FAULT DETECTION AND LOCALIZATION TECHNIQUES FOR CONCURRENT PROGRAMS

by

Jing Xu

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON

December 2016

Arlington, Texas

Abstract

FAULT DETECTION AND LOCALIZATION TECHNIQUES FOR CONCURRENT PROGRAMS

Jing Xu, PhD

The University of Texas at Arlington, 2016

Supervising Professor: Yu Lei

Concurrency faults are hard to detect and localize due to the nondeterministic behavior of concurrent programs. In this dissertation, we present three approaches to detecting and localizing faults in concurrent programs. The first approach identifies erroneous event patterns in a failed concurrent program execution. Given a failed execution, we characterize the execution as a sequence of context-switch points and then use controlled execution to distinguish erroneous context-switch points from benign context-switch points. Erroneous context-switch points are used to derive erroneous event patterns, which allow the user to quickly localize the actual fault. Our experiments were conducted on thirteen programs. Seven of them were made by students of a course and the others were from real-life programs. The results showed that our technique can effectively and efficiently localize the faults in twelve of the thirteen programs.

The second approach detects unbounded thread-instantiation loops in server applications that typically spawn a separate thread to handling incoming requests. It checks loops and conditions under which a thread instantiation may take place against several bounding iteration patterns and bounding condition patterns. A loop is considered bounded if a pattern match is found. Otherwise, it is considered unbounded. The results of our experiments showed that the approach could effectively detect 38 unbounded thread-instantiation loops from 24 real-life java server applications. In particular, 12 unbounded thread-instantiation loops detected by our approach were confirmed by the original developers.

The third approach minimizes stress tests for concurrent data structures. It applies delta debugging to identify threads and method invocations that can be removed from a stress test. When running a stress test reduced by removing some threads/method invocations, we control the execution of the reduced test in a way such that it is more likely to repeat the original failure. In our experiments, we applied the approach to the stress tests of sixteen real-life concurrent data structures. Each stress test had 100 threads and 100 method invocations in each thread to stress test the target data structure. All the stress tests were reduced to be no more than four threads and fourteen out of sixteen stress tests were reduced to have no more than five method invocations.

Acknowledgements

I would like to thank my supervising professor Dr. Jeff Lei for his training, guidance, and suggestions. Without his help and support, I could not imagine I could reach this finish line of my Ph. D. I also wish to thank my committee, Dr. Christoph Csallner, Dr. David Kung, Dr. Donggang Liu and Dr. Junzhou Huang for generously sharing their time and ideas.

I would also like to thank my parents for their support and unselfish love. Finally, thank my wife for her support and understanding, who has taken care of our baby and me without any complaints. I wish I could have more family time with you in future.

November 2016

Table of Content

List of Tables

List of Illustrations

Chapter 1. Introduction

As concurrent programs become widespread, it is important to have effective and efficient tools and techniques for testing and debugging concurrent programs. Concurrency faults are hard to find and fix due to the nondeterministic behavior of concurrent programs. A Microsoft survey [17] shows that nearly two-thirds of Microsoft developers have to deal with concurrency issues and over half of the developers detect, debug, and fix concurrency faults every month. Furthermore, over 60% of these faults take several days to fix. Also, failures caused by concurrency issues can have potentially devastating consequences. For example, a blackout in the northeastern U.S. in 2003 left tens of millions of people without electricity, due to a race condition in power plant monitoring software. [18]

In this dissertation, we present three approaches to detecting and localizing faults in concurrent programs.

1.1 Research overview

The first approach identifies erroneous event patterns in a failed concurrent program execution to help users localize the faults. After a failed execution is found, it still takes time to localize the fault. Especially to expose a concurrent fault, it typically involves interleavings of multiple threads, which makes the faults hard to localize and understand. If the failure-introducing context-switch point, or the failure-introducing pattern can be automatically found, it would make finding and fixing the concurrency bugs easier. The approach takes the trace of a failed execution as input and characterizes the execution as a sequence of context-switch points, or switch points, derived from the trace. A systematic search strategy is used to find the erroneous switch point that causes the execution to fail. The novelty of our approach is the use of the least concurrency mode to determine the correctness of a switch point. In the least concurrency mode, each thread is controlled to execute until it cannot proceed further, i.e., it either blocks or finishes. The motivation is to minimize the number of interleavings and thus reduce the chance of failure due to concurrency. To determine the correctness of a switch point, we perform a number of test executions in which we first replay all the events up to and including the switch point, which allows the switch point to be reproduced, and then let the program proceed in the least concurrency mode. If one of these executions passes, the switch point is likely to be a benign switch point. Otherwise, the switch point is likely to be an erroneous switch point. After identifying the erroneous switch point, our technique tries to find erroneous event patterns related to this erroneous switch point, which can help the user to quickly localize the faults. An experimental evaluation of our technique was conducted on thirteen Java benchmark programs. Seven of them were made by students of a course and the others were from real-life programs [5]. The results of our experiments showed that our technique could effectively localize the faults in twelve of the thirteen programs.

The second approach is a lightweight, static approach to detect unbounded thread-instantiation loops that may exist in a server application.In server applications, threads are created to handle incoming requests. Since threads consume significant resources including CPU cycles and memory, it is important to control the number of threads that are instantiated. If this number is unbounded, the application may respond slowly, or even crash, when there are a large number of incoming requests. A key observation is that the decision logic for thread instantiation is typically not complex. Our approach checks thread instantiation loops against some bounded thread-instantiation patterns. A loop is considered bounded if a pattern match is found. Otherwise, it is considered unbounded. An experiment on 24 real-life Java server applications was done using an Eclipse plugin ThreadBoundChecker that has been developed during this research. In the experiment, a total of 41 unbounded thread-instantiation loops were found. Of these, 12 loops were confirmed by the program developers to be unbounded. For 26 loops, we did not get a response from the developers but we verified them to be unbounded by a manual inspection of the code.

The third approach is to minimize stress tests for concurrent data structures. Stress testing is often used to test a concurrent data structure. However, the execution trace of a failed stress test that involves many threads executing many methods may contain a large number of execution events. If the size of a failing execution trace can be reduced, then faults can be localized faster and easier. Our approach is to remove some of the threads and/or method invocations from a stress test for a concurrent data structure to create a smaller test that still produces the same failure. We apply delta debugging to a failed stress test to identify the threads and method invocations that are essential to cause the failure. The other threads and method invocations in the original failed execution can be removed if the original failure can still be triggered after the removal. To increase the chance of triggering the original failure during the execution of a smaller stress test, we force the new execution to follow the original failed execution trace when possible, and guide the execution back to the failed trace when the execution diverges. A tool called TestMinimizer was implemented and it was applied to the stress tests of sixteen real-life concurrent data structures. Each stress test had 100 threads and each thread had 100 method invocations to stress test the target data structure. All the stress tests were reduced to be no more than four threads and fourteen out of sixteen stress tests were reduced to have no more than five method invocations.

1.2 Summary of publications

This dissertation is presented in an article-based format and includes three research papers.

In 错误！未找到引用源。, we present the paper titled, “A Dynamic Approach to Isolating Erroneous Event Patterns in Concurrent Program Executions”, which was published in IEEE first International Conference on Multicore Software Engineering, Performance, and Tools (MUSEPAT), in 2013. The paper reports the first approach in section 1.1, which identifies erroneous event patterns in a failed concurrent program execution.

Chapter 3 presents the paper titled, “A Lightweight, Static Approach to Detecting Unbounded Thread-Instantiation Loops”. The paper was published in IEEE eighth International Conference on Software Testing, Verification, and Validation (ICST), in 2015. The paper presents the second approach in section 1.1, which detects unbounded thread-instantiation loops in server applications.

Chapter 3 presents the paper titled, “Using Delta Debugging to Minimize Stress Tests for Concurrent Data Structures”. The paper was published in IEEE tenth International Conference on Software Testing, Verification, and Validation (ICST), in 2017.. The paper presents the third approach in section 1.1, which minimizes stress tests for concurrent data structures.

Chapter 2. A Dynamic Approach to Identifying Erroneous Event in Concurrent Program Executions

The chapter contains a paper published in IEEE first International Conference on Multicore Software Engineering, Performance, and Tools (MUSEPAT), in 2013.

A Dynamic Approach to Identifying Erroneous Event in Concurrent Program Executions[1]

Jing Xu[1], Yu Lei[1], Richard Caver[2], David Kung[1]

[1]Dept. of Computer Science and Engineering, University of Texas at Arlington, Arlington, TX, USA

jingxu@mavs.uta.edu, {ylei,kung}@uta.edu

[2]Dept. of Computer Science, George Mason University, Fairfax, VA , USA

rcarver@gmu.edu

Abstract. Concurrency bugs are hard to find due to the nondeterministic behavior of concurrent programs. In this paper, we present an algorithm for isolating erroneous event patterns in concurrent program executions. Failed executions are characterized as a sequence of switch points, which capture the interleaving of read and write events on shared variables. The algorithm inputs the sequence of a failed execution, and outputs erroneous event patterns. We implemented our algorithm and conducted an experimental evaluation on several Java benchmark programs. The results of our evaluation show that our approach can effectively and efficiently identify erroneous event patterns in failed executions.

Keywords: Concurrency, Fault Localization, Debugging

2.1 Introduction

As concurrent programs become widespread, it is important to have effective and efficient techniques and tools for testing and debugging concurrent programs. A survey from Microsoft [17] reveals that nearly two-thirds of Microsoft programmers have to deal with concurrency issues and over half of the programmers detect, debug, and fix

[1] Multicore Software Engineering, Performance, and Tools, A Dynamic Approach to Identifying Erroneous Event in Concurrent Program Executions, volume 8063, 2013, pp 97-109, Jing Xu, Yu Lei, Richard Carver, David Kung, with permission of Springer.

concurrency faults on at least a monthly basis. Furthermore, over 60% of these faults take several days to fix. Failures caused by concurrency faults can have potentially devastating consequences. For example, in 2003, a blackout in the northeastern U.S. left tens of millions of people without electricity, due to a race condition in the power plant monitoring software [18].

Much work has been reported on detecting two types of concurrency fault. One is atomicity violation (also referred to as serializability violation), which occurs when a code block that is expected to be executed atomically is executed non-atomically. The other is order violation, which occurs when code blocks are executed in an incorrect order. Recent work uses dynamic pattern analysis [2, 3, 6, 22] to recognize patterns of events that may be associated with these faults. Some pattern analysis techniques try to extract a set of benign patterns from passed executions and then identify violations of these benign patterns in failed executions [3, 6, 22]. However, some patterns may appear in both passed and failed executions [2]. These patterns will be classified as benign, which prevents them from being identified in failed executions.

We propose a dynamic approach to identifying the erroneous patterns in a failed execution. Our approach is independent from the underlying synchronization mechanisms such as shared variables, semaphores, and monitors. The main idea of our approach is described as follows. We record the trace of a failed execution as a sequence of events. A systematic search strategy is used to find the erroneous switch point that causes the execution to fail. The novelty of our approach is the use of a notion called least concurrency mode to determine the correctness of a switch point. In the least concurrency mode, each thread executes until it cannot proceed further, i.e., it either blocks or finishes. The motivation is to minimize the number of times a thread being interrupted by another thread and thus reduce the chance of failure due to concurrency.

To determine the correctness of a switch point s, we perform a number of test executions in which we first replay all the events up to and including s, which allows s to be reproduced, and then let the program proceed in the least concurrency mode. If one of these executions passes, s is likely to be a benign switch point. Otherwise, s is likely to be an erroneous switch point. After identifying the erroneous switch point, our technique tries to find erroneous event patterns related to this erroneous switch point, which can help users localize the faults.

This fault localization technique has been implemented in a tool called *Huatuo*, which was used to perform an empirical study on 13 benchmark Java programs. The results of the study show that our technique can effectively and efficiently localize the faults in 12 of the 13 programs.

2.2 Preliminaries

Let *s* be a totally-ordered sequence of read and write events in a failed execution of concurrent program CP with switch points P_1, P_2, … P_n, n≥1. Switch point P_i in *s* is *erroneous* if CP enters an incorrect internal state, called an *error* state, after P_i is executed, but before P_{i+1} is executed, and this error state is propagated to cause CP to output an incorrect result (a *failure*) [21].

After a switch point CP may enter an error state, which however may not propagate to the result. This case is possible because CP may recover from an error state, or the result is "coincidentally correct" and CP does not fail.

An atomicity violation occurs when a sequence of read and write events is executed in a way that cannot be serialized and this sequence violates the programmer's intention of atomicity. Figure 2-1 shows the five possible patterns of unserializable interleavings.

If two threads each access a shared variable, and at least one of the two accesses is a write access, then these two accesses comprise a conflicting interleaving pattern. An order violation occurs when a conflicting interleaving pattern is executed and this pattern violates the programmer's intended ordering.

An event pattern is *erroneous* if the appearance of this pattern causes the execution to enter an error state. Patterns that are not erroneous are *benign*.

	Interleaving	Description
1	T1:R R T2: W	Two reads by T1 were expected to have the same value.
2	T1:W R T2: W	The read by T1 was expected to read the value written byT1.
3	T1:W W T2 R	A temporary result written by T1 was not expected to be read by T2.
4	T1:R W T2 W	The value written by T2 was unexpectedly overwritten by T1.
5	T1:W W T2: W	The value written by T2 was unexpectedly overwritten by T1

Figure 2-1 Unserializable Interleaving Patterns [5].

