A Lightweight and Portable Approach to Making Concurrent Failures Reproducible

Qingzhou Luo[1], Sai Zhang[2], Jianjun Zhao[1], and Min Hu[1]

[1] School of Software, Shanghai Jiao Tong University
{seriousam,zhao-jj,minhu_fox}@sjtu.edu.cn
[2] Computer Science & Engineering Department, University of Washington
szhang@cs.washington.edu

Abstract. Concurrent programs often exhibit bugs due to unintended interferences among the concurrent threads. Such bugs are often hard to reproduce because they typically happen under very specific interleaving of the executing threads. Basically, it is very hard to fix a bug (or software failure) in concurrent programs without being able to reproduce it. In this paper, we present an approach, called `ConCrash`, that automatically and deterministically reproduces concurrent failures by recording logical thread schedule and generating unit tests. For a given bug (failure), `ConCrash` records the logical thread scheduling order and preserves object states in memory at runtime. Then, `ConCrash` reproduces the failure offline by simply using the saved information without the need for JVM-level or OS-level support. To reduce the runtime performance overhead, `ConCrash` employs a static data race detection technique to report potential possible race conditions, and only instruments such places. We implement the `ConCrash` approach in a prototype tool for Java and experimented on a number of multi-threaded Java benchmarks. As a result, we successfully reproduced a number of real concurrent bugs (e.g., deadlocks, data races and atomicity violation) within an acceptable overhead.

1 Introduction

The increasing popularity of concurrent programming has brought the issue of concurrent defect analysis to the forefront. Concurrent programs often exhibit wrong behaviors due to unintended interferences among the concurrent threads. Such concurrent failures or bugs - such as data races and atomicity violations - are often difficult to fix without being able to reproduce them. However, in a multi-threaded concurrent program, the number of possible interleavings is huge, and it is not practical to try them all. Only a few of the interleavings or even one specific interleaving actually produce the failure; thus, the probability of reproducing a concurrent failure is extremely low. A traditional method of reproducing concurrent failure is to repeatedly execute the program with the hope that different test executions will project in different interleavings. Unfortunately this approach is proved to be neither efficient nor reproducible in practice. Firstly, execution result of a concurrent program depends on the underlying operating system or the virtual machine for thread scheduling - it does not try to explicitly control the thread schedules; therefore, executions often end up with the same interleaving many times, which means getting an access to the buggy interleaving is always a time-consuming task. Moreover, many concurrent applications (like servers) often run a long time and serve for specific users. So it would be extremely hard to reproduce such environmental dependent bugs in off-line testing.

The high cost of reproducing concurrent failures has motivated the development of sophisticated and automated analysis techniques, such as [6–8, 10, 11, 14, 16, 18, 19]. Of particular interest for our work is the `ReCrash` approach proposed by Artzi et al [5]. `ReCrash` monitors every execution of the target (sequential) program, stores partial copies of method arguments, and converts a failing program execution into a set of deterministic unit tests, each of which reproduces the problem that causes the program to fail. The `ReCrash` approach is designed for sequential programs. However, the non-determinism in a multi-threaded concurrent program might disallow the unit tests generated by `ReCrash` to reproduce a concurrent failure.

The work described in this paper aims to reduce the amount of time a developer spends on reproducing a concurrent failure. A key element in designing such an approach is the ability to provide a deterministic thread executing order of a non-deterministic execution instance. In this paper, we propose `ConCrash`, an automated concurrent failure reproducing technique for Java. `ConCrash` handles all threads and concurrent constructs in Java, except for windowing events, I/O inputs and network events which are topics of our future work.

The `ConCrash` approach adapts the concept of *logical thread schedule* as described in [7]. It monitors each critical event to capture the thread execution order during one execution of a multi-threaded program. When the concurrent program fails, `ConCrash` saves both information about thread scheduling and current object states in memory and automatically generates an *instrumentation scheme* and a set of *JUnit tests*. The *instrumentation scheme* records the thread schedule information during the failing execution as pure text, and then enforces the exact same schedule when replaying the execution, while the JUnit tests captures the failed method invocation sequences. The `ConCrash` approach can be used on both client and developer sides. When a concurrent failure occurs, the user could send an *instrumentation scheme* as well as generated *JUnit tests* to developers. While developers could use a `ConCrash`-enabled environment to replay the thread execution order, step through execution, or otherwise investigate the root cause of the failure.

Unlike most of the existing replay techniques like [7], our `ConCrash` approach does not depend on JVM modification or existing OS-level support for replay. Instead, `ConCrash` instruments the compiled class files by modifying their bytecode. To reduce the runtime performance overhead, `ConCrash` also employs a static data race detection technique [14] to find potential possible race conditions, and only instruments such places. While starting to reproduce a failure, `ConCrash` eliminates the nondeterminacy of the program caused by JVM scheduler by transforming the compiled nondeterministic multi-threaded program into a deterministic sequential program without changing the semantics.

