Isolating Failure-Inducing Thread Schedules

Jong-Deok Choi
IBM T. J. Watson Research Center
P. O. Box 704
Yorktown Heights, NY 10598, USA
jdchoi@us.ibm.com

Andreas Zeller
Universität des Saarlandes
Lehrstuhl für Softwaretechnik
Postfach 15 11 50
66041 Saarbrücken, Germany
zeller@acm.org

ABSTRACT

Consider a multi-threaded application that occasionally fails due to non-determinism. Using the DEJAVU capture/replay tool, it is possible to record the thread schedule and replay the application in a deterministic way. By systematically narrowing down the difference between a thread schedule that makes the program pass and another schedule that makes the program fail, the Delta Debugging approach can pinpoint the error location automatically—namely, the location(s) where a thread switch causes the program to fail. In a case study, Delta Debugging isolated the failure-inducing schedule difference from 3.8 billion differences in only 50 tests.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*debugging aids, diagnostics, testing tools, tracing*; D.1.3 [**Programming Techniques**]: Concurrent Programming

1. INTRODUCTION

The increasing popularity of the Java programming language has made parallel programming more popular than ever. Unfortunately, concurrent programs are notoriously difficult to debug. Both the reproduction of failures and the subsequent isolation of errors impose additional challenges when applied to concurrent programs:

How do I reproduce a failure? Consider a concurrent application being run on some fixed input several times. Despite the input being constant, the application may fail occasionally. The reason is *non-determinism:* the *thread schedule* (or thread execution order) can vary from run to run. While having non-determinism is convenient for parallel programming, non-determinism makes it hard to reproduce a failure.

How do I isolate the error? Even if we can reproduce a failing program run deterministically, we still do not know the *cause of the failure*: How come one thread schedule makes the program fail, and another one makes the program pass? A thread schedule may be composed of 10,000 thread switches or more, yet only few of these switches may induce the specific interaction between threads that make the program fail.

In this paper, we present a novel approach that brings significant advances in addressing these problems. Our approach uses four automated building blocks, illustrated in Figure 1:

Deterministic replay. The DEJAVU tool [2] captures the execution of non-deterministic Java applications and allows the programmer to *replay* these executions deterministically—that is, input and thread schedules are reconstructed from the recorded execution. This effectively solves the problem of reproducing failures deterministically.

Test case generation. One of DEJAVU's features is that it allows the application to be executed under a given thread schedule. We use this to generate *alternate schedules:* For instance, we can alter an original passing (or failing) schedule until an alternate failing (passing) schedule is found.

Isolating failure causes. We use *Delta Debugging* [21] to automatically isolate the failure cause in a failure-inducing thread schedule. The basic idea is to systematically *narrow the difference* between the passing and the failing thread schedule, until only a minimal difference remains—a difference such as "The failure occurs if and only if thread switch #3291 occurs at clock time 47,539." This effectively solves the isolation problem.

Relating causes to errors. Each of the resulting thread differences occurs at a specific location of the program—for instance, thread switch #3291 may occur at line 20 of *foo.java*—giving a good starting point for locating thread interferences. In case understanding of the remaining behavior is required, DEJAVU can be used to replay the resulting thread schedules.

Altogether, these building blocks widely automate the testing and debugging process; at the same time, our approach is *purely experimental*, meaning that no knowledge of the program text is required. We estimate that our combined approach will considerably ease the debugging of multi-threaded applications.

This paper is organized as follows: We start with a motivating example in Section 2. Section 3 presents how DEJAVU captures and replays thread schedules. In Section 4, we show how to isolate failure-inducing thread schedules with Delta Debugging. In Section 5, we discuss the generation of alternate thread schedules. Section 6 presents the results of a case study, using a real-life Java program. Section 7 discusses related work; Section 8 closes with the conclusion and future work.

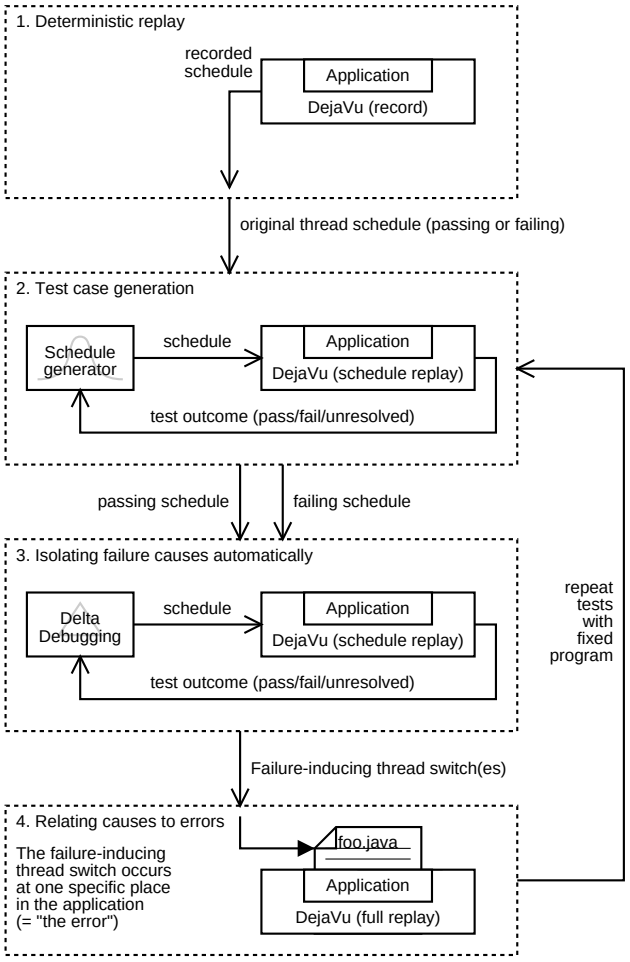

Figure 1: The testing and debugging process, as proposed in this paper. Using DEJAVU, we record a thread schedule of a program run (1). Starting from this schedule (either passing or failing), we randomly generate alternate thread schedules and execute them deterministically using DEJAVU (2). When we have found both a passing and a failing schedule, Delta Debugging isolates the failure-inducing difference (3). This difference occurs at a specific location in the program, which is typically the place to be fixed by the programmer (using DEJAVU to fully replay runs if needed) (4). After the fix, the tests are re-run.

2. A SIMPLE EXAMPLE

As a simple motivating example, consider the shared queue program *IntQueue.java* in Figure 2 (adapted from [19]). The *IntQueue* class realizes a queue of integers. *enqueue(elem)* enqueues an integer number *elem* into the queue; *dequeue()* returns and dequeues the first element from the queue.

Internally, the queue is realized as an integer array *link* where the entry *link*[*elem*] is the successor of *elem*; the attributes *head* and *tail* hold the first and last element, respectively. A value of 0 indicates a non-existing element; if *head* is 0, the queue is empty.

To allow concurrent access by multiple threads, the programmer of *IntQueue* has encapsulated all accesses to *link* into *critical sections* marked with the *synchronized* keyword; this Java feature prevents a thread from entering the critical section while the section is executed by another thread. (To be more precise, *synchronized(object)*

```
1  class IntQueue {
2     // The queue holds integers in the range
3     // of [1..numberOfElements - 1]
4     static final int numberOfElements = 100;
5
6     // link[N] is N's successor in the queue
7     int link[] = new int[numberOfElements];
8
9     int head;  // First element of queue
10    int tail;  // Last element of queue
11
12    // Constructor
13    IntQueue() {
14        head = 0;
15        tail = 0;
16        for (int i = 0; i < numberOfElements;
17             i++) {
18            link[i] = 0;
19        }
20    }
23
24    // Enqueue ELEM.
25    public void enqueue(int elem) {
26        link[elem] = 0;
27
28        if (head == 0)
29            head = elem;
30        else {
31            synchronized (this) {
32                link[tail] = elem;
33            }
34        }
35
36        tail = elem;
37    }
38
39    // Return first element of queue.
40    // No error checking.
41    public int dequeue() {
42        int elem = head;
43        if (elem == tail)
44            tail = 0;
45
46        synchronized (this) {
47            head = link[head];
48        }
49
50        return elem;
51    }
52
53    // Print elements of queue
54    public void print() {
55        for (int e = head; e != 0; e = link[e])
56            System.out.print(e + " ");
57        System.out.println();
58    }
59 }
```

Figure 2: *IntQueue.java*—an erroneous shared queue. This class may exhibit failures if multiple threads access the *enqueue* and *dequeue* methods concurrently.

places a *lock* on *object*, preventing other threads to enter the critical section using the same object as the lock; the lock is released when the thread leaves the synchronized block.)