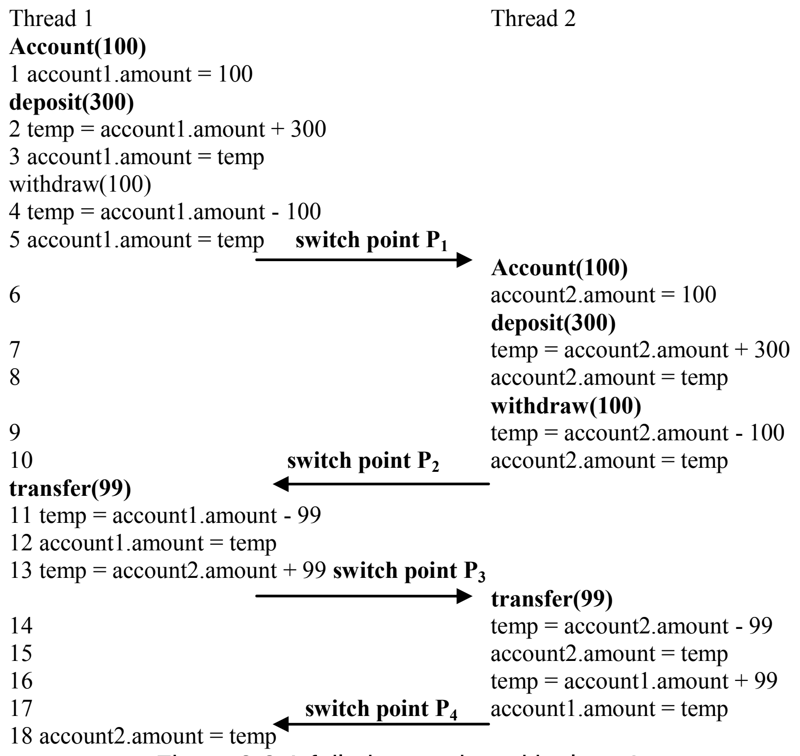
2.3 A Motivating Example

```
public class Account {
    double amount;
    String name;
    public Account(String nm, double amnt ) {
        amount = amnt;
        name = nm;
    }
    synchronized void deposit(double money){
        amount += money;
    }
    synchronized void withdraw(double money){
        amount -= money;
    }
    synchronized void transfer(Account ac, double mn){
        amount -= mn;
        ac.amount += mn;
    }
 }
```

Figure 2-2 An example (faulty) program

As a motivating example, we consider a Java class *Account* in Figure 2-2. This class is from the *ConTest* benchmark programs [19]. Class *Account* has two fields *amount* and *name*, and three methods *deposit*, *withdraw* and *transfer*. Method *deposit* adds a given amount of money, method *withdraw* withdraws a given amount of money and method *transfer* transfers a given amount of money from one account to another.

Figure 2-3 shows a failed execution in which two accounts, account1 and account2, are accessed concurrently by two threads. Thread 1 (or thread 2) initializes account1 (or account2) with 100, deposits 300, withdraws 100 and then transfers 99 to the other account. However, the final balance of *account2* is 399, instead of 300. This is because method *transfer* directly accesses *ac.amount,* where *ac* is an *Account* object



```
Thread 1                                          Thread 2
Account(100)
1 account1.amount = 100
deposit(300)
2 temp = account1.amount + 300
3 account1.amount = temp
withdraw(100)
4 temp = account1.amount - 100
5 account1.amount = temp    switch point P1
                          ------------------->
                                                  Account(100)
6                                                 account2.amount = 100
                                                  deposit(300)
7                                                 temp = account2.amount + 300
8                                                 account2.amount = temp
                                                  withdraw(100)
9                                                 temp = account2.amount - 100
10                          switch point P2       account2.amount = temp
transfer(99)                <-------------------
11 temp = account1.amount - 99
12 account1.amount = temp
13 temp = account2.amount + 99 switch point P3
                          ------------------->
                                                  transfer(99)
14                                                temp = account2.amount - 99
15                                                account2.amount = temp
16                                                temp = account1.amount + 99
17                           switch point P4      account1.amount = temp
18 account2.amount = temp   <-------------------
```

Figure 2-3 A failed execution with class Account.

passed to method *transfer* as an argument. In our example scenario, when Thread 1 calls method *transfer*, it only acquires the lock for *account1*. So the lock for *account2* can still be acquired by Thread 2, which can access and modify *account2* concurrently with thread 1.

Below we illustrate how our approach is used to identify the erroneous patterns in the example execution shown in Figure 2-3. Our technique has two phases. In the first phase, we identify the erroneous switch point. In the second phase, we identify the erroneous event patterns.

Switch points are checked in the reverse order of their occurrence in the failed execution. Thus, the first switch point checked is P_4. Step 1 of the controlled execution for P_4 replays the execution up to and including the execution of statement 18 in Thread 1 (We will explain how to obtain the replay portion of a failed execution in the Section IV.). This ensures that switch point P_4 appears in the new execution. In step 2 of the execution, a “least concurrency” policy is enforced. Since there are no more statements to execute, the least concurrency part of the controlled execution does not exercise any events. The resulting execution fails, allowing us to conclude that switch point P_4 or one of the switch points that precede P_4 is erroneous.

Next we generate a controlled execution to check switch point P_3 of the failed execution. Step 1 of the controlled execution replays the events up to and including the execution of statement 17 in Thread 2. This ensures that switch point P_3 appears in the new execution. Step 2 of the controlled execution enforces the “least concurrency” policy, which executes Thread 1 until it ends. The new execution fails, which allows us to conclude that switch point P_3 or one of the switch points that precede P_3 is erroneous.

Next we generate a controlled execution to check switch point P_2. Step 1 of the controlled execution replays the events up to and including the execution of statement 13 in Thread 1. This ensures that targeted switch point P_2 appears in the new execution. Step 2 of the controlled execution enforces the “least concurrency” policy, which can force Thread 1 to execute until it ends and then Thread 2 to execute until it ends. This controlled execution passes. We also can force Thread 2 to execute first in the least concurrency mode and the generated execution is failed. Since we find one passed execution for switch point P_2, switch point P_2 is benign, and we conclude that switch point P_3, which is the switch point immediately after benign switch point P_2, is the erroneous switch point.

Next, we try to detect erroneous event patterns related to erroneous switch point P_3. Our technique checks whether there are any unserializable interleaving patterns or conflicting interleaving patterns that are introduced by P_3. In Figure 2-3, statement 13 in Thread 1, statement 15 in Thread 2 and statement 18 in Thread 1 comprise an unserializable interleaving pattern of shared variable *account2.amount* introduced by switch point P_3. This implies that the value written by statement 15 in Thread 2 is unexpectedly overwritten by the write operation of statement 18 in Thread 1, i.e., the write operation of statement 15 in Thread 2 should not interrupt the execution of statements 13 and 18 in Thread 1. This pattern is output to the programmer to guide debugging.

Delta debugging [1] fails to point out P_3 is the erroneous switch point. Assume the alternative passed execution is generated with switch points 5,11,13,16, while those for the failed execution in Figure 2-3 are 5,10,13,17. Since both executions have a switch point P_3 at event 13 and delta debugging only detects the erroneous switch point from the difference between the schedules of two executions, delta debugging would point out that P_2 in the failed execution is the cause of failure, which however does not make the execution enter erroneous state. The fundamental reason is that delta debugging only identifies the switch point that differs between the passed and failed execution and that if reconciled, would flip the result of the executions. Such a switch point does not necessarily create an erroneous state.

2.4 Our Algorithm

In this section, we describe our algorithm for fault localization in detail. Figure 2-4 shows algorithm *LocalizeErroneousPatterns*. This algorithm takes as input a program P and a totally-ordered sequence F of read and write events exercised by a failed execution of P. The output is a set of erroneous event patterns that trigger the failure. *LocalizeErroneousPatterns* has two major phases: (1) identifying the erroneous switch point; and (2) identifying the erroneous event patterns.

LocalizeErroneousPatterns begins by identifying the switch points in sequence F. A prefix *of F* is generated for each of the identified $switchPoint_i$ (lines 2-4). For $switchPoint_i$, the prefix contains all the events up to and including the event that immediately precedes $switchPoint_{i+1}$. The prefix for $switchPoint_i$ is used to replay the portion of F that contains $switchPoint_i$.

```
Algorithm LocalizeErroneousPatterns:
Input: program P, a totally ordered sequence F from a failed execution of P.
Output: a set erroneousPatterns of erroneous event patterns

1. Let switchPoints be a sequence of switch points in their order in F;
2. for ($switchPoint_i$ in switchPoints){
3.    create the prefix for $switchPoint_i$
4. }
5. for (int i = number of switch points; i >= 1; i --) {
6.    for (int j = 0; j < number of shared variables; j ++) {
7.       let P replay $prefixFile_i$ first and then execute in the least concurrency mode
8.       if(the generated execution passes){
9.           record $switchPoint_{i+1}$ as the erroneous switch point
10.          break the outer for loop;
11.      }
12.   } // end inner for
13.} // end outer for
14. if(all the switch points are erroneous){
15.     record $switchPoint_1$ as the erroneous switch point
16. }
17. erroneousPatterns = {patterns collected with the erroneous switch point }
18. return erroneousPatterns
```

Figure 2-4 Algorithm LocalizeErroneousPatterns.

In phase 1, *LocalizeErroneousPatterns* checks switch points in the reverse order as they appear in sequence F (line 5-13). This process stops when it finds that *switchPoint*$_i$ is benign (line 8) and records *switchPoint*$_{i+1}$ as the erroneous switch point (line 9).

A controlled execution is used to replay the prefix generated for a targeted switch point and then force the execution to proceed in the least concurrency mode (line 7). The reason why the least concurrency mode is used after replaying the prefix is as follows. Atomicity violation and order violation are both due to interleavings of concurrent shared variable accesses. Assume that switch point P_i is targeted, and that switch point P_{i+1} is the erroneous switch point. When the prefix for P_i is replayed, and the least concurrency mode is used to complete the execution, interleavings of shared variable accesses are minimized. This means that additional switch points, including erroneous switch point P_{i+1}, can be avoided after P_i is replayed, allowing the execution to pass. If the prevention of P_{i+1} from being executed allows the execution to pass, switch point P_{i+1} is identified as the erroneous switch point.

Multiple controlled executions are used to determine whether a targeted switch point is benign or not. The following heuristic is used to identify benign switch points: if at least one controlled execution passes for a targeted switch point (it means the execution with the targeted switch point can pass), then we conclude that this switch point is benign and that the other failed controlled executions for this switch point are due to the event patterns introduced during the least concurrency mode. Likewise, if all the controlled executions for a targeted switch point fail, then this switch point is identified to be the erroneous switch point. With this heuristic, we ignore the possibility that the target switch point is benign and all the failures are due to the least concurrency part. From the empirical study results in section 5, our algorithm works effectively under this heuristic.

Based on this heuristic, *LocalizeErroneousPatterns* performs controlled executions until an execution passes, or a maximum number of executions, which equals the number of shared variables, have been performed (line 6). This limit is set to the number of shared variables because of the strategy we use to generate controlled executions, which is described as follows.

For each shared variable *s*, the last thread L that accessed *s* in the replay mode is allowed to execute first in the least concurrency mode and continue execution until it blocks or terminates. In this way, no accesses from other threads can interrupt the access of *s* by L and we can avoid any potential erroneous event patterns for *s* that may occur at the boundary between the replay part and the least concurrency part in the controlled execution. The threads that execute after L are randomly selected. The number of controlled executions required by this strategy is equal to the number of shared variables. Note that this strategy does not allow us to determine with certainty whether a targeted switch point is erroneous. However, the empirical study in Section 5 suggests that this strategy can be effective for many programs.

As we mentioned above, switch points are targeted in the reverse order of their appearance in F. If switch points were instead checked in the order as they appeared in F, we could not conclude that the first switch point P_i that makes all the controlled executions fail is the erroneous switch point in F. This is because some switch point P_j after P_i may allow the execution to recover, making a later switch point P_k the cause of the failure. When switch points are checked in the reverse order, we can conclude that the switch point P_{i+1} that follows the first benign switch point P_i is the switch point that caused the original execution to fail. This is because after P_{i+1} is introduced into the executions all the controlled executions fail, which indicates that P_{i+1} is the switch point that causes the failure.

We point out that binary search cannot be used to find the erroneous switch point. During our detection, the executions generated for each switch point can be all failed or contain at least one passed execution, because the execution can recover from some erroneous states. Since binary search can only be applied when the elements are sorted, our search process cannot use binary search. For example, if we test all the switch points in a failed execution, we may get the following result: P P F F F P F F F. (P represents the switch point is benign and F represents the switch point is erroneous) It indicates that the error introduced by switch point 3 can be recovered by switch point 6. Applying binary search, we will identify switch point 3 as the erroneous switch point. But this error cannot be seen from the output.

After an erroneous switch point is identified, phase 2 of *LocalizeErroneousPatterns* identifies the erroneous event patterns related to the erroneous switch point. Let switch point P_i be the erroneous switch point. Assume that thread A executes between switch point P_{i-1} and P_i, and thread B executes between switch point P_i and P_{i+1}. To identify unserializable interleaving patterns, which are associated with atomicity violations, for each shared variable we select one event from each of the following three blocks of events: (1) the events executed by Thread A between switch point P_{i-1} and P_i; (2) the events executed by Thread B between switch point P_i and $P_{i+1;}$ and (3) the events executed by Thread A between switch point P_j and P_{j+1}, where P_j is the first switch point at which control switched back to Thread A after switch point P_i. The reason why the algorithm can detect unserializable interleaving patterns from these three blocks of events is because all the unserializable interleaving patterns share the following property. Referring to the five unserializable interleaving patterns in Figure 2-1, for each pattern, after the first two events are exercised, the execution enters an erroneous state. For example, for the pattern (T1:R)-(T2:W)-(T1:R), the two reads by

T1 expect to read the same value, but after the execution of (T1:R)-(T2:W), the second read by T1 cannot read the same value as the first read. Thus, if a failed execution has an unserializable interleaving pattern, the erroneous switch point found by our algorithm is the switch point between the first and the second event in the pattern, and the third event will be executed by the same thread as the first event.

After the three blocks of events are identified, for each shared variable, the algorithm selects one event from each of the three blocks, which is the same scheme used by Falcon [5]. Preference is given to the selection of write events, since unserializable interleaving patterns require at lease one write event [5]. The algorithm then checks whether the selected events comprise an unserializable interleaving pattern. All the patterns for all the shared variables are output to the user to guide debugging. If we cannot find any unserializable interleaving patterns, we search for conflicting interleaving patterns, which are associated with order violations, with a similar process, but conflicting interleaving pattern only contains two events.

2.5 Empirical Study

Our fault localization algorithm has been implemented in a tool called *Huatuo*. As a proof-of-concept, we used *Huatuo* to conduct several empirical studies of our fault localization technique on a suite of faulty multithreaded Java programs. Our objective was to investigate the following two questions:

(1) What is the most effective and efficient technique for controlling thread executions during the least concurrency mode?

(2) How effective and efficient is algorithm *LocalizeErroneousPatterns* at finding the faults?

Since tools are not available for the techniques most closely related to ours, such as replay analysis [23], Falcon [5], and delta debugging [1], we are not able to compare experimental results from our technique with results from the other techniques. In section 6, we will compare our approach to these techniques.

We selected 13 programs for our empirical study, all of which are faulty concurrency programs that are used in [5]. Compared to the empirical study in [5], we removed the following programs: (1) Hedc, which has a bug hidden in the library code and we cannot instrument the library code; (2) Philo and Tsp, both of which never failed even though we executed them for four hours; and (3) TreeSet, which has the same bug with HashSet in the super class collection [9]. Note that for program BufWriter, the main() function calls Thread.sleep() to give its child threads time to finish. Our tool cannot deal with operation sleep(), so we modified BufWriter to use join() instead of sleep(). In order to conduct the empirical study, we rewrite these programs using the Modern Multithreading library. The failed executions were traced and recorded using the Modern Multithreading library. We manually inserted some assertions in the programs to determine whether a test execution fails or succeeds.

