Concurrent Breakpoints

Chang Seo Park
Koushik Sen

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2011-159
http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-159.html

December 18, 2011

Acknowledgement

Research supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227), by NSF Grants CCF-101781, CCF-0747390, CCF-1018729, and CCF-1018730, and by a DoD NDSEG Graduate Fellowship. The last author is supported in part by a Sloan Foundation Fellowship. Additional support comes from Par Lab affiliates National Instruments, Nokia, NVIDIA, Oracle, and Samsung.

Concurrent Breakpoints

Chang-Seo Park and Koushik Sen
EECS Department, UC Berkeley
Berkeley, CA 94720, USA
{parkcs,ksen}@cs.berkeley.edu

Abstract

In program debugging, reproducibility of bugs is a key requirement. Unfortunately, bugs in concurrent programs are notoriously difficult to reproduce compared to their sequential counterparts. This is because bugs due to concurrency happen under very specific thread schedules and the likelihood of taking such corner-case schedules during regular testing is very low. We propose concurrent breakpoints, a light-weight and programmatic way to make a concurrency bug reproducible. We describe a mechanism that helps to hit a concurrent breakpoint in a concurrent execution with high probability. We have implemented concurrent breakpoints as a light-weight library (containing a few hundreds of lines of code) for Java and C/C++ programs. We have used the implementation to deterministically reproduce several known non-deterministic bugs in real-world concurrent Java and C/C++ programs involving 1.6M lines of code. In our evaluation, concurrent breakpoints made these non-deterministic bugs almost 100% reproducible.

1 Introduction

A key requirement in program debugging is reproducibility. When a bug is encountered in a program execution, in order to fix the bug, the developer first needs to confirm the existence of the bug by trying to reproduce the bug. Developers also require that the bug can be reproduced deterministically so that they can run the buggy execution repeatedly with the aid of a debugger and find the cause of the bug. Reproducing bugs for sequential programs is relatively easy when given enough details. If all sources of non-determinism, such as program inputs (and values of environment variables in some cases), are recorded, a bug in a sequential program can be reproduced deterministically by replaying the program with the recorded inputs. Bugs in sequential programs can be reported easily to a bug database because a user only needs to report the input on which the sequential program exhibits the bug.

Unfortunately, concurrent programs are notoriously difficult to debug compared to their sequential counterparts. This is because bugs due to concurrency happen under very specific thread schedules and are often not reproducible during regular testing. Therefore, even if a user reports the input that caused a bug in a concurrent program, the developer may not be able to recreate the bug and debug the cause of the bug. Such non-deterministic bugs in concurrent programs are called *Heisenbugs*. One could argue that Heisenbugs could be made reproducible if the thread schedule is recorded along with program inputs during a program execution. Unfortunately, recording and replaying a thread schedule poses several problems: 1) It requires to observe the exact thread schedule either through program instrumentation or by using some specialized hardware. Instrumentation often incurs huge overhead and specialized hardware are often not easily available. 2) Replaying a thread schedule requires special runtime on the development side which could again incur huge overhead.

Nevertheless, we need some information about the thread schedule along with the program inputs to reproduce a Heisenbug. We, therefore, ask the question: is there a better way to provide necessary information about a thread schedule along with program inputs so that one can reproduce a Heisenbug? We would like the information about thread schedule to be portable so that we do not need a special runtime to reproduce the bug. In this paper, we propose a simple light-weight technique to specify enough information about a Heisenbug so that it can be reproduced with very high probability without requiring a special runtime or a full recording of the thread schedule.

Our technique for reproducibility is based on the observation that Heisenbugs can often be attributed to a set of program states, called *conflict states*. A program execution is said to be in a conflict state if there exists two threads such that both threads are either 1) trying to access the same memory location and at least one of the accesses is a write (i.e. a data race), or 2) they are trying to operate on the same synchronization object (e.g. contending to acquire the same lock). Depending on how a conflict state is resolved, i.e. which thread is allowed to execute first, a concurrent program execution could end up in different states. Such difference in program states often lead to Heisenbugs. Therefore, in order to reproduce a Heisenbug, one should be able to reach conflict states and control the program execution from those states.

In this paper, we propose *concurrent breakpoints*, a light-

weight and programmatic tool that facilitates reproducibility of Heisenbugs in concurrent programs. A concurrent breakpoint is an object that defines a set of program states and a scheduling decision that the program needs to take if a state in the set is reached. Typically, the set of states described by a concurrent breakpoint would be a set of conflict states. Formally, a concurrent breakpoint is a tuple of the form (ℓ_1, ℓ_2, ϕ), where ℓ_1 and ℓ_2 are program locations and ϕ is a predicate over the program state. A concurrent program execution reaches a concurrent breakpoint (ℓ_1, ℓ_2, ϕ) if there exists two threads t_1 and t_2 such that t_1 and t_2 are at program locations ℓ_1 and ℓ_2, respectively, and the states of t_1 and t_2 jointly satisfy the predicate ϕ. After reaching a state denoted by a concurrent breakpoint, the program makes a scheduling decision: it executes the next instruction of t_1 before t_2. In this paper, we consider concurrent breakpoints involving two threads; however, concurrent breakpoints could easily be generalized to more than two threads.

We show that concurrent breakpoints could represent all conflict states, i.e. they could represent data races and lock contentions. We also illustrate that concurrent breakpoints could represent other buggy states, such as a deadlock state or a state where an atomicity violation or a missed notification happens. We argue that the necessary information about a buggy schedule could be represented using a small set of concurrent breakpoints: if a program execution could be forced to reach all the concurrent breakpoints in the set, then the execution hits the Heisenbug. We show using probabilistic arguments that concurrent breakpoints are hard to reach during normal program executions. We propose a mechanism, called BTRIGGER, that increases the likelihood of hitting a concurrent breakpoint during a program execution. We provide a simplified probabilistic argument to show the effectiveness of BTRIGGER in hitting a concurrent breakpoint.

We have implemented concurrent breakpoints and BTRIGGER as a light-weight library (containing a few hundreds of lines code) for Java and C/C++ programs. We have used the implementation to reproduce several known Heisenbugs in real-world Java and C/C++ programs involving 1.6M lines of code. In our evaluation, concurrent breakpoints made these non-deterministic bugs almost 100% reproducible.

Concurrent breakpoints can be used as regression test cases for concurrent programs in a similar way we use data inputs as regression test cases for sequential programs. A set of concurrent breakpoints specifies the necessary information about a thread schedule that leads a program to a bug. After fixing a Heisenbug, the set of concurrent breakpoints denoting the Heisenbug can be kept as a regression test, in case a future change in the program leads to the same problem.

