execution trace, due to removal of threads/method invocations from the original stress test. For example, assume that the event that is expected to be executed next according to the original execution trace is to be executed by thread T, but thread T is unable to execute this expected event in the current execution of the reduced stress test. The reason for this mismatch between the event that is expected to be executed by thread T and the actual event that thread T can execute is:

1. The thread or method invocation that generated this expected event in the original stress test was removed from the original stress test to create the reduced stress test.
2. The thread or method invocation that generated this expected event in the original stress test is still in the reduced stress test. However, due to the removal, the control flow of the program under test has changed and the expected event should be skipped or executed later.

This problem is addressed in two steps.

*Step 1:* Remove from the original execution trace any event that cannot possibly be executed by the reduced stress test.

- If thread T is removed from the original stress test, then all of the events executed by T are removed from the original execution trace, since these events cannot be executed by the reduced stress test.
- If an invocation of a method M performed by thread T is removed from the original stress test, then we remove from the original execution trace all the events from the *entermethod* event for the method invocation that is executed by some thread T, up to and including the next *exitmethod* event for the method invocation that is executed by the same thread T. This will include any *enterMonitor*, *read*, *write*, etc events that are exercised during the execution of method M by thread T. Note that these events may not be consecutive in the original execution trace as there may be interleavings of events performed by other threads.

The execution trace that is resulted from removing events in step 1 is called the retained trace. However, the execution of the reduced stress test may still be unable to follow the retained trace.

***Step 2: Recover the execution of the reduced stress test by looking ahead in the retained execution trace.***

Assume that the length of the retained trace is *n*. It is possible that after the reduced stress test executes the first *i* < *n* events of the retained trace, event *i+1* of the reduced executed trace cannot be executed. This is because the removal of threads and/or method invocations may have affected the control-flow of the program so that events that were (were not) executed by the original stress test cannot (must) be executed by the new stress test. This problem is illustrated by the following two examples, which use class *Account* from Section II.

Example 1: Assume that the following execution is part of a failed execution of the original stress test. In this execution, T1 invokes *deposit* on *acc* and T2 invokes *withdraw* and *deposit* on *acc*. The sequence of methods that are executed and the corresponding trace of events are shown in Figure 3. Object *acc* is an *Account* object that is initialized to have a balance of 0.

Suppose we remove the invocation of *deposit* by T1 and create a new reduced stress test *s*. Then the *deposit* method invocation by T1 will be removed from the original execution trace. However, the removal of the *deposit* method invocation by T1 to create *s* has made it impossible for the execution of T2's *withdraw* to be completed by *s*, since the balance of Account *acc* is 0 and a withdrawal of $100 therefore cannot be made. This means that T2 can execute events *enterMethod*, *enterMonitor*, and the first read of *amount*. However, the next event executed by T2 must be an *exitMonitor* for *withdraw*, and this event will not match the next event in the retained execution trace, which is highlighted.

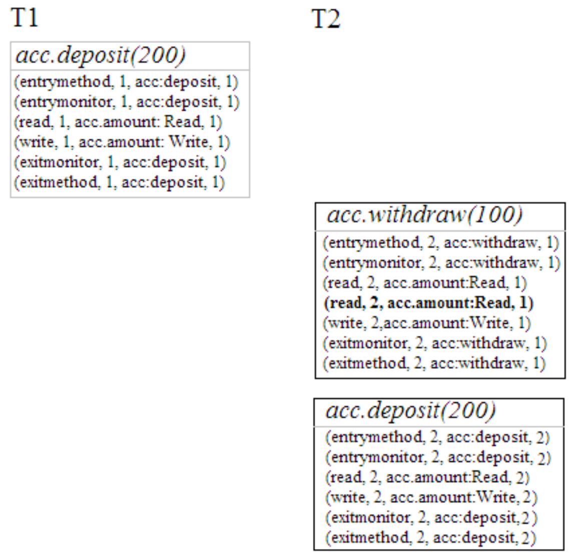
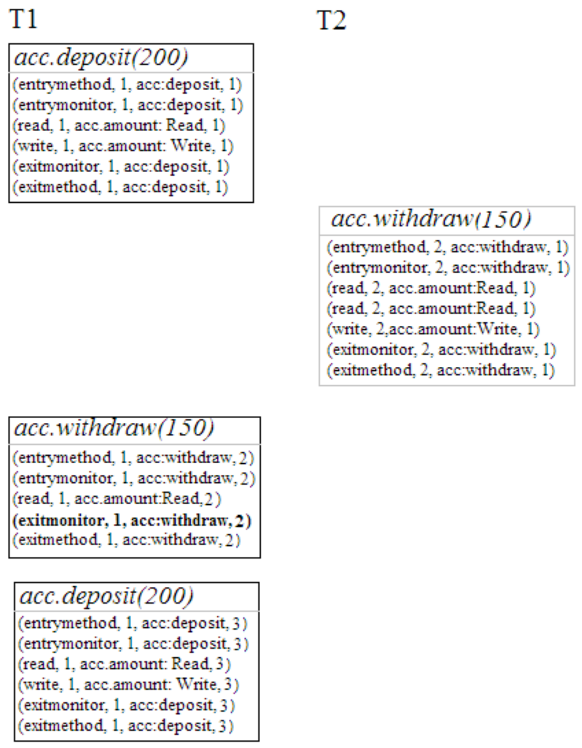