2.5.1 Study 1: Selecting the First Thread to Run in the Least Concurrency Mode

The goal of this empirical study was to answer Question (1). To answer Question (1), we implemented two techniques for controlling thread executions during the least concurrency mode. Technique 1 used the strategy described in Section 4 for controlling thread executions. This strategy makes a careful selection of the first thread to execute during controlled executions. Technique 2 was to perform five test executions for each switch point, with the first thread to execute randomly selected.

Table 2-1 shows the result of this study. The first column identifies the subject programs. The second column shows the size of each program in terms of lines of code (LOC). The third column shows the number of shared variables in each program. The shared variables were identified manually based on documentation and source code. The fourth column shows the number of threads for each program. The fifth column shows the total number of executions when technique 1 was used to search for the erroneous switch point. The sixth column shows whether technique 1 can successfully identify the erroneous event patterns. Columns 7 and 8 show the results for technique 2 in the same format.

Table 1 shows that both techniques can successfully find the erroneous switch points and the erroneous event patterns, except for program RayTracer. This is because RayTracer contains a large number of threads and at some switch points the controlled execution can pass only if a specific thread is executed first in the least concurrency mode. Thus, the probability that the first thread chosen by technique 2 results in a passed execution is low. As we mentioned above, five controlled executions were performed at each switch point using technique 2. However, this was not enough for technique 2 to generate a passed execution. Technique 1 generated a passed execution for RayTracer, and overall, technique 1 only required half of number of executions required by technique 2 for generating a passed execution for the subject programs.

Table 2-1 Comparison between two strategies for controlled executions

Program	LOC	# of shared variables	# of threads	Technique 1		Technique 2	
				Total # of executions	success	Total # of executions	success
Account	177	2	3	8	Yes	17	Yes
AirlineTickets	142	2	7	14	No	40	No
BubbleSort2	184	3	3	13	Yes	21	Yes
BufWriter	183	3	3	9	Yes	17	Yes
Lottery	154	2	3	11	Yes	23	Yes
MergeSort	375	3	4	10	Yes	16	Yes
Shop	226	11	3	10	Yes	22	Yes
Arraylist	5898	3	3	8	Yes	20	Yes
HashSet	7103	10	3	7	Yes	11	Yes
StringBuffer	1380	33	3	13	Yes	21	Yes
Vector	760	5	3	10	Yes	16	Yes
Cache4j	3976	2	2	3	Yes	6	Yes
RayTracer	2047	2	17	14	Yes	41	No

2.5.2 Study 2: Effectiveness and Efficiency

The results in Table 2-1 show that our technique correctly identified the erroneous event patterns for all the programs except program *AirlineTickets*. Program *AirlineTickets* fails even when the program executes serially and all of the passed executions need an extra common switch point that is missing in the failed executions. So all the controlled executions are failed, which make our algorithm to conclude that the first event after the first switch point is erroneous. But the real fault is that it needs a context switch at a specific point. Also, our algorithm identified a single erroneous event pattern for all the programs except for program *Bubblesort2*, for which two patterns were identified, which means one or both of the patterns can help the user localize the faults. In general, the algorithm does not make assumptions about what synchronization mechanisms are used in the concurrent program and can localize the faults for 12 out of 13 programs except the one that cannot pass when it is executed serially.

The instrumentation for the replay and the least concurrency mode slows down the executions. We performed 6 executions on the original version and the instrument version of program *RayTracer* which was the largest program in our benchmark. The slowdown factors are 22.4, 20, 21.5, 25.2, 26.4, and 25.25. On average, our instrumented executions took 23.5x longer than non-instrumented executions. This is faster than the results in [3, 11, 24], which reported average slowdowns from 25x [3] to more than 200x [24]. Although the slowdown of Falcon [2] is 9.9x, which is faster than our tool, our systematic search technique needs fewer executions than the techniques based on training [2, 3].

2.6 Related Work

In [23], the authors tried to distinguish benign races from erroneous ones after they detect all the data races. They execute a program twice for a given data race —

once for each of the two possible orders of conflicting memory operations. However, when testing the alternative order, there is no guarantee the alternative execution is feasible. As mentioned in section 4.2.1 in [23], the alternative execution is possible to follow a totally different data and control flow, making it impossible to execute the alternative order of conflicting memory operations. The authors classify this as a replay failure, and in their experiments, 29 benign data races were potentially harmful races. Using the least concurrency mode of execution, our technique can determine whether or not a data race is benign by only replaying the orders of conflicting memory operations that appear in the failed execution, without having to test the alternative orders.

The AVIO method [3] uses heuristics to automatically extract access interleaving invariants and detect violations of these invariants at run time. Defuse [22] uses training to learn definition-use invariants and considers violations of these invariants to be erroneous. Since both AVIO and Defuse are invariant-based approaches, they can only report erroneous patterns that only appear in failed executions. Our technique can identify erroneous patterns in the failed execution, even when these patterns also appear in passed executions. The reason why AVIO and Defuse may miss some patterns is because it is assumed that any execution that contains an erroneous pattern will fail. However, this is not always true. A pattern that triggered the failure in a failed execution can also appear in passed executions [5].

Falcon [5] monitors memory-access sequences among threads, detects data-access patterns associated with a program's pass/fail results, and ranks data-access patterns with regards to how suspicious they are. The main drawback of this technique is that highly suspicious patterns may not be the patterns that caused a failure; rather, they may be patterns that are resulted from the erroneous patterns. Our technique systematically tests the switch points in the failed execution one by one, and can pinpoint the actual erroneous pattern that triggers the failure.

By systematically narrowing down the difference between a failed thread schedule and a passed thread schedule, the Delta Debugging approach [1] can pinpoint the thread switch that differs between the two schedules and that if reconciled, would flip the result of the two schedules. As shown in section 3, such a thread switch may not actually produce an erroneous state. Our technique can find the switch point and the event pattern that actually produces an erroneous state which we believe is of more help for debugging.

2.7 Conclusion

In this paper, we presented an algorithm for identifying erroneous event patterns in concurrent executions. Failed executions are characterized as a sequence of events, which capture the interleaving of read and write events on shared variables. The algorithm inputs the sequence of switch points of a failed execution, and then uses controlled executions to distinguish erroneous switch points from benign switch points. The output of the algorithm is the erroneous event patterns. The event pattern can guide the user in locating the actual fault that triggered the failure. The algorithm is implemented in a tool called *Huatuo*. The results of our empirical study show that Huatuo can effectively and efficiently identify erroneous event patterns.

There are a number of venues to continue our work. First, we plan to conduct more experiments to evaluate the effectiveness of our approach. In particular, we want to conduct experiments on more complex real-life programs. Second, our approach currently deals with switch points, and a concurrent execution may consist of a large number of switch points. We will explore the idea of grouping switch points. Doing so will help reduce the number of switch points that have to be checked by our approach. Finally, we want to further develop our prototype tool and release it as an open-source tool.

Chapter 3. A Lightweight, Static Approach to Detecting Unbounded Thread-Instantiation Loops

This chapter contains a paper published in IEEE eighth International Conference on Software Testing, Verification, and Validation (ICST), in 2015.

A Lightweight, Static Approach to Detecting Unbounded Thread-Instantiation Loops*

Jing Xu[1], Yu Lei[1], Richard Caver[2], David Kung[1]

[1]Dept. of Computer Science and Engineering, University of Texas at Arlington, Arlington, TX, USA

jingxu@mavs.uta.edu, {ylei,kung}@uta.edu

[2]Dept. of Computer Science, George Mason University, Fairfax, VA , USA

rcarver@gmu.edu

***Abstract*—**In server applications, threads are created to handle incoming requests. Since threads consume significant resources including CPU cycles and memory, it is important to control the number of threads that are created. In this paper, we introduce a lightweight, static approach to detecting unbounded thread-instantiation loops that may exist in a server application. The key observation of our approach is that threads are objects of special significance and the decision logic for thread instantiation is typically not complex. Our approach checks loops against some bounded thread-instantiation patterns. A loop is considered bounded if a pattern match is found. Otherwise, it is considered unbounded. Our approach is heuristic by nature. That is, it does not guarantee to detect all the unbounded loops and may report unbounded loops that are actually bounded. To evaluate the effectiveness of our approach, we report an Eclipse plugin called *ThreadBoundChecker* which implements our approach and an experiment on 24 real-life Java server applications. The results of our evaluation show that our approach can effectively detect unbounded thread-instantiation loops in these

* Copyright © 2015 IEEE. Reprinted, with permission, from Jing Xu, Yu Lei, Richard Carver, David Kung, A Lightweight, Static Approach to Detecting Unbounded Thread-Instantiation Loops, IEEE International Conference on Software Testing, Verification and Validation (ICST), April 2015.

applications. In particular, 12 unbounded thread-instantiation loops detected by our approach are confirmed by the original developers.

3.1 Introduction

In server applications, threads are created to handle incoming requests. Since threads consume significant resources including CPU cycles and memory, it is important to control the number of threads that are created. If this number is unbounded, the application may respond too slowly, or even crash, when there are many incoming requests. This constitutes a vulnerability that can be potentially exploited by a hacker, e.g., to launch a denial of service attack. Some server applications use fixed-size thread pools to manage all the threads. However, others instantiate threads directly, on demand, which makes them vulnerable to unbounded thread-instantiation loops.

If we can prove that a thread-instantiation loop will terminate, then we can conclude the number of thread instantiations in the loop is bounded. However, there is no general procedure for determining whether a loop will terminate [25]. Effective termination-analysis techniques have been developed for certain classes of programs [26], but termination is difficult and costly to prove, especially for large applications. Moreover, in server application, loops that are used to accept incoming requests may intentionally be non-terminating, but these loops can still have a bound on the number of thread instantiations. In this paper, we propose a heuristic approach to detecting unbounded thread-instantiation loops regardless of whether they terminate or not.

The key observation of our approach is that threads are objects of special significance, and are usually created before business scenarios are actually handled. Consequently, for many applications, the decision logic for thread instantiation is not complex and is usually intended to use one of several common patterns for bounding the number of thread instantiations. Our goal is not to prove that all the thread-instantiation loops terminate, or generate an exact bound on the number of loop iterations. Instead, our approach employs several patterns and heuristics that are designed to be effective, efficient, and scalable for detecting unbounded thread-instantiation loops in real-life applications.

Our approach consists of three major steps. First, we identify all of the *thread* classes, which are the Java built-in *Thread* class and its subclasses. Next, we generate a reverse call graph for each of the constructors of the *thread* classes. A reverse call graph identifies all the methods that directly or indirectly call a *thread* constructor. From this graph, we can collect all the paths on which a *thread* constructor is called. Finally, we analyse the paths in these reverse call graphs, which are referred to as reverse call graph paths. For each reverse call graph path, we locate all of the loops that contain thread instantiation, i.e., loops that contain at least one statement that calls a *thread* constructor. For each loop, we check its iteration structure and the conditions under which a thread instantiation may take place at runtime against some bounded thread-instantiation patterns which are commonly used for bounding the number of thread instantiations. A thread-instantiation loop is considered bounded if a pattern match is found. Otherwise, an unbounded thread-instantiation loop is detected and reported to the user.

Our approach has been implemented in an Eclipse plugin called *ThreadBoundChecker*, which was used to perform an empirical study on 24 Java programs, including 9 web servers, 8 network servers, and 7 chat servers. All of the programs are real-life programs from *java-source.net* [27], 16 of which contain more than 10 thousand lines of code. The results of our study show that our approach detected unbounded thread-instantiation loops in 11 out of 24 programs. A total of 41 unbounded thread-instantiation loops were found. Of these, 12 loops were unbounded, as confirmed by the program developers. For 26 loops, we did not get a response from the developers but we verified them to be unbounded by a manual inspection of the code. Detailed information about these unbounded thread-instantiation loops, including scenarios that demonstrate why these loops are truly unbounded, is posted on our website [28]. There were only 3 false positives, i.e., 3 (out of 41) loops that were reported to be unbounded but were in fact bounded. Details about these false positive loops are shown in Section V and also published on our website. For most of the 24 programs, the *ThreadBoundChecker* plugin took only a few minutes to finish on a personal laptop.

The rest of the paper is organized as follows. Section 2 uses a simple example to motivate our work. Section 3 describes our approach in detail and introduces six patterns and four heuristics. Section 4 presents the design and implementation of our *ThreadBoundChecker* Eclipse plugin. Section 5 uses *ThreadBoundChecker* to evaluate our approach on the 24 Java programs. Section 6 reviews related work. Section 7 provides concluding remarks and presents our plan for future work

3.2 A Motivating example

As a motivating example, Figure 3-1 shows three methods of the *Tornado* program from *java-source.net* [27]. *Tornado* is a multi-threaded web server that provides a full implementation of HTTP 1.1. Here, we focus on path analysis; path collection will be explained in Section 3. Assume that a reverse call graph path is collected in which method *run* calls method *spawnThreads* and method *spawnThreads* calls method *addThread*, which instantiates a *ServerThread*. This reverse call graph path has two loops, the while-loop in method *run* and the for-loop in method *spawnThreads*. If either of these two loops can execute the thread-instantiation statement an unlimited number of times, then the number of thread instantiations is unbounded; otherwise, the number of thread instantiations is bounded.

The conditions in the while-loop that may affect the execution of method *spawnThreads* include the loop-condition *true* and the if-condition *idleThreads < minIdleThreads*. The loop-condition *true* obviously does not bound the number of thread instantiations. To determine whether the if-condition *idleThreads < minIdleThreads* bounds the number of thread instantiations, we check it against several commonly used bounded thread-instantiation patterns. In particular, consider a pattern in which the left-hand side is monotonically increasing, and the right-hand side is a constant.

```
public void run() {
    int minIdleThreads = Tornado.getConfig().getMinIdle();
     int maxIdleThreads = Tornado.getConfig().getMaxIdle();
    while (true) {
        try {
             Thread.sleep(this.sleepTime);
        } catch (InterruptedException e) {}
        int idleThreads = this.threadPool.getIdleThreads();
        Tornado.log.debug(idleThreads + " idle; "
                + this.threadPool.getBusyThreads()  + " busy");
        if (idleThreads < minIdleThreads) {
              spawnThreads(minIdleThreads - idleThreads);
              continue;
        }
        if (idleThreads <= maxIdleThreads)
               continue;
        killThreads(idleThreads - maxIdleThreads);
     }
 }

 private void spawnThreads(int num) {
     Tornado.log.debug(num + "new threads spawned");
     for (int i = 0; i < num; i++)
          this.threadPool.addThread();
 }

 public void addThread() {
     ServerThread t = new ServerThread(this.serverGroup,
                              this.taskPool, this);
     t.start();
 }
```

Figure 3-1 A Motivating Example.