In most situations, the *IntQueue* class works fine. Figure 3 shows how the queue is accessed by three threads: First, thread *A* enqueues the number 11; later, thread *B* dequeues it; and later again, following *A*, thread *C* enqueues the number 95.

There are situations, though, where *IntQueue* fails—for instance,

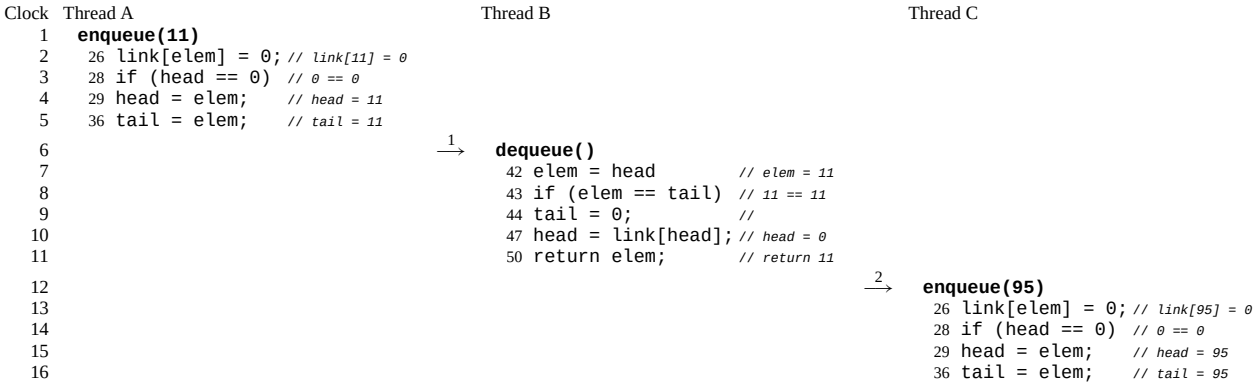

Figure 3: A passing thread schedule ⟨6, 12⟩ **(or** ⟨6, 12, 17, 17, 17⟩ **in the "padded" form)**

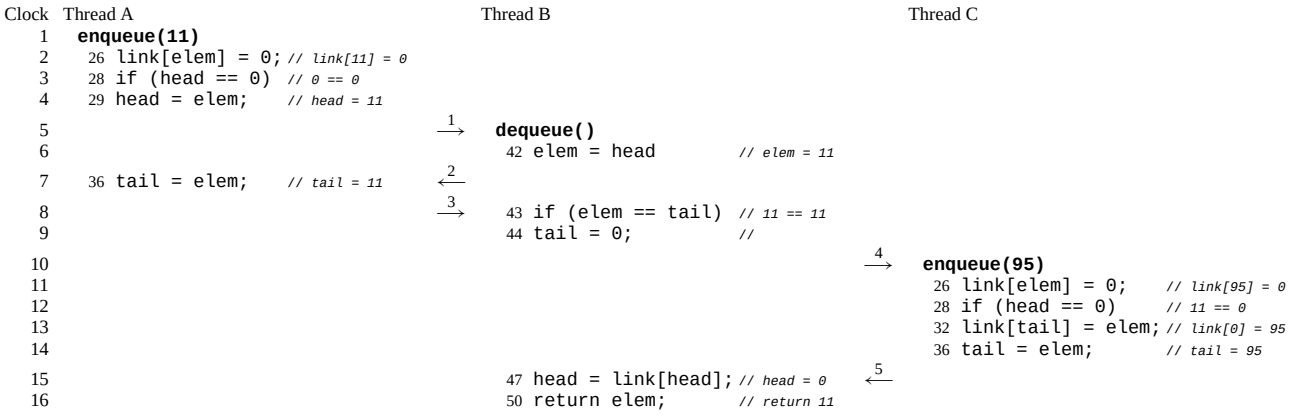

Figure 4: A failure-inducing thread schedule ⟨5, 7, 8, 10, 15⟩

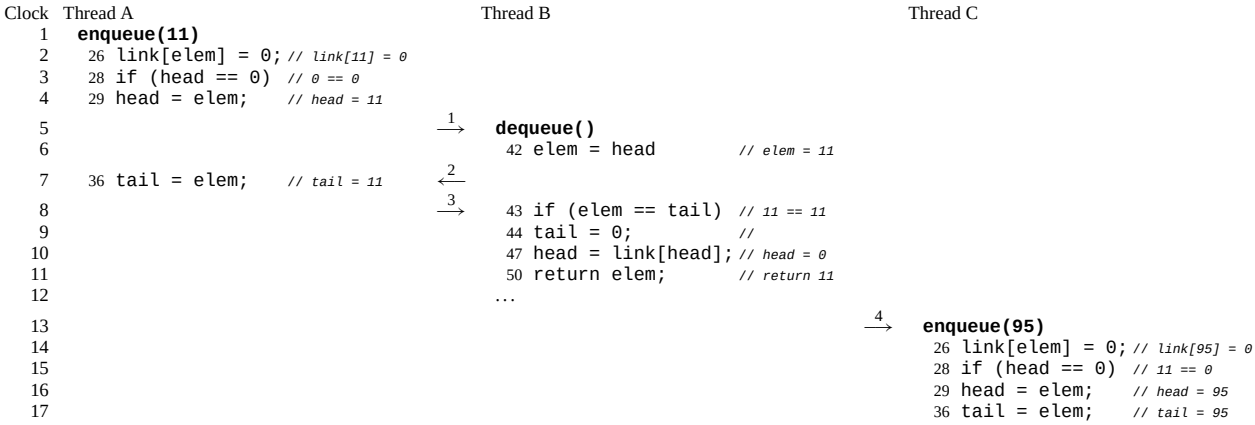

Figure 5: A generated passing schedule ⟨5, 7, 8, 13, 17⟩

the one in Figure 4. In this example, the *schedule* of the threads is different from Figure 3. In particular, *B*'s invocation of *dequeue* starts while *A*'s execution of *enqueue* has not finished yet; likewise, thread *C* starts its *enqueue* operation while *B*'s *dequeue* has not yet returned. At the end of this different schedule, we obtain a different result: As *head* is 0, the queue is empty—*C*'s enqueuing had no effect. Except for the different schedule, everything else is unchanged; as the schedule is non-deterministic, the program will show non-deterministic failures.

For a programmer, isolating the causes for non-deterministic failures in a concurrent program is among the least gratifying tasks. Hence, several approaches have been suggested that address the problem. *Static analysis* attempts to identify the statements that may happen in parallel or not [10, 11, 12, 13]; this is a prerequisite for detecting data races statically [18]. A wide range of methods has been proposed and evaluated for detecting deadlocks statically [4]. *Dynamic analysis* can detect shared memory accesses at run time [3, 8, 14, 16]. All these approaches require complete knowledge about the program to be analyzed.

In this paper, we promote a different approach, focusing on the *thread schedule* rather than on the program code. We look at the *difference* between a failure-inducing schedule (as the one in Figure 4) and another schedule where the failure does not occur (as the one in Figure 3). Our goal now is to relate the failure to a small set of *relevant differences*—differences that determine whether the failure occurs or not.

Why do we focus on differences? The schedule in Figure 5 is very much like the failing schedule in Figure 4. This schedule is successful, though—the element 95 is properly enqueued. This tells us that the first three thread switches at clock time 5, 7, and 8 are not relevant for producing the failure. The only remaining difference between the schedules in Figures 4 and 5 is whether thread *B* executes lines 47 before thread *C* takes over control or not. Thus, we can relate the failure to lines 47 in the *dequeue* method and their interference with *enqueue*—a certainly helpful hint within *IntQueue* and an even more helpful hint if *IntQueue* were part of a larger program.