Our idea about concurrent breakpoints is motivated by recent testing techniques for concurrent programs, such as CalFuzzer [17, 39, 31, 18], AssetFuzzer [20], CTrigger [32], Penelope [40], and PCT [5]. In such testing techniques, one first identifies potential program statements where a delay or a context switch could be made to trigger a Heisenbug. Delays or context switches are then systematically or randomly inserted at those program statements to see if the resulting thread schedule could lead to a bug. The goal of this work is not to systematically or randomly explore thread schedules based on some prior information; rather, concurrent breakpoints make sure that once a bug is found, the bug can be made reproducible using a simple programmatic technique. Concurrent breakpoints also ensure that anyone can reproduce the bug deterministically without requiring the original testing framework and its runtime.

For reproducibility of Heisenbugs in concurrent programs, a number of light-weight and efficient techniques have been proposed to record and replay a concurrent execution [29, 36, 7, 35, 44, 28, 24, 33, 2]. A record and replay system dynamically tracks the execution of a concurrent program, recording the non-deterministic choices made by the scheduler. A trace is produced which allows the program to be re-executed, forcing it to take the same schedule. If captured in a trace, a concurrency bug can be replayed consistently during debugging. Note that these record-and-replay systems are automatic, but require heavy-weight machinery to record or replay a buggy thread schedule. In contrast, concurrent breakpoints provide a manual mechanism to make a Heisenbug reproducible and it requires no special runtime or heavy-weight machinery.

2 Concurrent Breakpoint

We define a *concurrent breakpoint* as the tuple (ℓ_1, ℓ_2, ϕ), where ℓ_1 and ℓ_2 are program locations and ϕ is a predicate over the program state. A program execution is said to have triggered a concurrent breakpoint (ℓ_1, ℓ_2, ϕ) if the following conditions are met

- the program reaches a state that satisfies the following predicate:

$$\exists t_1, t_2 \in \mathit{Threads} \quad . (t_1.pc = \ell_1) \wedge (t_2.pc = \ell_2) \wedge (t_1 \neq t_2) \wedge \phi, \text{ and}$$

- from the above state, the program executes the next instruction of thread t_1 before the next instruction of thread t_2.

That is, we say that a concurrent program execution triggers a concurrent breakpoint (ℓ_1, ℓ_2, ϕ) if the program reaches a program state and takes an action in the state. The state is such that it satisfies the predicate ϕ and there exists two threads t_1 and t_2 such that t_1 and t_2 are at program locations ℓ_1 and ℓ_2 in the state, respectively. The action at the state executes the thread t_1 before the thread t_2. In practice, we will restrict the predicate ϕ to the local states of t_1 and t_2, i.e. ϕ can only refer to variables local to t_1 and t_2. If v is

```
 1: void foo (Point p1) {
 2:   ...
 3:   p1.x = 10;
 4:   ...
 5: }
 6:
 7: void bar (Point p2) {
 8:   ...
 9:   t = p2.x;
10:   ...
11: }
```

Figure 1: Data Race

a local variable of thread t_1, then we will denote the variable using $t_1.v$.

Note that in our definition, a concurrent breakpoint involves two threads. The definition can be easily extended to involve more than two threads. For example, a concurrent breakpoint $(\ell_1, \ell_2, \ell_3, \phi)$ involves three threads. Our implementation of concurrent breakpoints, as described in the subsequent section, can be extended accordingly. To simplify exposition, in this paper, we restrict to concurrent breakpoints involving two threads.

We next illustrate how we can trigger various kind of bugs in a concurrent program using concurrent breakpoints. We consider three kinds of common concurrency bugs: data races, deadlocks, and atomicity violations. For each kind of bug, we illustrate through an example, the representation of the bug using a concurrent breakpoint.

For example, we can trigger a feasible data race in a program, i.e. reach a state in which two threads are about to access the same memory location and at least one of them is a write, using a concurrent breakpoint as follows. Consider the program in Figure 1. The concurrent breakpoint $(3, 9, t_1.\texttt{p1} == t_2.\texttt{p2})$ represents the state where two threads are at lines 3 and 9, respectively, and are about to access the same memory location denoted by the field `x` of the object referenced by both `p1` and `p2` and at least one of the accesses is a write. Such a racy state or the concurrent breakpoint could be reached if `foo` and `bar` are executed in parallel by different threads on the same `Point` object. The concurrent breakpoint also specifies that if the racy state is ever reached, then the thread reaching line number 3 must execute its next instruction before the thread reaching line number 9 executes its next instruction. This forces the program to resolve the data race in a particular order.

A deadlock in a concurrent program can also be represented using a concurrent breakpoint. Consider the code of Jigsaw in Figure 2. When the http server shuts down, it calls cleanup code that shuts down the `SocketClientFactory`. The shutdown code holds a lock on the factory at line 867, and in turn attempts to acquire the lock on `csList` at line 872. On the other hand, when a `SocketClient` is closing, it also calls into the factory to update a global count. In this situation, the locks

```
org.w3c.jigsaw.http.socket.SocketClientFactory{
130: SocketClientState csList;

574: synchronized boolean decrIdleCount(){
       ...
     }

618: boolean clientConnectionFinished(...) {
623:   synchronized (csList) {
         ...
626:     decrIdleCount();
         ...
       }
     }

867: synchronized void killClients(...) {
872:   synchronized (csList){
         ...
       }
     }
}
```

Figure 2: Deadlock in Jigsaw

are held in the opposite order: the lock on `csList` is acquired first at line 623, and then on the factory at line 574. The concurrent breakpoint $(626, 872, t_1.\texttt{csList} == t_2.\texttt{csList} \wedge t_1.\texttt{this} == t_2.\texttt{this})$ represents this deadlock state. The deadlock state is reached if any two threads t_1 and t_2 reach the line numbers 626 and 872, respectively, the `csList` variable of both t_1 and t_2 point to the same object, and the `this` pointer refers to the same objects in both t_1 and t_2.

Atomicity has been shown as a widely applicable non-interference property. A block of code in a concurrent program can be specified as atomic. This means that for every program execution where threads are arbitrarily interleaved, there is an equivalent execution with the same overall behavior where the atomic block of code is executed serially, that is, the code block's execution is not interleaved with actions of other threads.

An atomicity violation can be represented using a concurrent breakpoint. For example, consider the code snippet in Figure 3 from the Java class `java.lang.StringBuffer`. In the `append` method, `sb.length()` returns the length of `sb` and stores it in the local variable `len`. A call to `sb.setLength(0)` at this point by a second thread will make the length of `sb` 0. This makes the value of `len` stale, i.e. `len` no longer has the latest value of the length of `sb`. Therefore, the call to `sb.getChars(0,len,value,count)` in the `append` method by the first thread will throw an exception. This is a classic atomicity violation of the `append` method of `StringBuffer`. A concurrent breakpoint, which would trigger this atomicity violation, is $(239, 449, t_1.\texttt{sb} == t_2.\texttt{this})$. The atomicity violation is triggered if two threads t_1 and t_2 reach the line numbers 449 and 239, respectively, the `sb` variable of t_1 and the

```
143: public synchronized int length(){...}

322: public synchronized void getChars(...){...}

437: public synchronized
       StringBuffer append(StringBuffer sb){
       ...
444:   int len = sb.length();
       ...
449:   sb.getChars(0, len, value, count);
       ...
     }

239: public synchronized void setLength(...){
240:       ...
     }
```

Figure 3: Atomicity Violation in StringBuffer

```
     Initially: o.x = 0;

void foo(XObject o1){   void bar(XObject o2){
1. synchronized(o1){    10. o2.x = 1;
2.  f1();               11. synchronized(o2){
3.  f2();               12.   f6();
4.  f3();               13. }
5.  f4();               }
6.  f5();
7. }
8. if (o1.x==0)
9.   ERROR;
}

thread1 executes foo(o) and
thread2 executes bar(o) concurrently
```

Figure 4: A program with a hard to reach concurrent breakpoint $(8, 10, t_1.\texttt{o1} == t_2.\texttt{o2})$

`this` variable of t_2 point to the same object, and the thread reaching line 239 is executed before the other thread from the state. This atomicity violation also results in an exception if `len > 0`.