The right-hand side, *minIdleThreads*, of the if-condition is a value that is read from a configuration file, and that does not appear on the left-hand side of an assignment statement in the loop. Thus the right-hand side of this condition is considered to be constant. Next we check whether the left-hand side *idleThreads* of the condition is monotonically increasing.

In general, it is hard to prove the trend of a variable. In our approach, we apply the following heuristic: if a variable appears on the left-hand side of an assignment statement for a simple computation, e.g., by an increment operator ++, we consider this variable to be monotonically increasing.

The idea behind this heuristic is two-fold. First, the fact that the developer is using a complex computation to update a variable is probably an indication that this is not a simple update and thus that the variable's values are not likely to be monotonically increasing. A monotonical counter variable *i* is typically updated by using a simple arithmetic operation such as ++. Second, this heuristic avoids potentially complex analysis that is required for a complex update, which is important for making our approach lightweight and scalable to large applications. As discussed in Section 5, this heuristic was shown to be effective in our experiments.

Since *idleThreads* is updated by a method, i.e., *getIdleThread*, instead of a simple computation, it is not considered to be monotonically increasing. Hence, condition *idleThreads* < *minIdleThreads* does not match the pattern. In fact, this condition does not match any other commonly used pattern either. Thus we conclude that this condition does not bound the number of thread instantiations. In this example, the heuristic works, as a manual inspection of method *getIdleThread* indicates that it returns the value obtained by subtracting the number of busy threads from the total number of threads and this value is not monotonically increasing.

Since neither of the two conditions in the while-loop is found to bound the number of thread instantiations, the number of thread instantiations in this loop is considered to be unbounded. Regardless of whether the for-loop bounds the thread-instantiation statement or not, we consider that this reverse call graph path contains an unbounded thread-instantiation loop. This vulnerability in *Tornado* has been confirmed by the developer, and the code has been subsequently updated to address this issue.

3.3 Our Approach

In this section, we first give an overview of our approach and then we present an algorithm that implements our approach.

3.3.1 Overview

Our goal is to check whether a loop in a server application can instantiate an unlimited number of threads. The first step of our approach is to identify all of the thread classes, which include the Java built-in *Thread* class and its subclasses. Threads are instantiated when the constructors of these classes are invoked. Note that a class R that implements *Runnable* can be used to provide a *run()* method, but an instance of R must be provided to a *Thread* class constructor to instantiate a thread. Thus, we do not need to collect classes that implement *Runnable*.

In order to determine whether a thread instantiation, i.e., a call to a constructor of a *Thread* class, can be executed an unlimited number of times, we need to determine whether this instantiation is in an unbounded thread-instantiation loop. Suppose that this instantiation is in a method M, but it is not in an unbounded thread-instantiation loop in M. Then we need to check if any method M’ that calls M can be executed in an unbounded thread-instantiation loop, or any method that calls M’, and so on.

Thus, the second step of our approach is to generate the reverse call graph for each constructor of a thread class. The *reverse call graph* for a method M is a graph rooted at M in which each node represents a method and each edge (*f*, *g*) indicates that method *f* is called by method *g*. Recall that a call graph represents the *calling* relation, i.e., all the methods that are directly or indirectly called by the method represented by the root node. In contrast, a reverse call graph represents the *called* relation, i.e., all the methods that directly or indirectly call the method represented by the root node. For the snippet of program *Tornado* in Section 2, the corresponding reverse call graph consists of the following single path: *ServerThread constructor → addThread() → spawnThreads() → run()*.

The final step of our approach is to analyse each path in each reverse call graph to detect unbounded thread-instantiation loops. Hereafter we refer to each path in a reverse call graph as a reverse call graph path. During the analysis, we identify *thread-instantiation loops*, which are loops that contain one or more thread-instantiation statements (i.e., calls to a thread constructor). Each of the iteration structures of the loops is first checked against bounding iteration patterns which are the loop structures commonly used to bound the number of iterations of the loop. If a match is found, the number of iterations of this loop is bounded, which bounds the number of thread instantiations. If no match is found, we further identify all the thread-instantiation statements in each of these loops. For each thread-instantiation statement, we identify its reachability condition and the conditions indirectly control the thread instantiation (e.g., the conditions for *return* and *break* statements in the loop), i.e., the condition under which the thread-instantiation statement is executed. We refer to such conditions as thread-instantiation conditions. A thread-instantiation condition typically involves the termination condition of the loop and may also involve some branching conditions inside the loop body. Each thread-instantiation condition is checked against several bounding condition patterns, which represent conditions commonly used for bounding the number of thread instantiations. If a pattern match is found for a thread instantiation condition, the number of thread instantiations is considered bounded under this condition. The loop is considered bounded if all the thread-instantiation conditions are bounded. Otherwise, it is considered unbounded.

3.3.2 The algorithm

Fig. 2 shows algorithm *CheckThreadBound*. This algorithm takes as input a program *P* under test, the maximum nesting level of a nested method call, *LimitOfNestedCalls*, and the maximum nesting level of a nested loop, *LimitOfNestedLoops*. As discussed later, these two limits are used to control the size of a reverse call graph. Algorithm *CheckThreadBound* returns a set of unbounded thread-instantiation loops.

Algorithm *CheckThreadBound* begins by identifying all of the subclasses of class *Thread* (line 1). These classes, and class *Thread*, are stored in *tclasses*. A thread is instantiated when a call to a constructor of a thread class is made. Thus, algorithm *CheckThreadBound* collects all of the constructors of each thread class and stores them in *tconstructors* (line 4). For each constructor, a reverse call graph is generated (line 6).

```
Algorithm CheckThreadBound
Input: P - the subject program, LimitOfNestedCalls – the
          maximum nesting level of a nested method call,
          LimitOfNestedLoops – the maximum nesting level of a
          nested loop
Output: a set uloops of unbounded thread-instantiation loops

1  let tclasses be the set of all the thread classes in P;
2  let uloops be an empty set;
3  for each class tclass in tclasses {
4      let tconstructors be the set of all the constructors of tclass;
5      for each method tconstructor in tconstructors {
6          build a reverse call graph for tconstructor whose size
             is controlled by LimitOfNestedCalls and
             LimitOfNestedLoops;
7          analyse each path to detect unbounded
             thread-instantiation loops and add the detected loops
             into uloops
8      }
9   }
10  return uloops
```

Figure 3-2 Algorithm CheckThreadBound

A key decision in the construction of a reverse call graph is to control the size of the graph. Constructing a complete graph for each thread constructor is often not practical for large applications. In our algorithm, *LimitOfNestedCalls* is the maximum nesting level of a nested method call and *LimitOfNestedLoops* is the maximum nesting level of a nested loop. The reverse call graph for a thread constructor is built by exploring all of the possible paths in which the constructor can be called. The exploration of a path is stopped when *LimitOfNestedCalls* or *LimitOfNestedLoops* is reached.

The limits specified by *LimitOfNestedCalls* and *LimitOfNestedLoops* need to be carefully selected. On the one hand, if the limits are too large, the graphs can be too expensive to build and explore. On the other hand, if the limits are too small, some unbounded thread-instantiation loops may be missed. The experiments reported in Section V show that setting *LimitOfNestedCalls* to 3 and *LimitOfNestedLoops* to 2 is effective for detecting unbounded thread-instantiation loops. The intuition behind these two limits is that if the code sets a bound on thread instantiations, the bound tends to be set in a location that is close to where a thread constructor is invoked. More discussion about this is provided in Section V.

For each reverse call graph, *CheckThreadBound* traverses the graph and analyses each path to detect unbounded thread-instantiation loops (line 7). The details of path analysis are presented in Section 3.3. The bounded thread-instantiation patterns are shown in Section 3.4. Algorithm *CheckThreadBound* returns all of the unbounded thread-instantiation loops that are detected by path analysis (line 10). (The algorithm can be changed to stop when the first unbounded thread-instantiation loop is detected.)

3.3.3 Path analysis

Let H be a path in the reverse call graph built for a constructor M of a thread class. This path is an abstract path, as it only contains a sequence of method calls. To analyse H, we first generate a concrete path H' consisting of a sequence of statements that executes the sequence of method calls in H.

Consider the example in Fig. 1. A path in the reverse call graph for constructor *ServerThread* consists of the following sequence of method calls, *ServerThread constructor → addThread() → spawnThreads() → run()*. A concrete path that can be generated for this abstract path is as follows: <25, 22, 21, 20, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2>.

In general, given an abstract path H = <M_1, M_2, … $M_{|H|}$>, a concrete path H' = $H_1 \bullet H_2 \bullet \ldots \bullet H_{|H|-1}$ is generated, where H_i is a (control-flow) path in M_{i+1} that begins with a statement that calls M_i and ends with the first statement of M_{i+1}. Since a method can be called in multiple statements of another method, multiple concrete paths can be generated for an abstract path.

For each concrete path H', our analysis checks whether constructor M can be called in a loop in H'. If M cannot be called in a loop, this path does not contain an unbounded thread-instantiation loop. Otherwise, let L be a loop in which M can be called. L can be a simple or nested loop. In the following, we assume that L is a simple loop. If L is a nested loop, then the check that we describe next can be repeated for each loop, starting from the innermost loop.

As mentioned in Section 3.1, in order to determine if the number of thread instantiations is bounded in a given loop, we check each loop against several bounded thread-instantiation patterns. There are two types of bounded thread-instantiation patterns, bounding iteration patterns and bounding condition patterns. Bounding iteration patterns are patterns on iteration structures, which bound the number of loop iterations. Bounding condition patterns are patterns on thread-instantiation conditions under which the number of thread instantiations is bounded. We note that bounding iteration patterns can also be expressed as bounding condition patterns. A thread-instantiation condition may involve both the termination condition of a loop and branching conditions inside the loop body. As an example, consider the following loop:

```
Example:
    while (condition1) {
      if (condition2) {
            return;
      }
      if (condition3) {
            createThread();
      }
    }
```

This loop contains one thread-instantiation statement. Thus there exists one thread thread-instantiation condition: *condition1 && !condition2 && condition3*. Note that a condition that guards a *break* or *return* statement should be negated in the thread-instantiation condition.

A thread-instantiation condition can be a simple condition which is a boolean expression that does not contain any boolean operators, or a complex condition that contains simple conditions connected by boolean operators. A simple condition is said to be a bounding condition if it matches one of our bounding condition patterns. Determining whether a complex condition is bounding is done recursively as follows. If a complex condition C is a conjunction of two simple/complex conditions, then C is bounding if at least one of the two simple/complex conditions is bounding. If a complex condition C is a disjunction of two simple/complex conditions, then C is bounding if both of the two simple/complex conditions are bounding.

3.3.4 Bounded thread-instantiation patterns

In this section, all the bounded thread-instantiation patterns used in our approach, including the bounding iteration patterns and the bounding condition patterns, are introduced.

3.3.4.1 Bounding iteration patterns

If a loop iterates through a fixed-size collection, then the number of thread instantiations in this loop is bounded.

Pattern 1. A for-each loop is bounding that iterates through a non-concurrent Java collection.

The size of a non-concurrent Java collection is fixed during iteration. (This is because any modification that changes the collection's size will trigger a ConcurrentModificationException). This bounds the number of iterations and hence the number of thread instantiations.

For a concurrent Java collection, it is possible that new elements are added into the collection from the current thread or other threads during an iteration. This may make the number of iteration unlimited.

Pattern 2. A loop is bounding if it uses an iterator other than a *ListIterator* to iterate through a non-concurrent Java collection.

The reason why the iterator cannot be a *ListIterator* iterator is that a *ListIterator* iterator can iterate backwards and add a new value at any point which may make the number of iterations unlimited.

3.3.4.2 Bounding condition patterns

As mentioned earlier, bounding condition patterns are defined for simple conditions. In Java, a simple condition has one of the following four types: a boolean literal, a boolean variable, a boolean method invocation, or a relational expression. We define our bounding condition patterns based on these types. In the following discussion, a simple condition is assumed to be a reachability condition (or part of it) for a thread instantiation statement or a condition that indirectly bounds the number of thread instantiations, e.g., the condition for *break* or *return* statement.

3.3.4.2.1 *Boolean literal*

The boolean literal *true* is not bounding, since it can never prevent, by itself, the execution of a statement. Note that when *true* is the condition of a while-loop, the condition for a *break* or *return* statement may bound the number of loop iterations and the number of thread instantiations in the loop. In this case the conditions for the *break* or *return* statement must be collected and checked against the bounding condition patterns (after negation). The boolean literal *false* is always bounding, since the statements under the condition are never executed. But it is rarely used as a bounding condition. Thus we do not provide bounded thread-instantiation pattern for it.

3.3.4.2.2 *Boolean variable*

Pattern 3. A boolean variable *b* or its negation *!b* is bounding, if the following two conditions are satisfied:

1) There exists an assignment statement *s* in the loop body that negates *b*;
2) The negation of the reachability condition for statement *s* matches a bounding condition pattern.

If *b* is negated, the loop will terminate and no more threads will be created. So the negation of the reachability condition for statement *s* needs to be checked against the bounding condition patterns.

```
Example:
    boolean continue = true;
    while (continue) {
        new Thread();
        if (condition) { continue = false; }
     }
```

In this example, *continue* is the reachability condition for the only thread instantiation statement, and *condition* is the reachability condition for the assignment statement that updates *continue* to false. A new thread is created only when *condition* is false. Thus, *!condition* should be checked against the bounding condition patterns to determine whether it bounds the number of thread instantiations.

3.3.4.2.3 *Boolean method invocation*

Pattern 4. A boolean method invocation that checks whether a collection *v* is empty in a loop is bounding if the following three conditions are satisfied:

1) There exists one or more methods that remove an element from collection *v* in the loop body;
2) There exists no methods that add an element into collection *v* in the loop body; and
3) Collection *v* is only accessed by a single thread.

If new elements are added, or no element is removed, during loop iterations, then a collection may not become empty and the loop may not terminate.

If a collection is accessed concurrently, new elements may be added into this collection from other threads during the iteration, which may make the number of iterations unbounded.

```
Example:
    List l = new LinkedList(elements);
    while (!l.isEmpty()) {
       MyThread t = new MyThread(l.remove());
    }
```

In each iteration, a new thread is created with an element in the linked list and the element is removed from the linked list. Also, the current thread is the only thread that accesses the linked list. Thus, the number of thread instantiations is consistent with the initial number of elements in the linked list.