To find such a relevant difference (and, among others, a schedule like the one in Figure 5), we use a *purely experimental* approach. That is, rather than reasoning about the program code, we run a series of *experiments* under *altered schedules* and test whether the failure still occurs or not. The advantages are that

- the program can be treated like a black box—it suffices that the program can be executed;
- the failure can be an arbitrary behaviour of the program; it suffices that one can distinguish failure from success.

The disadvantage is that our approach is *test-based* and hence inherits the disadvantages of tests when compared to static analysis—we can not determine properties for all runs of a program like the general absence of deadlocks. Like all tests, we require an observable failure. Should a failure occur, though, we can narrow down the cause automatically.

For this experimental approach, we need an infrastructure to *capture, replay, and alter* thread schedules of existing programs. Such an infrastructure exists, it is called DEJAVU and discussed in the following section.

3. CAPTURING AND REPLAYING THREAD SCHEDULES

DEJAVU[1] [2] is a tool for deterministic capture and replay of Java programs. It is part of Jalapeño [1], a research virtual machine for Java developed at the IBM T. J. Watson Research Center. The aim of DEJAVU is to make failures *reproducable:* Once a (non-deterministic) run is captured, it can be replayed deterministically again and again. DEJAVU can operate in three modes:

Record. When recording, DEJAVU executes a Java program, *recording* its input and its thread schedule to be replayed later.

Full Replay. When replaying, DEJAVU reads the original input and the thread schedule back again and executes the program in question such that the original (non-deterministic) behavior is restored deterministically.

[1] DEJAVU stands for Deterministic Java Replay Utility.

Schedule Replay. This mode is actually a mixture between record and full replay. In this mode, the program is executed and the input is recorded (as in record mode). The thread schedule, though, is *replayed* from a previously recorded run.[2]

To understand how DEJAVU captures and replays thread schedules, let us give a brief overview of Jalapeño's thread package. On a uniprocessor system, a *thread schedule* of a program is essentially a sequence of *time slices*. Each interval in this sequence contains execution events of a single thread. Consequently, interval boundaries correspond to points where *thread switches* occur and where control passes from on thread to another.

Three factors can cause thread switches in Jalapeño

1. timer interrupts (such as in *IntQueue.java*),
2. timed events (such as *sleep* and timed *wait*), and
3. synchronization events.

In Jalapeño, thread switches due to timer interrupts or timed events are *non-deterministic*, while thread switches due to synchronization events are deterministic. We will now discuss how DEJAVU records and replays these events.

3.1 Preemptive Thread Switching Due To Timer Interrupts

Jalapeño employs type-accurate garbage collectors to avoid memory leaks associated with conservative garbage collection and to allow copying garbage collection. This means that every reference to a live object must be identified during garbage collection. Identifying such references in the frames of a thread's activation stack is particularly problematic. Jalapeño *reference maps* specify these locations for predefined *safe points* in a method.

Definition 1 (Safe point) *A safe point in Jalapeño is a program location where the compiler that created the method body is able to describe where the live references exist.*

At garbage-collection time, Jalapeño guarantees that every method executing on every mutator thread is stopped at one of these safe points so that the garbage collector can have precise information on the references to live objects.

To make good on this guarantee, Jalapeño's own thread package performs quasi-preemptive thread switching only when the current running thread is at a predetermined *yield point.*

Definition 2 (Yield point) *A yield point in Jalapeño is a safe point located at a method prologue (i.e. a function invocation) or at a loop back-edge.*

To achieve some measure of fairness among Java threads, they are preempted at the first yield point after a periodic timer interrupt.

[2] The fourth possible mode, recording the thread schedule but replaying the input is infeasible as the non-deterministic thread schedule might alter the way the input is accessed, thus not matching the replayed input.

3.2 Replaying Preemptive Thread Switches

Since preemptive thread switches occur only at yield points, counting the number of yield points executed since the start of the execution can uniquely identify each thread switching event. DEJAVU uses this cumulative number of yield points executed at each preemptive thread switch as the global clock value of the thread switch.

- During *record mode*, DEJAVU captures and stores the global clock values of preemptive thread switches.[3]
- During *full replay mode* and *schedule replay mode*, DEJAVU reads a global clock from the recorded schedule, sets an internal counter to the read value, decrements the counter at every yield point, then forces a thread switch when the counter reaches zero. DEJAVU repeats this for all recorded global clock values until the program terminates.

3.3 Replaying Other Thread Switches

To ensure deterministic threading behavior during replay, DEJAVU records the wall-clock values read during execution in record mode and replays them during replay. In fact, reproducing wall-clock values is a special case of replaying non-deterministic events such as reading values from a random-number generator. DEJAVU captures these non-deterministic input values during record mode and reuses them during replay. Therefore, *timed thread events* that depend on wall-clock values, such as *sleep* and timed *wait*s, will execute deterministically, and will reproduce the recorded behavior.

In full replay mode, one must also take care to replay *synchronization events*. When DEJAVU fully replays an application up to a synchronization operation (say, *monitorenter*), it replays the entire program state of the Jalapeño JVM as well, including its thread package, which maintains the *lock state* of each thread and lock variable plus the dispatch queue of threads. Therefore, the synchronization operation will succeed or fail during replay mode depending on whether it succeeded or failed during record mode. If it fails, moreover, the next thread to be dispatched during replay mode (as determined by the thread package) will be the same thread dispatched during record mode. This is because the data structure used by the thread package in selecting the next active thread will also be exactly reproduced by DEJAVU—that is, the non-deterministic synchronization event will be faithfully reproduced.

3.4 Replaying Generated Thread Schedules

The simple replay mechanism of DEJAVU based on the global clock values also offers a simple mechanism for *alternate thread switching*. One can simply generate a sequence of global clock values for an alternate thread schedule. In schedule replay mode, DEJAVU then uses these values in deciding when to force a thread switch, as it does during full replay mode.

Obviously, a deterministic behavior of a program in schedule replay mode is desirable—that is, for a given schedule, a program should show the same behavior in every execution. It turns out that both factors for thread switches are deterministic during schedule replay mode:

1. *Preemptive thread switches* are determined by the given thread schedule.
2. *Synchronization events* are deterministic in Jalapeño.

There may be other sources for non-determinism; a program may read random values from external devices, for instance. Such sources will be *recorded* in schedule replay mode and be replayed deterministically in full replay mode. If the input in schedule replay mode is fixed, though (that is, the program can be automatically tested), the program behavior depends uniquely on the given thread schedule.

This is how DEJAVU becomes a foundation for our approach: We can use DEJAVU as *testbed* to determine how different schedules affect the outcome of the program.

4. NARROWING SCHEDULE DIFFERENCES

In this section, we show how to isolate failure-inducing schedule differences. We start with some formal notation for thread schedules and differences, abstracting somewhat from DEJAVU.

4.1 Identifying Thread Schedules

Let us start with a denotation of thread schedules. Within this paper, we are only interested in non-deterministic thread switches caused by timer interrupts. (As discussed in Section 3.4, both timed events and synchronization events are deterministic in DEJAVU.) Hence, we need not care about identifying specific threads; we denote thread schedules simply by the clock times where thread switches occur.

Definition 3 (Thread Schedule) *Let $\mathcal{T}$ be the set of all thread schedules. A* thread schedule *$T \in \mathcal{T}$ with $T = \langle t_1, t_2, \ldots, t_n \rangle$ is a list of n clock times $t_1, \ldots, t_n$. Each t_i is the clock time where a non-deterministic thread switch caused by a timer interrupt occurs.*

As an example, consider Figure 3. Here, we assume a simple logical clock counting executed statements. As thread switches occur at clock times 6 and 12, the thread schedule in Figure 3 can thus be denoted as $T = \langle 6, 12 \rangle$.