3 Triggering a Concurrent Breakpoint

Given a concurrent breakpoint (ℓ_1, ℓ_2, ϕ), it is very unlikely that two threads will reach statements labelled ℓ_1 and ℓ_2, respectively, at the same time in a concurrent execution, even though each thread could reach the statements independent of the other threads several times during the execution. Therefore, a concurrent breakpoint could be difficult to hit during a normal concurrent execution unless we have a mechanism to force an execution to the concurrent breakpoint.

For example, consider the two-threaded program in Figure 4. The program uses a shared memory location `o.x` which is initialized to 0. The important statements in this program are statements 8, 9, and 10. We add the other statements in the program to ensure that statement 8 gets executed after the execution of a large number of statements by `thread1` and statement 10 gets executed by `thread2` at the beginning.

Now consider the concurrent breakpoint $(8, 10, t_1.\texttt{o1} == t_2.\texttt{o2})$. If we run the program, then the likelihood of reaching a state where `thread1` is at line 8 and `thread2` is at line 10 is very low. Therefore, the concurrent breakpoint will hardly be hit by any program execution.

We next describe a mechanism that tries to force a program execution to a concurrent breakpoint. We call this mechanism BTRIGGER. We also provide a simplified probabilistic argument to justify the effectiveness of BTRIGGER. Recall that a concurrent breakpoint (ℓ_1, ℓ_2, ϕ) denotes a set of program states and an action, where a state in the set satisfies the following predicate:

$$\exists t_1, t_2 \in \mathit{Threads} \,.\, (t_1 \neq t_2) \wedge (t_1.pc = \ell_1) \wedge (t_2.pc = \ell_2) \wedge \phi$$

The predicate can be rewritten as follows:

$$\exists t_1, t_2 \in \mathit{Threads} \,.\, (t_1 \neq t_2) \wedge \phi_{t_1} \wedge \phi_{t_2} \wedge \phi_{t_1 t_2}$$

where ϕ_{t_1} only refers to local variables of thread t_1, ϕ_{t_2} only refers to local variables of thread t_2, and $\phi_{t_1 t_2}$ refers to local variables of both t_1 and t_2.

BTRIGGER works as follows. During the execution of a program, whenever a thread reaches a state satisfying the predicate ϕ_{t_i} where $i \in \{1, 2\}$, we postpone the execution of the thread for T time units and keep the thread in a set $\texttt{Postponed}_{t_i}$ for the postponed period. We continue the execution of the other threads. If another thread, say t, reaches a state satisfying the predicate ϕ_{t_j} where $j \in \{1, 2\}$ then we do the following. If there is a postponed thread, say t', in the set $\texttt{Postponed}_{t_i}$ where $i \neq j$ and local states of the two threads t and t' satisfy the predicate $\phi_{t_1 t_2}$, then we report that the concurrent breakpoint has been reached. Otherwise, we postpone the execution of the thread t by T time units and keep the thread in the set $\texttt{Postponed}_{t_j}$ for the postponed period. If the concurrent breakpoint is reached, we also order the execution of threads t and t' according to the order given by the concurrent breakpoint.

Note that we do not postpone the execution of a thread indefinitely because this could result in a deadlock situation if all threads reach either ℓ_1 or ℓ_2 and none of the breakpoint predicates are satisfied by any pair of postponed threads.

BTRIGGER ensures that if a thread reaches a state satisfying the concurrent breakpoint partly (i.e. reaches a state satisfying the predicate ϕ_{t_1} or ϕ_{t_2}), it is paused for a reasonable amount of time, giving a chance to other threads to catch up and create a state that completely satisfies the concurrent breakpoint. This simple mechanism increases the likelihood of hitting a concurrent breakpoint.

We now describe a simplified probabilistic argument to show the effectiveness of BTRIGGER. Consider a two

threaded program where each thread executes N steps and the execution of one thread does not affect the execution of the other thread, i.e. the two threads execute independently. In these N steps, let us assume that a thread t visits a state satisfying ϕ_t M times uniformly at random and visits a state satisfying the concurrent breakpoint m times, again uniformly at random. Note that $m \leq M$. Then the probability that the two threads will reach a state satisfying the concurrent breakpoint is

$$1 - \frac{^{N-m}\mathcal{C}_m}{^{N}\mathcal{C}_m}$$

The above probability follows from the following facts. Assume that thread t_1 has visited the m states satisfying the concurrent breakpoint at time steps $N_1, N_2, \ldots, N_m$, respectively. The number of possible ways in which t_2 could visit m states, where each state satisfies the concurrent breakpoint, within N steps is $^{N}\mathcal{C}_m$. Similarly, the number of ways in which t_2 could visit m states, where each state satisfies the concurrent breakpoint and is not visited at time steps $N_1, N_2, \ldots, N_m$, is $^{N-m}\mathcal{C}_m$.

The above probability is upper bounded by

$$1 - \left(1 - \frac{m}{N - m + 1}\right)^m$$

For $m \ll N$, this probability is approximately equal to (using the Binomial theorem)

$$\frac{m^2}{N - m + 1}$$

In BTRIGGER, we pause a thread t for T steps whenever it reaches a state satisfying ϕ_t. Therefore, a thread now takes $N + MT$ time steps to complete its execution. Then the probability that two threads t_1 and t_2 will reach a state satisfying the concurrent breakpoint is greater than (using similar reasoning as before)

$$1 - \frac{^{N+MT-M-mT}\mathcal{C}_m}{^{N+MT-M}\mathcal{C}_m}$$

which is lower bounded by

$$1 - \left(1 - \frac{m}{N + MT - M}\right)^{mT}$$

For $m \ll N$, this probability is approximately equal to

$$\frac{m^2 T}{N + MT - M}$$

Therefore, BTRIGGER increases the probability of hitting a concurrent breakpoint by a factor of at least

$$\frac{T(N - m + 1)}{N + MT - M}$$

```
abstract public class BTrigger {
  private String name;
  public BTrigger(String name) {
    this.name = name;
  }

  abstract public
  boolean predicateGlobal(BTrigger bt);

  abstract public
  boolean predicateLocal();

  public
  boolean triggerHere(boolean isFirstAction,
                      int timeoutInMS);
}
```

Figure 5: BTrigger API for Java

This factor increases if we increase T and decrease M which is lower bounded by m. We can decrease M if we make the concurrent breakpoint more precise, i.e. we make the concurrent breakpoint such that most states that satisfy $\phi_{t_1} \wedge \phi_{t_2}$ must also satisfy $\phi_{t_1 t_2}$. We can increase T by making a thread t pause longer at a state satisfying ϕ_t, but pausing a thread longer increases the running time of the program. The programmer can modify these two parameters (wait time T and the precision of predicates) if she cannot hit a bug with a concurrent breakpoint with high probability. The effects are confirmed empirically in section 6.