The following heuristic is used to determine whether a collection is accessed by a single thread or not.

Heuristic *1:*

If one of the following three conditions is satisfied, we consider that a collection is accessed by multiple threads; otherwise, it is accessed by a single thread:

1) A collection is a concurrent Java collection, or is synchronized by a synchronizing method, like *synchronizedCollection*, *synchronizedMap*, and so on.
2) An object of a type in the *java.util.concurrent* package is used to protect a method that accesses a collection object. For example, if a method that accesses a collection is protected by a variable of type *ReentrantLock* or *Semaphore*.
3) One or more methods that access a collection are either *synchronized* or *run* methods.

3.3.4.2.4 *Relational expression*

Pattern 5. A relational expression of the form *left relational_operator right* is bounding if the following three conditions are satisfied:

(1) *left* and *right* are integer operands;

(2) *relational operator* is one of >, >=, <, <=, ==, or !=;

(3) operands *left* and *right* satisfy any of the following constraints.

- A. when the operator is > or >=:
  - a. operand *left* is constant and *right* is monotonically increasing or
  - b. operand *left* is monotonically decreasing and *right* is constant.
- B. when the operator is < or <=:
  - a. operand *left* is constant and *right* is monotonically decreasing or
  - b. operand *left* is monotonically increasing and *right* is constant.
- C. when the operator is == or !=:

a. one operand is constant and the other is monotonically increasing or monotonically decreasing.

When the operator is == or !=, the operands of a relational expression can be booleans. If one side is a boolean variable or a boolean method invocation and the other side is a boolean literal, the expression is equivalent to a boolean variable or a boolean method invocation, Pattern 3 or Pattern 4 can be applied. In other cases (i.e., both sides are boolean variables or boolean method invocations), it is difficult to determine whether the expression is bounding or not, and such as expression is rarely used as a bounding condition. Thus, we do not provide any patterns for these cases.

Note that a relational expression with negation can be deduced to be a relational expression without negation, e.g., !(a < b) is equivalent with a >= b.

Next we provide several heuristics for determining whether an operand in a relational expression is constant, monotonically increasing, or monotonically decreasing.

Heuristic 2:

For an operand that is an *integer* variable to be constant, it can be a final variable or an variable (not a field of a class) that does not appear in the left-hand side of an assignment statement in the loop.

Heuristic *3:*

If a variable only appears in an increment operator; or on the left-hand side of an assignment statement whose right-hand side is a simple computation (like adding a constant or a variable does not appear on the left-hand side of an assignment statement) in the loop, we consider this variable to be monotonically increasing. A similar condition can be applied to monotonically decreasing variables.

This is inspired by the observation that when a variable is used as a counter, it is typically updated by using operator ++, or some other simple arithmetic expression. In the following example, *limit* is considered to be constant since it does not appear in any assignment statement in the *while*-body and *i* is considered to be a monotonically increasing integer since i is just updated by ++ in the *while*-body.

```
Example:
    int limit = 500;
    int i = 0;
    while (true) {
       if (i < limit) {
          Thread t = new Thread();
          i++;
       }
    }
```

Heuristic 4:

If the size() method of a collection is only accessed by a single thread, and there exist methods that remove elements from the collection, but no methods that add elements to the collection appear in the loop, then method size() is monotonically decreasing. A similar idea can be applied to monotonically decreasing and constant.

Pattern 6. A relational expression of form left == null is bounding if the following three conditions are satisfied:

1) left is a variable of type T;
2) left appears on the left-hand side of an assignment statement whose right-hand side is an instantiation expression;
3) left does not appear in the left-hand side of an assignment statement in the loop whose right-hand side is null.

Only when left is null, the condition is true and the thread can be initiated. Once a new instance is assigned to left which will no longer be assigned a null, the condition will be false. Thus, the number of thread instantiations is bounded.

```
Example:
    while (true) {
       if (t == null) { t = new Thread(); }
    }
```

In this example, only one thread can be instantiated, since a new instance is assigned to *t* after one iteration and *t* will no longer be assigned a *null*.

3.4 ThreadBoundChecker:An Eclipse Plugin

Eclipse is a multi-language, integrated development environment (IDE) consisting of a core workspace and an extensible plugin system for customizing the environment [29]. The core of Eclipse provides a basic user interface and internal control mechanisms. However, virtually every useful activity that can be performed in Eclipse relies on a plugin.

The Eclipse SDK includes the Eclipse Java development tools (JDT), which are a set of plugins that add the capabilities of a full-featured Java IDE and a full model of the user's Java source code.

JDT allows access to Java source code in two different ways - the Java Model [30] and the Abstract Syntax Tree (AST) [31] [32]. Each Java project is internally represented in Eclipse as a Java model. A Java model is a light-weight representation of the Java project that does not contain as much information as the Abstract Syntax Tree (AST) but a Java model can be created fast. The AST is a detailed tree representation of the Java source code. The AST defines an API to modify, create, read, and delete source code. To implement our *ThreadBoundChecker* plugin, both the Java Model and the Abstract Syntax Tree (AST) were used.

The *ThreadBoundChecker* plugin adds a new action item *AnalyseThreadBound* to the popup menu for each project in the project explorer viewer. By clicking on this action item, users can see all of the unbounded thread-instantiation loops detected by our approach with the default values of the two limits (i.e., the maximum nesting level of a nested method call is 3 and the maximum nesting level of a nested loop is 2). Users can also specify the values for these two limits.

ThreadBoundChecker has three major components:

- *Thread Class Finder*: This component identifies all the classes that extend the Java built-in *Thread* class. This is done by using class *ITypeHierarchy* in the Java Model. This class provides a way to navigate between a given type and its supertypes and subtypes in a program.
- *Reverse Call Graph Generator*: This component generates the reverse call graph for each constructor of each thread class identified earlier. This component uses the class *CallHierarchy* provided by the Java Development Tools (JDT), which allows one to find all the methods that call a given method.
- *Path Analyser*: This component is the main component of the *ThreadBoundChecker* plugin. This component is responsible for exploring a reverse call graph in a depth-first manner and analyses each path to detect unbounded thread-instantiation loops. As discussed in Section III, the analysis of each path is mainly conducted by checking against the bounding iteration and condition patterns.

3.5 Experiments

To evaluate the effectiveness of our approach, we conducted experiments on a set of Java server programs, which range from 1370 lines of code up to 823,376 lines. Table 3-I shows some statistics of the subject programs. All of these programs were obtained from *java-source.net*, which is a website that collects open source Java software [27]. The first two columns list the subject programs, which are grouped into three categories: *web server*, *network server*, and *chat server*. For the chat and network server categories, we included all the programs on *java-source.net* for which the source code is available. Since the web server category contained as many as 24 programs, we collected only the first 9 programs (as listed on java-source.net). The third column shows the size of each program in terms of the lines of code (LOC). The fourth column shows the number of classes that extend the *Thread* class.

The experiments were performed on a laptop with a 2.30GHz CPU and 4GB memory, running Windows 7(64-bit) and Sun's Java 1.5.

Our experiments consist of two major parts. The first part investigates the impact of reverse call graph size. The second part reports the results of analysing the 24 subject programs.

Table 3-1 Subject Programs

Program		LOC	# of Thread Classes
Web server	jetty	26605	2
	jicaralla	13077	1
	MJWS	25100	2
	Pygmy	7271	2
	reattore	12589	6
	resin	823376	12
	simple	36280	1
	Tomcat	362781	16
	Tornado	1577	3
Network server	ColoradoFTP	6792	1
	drftpd	33319	2
	ejbca	96269	1
	JGroups	101698	9
	jsocks	7281	1
	QuickServer	27400	5
	VeraxIPMI	25684	6
	xSocket	22538	3
Chat server	ace-app	16927	1
	ChipChat	2593	5
	ClarosChat	3138	4
	FreeCS	31321	13
	JavaMSNLibrary	28960	3
	LlamaChat	3887	3
	OpenChat	1370	1

Note: MJWS: MiniatureJavaWebServer

3.5.1 Impact of reverse call graph size

Recall that the size of a reverse call graph is controlled by using two parameters, i.e., *LimitOfNestedCalls* and *LimitOfNestedLoops*. During construction of a reverse call graph, the exploration of a path is stopped when either limit is reached. In the following, we first investigate the impact of *LimitOfNestedCalls*.

Table 3-2 shows the number of unbounded thread-instantiation loops detected by our approach, as well as the corresponding execution time, with *LimitOfNestedCalls* = 3, 4, 5 and *LimitOfNestedLoops* = 10. Since this experiment focused on the impact of *LimitOfNestedCalls*, the value of *LimitOfNestedLoops* was deliberately set to a big value, i.e., 10, so that the exploration of a path was likely to be stopped due to *LimitOfNestedCalls*, instead of *LimitOfNestedLoops*. The execution time was limited to 60 minutes.

Table 3-2 Impact of *LimitOfNestedCalls*

Program	#Unbounded # of Thread-Instantiation Loops			Execution Time		
	L3	L4	L5	L3	L4	L5
jetty	0	0	0	4’12	4’48	5’11
Jicaralla	0	0	0	3’57	4’18	4’56
MJWS	3	4	4	3’43	6’59	10’15
Pygmy	0	0	0	3’4	3’39	4’2
reattore	2	3	3	3’7	4’1	4’19
resin	0*	0*	0*	>60’	>60’	>60’
simple	0	0	0	3’16	4’11	4’37
Tomcat	1	3*	0*	52’5	>60’*	>60’*
Tornado	0	0	0	0’26	0’45	1’14
Colorado	3	4	5	1’40	3’9	5’52
Drftpd	1	1	1	0’58	1’8	1’24
ejbca	1	1	1	11’50	13’28	15’41
JGroups	1*	3*	2*	>60’	>60’	>60’
jsocks	2	3	4	1’38	1’59	2’35
QS	0	3	4	8’4	8’13	10’31
VeraxIPMI	0	0	0	2’21	2’38	4’15
xSocket	0	1	1*	18’14	31’18	>60’
ace-app	0	0	0	1’36	1’48	2’12
ChipChat	1	1	1	0’30	0’45	0’58
ClarosChat	0	0	0	0’33	0’36	0’54
FreeCS	0	0	2	7’23	7’40	8’1
JML	0	2	2	4’49	5’13	6’19
LlamaChat	1	1	1	0’55	1’27	1’35
OpenChat	2	2	2	0’14	0’16	0’17

Notes: (1) MJWS: MiniatureJavaWebServer; QS: QuickServer; JML: JavaMSNLibrary; (2) L3 indicates *LimitOfNestedCalls* = 3, L4 indicates *LimitOfNestedCalls* = 4, L5 indicates *LimitOfNestedCalls* = 5; (3) “*” indicates all the results obtained from a partial execution;

The results in Table 3-2 show that for 14 out of 24 programs, the number of unbounded thread-instantiation loops detected is the same when *LimitOfNestedCalls* = 3, 4 or 5. Furthermore, for 6 of the remaining 10 programs, at least one unbounded thread-instantiation loop is detected when *LimitOfNestedCalls* = 3. On the other hand, the execution time increases when *LimitOfNestedCalls* increases from 3 to 5. Such an increase is significant in several cases. For example, for program *MJWS*, the execution time increases from 3’43 when *LimitOfNestedCalls* = 3 to 10’15 when *LimitOfNestedCalls* = 5. This suggests that *LimitOfNestedCalls* = 3 is a reasonable choice, especially when time is constrained.

Next we investigated the impact of *LimitOfNestedLoops*. Table 3-3 shows the number of unbounded thread-instantiation loops detected by our approach, as well as the corresponding time, with *LimitOfNestedLoops* = 2, 3, 4 and *LimitOfNestedCalls* = 10. Again, the execution time was limited to 60 minutes.

Table 3-3 Impact of *LimitOfNestedLoops*

Program	#Unbounded Thread-Instantiation Loops			Execution Time		
	L2	L3	L4	L2	L3	L4
jetty	0	0	0	4’1	4’22	4’38
Jicaralla	0	0	0	3’44	4’7	4’27
MJWS	3	3	3	18’54	26’27	44’44
Pygmy	0	0	0	1’15	1’50	1’59
reattore	3	3	3	7’55	8’29	9’22
resin	0*	0*	0*	>60’	>60’*	>60’*
simple	0	0	0	3’8	2’50	3’00
Tomcat	3*	3*	3*	>60’*	>60’*	>60’*
Tornado	0	0	0	0’17	0’21	0’22
Colorado	5	5	5	6’25	8’26	11’40
Drftpd	1	1	1	0’49	1’03	1’12
ejbca	1	1	1	8’9	12’10	12’46
JGroups	1*	1*	1*	>60’*	>60’*	>60’*
jsocks	4	4	4	2’8	2’8	2’35
QS	4	4	4	14’19	15’20	15’48
VeraxIPMI	0	0	0	2’2	2’18	2’42
xSocket	1*	2*	2*	>60’*	>60’*	>60’*
ace-app	0	0	0	1’19	1’29	1’45
ChipChat	1	1	1	1’3	1’25	1’58
ClarosChat	0	0	0	0’25	0’27	0’29
FreeCS	7	8	8	11’25	14’37	14’58
JML	2	2	2	12’24	13’5	13’29
LlamaChat	1	1	1	1’23	1’34	1’58
OpenChat	2	2	2	0’15	0’17	0’21

Notes (1) MJWS: MiniatureJavaWebServer; QS: QuickServer;
JML: JavaMSNLibrary; (2) L2 indicates *LimitOfNestedLoops* = 2, L3 indicates *LimitOfNestedLoops* = 3, L4 indicates *LimitOfNestedLoops* = 4; (3) “*” indicates all the results obtained from a partial execution;

The results in Table 3-3 show that we detected almost the same number of unbounded thread-instantiation loops when *LimitOfNestedLoops* = 2, 3, 4 (the only exceptions are programs *XSocket* and *FreeCS*), whereas the execution time may increase significantly as the value of *LimitOfNestedLoops* increases. As an example, *MJWS* whose execution time is doubled when *LimitOfNestedLoops* is from 2 to 4 but no more unbounded thread-instantiation loops are detected. This suggests that *LimitOfNestedLoops* = 2 is a reasonable choice, especially when the time is constrained.

We emphasize that the results from this experiment are consistent with our intuition. That is, if the programmer intends to set a bound on the number of thread instantiations, he or she tends to set the bound in a location close to where a thread constructor is called.