We implement the `ConCrash` approach in a prototype tool for Java and experimented on a number of multi-threaded Java benchmarks. We successfully reproduced a number of real concurrent bugs (e.g., deadlocks, data races, atomicity violation) within an acceptable overhead. The main contributions of this paper are:

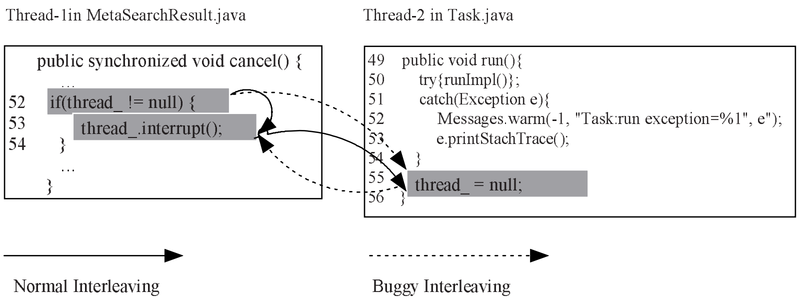
- A lightweight and portable technique that efficiently captures and reproduces multi-threaded concurrent failures, which can be integrated with various static analysis tools.
- Implementation of a prototype tool for Java. For the sake of applicable in long-running multi-threaded programs, we employ and extend a testing framework to support automatic generation of multi-threaded JUnit test cases.
- An empirical evaluation that shows the effectiveness and efficiency of `ConCrash` approach on Java benchmarks and real-world applications.

The rest of this paper is organized as follows. In Section 2, we give an overview of the `ConCrash` approach using a simple motivating example. We describe the details of the `ConCrash` approach in Section 3. In Section 4, we describe the implementation issues of `ConCrash` approach for Java and the results of our experiments, respectively. Related work is discussed in Section 5 followed by conclusion and future work.

2 Motivating Example

In this section, we use a real-world program to give an overview of our approach. Consider the two-threaded program snippet taken from *hedc* benchmark [14] in Fig. 1.

Two threads executing the code of `MetaSearchResearch.java` and `Task.java` have one shared variable `thread_`. There could be an unsynchronized assignment of null to field `thread_` (line 55 in `Task.java`), which could cause the program to crash with a `NullPointerException` (line 53 in `MetaSearchResult.java`) if the `Task` completes just as another thread calls `Task.cancel()`.



```
    public synchronized void cancel() {
       ...
52    if(thread_ != null) {
53        thread_.interrupt();
54    }
       ...
    }
```

```
49  public void run(){
50      try{runImpl()};
51      catch(Exception e){
52          Messages.warm(-1, "Task:run exception=%1", e");
53          e.printStachTrace();
54      }
55      thread_ = null;
56  }
```

Fig. 1. A Motivating Example

In this case, it would be nearly impossible to reproduce the failure by repeatedly executing the original program due to the huge volume of different thread execution orders. Moreover, since in typical OS and JVM design, the thread scheduler is nearly deterministic, executing the same program many times does not help, because the same interleaving is usually created. Actually, in our experiment environment (Section 4), executing the original program for more than 300 times could reproduce only one failure.

Suppose that the motivating example is running in a `ConCrash`-enabled environment. `ConCrash` first uses an existing static data race detection technique [14] to preprocess this program to find all possible race conditions (it is a one time cost). `ConCrash` then instruments the program at the reported race conditions. In this example, line 53 in `MetaSearchResult.java` and line 55 in `Task.java` are reported to be potential race conditions and `ConCrash` instruments these two statements to monitor the thread execution logical order. For each method invoked, `ConCrash` also maintains a state copy of the method receiver and arguments.

As soon as the (instrumented) program crashes, `ConCrash` will generate *an instrumentation scheme* and a set of *JUnit tests*. The user could send the *scheme* and *tests* with the initial bug report to developers. Upon receiving such a scheme and tests, the developer could use `ConCrash` (we provide an instrumentation tool in `ConCrash` design) to instrument the original program to resume the original thread schedule order. After that, the developer could run tests under a debugger to easily reproduce the failure and locate its cause.

For this motivating example, one of the generated JUnit test case is shown in Fig. 2. In lines 2 - 6 of Fig. 2, `ConCrash` first reads the current object (`thisObject`) from a trace file to resume the state. `ConCrash` then reads other thread objects from the recorded file (lines 7 and 8), and synchronizes them to restart at a certain checkpoint before the crash occurs (lines 9 and 10). Finally, `ConCrash` invokes the crashed method (and loads augments if there is any) on the deterministic replay program version.

```
1. public voidtest_MetaSearchResult_cancel_3()throws Throwable {
2.      //Read object from trace file
3.      //adpated from ReCrash implementation
4.      TraceReader.setMethodTraceItem(3);
5.      MetaSearchResult thisObject =
6.         (MetaSearchResult)TraceReader.readObject(0);
        // Resume thread execution orders
7.      ThreadEntity te = TraceReader.getStackTraceItem().threadEntity;
8.      Monitor.restartThreads(te.checkPoints,3);
9.      Monitor.waitForThreads();
        // Method invocation
10.     thisObject.cancel();
11. }
```

Fig. 2. A generated test case by ConCrash to reproduce the hedc failure

3 Approach

In this section we discuss our `ConCrash` approach in detail. The overview of our approach is shown in Fig. 3. Our approach consists of three stages: getting instrumentation sites (Section 3.1), instrumenting original program to generate record & replay version (Section 3.2), and generating JUnit test cases after a crash (Section 3.3).