4 A Concurrent Breakpoint Library

We have implemented BTRIGGER as a light-weight library for Java and C/C++ programs. The breakpoints are inserted as extra code in the program under test. The breakpoints can be turned on or off like traditional `assertions`.

We next describe the design and implementation of a concurrent breakpoint library for Java programs. The API for C/C++ programs is similar to that of Java. While one could specify a concurrent breakpoint using the notations described above and then use a compiler to weave the necessary code into a program, we provide a small library which programmers could use directly in the program to specify a concurrent library. Figure 5 shows the abstract concurrent breakpoint class `BTrigger` and Figure 6 describes an implementation of the abstract class. The implementation allows one to specify a breakpoint of the form $(\ell_1, \ell_2, t_1.\texttt{obj} == t_2.\texttt{obj})$.

For example, Figure 7 shows the code in Figure 1 after the insertion of the concurrent breakpoint $(9, 3, t_1.\texttt{p2} == t_2.\texttt{p1})$. Note that the breakpoint is split into two statements and inserted before line numbers 3 and 9, respectively. In general, if we have a concurrent breakpoint $(\ell_1, \ell_2, \phi_{t_1} \wedge \phi_{t_2} \wedge \phi_{t_1 t_2})$, we subclass `BTrigger` and encapsulate the local states of the thread that are relevant to compute $\phi_{t_1} \wedge \phi_{t_2} \wedge \phi_{t_1 t_2}$. We also implement its abstract method

```
class ConflictTrigger extends BTrigger {
  private Object obj;

  public ConflictTrigger(String name,Object obj){
    super(name);
    this.obj = obj;
  }

  public
  boolean predicateGlobal(BTrigger bt){
    if (name.equals(bt.name)) {
      if ((bt instanceof ConflictTrigger)
           &&obj==((ConflictTrigger)bt).obj){
        System.err.println("Conflict");
        return true;
      }
    }
    return false;
  }

  public boolean predicateLocal() {  }
}
```

Figure 6: Implementation of a concurrent breakpoint for triggering data races

```
 1:void foo (Point p1) {
 2:  ...
   (new ConflictTrigger("trigger1",p1)).
       triggerHere(false,Global.TIMEOUT);
 3:  p1.x = 10;
 4:  ...
 5:}
 6:
 7:void bar (Point p2) {
 8:  ...
   (new ConflictTrigger("trigger1",p2)).
       triggerHere(true,Global.TIMEOUT);
 9:  t = p2.x;
10:  ...
11:}
```

Figure 7: Concurrent breakpoint to trigger the data race in Fig. 1

`predicateLocal` so that it returns true if and only if ϕ_{t_1} (or ϕ_{t_2} depending on the context) is satisfied. Similarly, we implement the abstract method `predicateGlobal` so that it returns true if and only if $\phi_{t_1 t_2}$ is satisfied. The argument to the constructor of `BTrigger` uniquely identifies a concurrent breakpoint. Two `BTrigger` instances with the same name are considered to be part of the same concurrent breakpoint. We create two instances of the subclass and call their `triggerHere` methods just before line numbers ℓ_1 and ℓ_2, respectively. The first argument of `triggerHere` denotes the action: the first argument of the `triggerHere` at line number ℓ_1 is set to true and the first argument of the other `triggerHere` call is set to false. The second argument specifies the time for which a thread will pause if the `predicateLocal` is satisfied. `triggerHere` returns true if and only if the concurrent breakpoint is satisfied, i.e. if both `predicateLocal` and `predicateGlobal` of the instance on which `triggerHere` has been called returns true.

```
class DeadlockTrigger extends BTrigger{
   private Object lok1, lok2;

   public DeadlockTrigger(String name,
                Object lok1,Object lok2){
      super(name);
      this.lok1 = lok1;
      this.lok2 = lok2;
   }

   public
   boolean predicateGlobal(BTrigger bt){
    if (name.equals(bt.name)) {
      if ((bt instanceof DeadlockTrigger)
           &&lok1==((DeadlockTrigger)bt).lok2
           &&lok2==((DeadlockTrigger)bt).lok1){
        System.err.println("Deadlock");
        return true;
      }
    }
    return false;
   }

   public boolean predicateLocal() {  }
}
```

Figure 8: Implementation of a concurrent breakpoint for triggering deadlocks

Figure 8 shows the implementation of a concurrent breakpoint to trigger a deadlock state. The class records the lock objects `lok1`, the lock that the thread has already acquired and `lok2`, the lock that the thread is about to acquire. If for two instances, `c1` and `c2`, of this class `c1.lok1==c2.lok2 && c1.lok2==c2.lok1`, then the concurrent breakpoint denotes a deadlock. Usage of this class is shown in Figure 9. The concurrent breakpoint inserted in the figure triggers a deadlock in the Jigsaw webserver.

5 Inserting Concurrent Breakpoints

In this section, we describe two methodologies for inserting concurrent breakpoints in programs in order to make Heisenbugs reproducible. Heisenbugs happen due to one or more conflicting operations among various threads in a concurrent program. Two threads are said to be in a conflicting state if

- they are trying to access the same memory location and at least one of them is a write, or
- they are trying to operate on the same synchronization object, e.g. trying to acquire the same lock.

If the threads are trying to access the same memory location and at least one of them is a write, then the conflict state denotes a classic data race. If the threads are trying to acquire the same lock in a conflicting state, then the conflicting state

```
org.w3c.jigsaw.http.socket.SocketClientFactory{
130: SocketClientState csList;

574: synchronized boolean decrIdleCount(){
       ...
     }

618: boolean clientConnectionFinished(...) {
623:   synchronized (csList) {
         ...
         (new DeadlockTrigger("trigger2",csList,
             this)).triggerHere(true,Global.
             TIMEOUT);
626:     decrIdleCount();
         ...
       }
     }

867: synchronized void killClients(...) {
       (new DeadlockTrigger("trigger2",this,
           csList)).triggerHere(false,Global.
           TIMEOUT);
872:   synchronized (csList){
         ...
       }
     }
}
```

Figure 9: Concurrent breakpoint to trigger deadlock in Jigsaw

denotes a lock contention. Depending on how a conflict is resolved, i.e. which thread is allowed to execute first from a conflicting state, a concurrent program could end up in different states. Heisenbugs often arise due to such difference in program states. Therefore, if one could guess the conflicts that lead to a Heisenbug and resolve those conflicts in a way such that the Heisenbug could definitely be reached, then she could attribute those conflicts to the Heisenbug. We next describe two methodologies to guess such conflicting states.