T1
T2
acc.deposit(200)
(entrymethod, 1, acc:deposit, 1)
(entrymonitor, 1, acc:deposit, 1)
(read, 1, acc.amount: Read, 1)
(write, 1, acc.amount: Write, 1)
(exitmonitor, 1, acc:deposit, 1)
(exitmethod, 1, acc:deposit, 1)
acc.withdraw(100)
(entrymethod, 2, acc:withdraw, 1)
(entrymonitor, 2, acc:withdraw, 1)
(read, 2, acc.amount:Read, 1)
**(read, 2, acc.amount:Read, 1)**
(write, 2,acc.amount:Write, 1)
(exitmonitor, 2, acc:withdraw, 1)
(exitmethod, 2, acc:withdraw, 1)
acc.deposit(200)
(entrymethod, 2, acc:deposit, 2)
(entrymonitor, 2, acc:deposit, 2)
(read, 2, acc.amount:Read, 2)
(write, 2, acc.amount:Write, 2)
(exitmonitor, 2, acc:deposit, 2)
(exitmethod, 2, acc:deposit, 2)

Figure 3. Example 1

Example 2: Assume that the following sequence is part of a failed execution of the original stress test. In this execution, T1 invokes *deposit*, *withdraw*, and another *deposit* on *acc*, and T2 invokes withdraw on *acc*.



```
T1: acc.deposit(200)
(entrymethod, 1, acc:deposit, 1)
(entrymonitor, 1, acc:deposit, 1)
(read, 1, acc.amount: Read, 1)
(write, 1, acc.amount: Write, 1)
(exitmonitor, 1, acc:deposit, 1)
(exitmethod, 1, acc:deposit, 1)

T2: acc.withdraw(150)
(entrymethod, 2, acc:withdraw, 1)
(entrymonitor, 2, acc:withdraw, 1)
(read, 2, acc.amount:Read, 1)
(read, 2, acc.amount:Read, 1)
(write, 2,acc.amount:Write, 1)
(exitmonitor, 2, acc:withdraw, 1)
(exitmethod, 2, acc:withdraw, 1)

T1: acc.withdraw(150)
(entrymethod, 1, acc:withdraw, 2)
(entrymonitor, 1, acc:withdraw, 2)
(read, 1, acc.amount:Read, 2)
(exitmonitor, 1, acc:withdraw, 2)
(exitmethod, 1, acc:withdraw, 2)

T1: acc.deposit(200)
(entrymethod, 1, acc:deposit, 3)
(entrymonitor, 1, acc:deposit, 3)
(read, 1, acc.amount: Read, 3)
(write, 1, acc.amount: Write, 3)
(exitmonitor, 1, acc:deposit, 3)
(exitmethod, 1, acc:deposit, 3)
```

Figure 4. Example 2

In this execution trace, a complete withdrawal by T1 was not allowed, since the balance of *acc* at the time of T1's withdrawal was only $50. Assume we try to simplify the stress test by removing the *withdraw* method invocation by T2. Now the withdrawal by T1 can be completed, and T1 in the stress test will try to execute an additional read and write event on the account balance. However, this will create a mismatch between the execution of the stress test and the retained trace, because the additional *read* and *write* events that T1 must execute are not in the retained trace.

### *5.2 The Approach*

When a mismatch occurs, we try to guide the recovery of the execution by modifying the retained trace so that a match can occur. Two options mentioned below could be tried:

1) Events that are in the retained trace, but that cannot be executed by the reduced stress test, can be skipped.

2) Some events that are not in the retained trace, but that must be executed by the reduced stress test, can be added to the retained trace so that the trace will contain the events that must be executed by the reduced test.