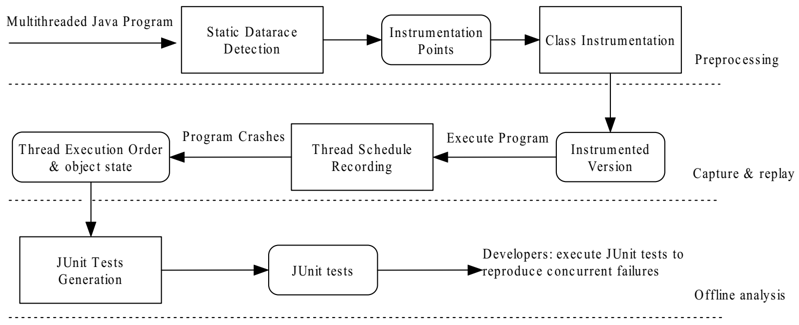

Multithreaded Java Program
Static Datarace Detection
Instrumentation Points
Class Instrumentation
Preprocessing
Thread Execution Order & object state
Program Crashes
Thread Schedule Recording
Execute Program
Instrumented Version
Capture & replay
JUnit Tests Generation
JUnit tests
Developers: execute JUnit tests to reproduce concurrent failures
Offline analysis

Fig. 3. An overview of the ConCrash approach

3.1 Stage One: Getting Instrumentation Sites

It is obviously not practical to instrument every statement in a program, because that would incur a huge slow down rate. Based on a recent study [12], most concurrent bugs can be categorized as *data race*, *atomicity violation* and *dead lock*. While *data race* bugs always have certain execution orders of raced statements, *dead lock* and *atomicity violation*[1] bugs also have specific orders of lock acquire/release operations. After recording such orders we would be able to replay those defects. We next present different strategies for handling these different types of concurrent bugs.

3.1.1 Statements Involved in Data Races

A data race condition is defined as two accesses to a shared variable in different threads without holding a common lock, at least one of which is a write. A data race condition often causes non-predictable behavior of a program and is usually considered as a defect. We are concerned about the execution order of the two accesses in a data race pair and employ a static race detection tool called Chord [14] to detect possible data race pairs. Chord reads the source code of a program and bytecode, performs four stage analysis and outputs the results in files. Though it reports some false positives, the experiments show that Chord is applicable for static race detection for most programs. We take the result of Chord as a part of our instrumentation sites.

►**Example.** In Fig. 1 Chord reports all those three shadowed lines as potential data race pairs. Then we instrument the program before the shadowed lines in the next stage. ◄

3.1.2 Lock Acquire and Release Operations

Atomicity violation bugs [10] are caused by concurrent non-atomic execution of a code region which was intended to run atomically. Admittedly a large part of atomicity violation are caused by data races, however, being data race free would not guarantee atomicity violation free. When the remote and local accesses in a atomicity violation are all well synchronized, it will not count for a data race. To address such problems,

[1] Here we refer to the part of *atomicity violations* that do not count for *data races*.

we also record the global order of lock acquire/release operations. With this information we should be able to reproduce atomicity violation bugs which are not reported by data race detection tools.

A *critical event* [7] is usually defined as a shared variable access or a synchronization event. Here we adapt this concept to the union set of the above two instrumentation sites. In our record and replay analysis, we only consider the temporal execution order of those points.