For convenience, we want schedules to be ordered (or *valid*):

Definition 4 (Valid Schedule) *A thread schedule $T = \langle t_1, \ldots, t_n \rangle$ is called* valid *if $t_i \leq t_{i+1}$ holds for all $1 \leq i \leq n$.*

4.2 Testing Thread Schedules

We assume a program run is uniquely determined by the specific thread schedule—that is, all other circumstances stay unchanged (and are being faithfully replayed by DEJAVU or a similar tool). Consequently, we can distinguish the outcome of a program run depending only on the schedule. According to the POSIX 1003.3 standard for testing frameworks [9], we distinguish three outcomes:

- The test *succeeds* (PASS, written here as ✔)
- The test has *produced the failure* it was intended to capture (FAIL, written here as ✘)
- The test produced *indeterminate results* (UNRESOLVED, written here as **?**).[4]

[3] For optimization purposes, only the *difference* between two consecutive clock values is actually stored.

[4] POSIX 1003.3 also lists UNTESTED and UNSUPPORTED outcomes, which are of no relevance here.

We assume the existence of an (automated) *testing function*:

Definition 5 (*stest*) *The function* $stest : \mathcal{T} \to \{\text{✘}, \text{✔}, \text{?}\}$ *determines for a thread schedule* $T \in \mathcal{T}$ *whether some specific failure occurs (✘) or not (✔) or whether the test is unresolved (?).*

In case of the *IntQueue* class, we would for instance define *stest* to return ✔ if the queue holds the value 95; to return ✘ if the queue is empty, and to return **?** in all other cases.

Let us now assume that for some program, we have a passing run, determined by a schedule $T_{\text{✔}}$, and a failing run, determined by a schedule $T_{\text{✘}}$. (In the *IntQueue* example, $T_{\text{✔}} = \langle 6, 12\rangle$ and $T_{\text{✘}} = \langle 5, 7, 8, 10, 15\rangle$ hold.) The notions of "passing" and "failing" run are determined by the test outcome:

Axiom 6 (Passing and Failing Runs) $stest(T_{\text{✔}}) = \text{✔}$ *and* $stest(T_{\text{✘}}) = \text{✘}$ *hold.*

In the *IntQueue* example, Axiom 6 holds as demonstrated in Figures 3 and 4.

Axiom 7 (Invalid Schedule) *If* T *is an invalid schedule,* $stest(T) = \text{?}$ *holds.*

4.3 Identifying Differences

Let us now turn to the *difference* between two schedules—the difference we eventually want to narrow. Formally, a difference is a mapping δ that can be applied to one schedule (in our case, $T_{\text{✔}}$) to obtain the other schedule ($T_{\text{✘}}$):

Definition 8 (Schedule Difference) *A* schedule difference *between two schedules* $T_{\text{✔}}$ *and* $T_{\text{✘}}$ *is a mapping* $\delta : \mathcal{T} \to \mathcal{T}$ *with* $\delta(T_{\text{✔}}) = T_{\text{✘}}$*. The set of all differences is denoted as* $\mathcal{C} = \mathcal{T}^{\mathcal{T}}$*.*

What is δ made of? In this paper, we assume a simple decomposition. First, we decompose δ into a number of *thread switch changes* δ_i, each representing the difference between the i-th thread switch of $T_{\text{✔}}$ and $T_{\text{✘}}$. For convenience, we assume that both schedules have the same length; this can be achieved by *padding* schedules with "dummy" thread switches that would occur after the execution of the program in question ended.

In our example, we end up with the "padded" schedule $T_{\text{✔}} = \langle 6, 12, 17, 17, 17\rangle$ (assuming execution ends at clock time 16) and the original schedule $T_{\text{✘}} = \langle 5, 7, 8, 10, 15\rangle$. The difference δ is $\delta = \delta_1 \circ \cdots \circ \delta_5$. Applying δ_i changes $t_{\text{✔}i}$ to $t_{\text{✘}i}$; for example, $\delta_2(T_{\text{✔}}) = \langle 6, 7, 17, 17, 17\rangle$ holds.

Definition 9 (Difference Decomposition) *A schedule difference* δ *between two schedules* $T_{\text{✔}} = \langle t_{\text{✔}1}, \ldots, t_{\text{✔}n}\rangle$ *and* $T_{\text{✘}} = \langle t_{\text{✘}1}, \ldots, t_{\text{✘}n}\rangle$ *is defined as* $\delta = \delta_1 \circ \delta_2 \circ \cdots \circ \delta_n$ *where*

- *each* $\delta_i : \mathcal{T} \to \mathcal{T}$ *maps* $t_{\text{✔}i}$ *to* $t_{\text{✘}i}$*; that is,* $\delta_i(T_{\text{✔}}) = \langle t_{\text{✔}1}, \ldots, t_{\text{✔}i-1}, t_{\text{✘}i}, t_{\text{✔}i+1}, \ldots t_{\text{✔}n}$
- *the composition* $\circ : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ *is defined as* $(\delta_i \circ \delta_j)(T) = \delta_i\big(\delta_j(T)\big)$.

To get precise results, we want the differences to be as small as possible. Hence, we decompose each δ_i into a number of *atomic changes* $\delta_{i,1}, \delta_{i,2}, \ldots$, each narrowing the difference between the i-th thread switch of $T_{\text{✔}}$ and $T_{\text{✘}}$ by one clock time unit (or, in other words, *moving* $t_{\text{✔}i}$ one clock unit towards $t_{\text{✘}i}$).

In our example, as $t_{\text{✔}2} = 12$ and $t_{\text{✘}2} = 7$, δ_2 is composed of $|t_{\text{✔}2} - t_{\text{✘}2}| = 5$ atomic changes $\delta_2 = \delta_{2,1} \circ \cdots \circ \delta_{2,5}$. Applying any $\delta_{2,j}$ to $T_{\text{✔}}$ decreases $t_{\text{✔}2}$ by one: for example, $\delta_{2,1}(T_{\text{✔}}) = \langle 6, 11, 17, 17, 17\rangle$ holds.

Definition 10 (Atomic Decomposition) *The differences* δ_i *in Definition 9 can be decomposed further into atomic differences*

$$\delta_i = \delta_{i,1} \circ \delta_{i,2} \circ \cdots \circ \delta_{i,|t_{\text{✔}i} - t_{\text{✘}i}|}$$

where each $\delta_{i,j}$ *is defined as*

$$\begin{aligned}\delta_{i,j}(T_{\text{✔}}) &= \delta_{i,j}\big(\langle t_{\text{✔}1}, t_{\text{✔}2}, \ldots, t_{\text{✔}n}\rangle\big)\\ &= \langle t_{\text{✔}1}, t_{\text{✔}2}, \ldots, t_{\text{✔}i-1}, t'_{\text{✔}i}, t_{\text{✔}i+1}, \ldots, t_{\text{✔}n}\rangle\end{aligned}$$

where $t'_{\text{✔}i}$ *is the value altered by* $\delta_{i,j}$*; that is,*

$$t'_{\text{✔}i} = \begin{cases} t_{\text{✔}i} + 1 & \text{if } t_{\text{✔}i} < t_{\text{✘}i}\\ t_{\text{✔}i} - 1 & \text{if } t_{\text{✔}i} > t_{\text{✘}i}\end{cases}$$

To round things up, let us prove the decomposition actually works:

Corollary 11 (δ maps) *Given two schedules* $T_{\text{✔}}$ *and* $T_{\text{✘}}$ *of length* n *and a thread difference* $\delta = \delta_1 \circ \ldots \delta_n$ *with* δ_i *and* $\delta_{i,j}$ *defined according to Definitions 8, 9, and 10, then* $\delta(T_{\text{✔}}) = T_{\text{✘}}$ *holds.*

PROOF. *Each* $\delta_{i,j}$*, as defined in Definition 10, decreases the difference between* $t_{\text{✔}i}$ *and* $t_{\text{✘}i}$ *by one. Each* δ_i *consists of* $|t_{\text{✔}i} - t_{\text{✘}i}|$ *elements* $\delta_{i,j}$ *(Definition 9). Consequently, each* δ_i *makes* $t_{\text{✔}i}$ *equal to* $t_{\text{✘}i}$*, and thus* δ *maps* $T_{\text{✔}}$ *to* $T_{\text{✘}}$*.* □

The number of atomic deltas can quickly become very large; in fact, the number is quadratic in proportion to the length of the schedules to be compared.