Ideally, both options could be tried when a mismatch occurs. However, always trying both options may result in an exponential number of executions. To address this problem we use information in the original execution trace to decide which option to explore.

When a mismatch occurs, denote the event in the retained trace that is expected to be executed as the *expected event*. Let T be the thread that is expected to execute this event. Denote the event that thread T is actually trying to execute as the *actual event*. We consider the following two cases:

Case 1: If the *actual event* occurs in the retained trace after the *expected event*, we can skip all the events in the retained trace up to the *actual event*. Now the *actual event* and the *expected event* match.

Case 2: Otherwise, the *actual event* is added to the retained trace and this added event becomes a matching *expected event*. By finding a match, the execution of the reduced stress test has recovered and can continue.

Referring again to Example 1, when the actual event to be executed by the reduced stress test is the *exitMonitor* event for withdraw, the expected event in the reduced execution sequence is the highlighted event, which is a mismatch. However, since we can find the *actual event* in the remaining retained trace, we can skip the infeasible events *read amount* and *write amount*, and allow the new matching expected event *exitMonitor withdraw* to execute. This skips the events that were executed in the original execution but that cannot be executed in the execution of the reduced stress test and guides the execution back towards the original failure.

In Example 2, when the actual event is the second *read amount* event by T1 during *withdraw*, there is a mismatch with the expected event *exitMonitor* for *withdraw*. Also, the actual event cannot be found in the remaining execution trace. In this case, we allow the actual events executed during the now able-to-complete withdrawal to execute. Eventually the expected event *enterMethod* for *deposit* matches the actual event executed by T1, and the execution is back on track.

We must choose an appropriate number of events that can be skipped when looking ahead for a match between the actual event and a future event in the expected execution trace. This is because the matching future event that we find may also be executable if we first execute some events that are added to the retained trace. If we look ahead too far and find a matching future event, we may mistakenly skip events in the retained trace.

On the other hand, it is also possible to set the lookahead to be too small. When the lookahead is too small, it is possible that it would be better to skip some events in the retained trace instead of adding some events to the retained trace. In the case studies reported in Section VII, we show the effect of different lookahead values.

Note that even if we could always find a perfect lookahead value, we could not guarantee that the guided execution would terminate and reproduce the failure. This is because, for example, if we remove a method invocation containing a *notify* operation, then a waiting thread that can only be notified by this *notify* operation would wait forever. The following is an example of such a case.

Assume the following failed execution occurs:

| Thread1 | Thread2 | Thread3 |
|---|---|---|
| e1 | | |
| e2 | | |
| wait | | |
| | e3 | |
| | e4 | |
| | notify | |
| | | e5 |
| e6 | | |

Suppose that we create a reduced stress test *s* by removing Thread2 from the original stress test and by removing all of the events executed by Thread2 from the original failed execution trace:

| Thread1 | Thread3 |
|---|---|
| e1 | |
| e2 | |
| wait | |
| | e5 |
| e6 | |

A forced execution of *s* with the retained trace can replay events e1, e2, *wait*, and e5. However, when we try to replay expected event e6, Thread1 is expected to execute *e6* but is blocked forever. This is because the *notify* event in Thread2 was removed, and, as a result, the execution cannot finish. In this case, we conclude that the failure cannot be reproduced by the reduced stress test.

We assume that we have a test oracle that can be used to detect failures, or that execution failures are detected by the failure of a user-specified assertion, or the raising of an exception. Thus, an original failure that is reproduced by a minimized stress test is triggered by the same assertion or exception.

In our approach and examples, we do not consider random inputs, which lead to more non-determinism. Executions with random inputs can be replayed if the random inputs are recorded.

## VI. TESTMINIMIZER: A PROTOTYPE TOOL

Our stress test minimization algorithm has been implemented in a tool called *TestMinimizer*. *TestMinimizer* was implemented using the Modern Multithreading library [8]. This library provides testing and debugging services for multithreaded Java programs.

The two components of *StreeTestMinimizer* are shown in Figure 5. The *remover* component takes as input the original *stress test*, the original *failed execution trace*, and two sets of threads/method invocations. The *removal set* is the set of threads/method invocations that can definitely be removed from the original stress test. The *candidate removal set* is the set of threads/method invocations that we are currently trying to remove from the original stress test. The *remover* removes all of the threads/method invocations in the two input sets from the original stress test and the original failed execution trace and outputs a *new stress test* and a *retained trace*.