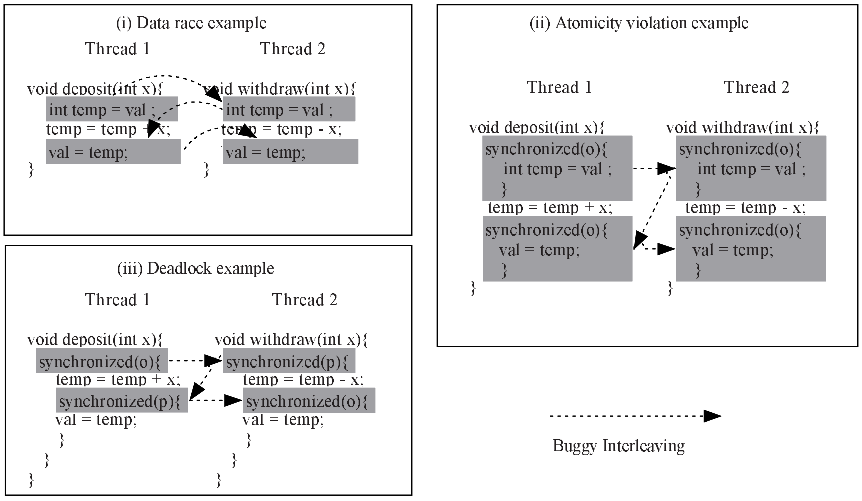


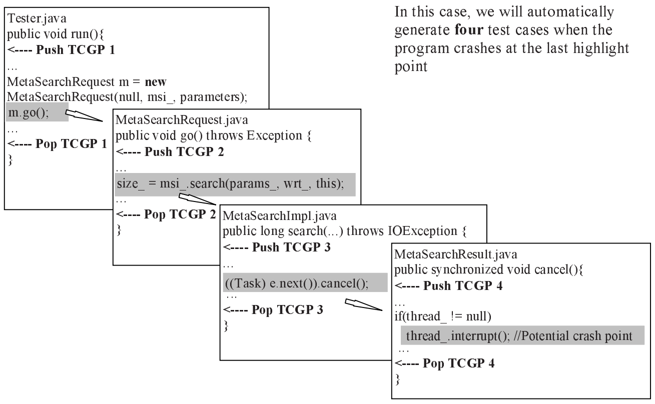
Fig. 4. Three different types of concurrency bugs and our instrumentation sites

►**Example.** Fig. 4 is a frequently used example in concurrency testing. In the first data race example, our instrumentation points consist of the four shadowed statements involved in the race. With execution order information among these statements we would be able to reproduce this data race bug. In the other two examples, the order of acquire/release lock operation is recorded, which consists of all the `synchronized` operations in shadowed lines. With the exact order of acquiring or releasing a lock the other two bugs could be also reproduced in following stages. ◄

3.1.3 Test Case Generation Points

Like ReCrash [5], we instrument a part of all the methods in the program at their entry and exit points. The purpose of this instrumentation is to get a copy of receiver object and method invocation arguments, which are used to generate test cases after a program crash. Each **T**est **C**ase **G**eneration **P**oint information (*TCGP* in short) is pushed into a stack at the entry point of a method. It is then popped at the exit point after successful execution of the method. See Fig. 5 for an example. A large part among all methods, like non-public methods and simple getter/setter methods are excluded because they are considered less likely to expose a bug, and also because of the concern about runtime performance.

We extend this technique to apply in multi-threaded program, that is, in each test case generation point we also make a copy of the *TCGP* stack of all other live threads. In the generated test cases this information is used to recreate all threads to simulate the crash scenario.



```
Tester.java
public void run(){
<---- Push TCGP 1
...
MetaSearchRequest m = new
MetaSearchRequest(null, msi_, parameters);
m.go();
...
<---- Pop TCGP 1
}
```

```
MetaSearchRequest.java
public void go() throws Exception {
<---- Push TCGP 2
...
size_ = msi_.search(params_, wrt_, this);
...
<---- Pop TCGP 2
}
```

```
MetaSearchImpl.java
public long search(...) throws IOException {
<---- Push TCGP 3
...
((Task) e.next()).cancel();
...
<---- Pop TCGP 3
}
```

```
MetaSearchResult.java
public synchronized void cancel(){
<---- Push TCGP 4
...
if(thread_ != null)
  thread_.interrupt(); //Potential crash point
  ...
<---- Pop TCGP 4
}
```

In this case, we will automatically generate **four** test cases when the program crashes at the last highlight point

Fig. 5. An example of test case generation points

3.2 Stage Two: Instrumenting the Original Program to Generate Record and Replay Version

After getting all instrumentation sites from the previous stage, we instrument the bytecode of a program at those sites to capture the logical thread schedule order for deterministic record and replay.

3.2.1 Logical Thread Schedule

Capturing the actual physical thread schedule information is neither feasible nor useful in our approach. Rather than doing so, we record the 'Lamport clock' of the thread as a logical thread schedule. As shown in [7], a logical thread schedule is used to record the begin and the end time stamps of a few of consecutive *critical events* in one thread, as the form <*FirstCriticalEvent, LastCriticalEvent*>. The interval in every tuple is the logical running time of the thread before a thread switch occurs.

3.2.2 Thread Execution Order Record and Replay

The Java record/replay algorithm in [7] could produce sufficient trace information to deterministically replay a multi-threaded program execution. Based on this algorithm [7], rather than extending a specific JVM, we instrument on bytecode level. From the view point of bytecode level, the different types of *critical event* can be divided into `putfield`, `getfield` instructions and `monitorenter`, `monitorexit` instructions. Our instrumentation is based on these individual bytecode instructions. Details of this technique can be found in following algorithms.

```
Input: number_of_interval, global_clock, local_clock
RecordOnStmt():
Enter monitor
Get thread_entity for current thread
gc_copy = global_clock
if thread_entity.local_clock < global_clock then
    Update FirstCriticalEvent and LastCriticalEvent array for current thread number_of_interval =
    number_of_interval + 1
end
Execute critical event
global_clock = global_clock + 1
thread_entity.local_clock = gc_copy
Exit monitor
```