Corollary 12 (Number of Atomic Deltas) *For two thread schedules* $T_{\text{✔}}$ *and* $T_{\text{✘}}$ *of length* n*, the number of atomic differences* $\delta_{i,j}$ *is* $\sum_{i=1}^{n} |t_{\text{✔}i} - t_{\text{✘}i}|$.

Between the schedule $T_{\text{✔}}$ in Figure 3 and the schedule $T_{\text{✘}}$ in Figure 4, there are $|6-5|+|12-7|+|17-8|+|17-10|+|17-15| = 1+5+9+7+2 = 24$ atomic differences, each moving one thread switch by one clock time closer to the other.

4.4 Testing Differences

Having established a notation for schedule differences, let us now define a function *test* that *applies* a number of differences to the passing schedule and tests the program in question under the altered schedule. For convenience, we define sets of atomic differences:

Definition 13 ($c_{\text{✔}}$, $c_{\text{✘}}$) *Let* $T_{\text{✔}}$ *and* $T_{\text{✘}}$ *be given, valid thread schedules; let* δ *be their difference as described. The set* $c_{\text{✘}}$ *is defined as the set of atomic differences in* δ*; the set* $c_{\text{✔}}$ *is defined as* $c_{\text{✔}} = \emptyset$*.*

Tests	$\delta_{1,1}$	$\delta_{2,1}\delta_{2,2}\delta_{2,3}\delta_{2,4}\delta_{2,5}$	$\delta_{3,1}\delta_{3,2}\delta_{3,3}\delta_{3,4}\delta_{3,5}\delta_{3,6}\delta_{3,7}\delta_{3,8}\delta_{3,9}$	$\delta_{4,1}\delta_{4,2}\delta_{4,3}\delta_{4,4}\delta_{4,5}\delta_{4,6}\delta_{4,7}$	$\delta_{5,1}\delta_{5,2}$	Schedule	Outcome
$T_{✔}$	·	· · · · ·	· · · · · · · · ·	· · · · · · ·	· ·	⟨6, 12, 17, 17, 17⟩	✔
$T_{✘}$	□	□ □ □ □ □	□ □ □ □ □ □ □ □ □	□ □ □ □ □ □ □	□ □	⟨5, 7, 8, 10, 15⟩	✘
(1)	□	□ □ □ □ □	□ □ □ □ □ □ □ □ □	· · · · · · ·	· ·	⟨5, 7, 8, 17, 17⟩	✔
(2)	□	□ □ □ □ □	□ □ □ □ □ □ □ □ □	□ □ □ □ □ □ □	· ·	⟨5, 7, 8, 10, 17⟩	✘
(3)	□	□ □ □ □ □	□ □ □ □ □ □ □ □ □	□ □ □ □ · · ·	· ·	⟨5, 7, 8, 13, 17⟩	✔
(4)	□	□ □ □ □ □	□ □ □ □ □ □ □ □ □	□ □ □ □ □ □ ·	· ·	⟨5, 7, 8, 11, 17⟩	✔
Result				□			

Figure 6: How Delta Debugging isolates a failure-inducing thread switch. Delta Debugging gradually narrows the difference between $T_{✔}$ (Figure 3) and $T_{✘}$ (Figure 4) until only one difference remains: Thread switch #4 at clock time 10 (instead of 11) causes the failure.

We can now define a *test function test* that determines the outcome for a given set of differences. This means that *test* must run the program under the given *generated* schedule.

Definition 14 (*test*) *The function* $test : 2^{c_{✘}} \to \{✘, ✔, ?\}$ *is defined as follows: Let* $c \subseteq c_{✘}$ *be a test case with* $c = \{\delta_{1,1}, \delta_{2,1}, \ldots, \delta_{n,m_n}\}$. *Then,* $test(c) = stest\big((\delta_{1,1} \circ \delta_{2,1} \circ \cdots \circ \delta_{n,m_n})(T_{✔})\big)$ *holds.*

Using Axiom 6, we can deduce the outcomes of $test(c_{✔})$ and $test(c_{✘})$:

Corollary 15 (Passing and failing test case) *The following holds:*

$$test(c_{✔}) = test(\emptyset) = stest(T_{✔}) = ✔$$
$$test(c_{✘}) = test\big(\{\delta_{1,1}, \delta_{2,1}, \ldots, \delta_{n,m_n}\}\big) = stest\big(\delta(T_{✔})\big) = ✘$$

4.5 Isolating Relevant Differences

Our next step now is to isolate a *minimal* set of differences that is relevant to produce the error. Unfortunately, this comes at a price: Relying on *test* alone, isolating a minimal set of differences is an NP-complete problem. The reason is simple: In the worst case, each subset of $c_{✘}$ must be tested, and $c_{✘}$ has $2^{|c_{✘}|}$ subsets.

In practice, though, we are already happy with an *approximation:* What we want is a set of atomic differences where each single remaining difference is *relevant* for the failure—that is, it cannot be removed without making the failure disappear. We call this property *1-minimality,* defined as

Definition 16 (1-minimal difference) *Let* $c'_{✔}$ *and* $c'_{✘}$ *be two sets of differences. Their difference* $\Delta = c'_{✘} - c'_{✔}$ *is* 1-minimal *if*

$$\forall \delta_i \in \Delta \cdot test\big(c'_{✔} \cup \{\delta_i\}\big) \neq ✔ \wedge test\big(c'_{✘} - \{\delta_i\}\big) \neq ✘$$

holds.[5]

To determine the sets $c'_{✔}$ and $c'_{✘}$ as well as their 1-minimal difference, we use the *Delta Debugging* approach. Delta Debugging [21] is a technique that automatically isolates failure-inducing circumstances; its main application is to simplify failure-inducing *program input.* The basic idea of Delta Debugging is to systematically narrow the difference between a passing and a failing program run, using test outcomes to direct the narrowing process.

Let us illustrate the use of Delta Debugging by applying it to our well-known example, as shown in Figure 6. At the top, we see the 24 atomic differences between $T_{✔}$ and $T_{✘}$. The first and second line shows the initial tests $T_{✔}$ (no difference applied, "·") and $T_{✘}$ (all differences applied, "□"), the resulting schedules and the test outcome. Now, Delta Debugging starts.

(1) The Delta Debugging algorithm splits the initial difference $\Delta = c_{✘} = \{\delta_{1,1}, \ldots, \delta_{5,2}\}$ into two subsets $\Delta_1 \cup \Delta_2 = \Delta$ with $\Delta_1 = \{\delta_{1,1}, \ldots, \delta_{3,9}\}$ and $\Delta_2 = \{\delta_{4,1}, \ldots, \delta_{5,2}\}$. First, Δ_1 is tested. The resulting schedule is ⟨5, 7, 8, 17, 17⟩. With this schedule, threads *B* and *C* do not interfere. The test passes ($test(\Delta_1) = ✔$), so we have narrowed down the failure-inducing difference to thread switch #4 and #5.

(2) The remaining set of differences is again split into two halves. In this example, we assume an "intelligent" splitting that splits differences according to the thread switches they are applied upon. It turns out that applying the differences for thread switch #4 alone causes the failure; whether thread switch #5 occurs immediately after thread *C* has finished enqueuing or later makes no difference.

(3) The remaining failure-inducing difference is now whether thread switch #4 occurs at clock time 10 or 17. Again, Delta Debugging splits the set of differences in two; making thread switch #4 occur at clock time 13 makes the program pass the test. This schedule is the one shown in Figure 5.

(4) Finally, the remaining difference is again split in two—and the final passing test has reduced the difference to a minimum. The failure is determined by whether thread switch #4 occurs at clock time 10 (failure) or clock time 11 (success). Looking up the involved code pinpoints the error: *C* begins enqueuing before *B* has finished updating *head*.