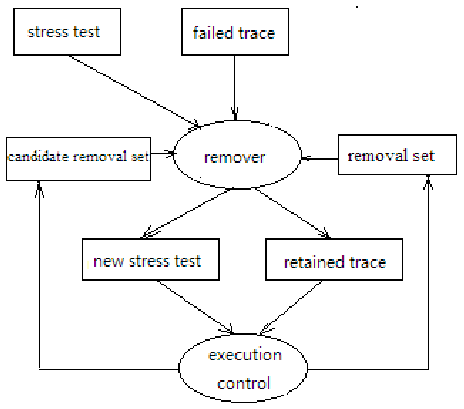


Figure 5. Architecture of TestMinimizer

The *new stress test* and *retained trace* are given to the *execution controller.* The *controller* guides the execution of the new stress test so that it follows the *retained trace* as closely as possible, adding, removing and executing events as necessary.

If the result of the controlled execution is the original failure, the threads/method invocations in the *candidate removal set* are added to *removal set* and *candidate removal set* is updated based on delta debugging. Otherwise, the threads/method invocations in the *candidate removal set* are not added to *removal set*, but the *candidate removal set* is still updated. This iterative process stops when all the threads/method invocations delta debugging wants to try to remove have been considered. The final output will be stored in *removal set*.

## VII. EXPERIMENTS

As a proof-of-concept, we used *TestMinimizer* to conduct several experiments of our minimization approach on a suite of failed stress tests for 16 faulty Java concurrent data structures. The objective of our experiments was to investigate the following two questions:

1. What is the effect of different lookahead values?
2. How effective and efficient is our approach?

The first nine of the 16 concurrent data structures were among the faulty programs used in [4]. These concurrent data structures lacked stress tests. Thus, we wrote stress tests for them using the common stress-testing framework described in Section II.

The last seven of the 16 programs used in our study were found in Github by searching for "concurrent data structure" and for concurrent data structures that were written in the Java language and that had stress tests written for them. Our query was matched in 20 projects and seven of them had stress tests. We selected all seven projects, and for each project, we selected the first matching concurrent data structure. Multiple threads were created to stress test the target concurrent data structure. Each thread repeatedly made a random selection of a public method to invoke. Note that only public methods that could be called with an integer value, or an instance of the target data structure, were invoked, i.e., we did not randomly generate instances of other types of objects. In fact, all of the public methods of the target data structures in the empirical study could be called with an integer value or an instance of the target data structure.

Table I. Subject Programs

| Program | LOC | Fault source | # faults | # runs |
|---|---|---|---|---|
| Account | 177 | Original | 1 | 1 |
| AirlineTickets | 142 | Original | 1 | 2 |
| BufWriter | 183 | Original | 1 | 1 |
| Lottery | 154 | Original | 1 | 3 |
| Shop | 226 | Original | 1 | 2 |
| Arraylist | 5898 | Original | 1 | 2 |
| HashSet | 7103 | Original | 1 | 2 |
| StringBuffer | 1380 | Original | 1 | 3 |
| Vector | 760 | Original | 1 | 1 |
| ConcurrentStack | 114 | [15] | 2 | 2 |
| BoundedBuffer | 126 | [16] | 1 | 3 |
| ConcurrentBST | 199 | [17] | 1 | 1 |
| ConcurrentLinkedList | 161 | [18] | 1 | 3 |
| ConcurrentQueue | 91 | [25] | 1 | 2 |
| ConcurrentQuadTree | 224 | [26] | 1 | 2 |
| ConcurrentHashMap | 206 | [27] | 2 | 1 |

In order to conduct the empirical study, we rewrote these programs using the Modern Multithreading library [9], which provided the services that we used for tracing and guiding executions. Then, we inserted faults into the programs based on descriptions of actual faults in similar programs that we found in the literature [15][16][17][18][25][26][27]. We also inserted assertions that were used as test oracles for determining whether test executions of the programs passed or failed. Table I shows the lines of code, the fault sources of the subject programs, the number of faults and also the number of runs to get the failed execution.

To determine, for each stress test, the number of threads and the number of method invocations for each thread, we investigated the default numbers of threads and method invocations used in existing stress tests for concurrent data structures. Some stress tests use a large number of threads with a small number of method invocations per thread, to simulate a high concurrency scenario. This was the case in [13], which creates 1000 threads, each invoking 1 method, for a total of 1000 method invocations. Other stress tests use a small number of threads that each executes a large number of method invocations, simulating a high workload for each thread. This was done in [14], which creates 10 threads that each invoke 500 methods, for a total of 5000 method invocations. We use numbers that are in between the ones used in [13] and [14] — we set the number of threads to 100, and the number of method invocations per thread to 100. A total of 10000 method invocations are executed in the stress tests.