Algorithm 1. Recording execution order of *critical events*

```
Input: number_of_interval, global_clock, local_clock
ReplayOnStmt():
Enter monitor
Get thread_entity for current thread
gc_copy = global_clock
while global_clock < FirstCriticalEvent[number_of_interval] do
    Wait for the execution of other threads
end
Execute critical event
global_clock = global_clock + 1
if global_clock >= LastCriticalEvent[number_of_interval] then
    number_of_interval = number_of_interval + 1
end
thread_entity.local_clock = gc_copy
Notify all other threads
Exit monitor
```

Algorithm 2. Algorithm for replaying *critical events*

3.2.3 Discussions about Effects of Instrumentation

Instrumentation is intrusive, which means that it could potentially affect the behavior of the original threads. However, the instructions we added only operate on their own data structures, and therefore would not change the control flow of the original program. The main impact is that it could possibly change the time slice allocated to a thread by the scheduler. Thus, as our algorithm only captures the linear order of shared variable (data race pairs) accesses and lock acquire/release operations of the original program, this linear order is also a legal order of the execution of the original program. Proof with details can be found in [9]. Time and space overhead of the instrumented program will be discussed in Section 4.

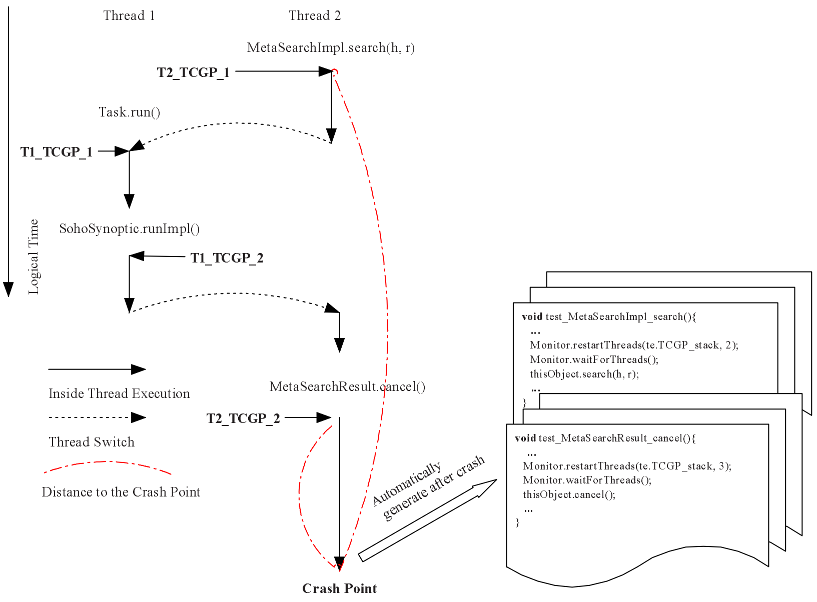
3.3 Stage Three: Generating JUnit Test Cases after a Crash

Testing long running multi-threaded programs is always a difficult task because of its inherent non-determinism and its expensive tracing overhead. `ConCrash` extends a unit test case generation framework to support multi-threaded application, which utilizes logical thread schedule information to deterministically trigger program crashes.

Specifically, we augment the unit tests generation technique in [5] to handle concurrent features. In `ConCrash`, the method call stack data is captured and used for tests generation when a crash happens. Whenever a method is called, it pushes the receiver object and parameters onto stack and then pop this information after normal exit of the method. After a crash happens, `ConCrash` can simply generate test cases by passing the same receiver object and method parameters, which are serialized to the disk before.

As `ReCrash` only monitors the execution of the main thread, if an exception is thrown in the `run()` method of another thread it would not be recorded. Instead `ConCrash` also supports concurrent features in Java by wrapping each `run()` method in a `try` block to capture exceptions thrown by each individual thread.

►**Example.** As seen in Fig. 6, two JUnit test cases are automatically generated after crash. Inside the method invocation `Monitor.restartThreads(te.TCGP_stack, 2)` another thread is created to simulate the behavior of thread 1 based on the time schedule. We use the argument `TCGP_stack` to pass all the necessary information. For example, in the `search` test case, thread 1 will be recreated and executed `Task.run()` afterwards concurrently with thread 2, while in another test case `SohoSynoptic.runImpl()` will be invoked. All the classes loaded in the two test cases are the replay version of the program, which forces them to execute with the same logical order when a crash happened. ◄



```
void test_MetaSearchImpl_search(){
  ...
  Monitor.restartThreads(te.TCGP_stack, 2);
  Monitor.waitForThreads();
  thisObject.search(h, r);
  ...
}
```

```
void test_MetaSearchResult_cancel(){
  ...
  Monitor.restartThreads(te.TCGP_stack, 3);
  Monitor.waitForThreads();
  thisObject.cancel();
  ...
}
```

Fig. 6. An example of multi-threaded JUnit test cases generation

The last issue we care about is how to recreate all corresponding threads in the generated test cases. Since JUnit does not support multi-threaded test cases up to now, we use a `WrapperThread` to achieve the goal. By using Java Reflection APIs, the `WrapperThread` has the ability to dynamically invoke the specified method when needed. After loading information such as method name and arguments from trace files, a `WrapperThread` is started at specific *TCGP*.