In this textbook example, Delta Debugging has required only 4 tests to isolate a minimal failure-inducing difference between a passing schedule and a failing schedule; in fact, Delta Debugging acted like a simple binary search. This may not necessarily be the case in all situations, as the following problems may occur:

Invalid schedules. In Figure 6, applying only $\delta_{5,1}$ and $\delta_{5,2}$ would result in the *invalid schedule* ⟨6, 12, 17, 17, 15⟩ and thus in a test outcome of **?**. Delta Debugging would then simply test the next alternative. Since an actual execution of the program is not required, such unresolved outcomes are cheap.

Other failures. A valid schedule may uncover another program behavior—neither the passing one from $T_{✔}$ nor the failing one from $T_{✘}$. Such outcomes can either be treated as failures (in case it does not matter which failure is induced by the difference to be found) or as unresolved outcomes (in which case Delta Debugging tries the next alternative).

[5] $A - B$ denotes the set difference between A and B.

Let $\mathcal{C}$ be the set of all possible circumstances (i.e. schedules). Let $test : 2^{\mathcal{C}} \to \{\text{✘}, \text{✔}, \text{?}\}$ be a testing function that determines for a test case $c \subseteq \mathcal{C}$ whether some given failure occurs (✘) or not (✔) or whether the test is unresolved (?).

Now, let $c_{\text{✔}}$ and $c_{\text{✘}}$ be test cases with $c_{\text{✔}} \subseteq c_{\text{✘}} \subseteq \mathcal{C}$ such that $test(c_{\text{✔}}) = \text{✔} \wedge test(c_{\text{✘}}) = \text{✘}$. $c_{\text{✔}}$ is the "passing" test case (typically, $c_{\text{✔}} = \emptyset$ holds) and $c_{\text{✘}}$ is the "failing" test case.

The *Delta Debugging algorithm* $dd(c_{\text{✔}}, c_{\text{✘}})$ isolates the failure-inducing difference between $c_{\text{✔}}$ and $c_{\text{✘}}$. It returns a pair $(c'_{\text{✔}}, c'_{\text{✘}}) = dd(c_{\text{✔}}, c_{\text{✘}})$ such that $c_{\text{✔}} \subseteq c'_{\text{✔}} \subseteq c'_{\text{✘}} \subseteq c_{\text{✘}}$, $test(c'_{\text{✔}}) = \text{✔}$, and $test(c'_{\text{✘}}) = \text{✘}$ hold and $c'_{\text{✘}} - c'_{\text{✔}}$ is *1-minimal*—that is, no single circumstance of $c'_{\text{✘}}$ can be removed from $c'_{\text{✘}}$ to make the failure disappear or added to $c'_{\text{✔}}$ to make the failure occur.

The *dd* algorithm is defined as $dd(c_{\text{✔}}, c_{\text{✘}}) = dd_2(c_{\text{✔}}, c_{\text{✘}}, 2)$ with

$$dd_2(c'_{\text{✔}}, c'_{\text{✘}}, n) = \begin{cases} dd_2(c'_{\text{✔}}, c'_{\text{✔}} \cup \Delta_i, 2) & \text{if } \exists i \in \{1, \ldots, n\} \cdot test(c'_{\text{✔}} \cup \Delta_i) = \text{✘} \\ dd_2(c'_{\text{✘}} - \Delta_i, c'_{\text{✘}}, 2) & \text{if } \exists i \in \{1, \ldots, n\} \cdot test(c'_{\text{✘}} - \Delta_i) = \text{✔} \\ dd_2\big(c'_{\text{✔}} \cup \Delta_i, c'_{\text{✘}}, \max(n-1, 2)\big) & \text{else if } \exists i \in \{1, \ldots, n\} \cdot test(c'_{\text{✔}} \cup \Delta_i) = \text{✔} \\ dd_2\big(c'_{\text{✔}}, c'_{\text{✘}} - \Delta_i, \max(n-1, 2)\big) & \text{else if } \exists i \in \{1, \ldots, n\} \cdot test(c'_{\text{✘}} - \Delta_i) = \text{✘} \\ dd_2\big(c'_{\text{✔}}, c'_{\text{✘}}, \min(2n, |\Delta|)\big) & \text{else if } n < |\Delta| \\ (c'_{\text{✔}}, c'_{\text{✘}}) & \text{otherwise} \end{cases}$$

where $\Delta = c'_{\text{✘}} - c'_{\text{✔}} = \Delta_1 \cup \Delta_2 \cup \cdots \cup \Delta_n$ with all Δ_i pairwise disjoint, and $\forall \Delta_i \cdot |\Delta_i| \approx (|\Delta| / n)$ holds.
The recursion invariant for dd_2 is $test(c'_{\text{✔}}) = \text{✔} \wedge test(c'_{\text{✘}}) = \text{✘} \wedge n \leq |\Delta|$.

Figure 7: The Delta Debugging algorithm in a nutshell. The function *dd* isolates the failure-inducing difference between two test cases $c_{\text{✔}}$ and $c_{\text{✘}}$. For a full description of the algorithm and its properties, see [21].

Multiple relevant thread switches. It may well be a failure is induced by applying *multiple schedule differences in conjunction* only and that applying a subset leads to unresolved test outcomes. Delta Debugging isolates this 1-minimal set of thread differences, but requires a larger number of tests.

In general, as a program is supposed to run under any given thread schedule, we expect very few unresolved test outcomes (and very few failure-inducing schedule differences), so the number of tests performed by Delta Debugging will typically be close to a binary search—that is, approximately $\log_2(n)$ tests for n atomic differences. Since n grows only quadratically with the length of the schedules, the number of tests will not grow without bounds. The formal definition of the Delta Debugging algorithm is shown in Figure 7; for a full discussion of the algorithm and its complexity, see [21].

Let us now put this approach into practice. We have the tool (DEJAVU), we have the method (Delta Debugging)—but we also need two schedules, a passing and a failing one. The next section discusses how to obtain these schedules.

5. GENERATING ALTERED SCHEDULES

Let us now assume we have a program test that fails. How do we get an *alternate thread schedule* that passes the test? Or vice versa: assume we have a program that passes. Can we try to obtain a schedule where the program fails?

One approach to obtain such alternate schedules could be to *generate random thread schedules*, replaying the program using DEJAVU with these schedules until an alternate outcome is found. However, we prefer an alternate schedule that is as *close as possible* to the original schedule, as this reduces the number of tests required to narrow down the failure-inducing difference.

Hence, we do not generate completely random schedules, but start from an *existing schedule* instead. Starting from a given schedule

$$T = \langle t_1, t_2, \ldots, t_n \rangle \ ,$$

we generate *fuzz schedules* of the form

$$T' = \langle f(t_1),\ f(t_2), \ldots,\ f(t_n) \rangle \ ,$$

where $f(t)$ is a *perturbation function* that randomly returns some time interval $t' = f(t)$ with $t' \in [0; \infty]$ with t being the most likely outcome—a simple *Gaussian distribution* centered around t, as depicted in Figure 8.

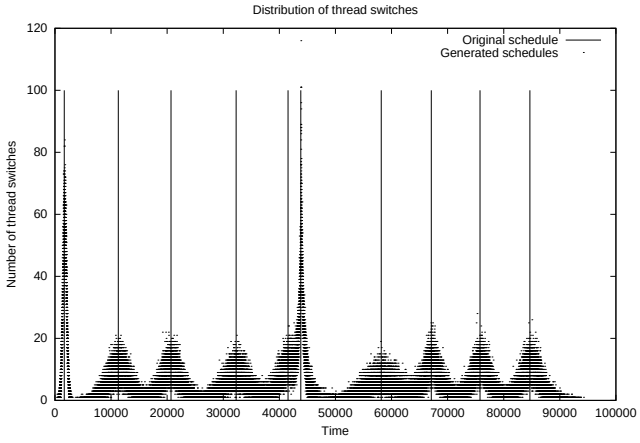

Figure 8: A sampling of 50,000 generated random schedules. Schedules are generated by moving thread switches, with smaller offsets being more likely than larger offsets.