The experiments were performed on a laptop with a 2.30GHz CPU and 4GB memory, running Windows 7(64-bit) and Sun's Java 1.8.

### *7.1 Impact of lookahead*

We considered lookahead values of 2, 4, 6, 8, and 10. Figures 6 to 9 show the result of this study. We show the number of threads retained, the number of method invocations retained, the total number of executions, and the total amount of running time. Detailed data can be found on our web site [30].

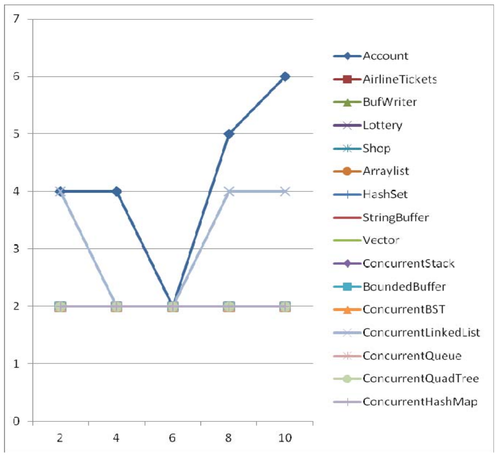


Figure 6. Impact of lookahead value on # of threads retained
x: the value of lookahead; y: # of threads retained

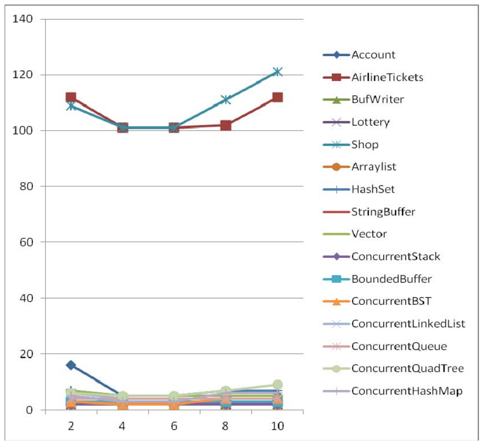


Figure 7. Impact of lookahead on # of method invocations retained
x: the value of lookahead; y: # of method invocations retained

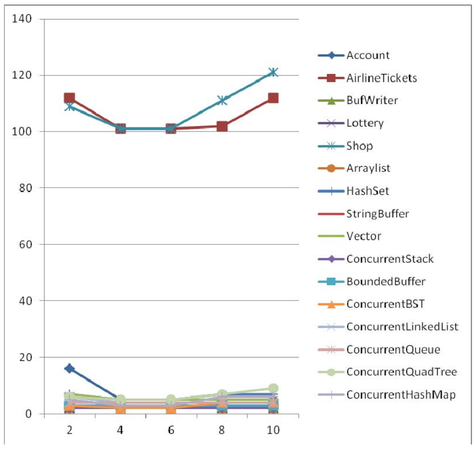


Figure 8. Impact of lookahead on # of executions
x: the value of lookahead; y: # of executions

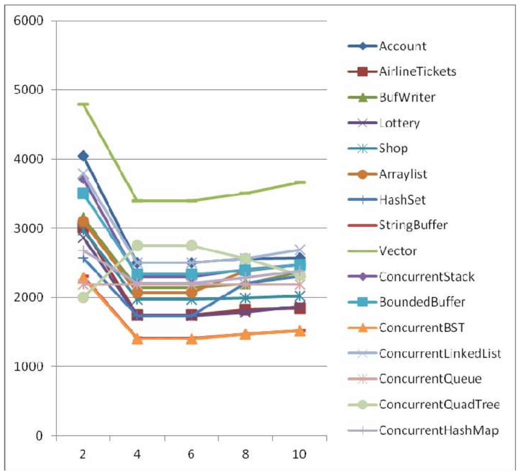


Figure 9. Impact of lookahead on total running time
x: the value of lookahead; y: total running time

The results in Figure 6 to 9 show that the lookahead value affects the number of executions, the number of threads, the number of method invocations that are retained and the total amount of running time. An observation is that having a lookahead value that is too small or too big leads to more executions, more threads and method invocations retained, and more total running time. We will discuss the factors that affect the total execution time in section 7.3.

### *7.2 Efficiency and Effectiveness of Guided Executions*

Our second experiment is to answer Question (2). In this experiment, we implemented three techniques: guided executions, uncontrolled executions and random executions.