3.3.1 Discussions about the Effectiveness of Generated Test Cases

To our intuition, the closer the start point of a test case to the fault locality, the more useful it is in the debugging process. As shown in Fig. 6 we use a term *Distance to the Crash Point* to indicate the distance for the test case to reach the crash point. Due to the selective instrumentation of `ConCrash`, a small set of test cases which are too far away from the crash point could possibly fail in reproducing that crash. In the worst case there is only one alive thread *t* at one *TCGP*, but after that a lot of new threads are created. If *t* crashes sometime later, then the test case generated at that specific *TCGP* could not be able to reproduce that crash, since it does not know which thread would be created in future. Our approach, though not sound, is demonstrated to be practically useful. In our experiment in Section 4, we show that the 'nearest' test cases are strongly guaranteed to reproduce crashes.

4 Implementation and Evaluation

We have implemented our approach for concurrent Java programs using the Chord static datarace detection tool [14], `ReCrash` tool [5], XStream [3] framework, and Soot analysis toolset [2]. The current implementation of our approach supports Java version 1.6. The preprocessing stage (includes identifying data race and lock/release operations) are built on top of Soot and Chord. We also use Soot to instrument all the class files containing instrumentation sites. By modifying `reCrashJ` [1] we implement `ConCrash` to support multi-threaded features. Like `reCrashJ`, the `ConCrash` implementation uses the stored shadow stack to generate a suite of JUnit tests. Each test in the suite invokes all alive threads at each *TCGP* and loads the receiver and method arguments from the serialized shadow stack.

To evaluate the effectiveness and efficiency of our proposed technique, we experimented on a number of multi-threaded Java benchmarks. Table 1 summarizes the details about each benchmark. Most of the benchmarks and failures are representative and frequently used in previous work [10] [6] [18] [14]. All the failures could not be reproduced by `ReCrash`, for it does not support multi-threaded applications.

Table 1. Subject Programs

Programs	#Loc	#Classes	#Methods	#Threads	Bug type	Brief description
XtangoAnimator	2088	31	220	3	Deadlock	Animation library
ftp	21897	118	1114	3	Data race	Apache ftp server
raytracer	1308	21	72	3	Data race	Ray tracing program
Hedc	29948	136	1552	6	Data race	Web crawler
Shop	299	3	11	3	Atomicity	Simulated shop
Pingpong	303	4	12	3	Data race	Simulated ball game

4.1 Procedure

We focus on the crash reproducibility and performance overhead of `ConCrash` in the evaluation. For each subject program, we first use Chord and Soot to get all instrumentation sites, then instrument the program and deploy it into a `ConCrash`-enabled environment.

Most of the bugs in the subject programs happen in very rare situations: 100 times of normal execution will not manifest those bugs. We seed a few `sleep()` operations in the program in the recording phase to trigger specific bugs. As demonstrated in [9], the thread schedule of seeded version is also a legal schedule of the original program, which guarantees the buggy schedule we capture could also happen in production run. When the program crashes within an exception, `ConCrash` outputs an instrumentation scheme which captures the thread execution orders (when failure occurs), and a set of JUnit tests. We then construct the replay version by using the instrumentation scheme, and then run all generated JUnit tests on it.

To measure the recording overhead we run each benchmark for a certain task for 100 times and compare the execution time of the original program with the execution time of the recorded version. The average time cost is taken between all the runs. At the same time the size of trace files are also recorded.

4.2 Results

The results for the experiments of concurrent crash reproducibility and performance overhead are shown in Tables 2 and 3, respectively. The experiments were done on a 3.00 GHz machine with 1GB memory and Sun JVM 1.6.0 in Windows XP system. In Table 2, we list the number of instrumentation sites and the crash type for each buggy program. To evaluate the effectiveness of `ConCrash`, we count the number of generated JUnit test cases with regard to the reproducible ones and reproduce rate. In Table 3 we compare the running time of the original program with instrumented ones. The time was measured in realtime execution time, and network related benchmarks (like **ftp** and **hedc**) were set in a local network to reduce the effects of web transmission. Slow down rate and the size of the trace files are also considered.

Table 2. Crash reproducibility study result

Programs	#Instrumentation sites data race	lock release	Exception type	# generated test cases	# reproducible test cases
XtangoAnimator	N.A.[1]	49	CustomException	1	1
ftp	51	61	NullPointerException	4	2
raytracer	52	17	AssertionError	1	1
Hedc	19	394	NullPointerException	4	3
Shop	10	3	ArrayOutOfIndex	3	2
Pingpong	4	0	NullPointerException	2	2

[1]Chord does not terminate after 10 hours in **XtangoAnimator**

4.2.1 Failure Reproducibility

We can observe from Table 2 that `ConCrash` was able to reproduce all three kinds of typical concurrent crashes (atomicity violation, deadlock, and data race) in the subject programs investigated.

In most cases, the generated tests of `ConCrash` could reproduce concurrent failures. As mentioned before, we do not record for every operation but rather for a small part of them. The failed cases are always because of the long distance between the *TCGP* to the crash point. However, our results showed that in our experiments all the failures could be reproduced by the last two closest test cases.