3.5.2 Detection Results

Table 3-4 shows the detection results of applying our *ThreadBoundChecker* tool to the 24 subject programs. In these experiments, *LimitOfNestedCalls* is set to 3, and *LimitOfNestedLoops* is set to 2. With these two limits, we cannot detect all of the unbounded thread-instantiation loops. However, by investigating all the detected unbounded thread-instantiation loops, we can still assess the effectiveness of our patterns and heuristics.

Column 1 shows the subject programs. Column 2 shows the number of bounded thread-instantiation loops. Column 3 shows the number of unbounded thread-instantiation loops. Column 4 shows the number of false positives. Column 5 shows the execution time. *ThreadBoundChecker* did not find any thread-instantiation loops for four programs, i.e., *jetty*, *Jicaralla*, *Simple* and *ace-app*. The first three programs use a 3rd-party thread pool class from a jar file for which we did not have source code access. The last program uses the *main* thread to handle incoming requests, and thus does not instantiate any new threads.

ThreadBoundChecker detected a total of 41 unbounded thread-instantiation loops. We reported 20 of the 41 unbounded loops to the open source developers for whom contact information was available. The developers responded to 14 of the reports. Their responses confirmed 12 of the 14 unbounded loops to be truly unbounded, which makes them vulnerable, and the other 2 to be false positive. We verified the other 27 unbounded loops by a manual inspection, and found 1 of them to be a false positive. We have created a web page [28] that contains more detailed information about these unbounded loops and scenarios that demonstrate that these loops are truly unbounded.

Table 3-4 Detection Results with *LimitOfNestedCalls* = 3 and *LimitOfNestedLoops* = 2

Subject	# of bounded thread-instantiation loops	# of unbounded thread-instantiation loops	# of false positives	Execution time
jetty	0	0	0	3’46
Jicaralla	0	0	0	3’28
MJWS	0	3	0	4’56
Pygmy	4	0	0	2’5
reattore	0	2	1	2’29
resin	3	6	0	442’23
simple	0	0	0	2’51
Tomcat	6	0	0	59’7
Tornado	3	1	0	0’22
Colorado	0	2	0	1’34
drftpd	1	2	0	1’4
ejbca	1	1	1	8’6
JGroups	201	18	0	1015’59
jsocks	1	2	0	2’5
QuickServer	0	0	0	7’14
VeraxIPMI	3	0	0	2’28
xSocket	3	0	0	15’47
ace-app	0	0	0	1’26
ChipChat	0	1	0	0’24
ClarosChat	0	0	0	0’24
FreeCS	4	0	0	8’22
JML	0	0	0	5’37
LlamaChat	0	1	0	0’42
OpenChat	0	2	1	0’13
Total	230	41	3	

Notes (1) MJWS: MiniatureJavaWebServer; JML: JavaMSNLibrary

In the following we discuss the three false positives reported by our approach. Since we do not know any faults that are not detected by our tool, we cannot show the number of false negatives. Our approach is very likely to provide false negatives. But the problem of precisely determining unbounded thread-instantiation loops is undecidable. Thus, no optimal solution exists that guarantees no false negatives or positives.

False Positive 1: In the *reattore* program, the following while-loop appears in the class *ListVar* of package *juju.reattore.perfcap.var.impl*.

The threads are instantiated in method *begin()*. Since the initial value of *current* is null, the first iteration of the while-loop will be executed. If *it.hasNext()* returns *true*, *current* will be assigned a non-null value and *begin()* will be invoked during this iteration. Only when *current.hasNext()* returns *false* will the second iteration be executed. Then *it.hasNext()* will return *false* and the loop will exit. Thus, the while-loop is bounded. However, this is reported as an unbounded thread-instantiation loop by our approach, since this loop does not match any of our thread-instantiation patterns.

```
public boolean hasNext() throws Exception {
  while (current == null || current.hasNext() == false) {
    if (current != null) {
      current.end();
      current=null;
    }
    if (it.hasNext() == false) {
      return false;
    }
    current=(Variable)it.next();
    current.begin();
  }
  return true;
}
```

False Positive 2: In *ejbca*, the false positive is attributed to performance testing code that will not execute when the application is deployed. The tester intentionally makes the test be able to run forever until the test is forced to stop. Note that it is a true unbounded loop if the code is executed.

False Positive 3: In *openChat*, there is a loop that uses *Integer.parseInt (getProperty("CHAT_SERVER_WORKERS"))* as a bound, which is a method invocation. Although the value returned by *parseInt* never changes, our current condition patterns are unable to classify the return value as a constant. In the future we plan to add a heuristic that if a return value by a method is read from an external input, like a configuration file or a property file, then this return value is a constant value.

3.6 Related work

Termination analysis is an area of work that is related to ours. Termination analysis techniques are typically based on ranking functions, which map program states to the elements of a well-founded domain. The ranking function strictly decreases on each computation step, thus guaranteeing termination. In general, it is hard to find and validate ranking functions. Many techniques [26, 33, 34, 35, 36, 37, 38] have been developed in this area. These techniques are typically based on assumptions that may not hold in practice, e.g., loops are not nested [38]. Most of the techniques ignore non-linear arithmetic and do not scale well to large programs. Our approach is based on the assumption that the structures used to instantiate threads are simple patterns and can be efficiently detected. Whether variables are updated linearly or non-linearly does not affect our approach.

A second area of work that is related to ours is worst case execution time (WCET) analysis. To statically derive a bound on the execution time of a program, a bound on the number of loop iterations must be derived. Consequently, a lot of WCET research has been done on automatic loop-bound analysis. The WCET tool provided in [39] performs loop-bound analysis using interval-based abstract interpretation and pattern matching. The loop-bound analysis of the Bound-TWCET tool [40] is based on Presburger arithmetic. Different loop bounds can be obtained for different calling contexts, since the loop bounds are context sensitive. SWEET [41] uses value analysis, abstract execution, and syntactical analysis to get a loop bound. The purpose of loop-bound analysis in WCET is to get a concrete bound on the execution time of a loop. These analysis techniques are complex and time consuming, and are limited to specific types of loops. Looper [42] uses a dynamic method to detect infinite loops in general loops, but requires an input generated by an SAT solver.

The goal of our analysis technique is to determine whether the programmer intended the thread instantiations in a loop to be bounded or not. That is, the specific number of bounds is not of our concern. Our approach is heuristic, i.e., it is not guaranteed to detect all unbounded thread-instantiation loops, but it is efficient and effective and can be applied to large programs.

Our work is also related to work on stress testing, which focuses on how to generate test cases and how to analyse the results. A Markov Model can be used to generate load test suites automatically [43, 44, 45, 46, 47]. Zhang presents a mixed symbolic execution approach aimed at discovering execution paths that contribute to high program loads while ensuring path diversity [48]. Malk presents a methodology to help automatically identify important performance counters for load testing and compare the counters across tests to find performance gain/loss [49]. Jiang et al. presents an approach which mines the execution logs of an application to identify the dominant behaviour of an application and then flag anomalous application behaviours that require closer analysis by domain experts [50]. The above techniques do not detect unbounded thread-instantiation loops. Furthermore, as a static approach, we do not need to generate test cases to simulate stressful scenarios or analyse log files.

Finally, we mention that work on defending against denial of service attacks focuses on analysing the route and IP addresses of requests and other similar information so that spurious requests can be rejected [51]. However, due to the nature of the problem, a single measure is unlikely to completely solve the problem. In particular, spurious requests may be difficult to recognize, as they are often application-specific. Thus, rejecting “all and only” spurious requests may not always be possible. We focus on the program’s source code and try to detect unbounded thread-instantiation loops that allow unlimited thread resources to be allocated, which may crash the server application. We believe that our approach is complementary to measures such as rejecting spurious requests.

3.7 Conclusions

In this paper, we presented a lightweight, static analysis approach for detecting unbounded thread-instantiation loops in server applications. The key insight behind our approach is that loop structures for bounding thread instantiations are often simple for practical applications due to the special nature of threads. Our approach checks loops and conditions under which a thread instantiation may take place against several simple bounding iteration patterns and bounding condition patterns. Complex patterns will likely require complex analysis, e.g., symbolic analysis and/or context-sensitive analysis. This would significantly limit the scalability of our approach. We also avoid complex termination proofs that are often difficult to perform. Our experimental results show that our approach is very effective at quickly locating unbounded thread-instantiation loops in real-life programs and has detected real problems confirmed by the developers, which would not be possible if we do complex analysis.

We plan to continue our work in three directions. First, we plan to conduct more experiments to evaluate the effectiveness of our approach. In particular, we want to conduct more experiments on the impact of the two parameters, i.e., *LimitOfNestedCalls* and *LimitOfNestedLoops*. Second, we plan to develop an open framework that allows the users to define new bounded thread-instantiation patterns as they are discovered. Finally, we plan to extend our approach so that it can detect unbounded loops for allocating other types of resources, such as sockets and significant data structures.

Chapter 4. Using Delta Debugging to Minimize Stress Tests for Concurrent Data Structures

This chapter contains a paper published in IEEE tenth International Conference on Software Testing, Verification, and Validation (ICST), in 2017.

Using Delta Debugging to Minimize Stress Tests for Concurrent Data Structures[*]

Jing Xu[1], Yu Lei[1], Richard Caver[2], David Kung[1]

[1]Dept. of Computer Science and Engineering, University of Texas at Arlington, Arlington, TX, USA

jingxu@mavs.uta.edu, {ylei,kung}@uta.edu

[2]Dept. of Computer Science, George Mason University, Fairfax, VA , USA

rcarver@gmu.edu

***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. However, the execution trace of a failed stress test that involves multiple threads making many method invocations may be very long, making it time-consuming to replay the failure and localize the fault.

In this paper, we introduce an approach to minimizing stress tests for concurrent data structures. Our approach is to remove some of the threads and/or method invocations from a stress test to create a smaller test that still produces the same failure. We apply delta debugging to a failed stress test to identify the threads and method invocations that are essential for causing the failure. Other threads and method invocations in the original failed execution 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

[*] Copyright © 2017 IEEE. Reprinted, with permission, from Jing Xu, Yu Lei, Richard Carver, Using Delta Debugging to Minimize Stress Tests for Concurrent Data Structures, IEEE International Conference on Software Testing, Verification and Validation (ICST), April 2017.

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. Each data structure was stress test by 100 threads and each thread had 100 method invocations. The results of our evaluation showed that *TestMinimizer* can effectively and efficiently minimize the stress tests for these concurrent data structures. All the stress were reduced to be no more than four threads and fourteen out of sixteen stress tests had no more than five method invocations left.

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

4.1 Introduction

A concurrent data structure stores and organizes data that is accessed by multiple computing threads (or processes) [54]. 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 only be exposed by certain, rare interleavings of instructions [52].

Stress testing is often employed to test a concurrent data structure so that rare interleavings can be exercised. 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 that involves many threads executing many methods may contain a large number of execution events. This makes replaying the failure and localizing the fault very time consuming. If the size of failing execution traces can be reduced, then faults can be localized faster and easier.

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, failing 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 program 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, simpler test execution can be used to localize the fault. Otherwise, a different removal set can be identified and used to derive a new test execution. This process can be repeated until no more threads and/or method invocations can be removed.

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 stress test of smaller size 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 [53].

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, failing execution trace to guide the reduced test execution towards repeating the failure. Since the reduced test will not execute the events in the removal set, we remove them from the original failing 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 analyses 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 chances for the guided execution to repeat 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 schedules for a reduced test; we only try to follow the schedule generated by the guided execution. When the guided execution passes, there is no guarantee that other executions of the reduced test would not fail. In this case, the reduced stress test we generate 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 [55]. These nine concurrent data structures were developed by students in a programming course. 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 with stress tests, 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 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 our framework for stress testing concurrent data structures and shows an example program. Section 3 presents our execution model. Section 4 describes the delta debugging-based framework. 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 presents our plan for future work.

4.2 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 [56]. A search for “concurrent data structure” and the “Java” language 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 manipulates the target data structure by repeatedly invoking public methods of the data structure. This same design was used for stress testing the concurrent data structures in the Java concurrency utilities [57]. Note that, typically, all of the threads manipulate a single instance of the target data structure. When some public methods also need instances of the target data structure for use as method parameters, multiple instances may be created.

Figure 4-1 shows an example stress test set from the *ConTest* benchmark programs [58].

```
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 StressTestAccount {
     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 4-1 A Motivating Example

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.

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 are created to stress test three instances of *Account*. Thread *i* randomly calls methods *deposit* and *withdraw* on a random account, and method *transfer* on a random source *Account account* and destination *Account dest_account*.

Assume that in the execution of the stress test, each thread threads[*i*] deposits $400 and then withdraws $100 from the *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[1],* instead of $300. This is because method *transfer* directly accesses *ac.amount*, where *ac* is the destination *Account* object passed to method *transfer* as an argument.

4.3 Execution model

In this section, we present our execution model, which focuses on monitor-based programs [61]. Note that most concurrent Java programs are monitor-based programs, as monitors are the main synchronization construct provided in Java.

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 *conditionVariable*.*signal*() 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.

Our execution model provides sufficient information for replaying an execution. Replay techniques for monitor-programs have already been developed [62][63]. Our execution model contains all the information required by these techniques and some additional information that is required by our stress testing technique.

As described in Section II, executions of a stress test for a concurrent data structure involve multiple threads. Each thread manipulates the target data structure by iteratively invoking its 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, and *iteration ID* indicates which loop iteration was being performed by the executing thread when it executed the event. In our stress test programs for concurrent data structures, each iteration invokes one public method. Thus *iteration ID* is also *method ID*. *Event type*, *thread ID*, and *name* are required by the replay technique [62], which tracks all of the synchronization actions and operations on shared variables that are exercised during an execution. The *iteration ID* is required by our stress testing technique to identify a method invocation that was issued in the body of a loop executed by a thread.

The valid event types are: *enterMonitor, exitMonitor, enterMethod*, *exitMethod*, a*wait*, *signal*, *reenterMonitor*, *read*, and *write*. Event *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 signaled. 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 [62] 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 (await, signal, 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).
```

4.4 The Framework

Assume that the execution of a stress test *s* detects a failure in a concurrent data structure. Our objective is to remove threads and/or method invocations from *s* to create a stress test that is as small as possible but that still detects the same failure.

In this section, first we give an overview of the delta debugging technique. Then we present a delta debugging-based framework, which applies delta debugging [53] to identify the failure inducing threads and methods more efficiently than randomly choosing which threads and method invocations to remove.

4.4.1 Overview of delta debugging

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

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-introducing. Otherwise, we partition *c* into two subsets *c*1 and *c*2 and test each of them. If either test of *c*1 or test of *c*2 fails, we can simply continue to search in the failing subset. If both of them pass, 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. 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 $O(\log n)$ [53].