We start with a very narrow distribution around the thread switches of the original schedule, and continually widen the distribution (and thus increase the differences to the original schedule) until an alternate outcome is found. Eventually, with a sufficient wide Gaussian distribution, we obtain completely random schedules.

```
25  public class Scene { ...
44      private static int ScenesLoaded = 0;
45      (more methods...)
81      private
82      int LoadScene(String filename) {
84          int OldScenesLoaded = ScenesLoaded;
85          (more initializations...)
91          infile = new DataInputStream(...);
92          (more code...)
130         ScenesLoaded = OldScenesLoaded + 1;
131         System.out.println("" +
                ScenesLoaded + " scenes loaded.");
132         ...
134     }
135     ...
733 }
```

Figure 9: Introducing a race condition in *205_raytrace*. The code in bold face, added to the original code, introduces a race condition on *ScenesLoaded*.

Unfortunately, the chances of obtaining an alternate schedule cannot be determined in advance. Confidence in a program increases, though, with the number of alternate schedules tested. As soon as an alternate schedule is found, we can pass it over to Delta Debugging to isolate the failure-inducing schedule difference.

6. A CASE STUDY

Let us now put all building blocks together and apply them on a real program. Test #205 of the SPEC JVM98 Java test suite [17], named *205_raytrace* is a multi-threaded ray-tracing program, processing a 3D-scene depicting a dinosaur. Being part of a test suite, *205_raytrace* has no known errors; a failure would typically indicate an error in the Java tool chain being tested.

In *205_raytrace*, the file *Scene.java* contains an interesting comment. Each ray-tracing thread calls the method *LoadScene* to be rendered once. This can lead to problems if shared data is accessed, which is why *LoadScene* is marked as *synchronized*. The comment says that the programmer attempted to change the code "so the MT [multi-threaded] version could have the data only read once, but this did not work." We simulated this failure by making *LoadScene* non-synchronized (removing the keyword) and introducing a simple observable race condition in *LoadScene*, as shown in Figure 9. Whenever a thread switch would occur during execution of *LoadScene*, causing the method to be called again, the *ScenesLoaded* variable would not be properly updated.

This code change leads to a failure the first time it is executed—the shared variable *ScenesLoaded* never increased to more than 1. Using DEJAVU, we recorded the failing thread schedule $T_{\text{✘}}$ (containing 3770 thread switches); DEJAVU was able to replay the failing schedule (and the failure) accurately.

Using the fuzz approach described in Section 5, we generated random schedules, starting from the failing one, until, after 66 tests, we had generated an alternate schedule $T_{\text{✓}}$ where the failure would not occur. Both $T_{\text{✓}}$ and $T_{\text{✘}}$ are shown in Figure 10—it turns out that $T_{\text{✓}}$ has a far higher granularity than $T_{\text{✘}}$, meaning that the amount of time between thread switches is larger.

Comparing $T_{\text{✓}}$ and $T_{\text{✘}}$ reveals that the average distance between a thread switch in $T_{\text{✓}}$ and the matching thread switch in $T_{\text{✘}}$ is more than a million yield points. Overall, 3,842,577,240 atomic deltas,

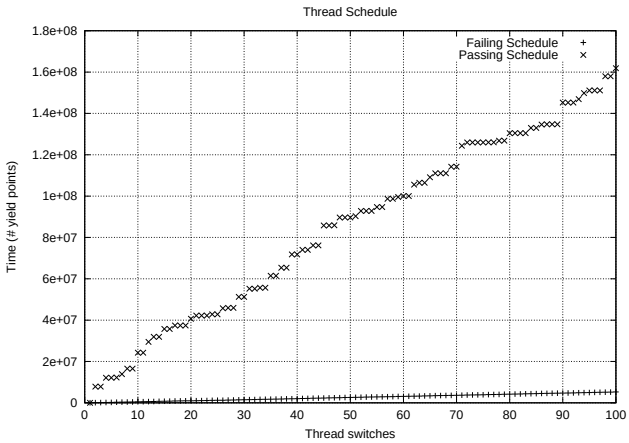

Figure 10: A passing and a failing schedule of the SPEC JVM98 ray-tracer program. This difference has to be minimized in order to isolate the failure cause.

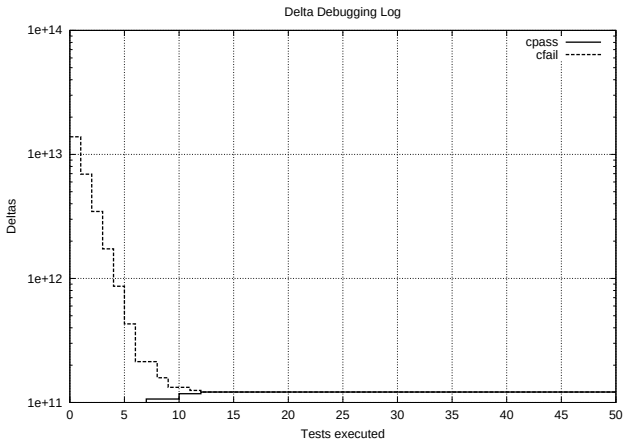

Figure 11: Narrowing down the failure-inducing thread switch. After only 50 tests, Delta Debugging isolates the single failure-inducing difference from 3,842,577,240 atomic differences.

each moving a thread switch by one yield point, have to be applied to turn $T_{\text{✓}}$ into $T_{\text{✘}}$. Some of these 3.8 billion schedule differences are relevant for the failure. So, as described in Section 4, we used Delta Debugging to narrow down the difference.

The Delta Debugging run is summarized in Figure 11. As in Figure 6, we grouped deltas according to their respective thread switch. After just 12 tests (or 408 seconds)[6], only one group of deltas remained, all applying to thread switch #33. Yet, this group still consisted of 53,976,462 deltas—that is, after 12 tests, the two schedules were still 53 million yield points apart. The later 38 tests subsequently halve this distance, such that eventually, after 50 tests (28 minutes), only one difference remains: The failure occurs if and only if thread switch #33 occurs at yield point 59,772,127 (instead of 59,772,126).

But which is the code that is executed at yield point 59,772,127? For this purpose, we extended DEJAVU by a "query mode", report-

[6] A single DEJAVU-controlled run of *205_raytrace* requires 34 seconds of real time on a powerpc-ibm-aix4.3.3.0 machine. Invalid schedules have been ignored.

ing the current backtrace for a given set of yield points. It turns out that yield point 59,772,127 occurs at the location *spec.benchmarks._205_raytrace.Scene.LoadScene (Scene.java:91)*, that is, at line 91 of *Scene.java*.

Line 91 of *Scene.java* is the first method invocation (and thus yield point) after the initialization of *OldScenesLoaded*. Likewise, the alternative yield point 59,772,126 (with a successful test outcome) is the invocation of *LoadScene* at line 82 of *Scene.java*—just *before* the variable *OldScenesLoaded* is initialized. So, by narrowing down the failure-inducing schedule difference to one single difference, we have successfully re-discovered the location where we originally introduced the error.

What does this case study tell us? For one thing, that Delta Debugging is able to handle even *very large schedule differences* and still isolate the failure-inducing difference.

The second thing is that Delta Debugging treated *205_raytrace* like a *black box*—only the schedule was subject to observation and alteration. Nonetheless, we could easily associate the failure-inducing thread switch with the appropriate piece of code.

The third observation is that Delta Debugging is *very efficient when applied to thread schedules*, essentially working like a binary search. This is so because (except from invalid schedules, which can be excluded right away) there are few unresolved test outcomes, if any. And this, again, is so because programs are "mostly correct" with regard to the thread schedule—it is unlikely that there is a third outcome besides passing the test and showing the failure in question. (And even so, such third outcomes would frequently be classified as successes or failures.)

The downside of our experimental approach, of course, is that a significant number of (automated) tests are required—both for finding alternate schedules, and for isolating the failure-inducing difference. On the other hand, our approach is fully automatic and, furthermore, orthogonal to analytical approaches to detect trouble spots in threaded programs. These will be discussed in the next section.