Guided executions apply delta debugging to remove threads/method invocations and exercise additional runtime control to guide stress test executions. In this experiment, we chose a lookahead value of 5. This choice is based on the study in section 7.1, which showed that a lookahead of 4 to 6 results in less threads, fewer retained method invocations, and lower total running time. Note that this choice may not be the optimal choice for all of the target stress tests.

Uncontrolled executions also apply delta debugging, but the execution of a stress test is allowed to run non-deterministically, i.e., without any execution control.

Random executions are a baseline technique that tries to remove threads/methods randomly, without applying delta debugging. Also executions are not controlled; instead the execution is allowed to run non-deterministically. To allow a fair comparison, the number of random executions performed was the same as the number of guided executions. During the thread removal stage, a random execution removed a random number of threads from the remaining threads. During the method removal stage, a random execution removed a random number of method invocations from the remaining method invocations of a random thread. When the new execution failed, the selected threads/method invocations could be removed; otherwise the selected threads/method invocations were restored.

Table II shows the total number of threads and method invocations left in the reduced stress tests, the number of executions (the number of reduced stress tests tried), and the total execution time for guided executions, uncontrolled executions, and random executions, respectively. All of the original failures were reproduced by the reduced stress tests. Otherwise, the removal set would have been restored.

The results in Table II show that for all of the subject programs, random execution retained more threads and method invocations than the other two techniques, with less execution time. Random executions did not remove threads/method invocations systematically, and different attempts at removal sometimes overlapped each other. Guided executions were able to remove more threads and method invocations, with fewer executions. than uncontrolled execution. The total number of threads left by uncontrolled executions was 1.9 times that left by guided executions, while the total number of method invocations left by uncontrolled executions was 4.1 times that left by guided executions.

The reason that uncontrolled executions are more effective during the thread removal stage than the method-invocation removal stage is that during the thread removal stage, even when only a few threads are left, uncontrolled executions are still likely to expose the fault since each thread left has 100 method invocations. However, during the method removal stage, with the methods removed, it is less likely for uncontrolled executions to expose the fault, since the stress is insufficient. In other words, guided executions are more effective than uncontrolled executions when stress is low. For 14 of 16 programs, the run time of the guided executions was longer than that of the uncontrolled executions. We will discuss the factors that affect the total execution time in next subsection.

### *7.3 Discussion*

The factors that affected the total execution time include the instrumentation overhead, the total number of executions and the number of executions that could not terminate. The programs under test were instrumented to allow controlled executions that follow the original failed trace. This adds execution time overhead. The number of random executions and guided executions were kept the same for the sake of comparison. The number of uncontrolled executions was generally greater than the number of guided executions, since guided executions are more likely to reproduce the failure than uncontrolled executions. Some executions could not terminate. For example, a *notify* event in a method could be removed, or a control flow change causes a notify event to be skipped. As a result, a waiting thread could not be awakened We set a timeout of 1 minute for each execution. When an execution was stuck in a busy-waiting loop, it terminated after the timeout.

After the thread removal phase, no additional threads could be removed. However, after removing method invocations, it was possible that all of the method invocations of some thread were removed, effectively removing the thread.

Table II Comparison results between guided, uncontrolled, and random executions

| Program | Guided executions | | | | Uncontrolled executions | | | | Random executions | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | # of threads left | # of methods left | # of executions | Time (s) | # of threads left | # of methods left | # of executions | Time (s) | # of threads left | # of methods left | # of executions | Time (s) |
| Account | 4 | 5 | 86 | 323 | 8 | 42 | 156 | 314 | 27 | 635 | 86 | 192 |
| ATickets | 2 | 101 | 328 | 1345 | 5 | 131 | 512 | 1324 | 19 | 262 | 328 | 644 |
| BufWriter | 2 | 2 | 45 | 212 | 2 | 6 | 61 | 148 | 16 | 589 | 45 | 93 |
| Lottery | 2 | 2 | 59 | 382 | 2 | 5 | 66 | 101 | 41 | 881 | 59 | 110 |
| Shop | 2 | 101 | 341 | 1431 | 4 | 133 | 552 | 1422 | 21 | 307 | 341 | 710 |
| Arraylist | 2 | 2 | 45 | 248 | 2 | 3 | 46 | 77 | 31 | 581 | 45 | 70 |
| HashSet | 2 | 5 | 45 | 179 | 4 | 65 | 83 | 148 | 44 | 816 | 45 | 89 |
| SB | 2 | 3 | 48 | 312 | 3 | 9 | 326 | 256 | 27 | 475 | 48 | 101 |
| Vector | 2 | 5 | 57 | 547 | 2 | 9 | 294 | 478 | 12 | 421 | 57 | 115 |
| ConStack | 2 | 2 | 29 | 104 | 2 | 7 | 37 | 565 | 36 | 59 | 29 | 59 |
| BB | 2 | 3 | 31 | 175 | 2 | 4 | 61 | 448 | 33 | 302 | 31 | 68 |
| ConBST | 2 | 2 | 30 | 240 | 4 | 16 | 57 | 175 | 29 | 384 | 30 | 62 |
| ConLL | 2 | 4 | 44 | 155 | 2 | 4 | 42 | 132 | 39 | 297 | 44 | 98 |
| ConQ | 2 | 4 | 45 | 249 | 4 | 9 | 173 | 213 | 37 | 491 | 45 | 85 |
| ConQuad | 2 | 5 | 57 | 496 | 5 | 13 | 197 | 393 | 13 | 205 | 57 | 119 |
| ConHM | 2 | 3 | 48 | 353 | 14 | 32 | 312 | 329 | 19 | 291 | 48 | 97 |