An example execution of delta debugging is shown below. 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 also 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 set of 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 we find that a set 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 4-2 A Failed Execution

4.4.2 Applying delta debugging

The failure-inducing factors for the stress test of a concurrent data structure are threads and 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 program. 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 non-deterministically may not repeat the original failure due to the non-deterministic nature of the stress test which is a concurrent program. 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.

4.5 Guided Execution

In this section, we introduce guided execution, which controls the execution of a reduced stress test so that it follows the original failed execution trace whenever possible. Doing so allows the execution more likely to reproduce the original failure than an uncontrolled execution that is inherently non-deterministic.

4.5.1 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 threads/method invocations that were removed to create the reduced test. For example, assume that the next event to be executed in the original execution trace is to be executed by thread T, but thread T is unable to execute this event next in the current execution of the reduced stress test. The reason for this mismatching between the expected event and the actual event can be

1. The expected event is removed in the current reduced stress test program.
2. Due to the removal, the control flow of the current reduced stress test program changes and the expected events should be skipped or executed later.

This problem is addressed in two steps.

Step 1: remove the events of the removed threads/method invocations from the original execution trace

The first step is to remove events from the original execution trace that cannot possibly be executed by the reduced stress test.

- If thread T is a thread in the removal set, 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.

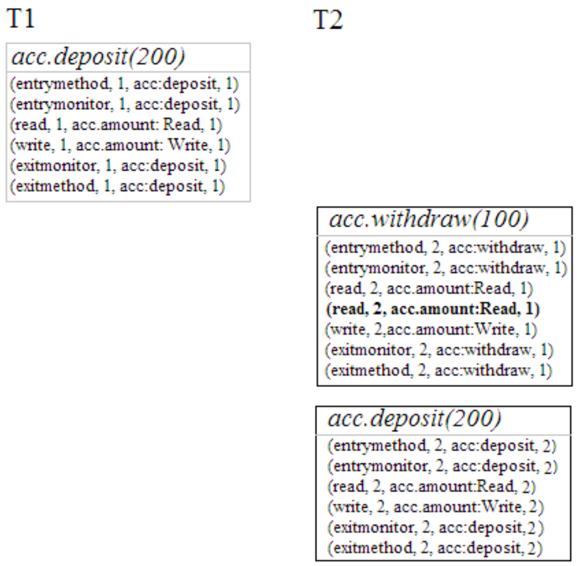
- If M is a method whose invocations are in the removal set, then all of the events in the original execution trace, from any *entermethod* event for method M that is executed by some thread T, up to and including the next *exitmethod* event for method M that is executed by the same thread T, are removed. 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.

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

Step 2: recover execution using lookahead

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 affect 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 is illustrated by the following two examples, which use class *Account* from Section II.

Example 1: Assume that the following execution is part of the failing 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 below. Object *acc* is an *Account* object that is initialized to have a balance of 0.


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

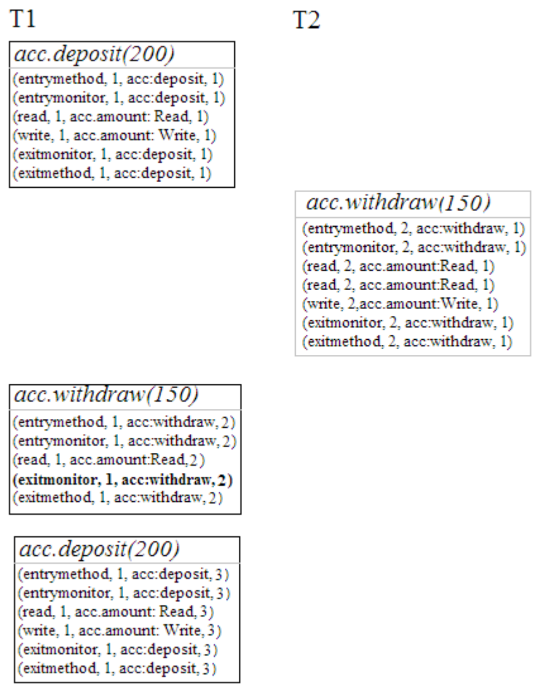
acc.withdraw(100)
(entrymethod, 2, acc:withdraw, 1)
(entrymonitor, 2, acc:withdraw, 1)
(read, 2, acc.amount:Read, 1)
(read, 2, acc.amount:Read, 1)
(write, 2,acc.amount:Write, 1)
(exitmonitor, 2, acc:withdraw, 1)
(exitmethod, 2, acc:withdraw, 1)

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 4-3 Example 1

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.

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



```
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(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)

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)

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 4-4 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. If 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.

4.5.2 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. Both possibilities mentioned below can be tried:

1. Events that are in the retained trace, but that cannot be executed by the reduced stress test, are skipped.
2. Some events that are not in the retained trace but that must be executed by the program during the reduced stress test are added to the retained trace so that the trace contains events that must be executed by the program.

Ideally, both possibilities are tried when a mismatch occurs. However, always trying both possibilities may result in an exponential number of executions. To address this problem we use information in the original execution trace to decide which possibility 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*, *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 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 and this may make the execution of the matching future event impossible.

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, but instead we add some events to the retained trace. This may prevent the execution from getting back on track. In the case studies reported in Section VII, we show the effect of different lookahead values.

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

Assume the following failed execution occurs:

```
Thread1   Thread2   Thread3
e1
e2
await
            e3
            e4
            signal
                       e5
e6
```

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