Methodology I. If a Heisenbug is present in a concurrent program, concurrent breakpoints can be inserted in the program with the aid of a testing tool as follows. First a testing technique for concurrent programs, such as CalFuzzer [17] and CTrigger [32], can be used to find various kinds of Heisenbugs, such as data races, deadlocks, and atomicity violations, in a concurrent program. These tools also provide useful diagnostic information whenever they find a bug. For example, if CalFuzzer finds a data race, it reports the shared memory involved in the data race and the line numbers of the program statements where the data race happened. A sample report is shown below:

```
Data race detected between
  access of x.f at sample/Test1.java:line 15, and
  access of y.f at sample/Test1.java:line 20.
```

Once we have such a bug report, we can easily insert a concurrent breakpoint in the program to force the bug. For example, for the above sample report, we insert the program statements

```
(new ConflictTrigger("trigger1",x)).triggerHere(
    true,Global.TIMEOUT);
```

and

```
(new ConflictTrigger("trigger1",y)).triggerHere(
    false,Global.TIMEOUT);
```

at lines 15 and 20 of `sample/Test1.java`, respectively. This denotes the concurrent breakpoint $(15, 20, t_1.\texttt{x}== t_2.\texttt{y})$. The concurrent breakpoint makes the data race reproducible and resolves the race in one particular way. Similarly, we can insert concurrent breakpoints to make deadlocks and atomicity violations reproducible using bug reports produced by a testing tool. For example, a deadlock report produced by CalFuzzer for the deadlock in Jigsaw (see Figure 2) is

```
Deadlock found:
 Thread10 trying to acquire lock this while
     holding lock csList at org/w3c/jigsaw/http/
     socket/SocketClientFactory.java:line 623
 Thread15 trying to acquire lock csList while
     holding lock this at org/w3c/jigsaw/http/
     socket/SocketClientFactory.java:line 872
```

To make this deadlock reproducible, we insert the program statements

```
(new DeadlockTrigger("trigger2",csList,this)).
    triggerHere(true,Global.TIMEOUT);
```

and

```
(new DeadlockTrigger("trigger2",this,csList)).
    triggerHere(false,Global.TIMEOUT);
```

at lines 623 and 872 of `org/w3c/jigsaw/http/socket/SocketClientFactory.java`, respectively.

In our experiments, we used this methodology to make previously known bugs found by CalFuzzer reproducible. Note that one can argue that there is no need to insert a concurrent breakpoint because the cause of the bug has already been found and can be reproduced by the testing tool that discovered the bug. This is true; however, these testing tools often use heavy machinery and program instrumentation to reproduce the bug. Concurrent breakpoints make reproduction of bugs light-weight; it does not require instrumentation of the entire program or any sophisticated record-and-replay mechanism.

Methodology II. Our second methodology requires more manual effort than our first methodology. This methodology is useful for Heisenbugs, such as missed notifications, that cannot be detected easily using concurrency testing techniques. If a concurrent program shows a Heisenbug rarely during stress testing, we run an off-the-shelf data race detector such as Eraser to find all potential conflicting states, i.e. data races as well as lock contentions and contentions over synchronization objects. Note that a data race detector

could trivially be modified to detect all potential lock contentions and contentions over synchronization objects. CalFuzzer [17] has such an implementation. We go over the reported list of potential conflict states and try to find if some conflict states look suspicious and could be the cause for the Heisenbug. We insert concurrent breakpoints corresponding to those conflict states one-by-one. For each concurrent breakpoint, we also try both actions and see if the Heisenbug manifests more frequently during repeated executions. If for a concurrent breakpoint the Heisenbug starts manifesting more frequently, we try either to increase the pause time or to improve the precision of the breakpoint so that the frequency of the Heisenbug increases. Note that our probabilistic analysis in the previous section suggests that increasing the pause time or improving the precision of the concurrent breakpoint should increase the probability of hitting a concurrent breakpoint. For example, we increase the pause time in BTRIGGER from 100 milliseconds to 1 second or 10 seconds and see if the Heisenbug becomes more frequent. We also try to make the concurrent breakpoint more precise by adding more context under which the concurrent breakpoint should reach. For example, we could say that the concurrent breakpoint could only be true if some particular statement has been executed more than 7200 times. We repeat this process with other conflict states that look suspicious until we reach a stage where the Heisenbug becomes frequent, i.e. happens on almost every execution.

We next describe this methodology on a missed notification bug in log4j 1.2.13, a logging library in Java.

1. First, we observed that in 5 out of 100 test executions, the program would stall.
2. After running a conflict detector, we got the following lock contentions:

```
Lock contention:
 org/apache/log4j/AsyncAppender.java:line 100,
 org/apache/log4j/AsyncAppender.java:line 309
Lock contention:
 org/apache/log4j/AsyncAppender.java:line 236,
 org/apache/log4j/AsyncAppender.java:line 309
Lock contention:
 org/apache/log4j/AsyncAppender.java:line 100,
 org/apache/log4j/AsyncAppender.java:line 236
Lock contention:
 org/apache/log4j/AsyncAppender.java:line 277,
 org/apache/log4j/AsyncAppender.java:line 309
```

   Line 100 is in the `append` function that has a `wait` and `notify` within the synchronized block. Line 236 is in the `setBufferSize` function. Line 277 is in the `close` function with a `notify` in the synchronized block. Line 309 is inside the `run` function of the Dispatcher thread, with a `wait` and `notify` in the synchronized block.

3. For each of the conflicts, we added concurrent breakpoints before the lock acquisitions. We resolved the contention in both ways, i.e. the left lock (in the first column of the table below) acquisition is allowed before the right lock acquisition and vice-versa. The results of this experiment are summarized in the following table.

Conflict resolve order	System stall (%)	BP hit (%)
100 → 309	0	100
309 → 100	0	100
236 → 309	100	100
309 → 236	0	100
100 → 236	0	100
236 → 100	0	100
309 → 277	97	3
277 → 309	99	1

4. From the table in step 3, we can infer the following.
   (a) From the second pair, we inferred that when the lock at line 236 is acquired before line 309, the concurrent breakpoint was always triggered and resulted in a system stall. If the locks are acquired in the opposite order, the system did not stall.
   (b) From the fourth pair, we can see that adding the breakpoint resulted in much more frequent system stalls; however, the concurrent breakpoint was not reached in most of the executions. Therefore, we can conclude that the system stall happens because of a different set of conflicts.
   (c) Note that when we added concurrent breakpoints for the first and third pair of statements, the system did not stall irrespective of the order in which we resolved the lock conflicts.
5. Based on the above observations, we can insert a breakpoint at the second pair of statements and make the stall bug reproducible.