Table 3. Performance overhead study

Programs	Original(ms)	Instrumented(ms) data race/lock release/All			Slow down	File size(kb)
XtangoAnimator	367.1	N.A.	439	439	19.6%	1.52
ftp	3206	3946	4009	4565	42.4%	45.2
raytracer	47	75	51	78	66%	1.4
Hedc	1501.4	1676.5	1701.6	1731.3	15.3%	27.9
Shop	278.1	335.8	309.3	367.1	32%	3.1
Pingpong	315.6	367.2	N.A.[1]	367.2	16.3%	2.34

[1]There is no lock acquire/release operations in **Pingpong**

4.2.2 Performance Overhead

We compare the execution time of the original program and `ConCrash`-instrumented version in Table 3. All the applications are set for a certain task, except for **ftp** we wrote a test harness program to start a new user and download a specific file. The data is then collected in 100 times' **normal** runs for each program. We also provide a comparison between different instrumentation strategies. Our results showed `ConCrash` has a runtime overhead from 15% to 66%. We believe such overhead is acceptable for real-world use, though there is still improvement space if more selective instrumentation strategies are adapted. We also believe that the performance of `ConCrash` could be further improved with the integration of better data race/atomicity violation detection tools.

The size of the trace files is relatively small - 45kb for the largest benchmark, including both schedule files and JUnit tests. It could be easily sent via Internet to developers when a crash happens.

4.3 Threats to Validity

Like any empirical evaluation, this study also has limitations which must be considered. Although we have experimented with several well-known multi-threaded Java benchmarks, in which the largest one is over 30KLOC, they are smaller than traditional Java software systems. For this case, we can not claim that these experiment results can be necessarily generalized to other programs. On the other hand, the systems and crashes we chose to investigate might not be representative. Though we experimented on reproducing several typical concurrent failures, such as data race, deadlock, and atomicity violation, we still can not claim `ConCrash` could reproduce an arbitrary concurrent crash.

Another threat is that we have not conducted any readability study about the generated test case on software developers (though the readability of the test case is secondary to reproducibility). For this reason, we might not be able to claim `ConCrash` can be applied to real-world development process.

The final threat to internal validity maybe mostly lies with possible errors in our tool implementation and the measure of experiment results. To reduce these kind of threats, we have performed several careful checks.

4.4 Discussion

When designing the `ConCrash` approach, we choose bytecode level instrumentation to make concurrent failure be deterministically reproduced. If we use OS-level support or JVM-level capture and replay, `ConCrash` would have to be deployed on a specific environment, which will deteriorate the portability of our approach. In current `ConCrash` implementation, no other program or configuration is needed, and the pre-instrumentation also permits the high comparability of `ConCrash`-enabled environment.

When using Chord to report potential data races, the coverage of Chord (or other analysis techniques used for preprocessing) might affect the precision/effectiveness of the `ConCrash` approach. Investigating the tradeoffs would be one of our future directions.

Current implementation of `ConCrash` uses shallow (depth-1) copying strategy as default mode like [5]. However, in some cases (e.g., the Hedc benchmark) an argument is side-effected, between the method entry and the crash point, in such a way that will prevent the crash from reproducing.

5 Related Work

In this section, we discuss some closely related work in the areas of multi-threaded program analysis, testing, and debugging. We also compare several similar tools in Table 4.

Much research has been done on testing and debugging multi-threaded programs. Researchers have proposed analysis techniques to detect deadlocks [15], data races [14], and atomicity violations [10]. The problem of generating different interleavings for the purpose of revealing concurrent failures [15] and record/replay techniques [7, 13, 17, 20] have also been examined. Moreover, systematic and exhaustive techniques, like model checking [11], have been developed recently. These techniques exhaustively explore all interleavings of a concurrent program by systematically switching threads at synchronization points.

Choi *et al.* [7], presented the concept of logical thread schedule and proposed an approach to deterministically replay a multi-threaded Java program. They are able to reproduce race conditions and other non-deterministic failures. However, Their method relies on the modification of the underlying JVM, while our method uses bytecode level instrumentation to capture the thread execution orders. Moreover, our approach generates a series of JUnit tests, which help developers to debug the program.

Recently, Park *et al.* [17] also proposed the idea of deterministic replay of concurrency bugs. They used the feedback of previous failed replay attempt of the program to reproduce concurrency bugs. They also presented five sketching methods for recording the execution of concurrent programs, while our approach employs a static analysis tool as frontend to identify recording points. ConCrash tries to reproduce concurrency failures with JUnit tests at those points, which is also different from their feedback approach.

The most similar work to us is the `ReCrash` approach [5] proposed by Artzi *et al.*. `ReCrash` generates tests by utilizing partial snapshots of the program states captured on each method execution in the case of a failure. The empirical study shows that `ReCrash` is easy to implement, scalable to large program, and generate simple but helpful tests. Our work on `ConCrash` aims to handle concurrent failures. We use the concept of logical thread schedule to capture the execution order, making the behavior of multi-threaded program deterministic when reproducing the concurrent failure.