For *AirlineTickets* and *Shop*, the faults are exposed when the capacity is reached. The capacity is set to 100 when the data structure is initialized. So for these programs, we need more than 100 method invocations to trigger the failure.

## VIII. RELATED WORK

Several tools minimize failure-introducing inputs to concurrent systems without controlling sources of non-determinism [19, 20]. Since no additional runtime control is exercised, these approaches require fewer instrumentations and are fast in term of running a single execution. However, some failures may rarely happen. Without execution control, it is hard to reproduce the failure and minimize the inputs. Other techniques seek to only minimize thread interleavings leading up to concurrency bugs [21, 22, 29, 31], but do not minimize the execution trace so that the trace is still lengthy and takes time to reproduce the failure and fix the bug. By guiding the execution with the original failed trace, our approach efficiently removes unrelated events to minimize the execution trace and reproduce the failure.

The work most closely related to our work was done by Scott et al. [28] for distributed systems. They first applied delta debugging to prune external events of distributed systems. To check each external-event subsequence chosen by delta debugging, they use a stateful version of dynamic partial-order reduction. They first explore a uniquely defined schedule that closely matches the original execution. (If an internal message from the original execution is not pending at the point that an internal message should be delivered, they skip over the message and move to the next message from the original execution.). If the schedule cannot reproduce the original failure, they try other schedules. These schedules prioritize backtrack points that match the type of the corresponding message from the original trace (as determined by the language-level type-tag of the message object, which is available to the RPC layer at runtime), within a user-defined duration. They also spend the remaining time budget attempting to minimize internal events. Our approach, tries only one schedule, which is dynamically generated based on the original failed trace and our recovery strategy. From the results of our empirical study, our approach can effectively and efficiently minimize stress tests for concurrent data structures.

Delta debugging [2] is an automated debugging approach based on systematic testing. Delta debugging automatically finds minimal, failure-inducing circumstances automatically, for circumstances such as program inputs, changes to the program code, or program executions. The input of delta debugging is a failed test case and the output is a 1-minimal failed test case. A failed test case *c* composed of *n* changes is 1-minimal if removing any single change causes the failure to disappear. While removing two or more changes at once may result in an even smaller, still-failed test case, every single change on its own is significant in reproducing the failure.

Our work is the first to apply delta debugging to minimize stress tests for concurrent data structures. Also, in our approach, we control program executions to recover from mismatches.

## IX. Conclusions

In this paper, we presented an approach for minimizing stress tests for concurrent data structures. A stress test typically involves multiple threads that repeatedly invoke methods of the target data structure. Our approach is to remove as many threads and method invocations as possible from a failed stress test, while ensuring that the original failure will still occur. We apply delta debugging to identify sets of threads and method invocations to remove. We then control the execution of the new test so that the original failure is more likely to be repeated. The results of our empirical studies show that our approach is effective and efficient at minimizing real-life stress tests for concurrent data structures.

In the future, we plan to conduct more experiments to evaluate the effectiveness of our approach. In particular, we will conduct more experiments on the impact of the lookahead value. Second, we plan to integrate our approach with an automated stress test generation approach so that the users only need to provide a target data structure. A minimized stress test with a failed execution will be reported to them, if a failed execution is found in the stress test.

## References

[1] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from Mistakes --- A Comprehensive Study on Real World Concurrency Bug Characteristics", 13th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS'08).