6 Evaluation

We have applied BTRIGGER to 15 Java benchmark programs and libraries and three C/C++ programs. The Java programs and libraries had previously known bugs such as data races, atomicity violations, deadlocks, stalls, and exceptions. We used bugs reports produced by CalFuzzer [17] to insert concurrent breakpoints in these programs using methodology I from section 5 except for three bugs. The three bugs—stalls in jigsaw, lucene, and log4j due to missed notifications—were reported in bug databases and the breakpoints were added using methodology II. We added a total of 31 breakpoints to 15 Java programs with over 850K total lines of code. After the insertion of suitable concurrent breakpoints, i.e. breakpoints that make a Heisenbug almost reproducible, we ran each program with the breakpoints 100 times to measure the empirical probability of hitting the breakpoint. We performed the experiments on a dual socket quad-core Intel Xeon 2GHz Linux server with 8GB of RAM.

We used the following benchmarks for our evaluation: `cache4j`, a fast thread-safe implementation for caching Java objects; `hedc`, a web-crawler from ETH; three computationally intensive multi-threaded applications: `moldyn`, `montecarlo`, and `raytracer` from the Java Grande Forum [16]; and `Jigsaw`, W3C's leading-edge Web server platform. For Jigsaw, we used a test harness that simulates multiple clients making simultaneous web page requests and sending administrative commands to control the server.

We also evaluated our tool on the following libraries with known bugs: `java.lang.StringBuffer`, `java.util.Collections$SynchronizedList` backed by an `ArrayList`, and `Collections$SynchronizedMap` backed by a `LinkedHashMap`, `java.util.logging`, which provides core logging facilities, `javax.swing`, which provides portable GUI components, `log4j`, an elaborate logging library, `lucene`, an indexing and searching library, and `pool`, an object pooling API.

We applied BTRIGGER for C/C++ on pbzip2, a parallel compression tool, Apache httpd 2.0.45 web server, and various versions of MySQL servers. The details of the bugs were retrieved from related papers [23, 45] and from the actual bug reports in the bug databases. Based on this knowledge, the breakpoints were added manually at the conflict states such that each bug is repeatedly reproducible.

6.1 Results

Table 1 summarizes the results for our experiments on Java programs. For each benchmark, we report the lines of code. We report the normal runtime (in seconds) of each benchmark in the third column. The fourth column reports the runtime of each benchmark with concurrent breakpoints added. The fifth column reports the overhead (in percentage) of running each benchmark with concurrent breakpoints. The sixth column reports the type of bug for which we added the concurrent breakpoints, with numbers to distinguish between different bugs. The seventh column describes the error produced due to the bug. The eighth column gives the empirical probability of triggering the breakpoints and causing the bug, which is the fraction of executions that hit the bug over 100 executions. The ninth column states any additional parameters or conditions that were used to make the concurrent breakpoints more precise.

For most of the benchmarks, the overhead of running the program with the concurrent breakpoint library was within 40% of the normal runtime. However, in some cases where we increased the waiting time to achieve a higher probability of hitting the breakpoint, the overhead became as large as 13x. We will discuss this issue in section 6.2.

For Jigsaw, we cannot accurately measure the running time due to its interactiveness as a server and we do not report it. In cases when the system stalled, we report the time that we first detected the stall, e.g. when the deadlock conditions have been met. Stalls due to missed notifications are detected by large timeouts; therefore, the runtime and overhead for such errors are omitted.

In some cases, the overhead is negative. `hedc` is a web crawler whose running times can fluctuate with the network status and cause inaccuracies. For some of the smaller benchmarks such as `synchronizedMap`, the overhead is negligible and within the margin of error.

Table 2 summarizes the results for our experiments on C/C++ programs. We report the size of the program in lines of code and the type of error that is reproduced using BTRIGGER. Since the programs we experimented on are continuously running servers (except pbzip2), the runtime of the program and probability of reproducing the bug is meaningless because all the bugs are eventually reproduced given enough client requests. Instead, we report the mean-time-to-error (MTTE) in column 4, to show the effectiveness of BTRIGGER in reproducing bugs quickly. Column 5 denotes the number of concurrent breakpoints that were required to consistently reproduce the error.

The results show that each bug can be reproduced very quickly. For the programs that crashed due to the bug, the MTTE corresponds to the mean time to the next crash after re-execution. For non-crashing bugs, the MTTE is calculated by counting the visible artifacts of the bug during the span of its execution. All the bugs are reproduced within a few seconds, showing the effectiveness of using concurrent breakpoints for reproducing complex bugs.

6.2 Increasing the Pause Time of Breakpoints

Most of the bugs in Java programs were triggered with probability close to 1.0, when we added a concurrent breakpoint. However, some bugs were triggered with lower probability, as low as 0.63. For such bugs, we increased the pause time of the breakpoints. For example, in race1 for `hedc`, the breakpoint was triggered with probability 0.87 when the pause time was 100 milliseconds. When we increased the pause time to 1 second, the breakpoint triggered with probability 1.0. Similarly for Swing, the deadlock occurred with probability 0.63 when the pause time was 100 milliseconds and the probability increased to 0.99 when the pause time was increased to 1 second. In both of these cases, the probability was increased by lengthening the pause time at the breakpoints, but the runtime overhead increased as well.

6.3 Improving the Precision of Breakpoints

In benchmarks where breakpoints increased the execution time significantly, we found that refining the local predicates was useful to prevent unnecessary waiting. In `cache4j`, the test harness generates a fixed number of objects during initialization. The constructor of `CacheObject` is involved in an atomicity violation, but with a weak local predicate (based on the program counter and lock object), the break-

Benchmark	LoC	Runtime (seconds)			Breakpoint type/no.	Error	Prob.	Comments
		Normal	w/ ctr	Overhead (%)				
cache4j	3897	1.992	2.089	4.9	race1		1.00	
			2.116	6.2	race2		0.99	
			2.101	5.5	race3		1.00	
			2.051	3.0	atomicity1		1.00	ignoreFirst=7200
hedc	29,947	1.780	2.042	14.7	race1		0.87	wait=100ms
			3.835	115.4	race1		1.00	wait=1000ms
			1.659	-6.8	race2		0.96	wait=1000ms
jigsaw	160K	-	-	-	deadlock1	stall	1.00	
					deadlock2	stall	1.00	
					missed-notify1	stall	1.00	Meth. II
					race1	stall	1.00	
					race2		1.00	
log4j 1.2.13	32,095	0.190	0.208	9	deadlock1	stall	1.00	
		0.135	-	-	missed-notify1	stall	1.00	Meth. II
logging	4250	0.140	0.140	0	deadlock1	stall	1.00	
lucene	171K	0.136	0.159	17	deadlock1	stall	1.00	
moldyn	1290	1.098	1.204	9.7	race1		1.00	bound=4
			1.302	18.6	race2		1.00	bound=10
montecarlo	3560	1.841	2.162	17.4	race1		1.00	bound=10
pool	11,025	0.131	-	-	missed-notify1	stall	1.00	Meth. II
raytracer	1860	1.097	1.274	16.1	race1	test fail	1.00	
			1.196	9.0	race2	test fail	1.00	
			1.360	24.0	race3		1.00	
			1.428	30.2	race4		1.00	
stringbuffer	1320	0.131	0.159	21	atomicity1	exception	1.00	
swing	422K	0.902	5.597	521	deadlock1	stall	0.63	wait=100ms
			12.003	1230	deadlock1	stall	0.99	wait=1000ms
synchronizedList	7913	0.134	0.142	6	atomicity1	exception	1.00	
		0.131	0.134	2	deadlock1	stall	1.00	
synchronizedMap	8626	0.132	0.173	31	atomicity1		1.00	
		0.133	0.131	-2	deadlock1	stall	1.00	
synchronizedSet	8626	0.132	0.183	39	atomicity1	exception	1.00	
		0.132	0.134	2	deadlock1	stall	1.00	