Table 4. A comparison between closely related testing tools for concurrent programs

Items	Instrumentation level	Running environment[1]	Deterministic replay in multi-threaded programs	Unit test cases generation
ReCrash [5]	byte code	both	No	Yes[2]
DejaVu [7]	JVM	developer site	Yes	No
RaceFuzzer [19]	byte code	developer site	No[3]	No
ConTest [9]	source code	developer site	Yes	No
ConCrash	byte code	both	Yes	Yes

[1] Running environment consists of user site and developer site

[2] ReCrash's generated test cases could not be applied in multi-threaded programs

[3] Depends on RaceFuzzer's random scheduler

6 Conclusions and Future Work

In this paper, we presented a lightweight and portable approach, called `ConCrash`, to making concurrent failures reproducible. `ConCrash` records the logical thread scheduling order and preserves object states in memory at runtime. When a crash occurs, it reproduces the failure offline by simply using the saved information without the need for JVM-level or OS-level support. To reduce the runtime overhead, `ConCrash` uses an effective existing data race detection technique to report all potential race conditions, and only instruments such places plus lock acquire/release points. We implemented the `ConCrash` approach in a prototype tool for Java. Our experiments on several well-known multi-threaded Java benchmarks indicate that `ConCrash` is effective in reproducing a number of typical concurrent bugs within an acceptable overhead.

We recommend the `ConCrash` approach be an integrated part of the existing `ReCrash` technique. As our future work, we would like to examine alternative techniques like dynamic program slicing [4] to improve the performance of `ConCrash`. We also intend to investigate the cost/effectiveness tradeoffs when reproducing concurrent failures at the application level.

Acknowledgements. We thank anonymous reviewers for useful comments. We also thank Shan Lu and STAP group members for their valuable discussions. This work was supported in part by National Natural Science Foundation of China (NSFC) (Grant No. 60970009).

References

1. reCrashJ implementation, `http://groups.csail.mit.edu/pag/reCrash/`
2. Soot Homepage, `http://www.sable.mcgill.ca/soot/`
3. XStream Project Homepage, `http://xstream.codehaus.org/`
4. Agrawal, H., Horgan, J.R.: Dynamic program slicing. In: PLDI 1990, pp. 246–256 (1990)
5. Artzi, S., Kim, S., Ernst, M.D.: Recrash: Making software failures reproducible by preserving object states. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 542–565. Springer, Heidelberg (2008)
6. Choi, J.D., Lee, K., Loginov, A., O'Callahan, R., Sarkar, V., Sridharan, M.: Efficient and precise datarace detection for multithreaded object-oriented programs. In: PLDI 2002, pp. 258–269 (2002)
7. Choi, J.D., Srinivasan, H.: Deterministic replay of Java multithreaded applications. In: SPDT 1998, pp. 48–59 (1998)
8. Choi, J.D., Zeller, A.: Isolating failure-inducing thread schedules. SIGSOFT Softw. Eng. Notes 27(4), 210–220 (2002)
9. Edelstein, O., Farchi, E., Nir, Y., Ratsaby, G., Ur, S.: Multithreaded Java program test generation. IBM Systems Journal 41(1), 111–124 (2002)
10. Flanagan, C., Freund, S.N.: Atomizer: a dynamic atomicity checker for multithreaded programs. In: POPL 2004, pp. 256–267 (2004)
11. Freund, S.N.: Checking concise specifications for multithreaded software. Journal of Object Technology 3, 81–101 (2004)
12. Lu, S., Park, S., Seo, E., Zhou, Y.: Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. SIGARCH Comput. Archit. News 36(1), 329–339 (2008)
13. Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Arumuga Nainar, P., Neamtiu, I.: Finding and reproducing heisenbugs in concurrent programs. In: OSDI 2008, pp. 267–280 (2008)
14. Naik, M., Aiken, A., Whaley, J.: Effective static race detection for Java. In: PLDI 2006, pp. 308–319 (2006)
15. Naik, M., Park, C., Sen, K., Gay, D.: Effective static deadlock detection. In: ICSE 2009, pp. 386–396 (2009)
16. O'Callahan, R., Choi, J.D.: Hybrid dynamic data race detection. SIGPLAN Not. 38(10), 167–178 (2003)
17. Park, S., Zhou, Y., Xiong, W., Yin, Z., Kaushik, R.H., Lee, K., Lu, S.: PRES: probabilistic replay with execution sketching on multiprocessors. In: SOSP 2009, pp. 177–192 (2009)
18. von Praun, C., Gross, T.R.: Object race detection. In: OOPSLA 2001, pp. 70–82 (2001)
19. Sen, K.: Race directed random testing of concurrent programs. In: PLDI 2008, pp. 11–21. ACM, New York (2008)
20. Steven, J., Chandra, P., Fleck, B., Podgurski, A.: jRapture: A capture/replay tool for observation-based testing. SIGSOFT Softw. Eng. Notes 25(5), 158–167 (2000)