[2] A. Zeller and R. Hildebrandt, "Simplifying and Isolating Failure-Inducing Input", IEEE Transactions on Software Engineering28(2), February 2002, pp. 183-200.

[3] Concurrent data structure, https://en.wikipedia.org/wiki/Concurrent_data_structure

[4] S. Park, R. Vuduc, and M. Harrold. Falcon, "Fault Localization in Concurrent Programs", Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, pp. 245-254, 2010.

[5] Github, https://www.github.com

[6] Java concurrent utilities, http://gee.cs.oswego.edu/cgi-bin/viewcvs.cgi/jsr166/src/test/tck/Collection8Test.java?revision=1.2&view=markup

[7] Y. Eytani, K. Havelund, S. D. Stoller, and S. Ur, "Towards a framework and a benchmark for testing tools for multi-threaded programs", Concurr. Comput. : Pract. Exper., 19(3):267–279, 2007.

[8] R. Carver, and K. C. Tai, Modern Multithreading, Wiley, 2006.

[9] R. Carver, and Y. Lei, "A Class Library for Implementing, Testing, and Debugging Concurrent Programs", International Journal on Software Tools for Technology Transfer: Volume 12, Issue 1 (2010), Page 69-88.

[10] Y. Lei, and R. Carver, "Reachability Testing of Concurrent Programs", IEEE Transactions on Software Engineering, Volume 32, No. 6, 2006, pp. 382-403.

[11] R. Carver and K. C. Tai, "Replay and testing for concurrent programs", IEEE Software, Vol. 8 No. 2, Mar. 1991, 66-74.

[12] K. C. Tai, R. H. Carver, and E. Obaid, "Debugging concurrent Ada programs by deterministic execution", IEEE Trans. Software Engineering, 17(1):45-63, 1991.

[13] Lock free binary search tree. https://github.com/shreya-inamdar/concurrent-data-structures/blob/master/LockFreeBST/src/Test.java

[14] Lock free concurrent stack. https://github.com/mdtareque/concurrentDataStructures/blob/master/src/CStackLockFreeTester.java

[15] Lock free stack. http://stackoverflow.com/questions/5614599/simple-lock-free-stack

[16] Lock free code a faulse sense of security. http://www.drdobbs.com/cpp/lock-free-code-a-false-sense-of-security/210600279?pgno=5

[17] Java theory and practice: Going aomic. http://www.ibm.com/developerworks/library/j-jtp11234/

[18] Java concurrency programming 5: lock free data structure. http://blog.csdn.net/b_h_l/article/details/8704480

[19] T. Arts, J. Hughes, J. Johansson, and U. Wiger, "Testing Telecoms Software with Quviq QuickCheck", Erlang '06.

[20] J. Clause and A. Orso. A Technique for Enabling and Supporting Debugging of Field Failures. ICSE '07.

[21] J. Choi and A. Zeller. Isolating Failure-Inducing Thread Schedules. SIGSOFT '02.

[22] M. A. El-Zawawy and M. N. Alanazi. An Efficient Binary Technique for Frace Simplifications of Concurrent Programs. ICAST '14.

[23] A. Brito,, et al. "Scalable and low-latency data processing with stream map reduce." Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on. IEEE, 2011.

[24] J. Xu, et al. "A Dynamic Approach to Isolating Erroneous Event Patterns in Concurrent Program Executions." Multicore Software Engineering, Performance, and Tools. Springer Berlin Heidelberg, 2013. 97-109.

[25] Writing a generalized concurrent queue, http://www.drdobbs.com/parallel/writing-a-generalized-concurrent-queue/211601363

[26] Threadsafe quadtree without locking, https://hub.jmonkeyengine.org/t/threadsafe-quadtree-without-locking/8035

[27] Concurrenthashmap, http://codereview.stackexchange.com/questions/96686/concurrenthashmap-implementation

[28] C. Scott, et al. Minimize faulty executions of distributed systems. In Proceedings of the 13th Usenix Symposium on Networked Design and Implementation (Santa Clara, CA, Mar. 16–18, 2016) 291–309.

[29] J. Huang,, and C. Zhang. "An efficient static trace simplification technique for debugging concurrent programs." International Static Analysis Symposium. Springer Berlin Heidelberg, 2011.

[30] TestMinimizer, http://barbie.uta.edu/~jxu/testminimizer.htm

[31] N. Jalbert, and K. Sen. "A trace simplification technique for effective debugging of concurrent programs." Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering. ACM, 2010.