Table 1: Experimental results for Java programs

point will wait too many times causing a large execution overhead. In this case, we observed that the breakpoint was triggered after some fixed large number of timeouts for the breakpoint in the constructor. Therefore, we ignored the first n timeouts by adding the predicate $thisBreakpointHit > n$ (where $n = 7200$ in our case) to the local predicate in the constructor of `CacheObject`. This helped us to reduce the execution time by removing the unnecessary waits.

Another cause for increased execution time was that some breakpoints were triggered many times in an execution. For example, in `moldyn` the breakpoint that causes a data race was observed hundreds of times within a single execution. We added a bound to the local predicate, i.e. $triggers < bound$, and increased $triggers$ whenever the race breakpoint was triggered. This decreased the execution time to a reasonable level as reported in the table.

Refining the local predicate for deadlocks also helped in cases where the deadlock depended on the context of the breakpoint. For example, in `Swing`, the method `addDirtyRegion0()` of class `RepaintManager` is called within many contexts, but the deadlock occurs only if the corresponding `BasicCaret` lock is held. We increased the pause time of the breakpoint in `addDirtyRegion0()` to get higher probability, but this caused the overall runtime to increase significantly because the thread was pausing at the breakpoint even when a lock on `BasicCaret` was not held and the deadlock could not possibly happen. Therefore, to eliminate this unnecessary overhead, we added the predicate $isLockTypeHeld(type)$, where `type=BasicCaret`, to the local predicate of the breakpoint in `addDirtyRegion0()`. The refinement decreased the execution time significantly, without sacrificing

Benchmark	LoC	Error	MTTE[1] (sec)	#CBR[2]	Comments
pbzip2 0.9.4	2.0K	program crash	1.2	2	null pointer dereference
Apache httpd 2.0.45	270K	log corruption	0.14	1	(Bug #25520)
		server crash	0.33	3	buffer overflow
MySQL 4.0.12	526K	log omission	0.12	2	(Bug #791)
MySQL 3.23.56	468K	log disorder	0.065	1	(Bug #169)
MySQL 4.0.19	539K	server crash	2.67	3	null pointer dereference (Bug #3596)

Table 2: Experimental results for C/C++ programs ([1] Mean Time To Error. [2] No. of concurrent breakpoints required.)

the probability of hitting the deadlock.

7 Related Work

Record and replay systems [21, 29, 36, 7, 34, 26] have been used to reproduce non-deterministic bugs and enable cyclic debugging. However, most record and replay systems require instrumentation of the program source or binary and other heavy-weight techniques incurring high overhead. BTRIGGER, on the other hand, is a light-weight technique for bug reproducibility requiring small manual programmatic additions instead of instrumentation of the entire program. ODR [2] and PRES [33] reduce overhead by recording only partial execution information, but require a potentially expensive offline search for reproducing executions.

There is a large body of work [38, 8, 9, 42, 37, 6, 1, 4, 11, 25, 10] on finding concurrency bugs, using static and dynamic analyses. Random testing has also been proposed to find real bugs in concurrent programs. Active testing [39, 18, 17] uses a biased random model checker to guide the scheduler towards buggy states. ConTest [30] adds random noise to the scheduler to increase the probability of hitting concurrent bugs. PCT [5] uses a randomized scheduler to probabilistically guarantee finding concurrency bugs of a certain depth. Although these tools are largely successful in finding bugs, it is sometimes hard to convey to the developer the information required to deterministically reproduce bugs. Using BTRIGGER, we can specify the relevant parts of the schedule required to reproduce the bug in a light-weight programmatic manner that does not require the developer to download and run any tools.

IMUnit [15] proposes a novel language to specify and execute schedules for multithreaded tests. The proposed language is based on temporal logic and testing requires instrumentation of code. IMUnit is less intrusive as it does not require to modify code. BTRIGGER can be also seen as a mechanism to constrain thread schedules; however, we do not introduce a new language to write concurrent breakpoints and our approach requires no code instrumentation, but it requires manual breakpoint insertion.

Model checking is a promising technique to find concurrency bugs in programs before they manifest in the wild; however, the cause of the bug can be difficult to pinpoint in an error trace returned by a model checker. A number of researchers have tried to minimize an error trace and extract useful counterexamples when a bug is found [3, 14, 13]. Statistical sampling techniques can find bugs in the sequential setting [22], and extensions have been proposed to discover concurrency bugs [41]. Program slicing [43, 46] is a popular debugging approach that determines which parts of a program are relevant to a particular statement (e.g. a bug). Precise slicing for concurrent programs is undecidable in general but a number of work have investigated efficient approximate approaches for debugging [19, 27, 12].

8 Discussion

Traditionally, programmers have used various ad-hoc tricks, such as inserting sleep statements and spawning a huge number of threads, to make a Heisenbug reproducible. These tricks are often found in various bug reports that are filed in open bug databases. We proposed a more scientific and programmatic technique to make a Heisenbug reproducible. We described how one can use concurrent breakpoints to reproduce Heisenbugs in concurrent programs. We empirically showed that concurrent breakpoints could be used to make Heisenbugs in real-world programs almost always reproducible in a light-weight and programmatic way. We believe that concurrent breakpoints could also be used in other novel ways.

For example, concurrent breakpoints could be used to constrain the thread scheduler of a concurrent program. This is because a concurrent breakpoint can denote a conflict state, i.e. a program state where an execution can make a non-deterministic choice, and specify a particular resolution of the conflict state. Therefore, if we can denote all possible conflict states in a program using concurrent breakpoints and specify their resolutions, we can force the program to take a unique thread schedule. Therefore, concurrent breakpoints could be used to write concurrent units tests that exercise a specific thread schedule. In general, one could use a few concurrent breakpoints to limit the number of allowed thread schedules.