```
Thread 1             Thread3
e1
e2
await
                       e5
e6
```

A forced execution of *s* with the retained trace can replay events e1, e2, await 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 *signal* event in Thread2 was removed. 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.

4.6 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

[59]. This library provides testing and debugging services for multithreaded Java programs.

The three components of *StreeTestMinimizer* are shown in Figure 4-5.

The *remover* component takes as input the original stress test, the original failing execution trace, and two sets of threads/method invocations. Set *removal* is the set of threads/method invocations that can definitely be removed from the original stress test. Set *try removal* 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.

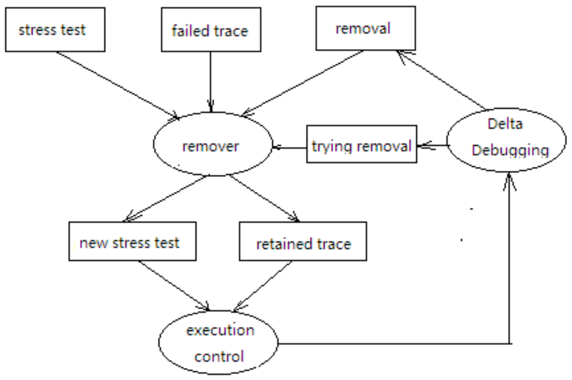


Figure 4-5 Architecture

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, removing and executing events as necessary.

The result of the controlled execution is given to *Delta debugging*. If the result is the original failure, Delta debugging adds the threads/method invocations in *try removal* to set *removal and* updates *try removal.* 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 set *removed*.

4.7 Experiments

As a proof-of-concept, we used *TestMinimizer* to conduct an empirical study of our minimization technique on a suite of failed stress tests for 16 faulty Java concurrent data structures.

The first nine of the 16 concurrent data structures were among the faulty programs used in [55]. 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 makes a random selection of a public method to invoke. Note that currently, we only invoke the public methods which could be called with an integer value or an instance of the target data structure, i.e., we do not randomly generate instances of other types. In fact, all the public methods of our target data structures in the empirical study could be called with an integer value or an instance of the target data structure.

In order to conduct the empirical study, we rewrote these programs using the Modern Multithreading library [60], 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

[66][67][68][69][76][77][78]. We also inserted assertions that were used as test oracles for determining whether test executions of the programs passed or failed. Table 4-1 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..

Table 4-1 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	[66]	2	2
BoundedBuffer	126	[67]	1	3
ConcurrentBST	199	[68]	1	1
ConcurrentLinkedList	161	[69]	1	3
ConcurrentQueue	91	[76]	1	2
ConcurrentQuadTree	224	[77]	1	2
ConcurrentHashMap	206	[78]	2	1

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 [64], 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 [65], 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 [64] and [65] ─ we set the number of created threads to be 100, and the number of method invocations per thread to be 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.

The objective of our empirical studies was to investigate the following two questions:

(1) What is the effect of different lookahead values?

(2) How effective and efficient is our approach?

4.7.1 impact of lookahead

The goal of our first empirical study was to investigate the impact of the lookahead value on our stress minimization technique. We considered lookahead values of 2, 4, 6, 8 and 10. Figure 4-6 to 4-9 shows the result of this study. We show the number of threads retained, the number of method invocations retained, and the total number of executions and total running time. Detailed data could be found on our web site [81].

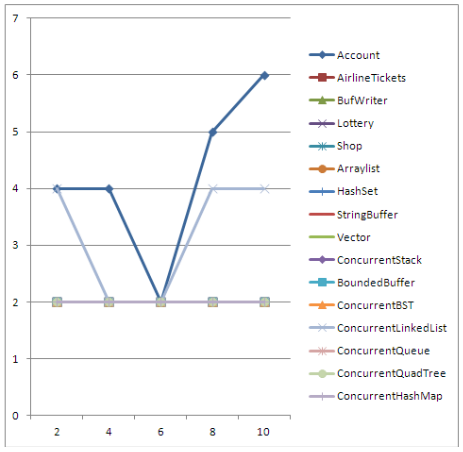


Figure 4-6 Impact of lookahead on # of threads retained

x: the value of lookahead; y: # of threads retained

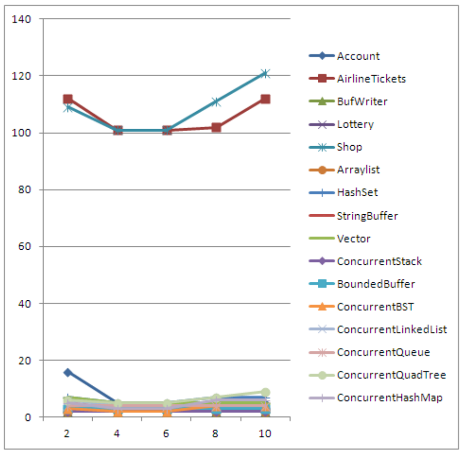


Figure 4-7 Impact of lookahead on # of method invocations retained

x: the value of lookahead; y: # of method invocations retained

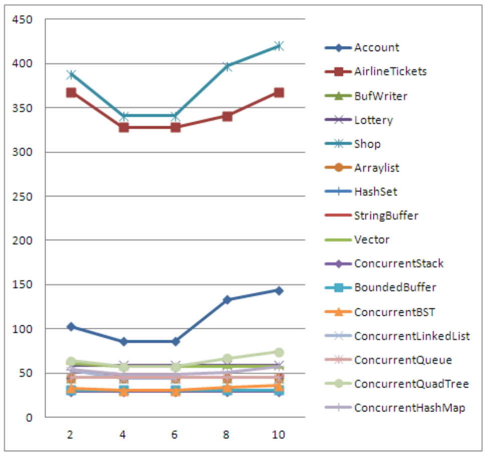


Figure 4-8 Impact of lookahead on # of executions

x: the value of lookahead; y: # of executions

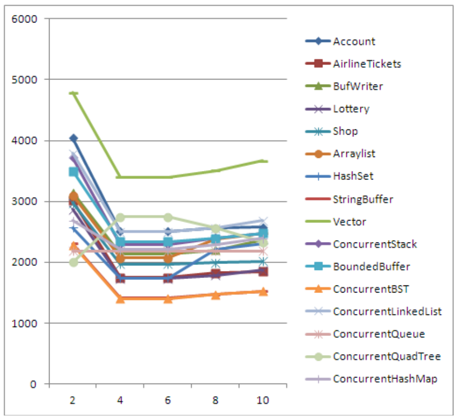


Figure 4-9 Impact of lookahead on total running time

x: the value of lookahead; y: total running time

The results in Figure 4-6 to 4-9 show that the lookahead value affects the number of executions, the number of threads, the number of method invocations that are retained and total running time. An observation is that either the lookahead value is too small or too big would lead to more executions, more threads, method invocations retained and more total running time.

When the lookahead value is too small, it is possible that some events in the execution trace should be skipped but instead extra events are executed without skipping events. When the lookahead is too big, it is possible that extra events should be executed, but instead some events in the execution trace are skipped. For example, the matching event is supposed to be executed in the next iteration of a loop, i.e., some extra events should be added before this matching event executes in next iteration, but the matching event is instead executed in the current iteration. Thus, a lookahead value that is either too big or too small may lead the execution to diverge from the original failed one. For example, after skipping some events mistakenly, the current thread may have to execute until it terminates or is blocked, since it may not be possible for the current thread to find a matching event, which make it execute extra events until end. So the failed pattern in the original failed trace may not be able to be reproduced. Thus, a lookahead value that is either too big or too small could result in more executions and more threads and method invocations left. We will discuss the factors that affects the total execution time in section 7.3.

4.7.2 Efficiency and Effectiveness of Guided Executions

Our second empirical study is to answer Question (2). In this study we implemented three techniques, guided execution, uncontrolled execution and random execution.

Guided execution applies delta debugging to try to remove different threads/method invocations and uses our execution control approach to guide the stress test execution. In this experiment, we use lookahead value 5. This choice is based on the study in section 7.1, which shows us when lookahead is around 4 to 6, less threads, method invocations are retained with less total running time. Note that this choice may not be optimal choice for all the target stress tests. This technique forces the execution of the stress test to follow the retained trace until a mismatch occurs. When a mismatch occurs and the execution is unable to follow the retained trace, guided execution then guides the execution by skipping some events in the retained trace or by executing some extra events, until the retained trace can be followed once again.

Uncontrolled execution also applies delta debugging, but allows the execution of the stress test to run non-deterministically, i.e., without any execution control.

Random execution is a baseline technique which tries to remove threads/methods randomly, but not applies delta debugging. Also it does not use our execution control technique, but allows the execution to run non- deterministically. For comparison, random execution runs the same number of executions as guided execution. For each execution for thread removal, random execution randomly removes a random number of threads from the remained threads. For each execution for method removal, random execution randomly removes a random number of method invocations from the remained method invocations in a random thread. When the new execution fails, the selected threads/method invocations could be removed. Otherwise, they will be restored.

Table 4-2 shows the total number of threads and method invocations left in the minimized stress tests, the number of executions (the number of reduced stress tests tried), and the total execution time for guided execution, uncontrolled execution and random execution respectively. All of the original failures were reproduced by the reduced stress test. Otherwise, the removal set would have been restored.

The results in Table 4-2 show that for all the subject programs, random execution left much more threads and method invocations than the other two techniques with less execution time. Random execution does not remove threads/method invocations systematically and different tries of removal may overlap with each other. Guided execution was able to remove more threads and method invocations with less executions than uncontrolled execution. The number of threads left by uncontrolled execution is 1.9 times of that by guided execution, while the number of method invocations left by uncontrolled execution is 4.1 times of that by guided execution. The reason that uncontrolled execution is more effective in thread removal stage than method invocation removal stage is that in thread removal stage, even when only a few threads are left, uncontrolled execution is still likely to expose the fault since each left thread has 100 method invocations. However, in method removal stage, with the methods being removed, it is less likely for uncontrolled execution to expose the fault, since the stress is not enough. In other words, guided execution is more effective than uncontrolled execution when stress is low. For 14 of 16 programs, the run time of guided execution was longer than that of uncontrolled execution. We will discuss the factors that affects the total execution time in next subsection.

Table 4-2 guided vs uncontrolled vs random

Program	guided execution				uncontrolled execution				Random execution			
	# 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

4.7.3 Discussion

The factors that affects the total execution time include the instrumentation overhead, total number of executions and the number of unterminatable executions. The stress test programs for guided execution are instrumented to let us control the execution follow the original failed trace, which adds execution time overhead. The number of executions for random execution is set to be the same as that for guided execution for comparison. The number of executions for uncontrolled execution is generally greater than that for guided execution, since guided execution is more likely to reproduce the failure than uncontrolled execution, which makes more executions fail. Some executions could not terminate. The reason for this could be that the methods contains notify event are removed so that the waiting thread could not be awaked or the control flow change makes the notify events are skipped. We set a timeout of 1 min for each execution. When an execution got stuck in a busy-waiting loop, it terminated after a timeout.

After the thread removal phase, no additional threads can be removed. However, after some method invocations are removed, it may be possible to remove all of the method invocations of some thread., effectively removing the thread.

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.

4.8 Related work

Several tools minimize failure-introducing inputs to concurrent systems without controlling sources of non-determinism [70, 71] Since there is no control of the execution, these approaches require fewer instrumentations and are fast in term of running a single execution. However, some failures may rarely happen. Without controlling the execution, it is hard to reproduce the failure and minimize the inputs. Other techniques seek to only minimize thread interleavings leading up to concurrency bugs [72, 73, 80, 82], 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. [79] 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, which prioritize backtrack points that match the type (the language-level type tag of the message object, which is available to the RPC layer at runtime) of the corresponding message from the original trace, 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 our empirical study results, we can see our approach could effectively and efficiently minimize the stress test program for concurrent data structures.

Delta debugging [53] 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 failing test case and the output is a 1-minimal failing test case. A failing 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-failing 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 test programs for concurrent data structures. Also, in our approach, we control program execution to recover from a mismatching.

4.9 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 to make it more likely that the original failure is 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 future, we plan to conduct more experiments to evaluate the effectiveness of our approach. In particular, we want to conduct more experiments on the impact of the *lookahead* value. Second, we plan to integrate our approach with an automatic stress test generation approach. Then users only need to provide a target data structure and a minimized stress test with a failed execution will be reported to them, if a failed execution is found in the stress test.

.

Chapter 5. Conclusion

In this dissertation, we present three approaches to detecting and localizing faults in concurrent programs.

Given a failed execution of a concurrent program, the first approach identifies the failure-introducing patterns, which make the users finding and fixing the bugs easier. The novelty of this approach is the use of the least concurrent mode. In the least concurrent mode, each thread is forced to execute until it cannot proceed further, i.e., either blocks or finishes. The motivation is to minimize the number of interleavings and thus reduce the chance of atomicity violations and order violations. The empirical study conducted by our tool *Huatuo* showed that our approach was effective, i.e., localized the faults in twelve of the thirteen programs, and efficient, i.e., had an average slowdown factor of 25x for the largest program in our experiments.

In the future, we plan to conduct more experiments on more complex real-life programs to evaluate the effectiveness of our approach. Also, we plan to explore ideas which could make our approach more efficient. For example, we will try to apply binary search. By applying binary search we may localize a switch point that does not trigger the failure. But it could still be useful, since it makes the execution enter an erroneous state.

The second approach detects unbounded thread-instantiation loops in server applications. The key observation is that the number of thread instantiations is usually bounded by simple patterns. Our approach checks loops and conditions for a thread instantiation against several simple bounding patterns. Our experimental results show that our approach was very effecient. The execution times for 20 out of 24 real-life programs are within 10 mins. It is also very effective. 38 unbounded thread-instantiation loops detected by our approach and 12 of them are confirmed by the original developers.

In the future, we plan to conduct more experiments on the impact of the two parameters, i.e., *LimitOfNestedCalls* and *LimitOfNestedLoops*. And we would like to explore the idea on improving the efficiency of our approach when these two parameters are big. Also, we plan to develop an open framework so that when the users find new patterns they can define them by themselves. Finally, we plan to extend our approach to detect unbounded loops for allocating other types of resources, e.g., sockets, etc.

The third approach is to minimize stress tests for concurrent data structures. The novelty of the approach is the use of *lookahead* value to guide reduced stress test to follow the original failed execution as much as possible to determine whether some threads/method invocations are removable. The results of our empirical studies showed that our approach could reduce the number of threads from 100 to no more than 4 for all 16 stress tests of the target data structures and reduce the number of method invocations to be no more than 5 method invocations for 14 out of 16 stress tests.

In the future, we plan to conduct more experiments to evaluate the impact of the *lookahead* value. Second, we plan to integrate our approach with an approach which generates stress test for a concurrent data structure automatically, After this integration, users only needs to provide a data structure. Our tool will report a minimized stress test with a failed execution, if a failed execution is found by stress test .

References

1. J. Choi, and A. Zeller. Isolating Failure-Inducing Thread Schedules. Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis, pp. 210-220, Jul. 2002.
2. S. Park, R. Vuduc, and M. Harrold. A Unified Approach for Localizing non-deadlock Concurrency Bugs. Proceedings of the Software Testing,

Verification and Validation (ICST), 2012 IEEE Fifth International Conference, pp.51-60, Apr. 2012.

3. S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. Proceedings of the 12th international conference on Architectural support for programming languages and operating systems, pp. 37-48, Dec. 2006.
4. S. Park, S. Lu, and Y. Zhou. CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places. Proceedings of ISSTA '02 Proceedings of the 14th international conference on Architectural support for programming languages and operating systems, pp. 25-36, Mar. 2009.
5. 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.
6. Hammer, J. Dolby, M. Vaziri, and F. Tip. Dynamic detection of atomic-set-serializability violations. Proceedings of the ICSE '08. ACM/IEEE 30th International Conference, pp. 231-240, May. 2008.
7. Z. Lai, S. Cheung, and W. Chan. Detecting atomic-set serializability violations in multithreaded programs through active randomized testing. Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, pp. 235-244, 2010.
8. M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. Proceedings of the Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 334-345, 2006.

9. K. Sen. Race Directed Random Testing of Concurrent Programs. Proceedings of PLDI '08, pp. 11-21, 2008.
10. R. Carver and Y. Lei, A Class Library for Implementing, Testing, and Debugging Concurrent Programs, Int. Journal on Software Tools for Tech. Transfer: Vol. 12, Issue 1, 2010, pp. 69-88.
11. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In POPL, pages 256-267, 2003
12. L. Wang and S. D. Stoller. Static analysis for programs with non-blocking synchronization. In PPoPP, 2005
13. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. Trans. Comput. Syst.,15(4):391–411, 1997.
14. M. Ronsse and K. D. Bosschere. RecPlay: a fully integrated practical record/replay system. Trans. Comput. Syst., 17(2):133–152, 1999.
15. M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In PLDI pages 446–455, June 2007.
16. C. Flanagan and S. N. Freund. Type-based race detection for java. In PLDI, pages 219–232, June 2000.
17. P. Godefroid and N. Nagappan. Concurrency at Microsoft: An exploratory survey. In Workshop on Exploiting Concurrency Efficiently and Correctly, 2008.
18. K. Poulsen. Tracking the blackout bug. SecurityFocus, February 2004. http://www.securityfocus.com/news/8412.

19. 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.

20. C. Artho, K. Havelundand and A. Biere. A high-level data race. Journal on Software Testing, Verification & Reliability, 2003

21. P. Ammann and J. Offutt., Introduction to Software Testing, Cambridge University Press, Cambridge, UK, 2008.

22. Y. Shi, S. Park, Z. Yin, and S. Lu. Do I use the wrong definition DeFuse definition-use invariants for detecting concurrency and sequential bugs. Proceedings of the OOPSLA, pp. 160-174, 2010.

23. S. Narayanasamy, Z. Wang, and J. Tigani. Automatically classifying benign and harmful data races using replay analysis. Proceedings of the PLDI '07 Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation, pp. 22-31, 2007.

24. L. Wang and S. D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, (PPoPP'06), 2006.

25. M. Sipser, Introduction to the theory of computation, 3rd Edition, pp. 504, 2013 published.

26. C. Păsăreanu and W. Visser. A survey of new trends in symbolic execution for software testing and analysis. International journal on software tools for technology transfer 11.4 (2009): 339-353.

27. Open source software in Java, http://java-source.net/

28. Thread Bound Checker Project, http://barbie.uta.edu/~jxu/threadboundchecker.htm

29. Java eclipse, http://en.wikipedia.org/wiki/Java_eclipse

30. Java Model, http://help.eclipse.org/juno/index.jsp?topic=%2Forg.eclipse.jdt.doc.isv%2Fguide%2Fjdt_int_model.htm.

31. Abstract syntax tree, http://help.eclipse.org/indigo/index.jsp?topic=%2Forg.eclipse.jdt.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fjdt%2Fcore%2Fdom%2FAST.html

32. Abstract syntax tree, http://www.eclipse.org/jdt/core/index.php

33. C. Otto, M. Brockschmidt, C. Von Essen, J. Giesl. Automated termination analysis of java bytecode by term rewriting. RTA. Vol. 10. 2010.

34. MA. Colón, and HB. Sipma. Practical methods for proving program termination. Proceedings of the Conference on Computer Aided Verification. Springer -Verlag, 2002.

35. B. Cook, A. Podelski, and A. Rybalchenko. Proving program termination. Communications of the ACM 54.5 (2011): 88-98.

36. A., P. Arenas, M. Codish, and S. Genaim. Termination analysis of Java bytecode. Proceedings of the International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'08). G. Barthe and F. S. de Boer, Eds. Lecture Notes in Computer Science, vol. 5051. Springer, 2--18.

37. Kroening, N. Sharygina, and A. Tsitovich. Termination analysis with compositional transition invariants. Proceedings of the Conference on Computer Aided Verification. Springer Berlin Heidelberg, 2010.

38. MA. Colóon, and HB. Sipma. Synthesis of linear ranking functions. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 67–81. Springer, Heidelberg (2001).

39. T. Stephan. Safe and precise WCET determination by abstract interpretation of pipeline models. Diss. Universitätsbibliothek, 2004.

40. N. Holsti, T. Långbacka, S. Saarinen. Worst-case execution-time analysis for digital signal processors. Proceedings of the EUSIPCO 2000 Conference (X European Signal Processing Conference). 2000.

41. J. Gustafsson, A. Ermedahl. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. Proceedings of the 27th IEEE International Real-Time Systems Symposium, p.57-66, December 05-08, 2006.

42. J. Burnim, N. Jalbert, C. Stergiou, and K. Sen, Looper:Lightweight detection of infinite loops at runtime, Proceeedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE), 2009, pp. 161–169.

43. Casale, A. Kalbasi, D. Krishnamurthy, and J. Rolia. Automatic stress testing of multi-tier systems by dynamic bottleneck switch generation. Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware, article no. 20, 2009.

44. Avritzer, and E. Weyuker. Generating test suites for software load testing. Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 44 - 57, 1994.

45. Avritzer, and B. Larson. Load testing software using deterministic state testing. Proceedings of the 1993 ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 82 - 88, Sept. 1993.

46. D. Krishnamurthy, J. Rolia, and S. Majumdar. A Synthetic Workload Generation Technique for Stress Testing Session-Based Systems., IEEE Transactions on Software Engineering, pp.868-882, Nov. 2006.

47. Avritzer, and E. Weyuker. The automatic generation of load test suites and the assessment of the resulting software. IEEE Trans. Software Engineering, pp.705-716, Sep. 1995.

48. P. Zhang, S. Elbaum, and M. Dwyer. Automatic generation of load tests. Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 43-52, Nov. 2011.

49. Malk. A methodology to support load test analysis. Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, pp. 421-424, 2010.

50. Z. Jiang, A. Hassan, and G. Hamann. Automatic identification of load testing problems. Proceedings of the Conference on Software Maintenance, pp.307-316, Nov. 2008.

51. DKY. Yau, J. Lui, F. Liang, and Y. Yam. Defending against distributed denial-of-service attacks with max-min fair server-centric router throttles. IEEE/ACM Transactions on Networking (TON) 13.1 (2005): 29-4

52. 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).

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

54. Concurrent data structure, https://en.wikipedia.org/wiki/Concurrent_data_structure

55. 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.

56. Github, https://www.github.com

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

58. 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.

59. R. Carver, and K. C. Tai, Modern Multithreading, Wiley, 2006.

60. 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.

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

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

63. 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.

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

65. Lock free concurrent stack, https://github.com/mdtareque/concurrentDataStructures/blob/master/src/CStackLockFreeTester.java

66. Lock free stack, http://stackoverflow.com/questions/5614599/simple-lock-free-stack

67. Lock free code a false sense of security, http://www.drdobbs.com/cpp/lock-free-code-a-false-sense-of-security/210600279?pgno=5

68. Java theory and practice: Going aomic, http://www.ibm.com/developerworks/library/j-jtp11234/

69. Java concurrency programming 5: lock free data structure, http://blog.csdn.net/b_h_l/article/details/8704480

70. T. Arts, J. Hughes, J. Johansson, and U. Wiger. Testing Telecoms Software with Quviq QuickCheck. Erlang ’06.

71. J. Clause and A. Orso. A Technique for Enabling and Supporting Debugging of Field Failures. ICSE ’07.

72. J. Choi and A. Zeller. Isolating Failure-Inducing Thread Schedules. SIGSOFT ’02.

73. M. A. El-Zawawy and M. N. Alanazi. An Efficient Binary Technique for Frace Simplifications of Concurrent Programs. ICAST ’14.

74. Brito, Andrey, 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.

75. 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.

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

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

78. Concurrenthashmap, http://codereview.stackexchange.com/questions/96686/concurrenthashmap-implementation

79. 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.

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

81. TestMinimizer, http://barbie.uta.edu/~jxu/testminimizer.htm

82. 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.