7. RELATED WORK

As stated in the introduction, we are unaware of any other technique that would automatically isolate failure-inducing differences between schedules. Nonetheless, there is several related work:

Manipulating schedules. The core idea of this paper, altering schedules to isolate failure causes, has first been suggested by Stone [19] as *speculative replay*. Her idea was to "reduce the investigation of all possible [schedule] orderings to that of a few selected partial orderings" by guiding the replay process according to (human-inferred) thread dependencies. In contrast, our method is fully automatic; instead of having programmers speculate about thread dependencies, we isolate the failure-inducing schedule difference(s) automatically.

Testing alternate schedules. The generation of alternative thread schedules to trigger failures in concurrent programs has first been suggested by Edelstein et al. [7]. In contrast to replay altered schedules using a replay tool like DEJAVU, they *seed* the Java byte code with random *sleep*, *yield*, or *priority* primitives. Their focus, though, is on obtaining *coverage* rather than isolating failure causes. In principle, the seeding technique could be a good alternative to alter schedules where a DEJAVU-like tool is not available; nonetheless, deterministic schedule replay is a must.

Static analysis. Obviously, it is preferable to detect as many errors in the source code as possible rather than inferring errors from non-deterministic failures. In general, only a *conservative approximation* is feasible. For instance, one can have either context-sensitive program analysis or synchronization statements, but not both [15]. (*Context-insensitive* program analysis under concurrency is feasible, though [10]).

Several approximations do exist that detect which statements may happen in parallel (*concurrency analysis*) [12, 13] or may not (*non-concurrency analysis*) [11]. Also, several dedicated analysis methods for detecting *deadlocks* have been suggested and evaluated [4]. Like any analysis, these methods require complete knowledge of the whole program. The resulting static information can easily be exploited in both Delta Debugging and schedule generation by focusing the search on potential trouble spots.

Dynamic analysis. If one is willing to pay the overhead, *data races* like the examples in this paper can also be detected dynamically, for instance by monitoring all shared-memory references [8, 16]. The overhead of dynamic detection can be considerably reduced by combining it with static analysis [3, 14]. However, data races are just one class of problems induced by concurrency, and each problem class must be addressed by an individually designed dynamic analysis. Our approach, in contrast, is not restricted to a specific problem class—but it requires that the concurrency problem manifests itself as a failure.

8. CONCLUSION AND FUTURE WORK

We have presented a method that automatically isolates the failure-inducing difference(s) between a passing and a failing schedule, thus pinpointing the cause of a failure. Our method is purely experimental, meaning that analysis of the program in question is not required. It requires the ability to execute a program under altered thread schedules, such as provided by DEJAVU, and it requires a small number of automated tests. We expect that the basic observations from both the shared-queue example and the ray-tracer case study can easily be transferred to larger programs, too, as we expect programs to be "mostly correct" with regard to the thread schedule.

We recommend that capturing, replaying and isolating thread schedules be an integrated part of testing and debugging concurrent applications. Each time a test fails, delta debugging could be used to isolate the failure-inducing schedule difference. Given a capture/replay tool like DEJAVU, the approach presented in this paper is straightforward and easy to implement.

There is more to do, though. Our future work will concentrate on the following topics:

Cause-effect chains. Formally, the isolated schedule differences are *root causes* of the failure—they are a cause because the failure occurs if and only if the differences are applied, and they are a root cause because they are not an effect of some other event. Nonetheless, the isolated differences cause the failure only in conjunction with other root causes, such as the program code or its input.

We expect that in most cases, the code affected by the schedule differences is directly connected to the error. However, it may well be that the affected code is only the beginning of some cause-effect chain within the program run, triggering a failure that must be fixed at a very different location. Such cause-effect chains can be isolated by applying Delta Debugging on the program state [20].

Other circumstances. There may be other circumstances that *interfere* with the thread schedule. For instance, a specific thread schedule may cause the program to read some different input, resulting in an error. In such a situation, it is unclear whether the difference in the schedule or the difference in the input should be called "the" cause of the error. In principle, differences between thread schedules and differences between input can be handled the same way using Delta Debugging. Nonetheless, such interferences must be further examined.

Experiments vs. analysis. In general, research in program understanding has focused on *analytical approaches* so far. However, reasoning about a system is only one way to gather knowledge. The other way is experimentation. Automated *experimental approaches* like Delta Debugging offer additional means to isolate and understand the concrete behavior of systems. In future, we expect a fruitful intertwining of static analysis, dynamic analysis and automated experiments to widely automate program comprehension.

More case studies. The intertwining of different failure-inducing circumstances must be thoroughly examined in practice; the same applies for future combinations of analytical and experimental approaches. All these approaches must be thoroughly evaluated using real-life concurrent programs with (hopefully) real-life errors. DEJAVU is currently being extended from a prototype to a full product that will be able to capture and replay large-scale Java programs, including the GUI. As soon as this is done, we will have access to a wealth of case studies—and then ease the loathed debugging of concurrent systems.

Acknowledgments. This work was carried out during two visits of A. Zeller at IBM T. J. Watson Research Center in October 2000 and October 2001; support of IBM Research is gratefully acknowledged. Jens Krinke and Holger Cleve provided valuable comments on earlier revisions of this paper.

This research was supported by Deutsche Forschungsgemeinschaft, grant Sn 11/8-1.

Further information on Delta Debugging and DEJAVU is available online [5, 6].

9. REFERENCES

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P.Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM System Journal*, 39(1):211–238, Feb. 2000. Available online at http://www.research.ibm.com/journal/sj39-1.html.

[2] J.-D. Choi, B. Alpern, T. Ngo, M. Sridharan, and J. Vlissides. A perturbation-free replay platform for cross-optimized multithreaded applications. In *Proceedings of the 15th IEEE International Parallel & Distributed Processing Symposium*, April 2001.

[3] 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 *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2002. To appear.

[4] J. C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 22(3):161–180, 1996.

[5] Delta debugging web site. http://www.st.cs.uni-sb.de/dd/.

[6] Dejavu web site. http://www.research.ibm.com/jalapeno/dejavu/.

[7] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded Java program test generation. *IBM Systems Journal*, 41(1):111–125, Feb. 2002. Available online at http://www.research.ibm.com/journal/sj41-1.html.

[8] K. Havelund. Using runtime analysis to guide model checking of Java programs. In *Proc. of the 7th SPIN Workshop on Model Checking of Software*, volume 1885 of *Lecture Notes in Computer Science*, pages 245–264, Stanford University, California, Aug. 2000. Springer.

[9] IEEE, New York. *Test Methods for Measuring Conformance to POSIX*, 1991. ANSI/IEEE Standard 1003.3-1991. ISO/IEC Standard 13210-1994.

[10] J. Krinke. Static slicing of threaded programs. In *Proc. ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 35–42, Montreal, Canada, June 1998.

[11] S. P. Masticola and B. G. Ryder. Non-concurrency analysis. In *Proceedings of the Fourth ACM SIGPLAN Symposium on on Principles and Practices of Parallel Programming*, pages 129–138, May 1993.

[12] G. Naumovich and G. S. Avrunin. A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel. In *Proceedings of the ACM SIGSOFT Sixth International Symposium on the Foundations of Software Engineering (FSE)*, pages 24–34, November 1998.

[13] G. Naumovich, G. S. Avrunin, and L. A. Clarke. An efficient algorithm for computing *MHP* information for concurrent java programs. In *Proceedings of the Seventh European Software Engineering Conference and Seventh ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 338–354, September 1999.

[14] C. v. Praun and T. Gross. Object race detection. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2001.

[15] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Transactions on Programming Languages and Systems*, 22(2):416–430, March 2000.

[16] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[17] Standard Performance Evaluation Corporation (SPEC), Warrenton, Virginia. *JVM98 Benchmarks*, 1.03 edition, 1999.

[18] N. Sterling. Warlock: A static data race analysis tool. In *USENIX Winter Technical Conference*, pages 97–106, 1993.

[19] J. M. Stone. Debugging concurrent processes: A case study. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 145–154, June 1988.

[20] A. Zeller. Isolating cause-effect chains from computer programs. Technical report, Universität des Saarlandes, FR Informatik, Mar. 2002. Submitted for publication; available online [5].

[21] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, Feb. 2002.