9 Acknowledgements

Research supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227), by NSF Grants CCF-101781, CCF-0747390, CCF-1018729, and CCF-1018730, and by a DoD NDSEG Graduate Fellowship. The last author is supported in part by a Sloan Foundation Fellowship. Additional support comes from Par Lab affiliates National Instruments, Nokia, NVIDIA, Oracle, and Samsung.

References

[1] R. Agarwal, A. Sasturkar, L. Wang, and S. D. Stoller. Optimized run-time race detection and atomicity checking using partial discovered types. In *20th IEEE/ACM International Conference on Automated software engineering (ASE)*, pages 233–242. ACM, 2005.

[2] G. Altekar and I. Stoica. ODR: output-deterministic replay for multicore debugging. In *ACM SIGOPS 22nd symposium on Operating systems principles (SOSP)*, pages 193–206. ACM, 2009.

[3] T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: localizing errors in counterexample traces. In *30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pages 97–105. ACM, 2003.

[4] S. Bensalem and K. Havelund. Scalable dynamic deadlock analysis of multi-threaded programs. In *Parallel and Distributed Systems: Testing and Debugging 2005 (PADTAD'05)*, 2005.

[5] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 167–178, New York, NY, USA, 2010. ACM.

[6] J. D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Programming language design and implementation (PLDI)*, pages 258–269, 2002.

[7] J.-D. Choi and H. Srinivasan. Deterministic replay of java multithreaded applications. In *In Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 48–59, 1998.

[8] A. Dinning and E. Schonberg. Detecting access anomalies in programs with critical sections. In *Proc. of the ACM/ONR Workshop on Parallel and Distributed Debugging*, 1991.

[9] M. B. Dwyer, J. Hatcliff, Robby, and V. P. Ranganath. Exploiting object escape and locking information in partial-order reductions for concurrent object-oriented programs. *Form. Methods Syst. Des.*, 25(2–3):199–240, 2004.

[10] A. Farzan and P. Madhusudan. The complexity of predicting atomicity violations. In *15th International Conference Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2009)*, 2009.

[11] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *31st Symposium on Principles of Programming Languages (POPL)*, pages 256–267, 2004.

[12] D. Giffhorn and C. Hammer. Precise slicing of concurrent programs. *Automated Software Engg.*, 16(2):197–234, 2009.

[13] A. Groce, S. Chaki, D. Kroening, and O. Strichman. Error explanation with distance metrics. *Int. J. Softw. Tools Technol. Transf.*, 8(3):229–247, 2006.

[14] A. Groce and W. Visser. What went wrong: Explaining counterexamples. In *10th International SPIN Workshop on Model Checking of Software*, pages 121–135, 2003.

[15] V. Jagannath, M. Gligoric, D. Jin, Q. Luo, G. Rosu, and D. Marinov. Improved multithreaded unit testing. In *8th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2011.

[16] Java Grande Forum. `http://www.javagrande.org/`.

[17] P. Joshi, M. Naik, C.-S. Park, and K. Sen. CalFuzzer: An extensible active testing framework for concurrent programs. In *CAV '09: Proceedings of the 21st International Conference on Computer Aided Verification*, pages 675–681, Berlin, Heidelberg, 2009. Springer-Verlag.

[18] P. Joshi, C.-S. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09)*, 2009.

[19] J. Krinke. Context-sensitive slicing of concurrent programs. In *9th European software engineering conference/11th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE)*, pages 178–187. ACM, 2003.

[20] Z. Lai, S. C. Cheung, and W. K. Chan. Detecting atomic-set serializability violations in mulithreaded programs through active randomized testing. In *32nd International Conference on Software Engineering (ICSE)*. ACM/IEEE, 2010.

[21] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. Comput.*, 36(4):471–482, 1987.

[22] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 15–26. ACM, 2005.

[23] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 329–339, New York, NY, USA, 2008. ACM.

[24] P. Montesinos, L. Ceze, and J. Torrellas. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *35th International Symposium on Computer Architecture (ISCA)*, pages 289–300. IEEE, 2008.

[25] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 446–455. ACM, 2007.

[26] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *8th USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 267–280. USENIX Association, 2008.

[27] M. G. Nanda and S. Ramesh. Interprocedural slicing of multithreaded programs with applications to java. *ACM Trans. Program. Lang. Syst.*, 28(6):1088–1144, 2006.

[28] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *32nd annual international symposium on Computer Architecture (ISCA)*, pages 284–295. IEEE, 2005.

[29] R. H. B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *ACM/ONR workshop on Parallel and distributed debugging (PADD)*, pages 1–11. ACM, 1993.

[30] Y. Nir-Buchbinder, R. Tzoref, and S. Ur. Deadlocks: From exhibiting to healing. In *8th Workshop on Runtime Verification*, 2008.

[31] C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *16th International Symposium on Foundations of Software Engineering (FSE'08)*. ACM, 2008.

[32] S. Park, S. Lu, and Y. Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 25–36. ACM, 2009.

[33] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: probabilistic replay with execution sketching on multiprocessors. In *22nd symposium on Operating systems principles (SOSP)*, pages 177–192. ACM, 2009.

[34] M. Ronsse, M. Christiaens, and K. D. Bosschere. Cyclic debugging using execution replay. In *ICCS '01: Proceedings of the International Conference on Computational Science-Part II*, pages 851–860, London, UK, 2001. Springer-Verlag.

[35] M. Ronsse and K. De Bosschere. RecPlay: a fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 17(2):133–152, 1999.

[36] M. Russinovich and B. Cogswell. Replay for concurrent non-deterministic shared-memory applications. In *ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 258–266. ACM, 1996.

[37] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[38] E. Schonberg. On-the-fly detection of access anomalies. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, PLDI '89, pages 285–297, New York, NY, USA, 1989. ACM.

[39] K. Sen. Race directed random testing of concurrent programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, 2008.

[40] F. Sorrentino, A. Farzan, and P. Madhusudan. Penelope: weaving threads to expose atomicity violations. In *Eighteenth ACM SIGSOFT international symposium on Foundations of software engineering (FSE)*, pages 37–46, New York, NY, USA, 2010. ACM.

[41] A. Thakur, R. Sen, B. Liblit, and S. Lu. Cooperative crug isolation. In *7th International Workshop on Dynamic Analysis (WODA)*, pages 35–41, 2009.

[42] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *15th International Conference on Automated Software Engineering (ASE)*. IEEE, 2000.

[43] M. Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.

[44] M. Xu, R. Bodik, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *30th annual international symposium on Computer architecture (ISCA)*, pages 122–135. ACM, 2003.

[45] W. Zhang, C. Sun, and S. Lu. ConMem: detecting severe concurrency bugs through an effect-oriented approach. In *ASPLOS '10: Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, pages 179–192, New York, NY, USA, 2010. ACM.

[46] X. Zhang, R. Gupta, and Y. Zhang. Efficient forward computation of dynamic slices using reduced ordered binary decision diagrams. In *26th International Conference on Software Engineering (ICSE)*, pages 502–511. IEEE, 2004.