International Symposium on Software Testing and Analysis (ISSTA), Rome, Italy, 2002

Isolating Failure-Inducing Thread Schedules

Andreas Zeller

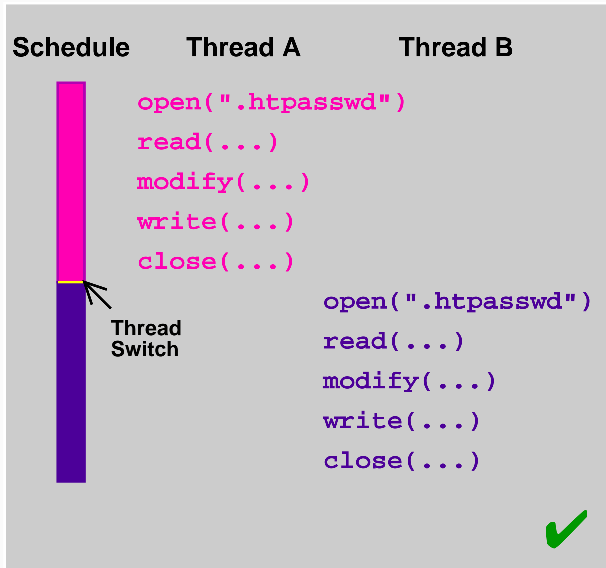
Lehrstuhl für Softwaretechnik
Universität des Saarlandes, Saarbrücken

Jong-Deok Choi

IBM T. J. Watson Research Center
Yorktown Heights, New York

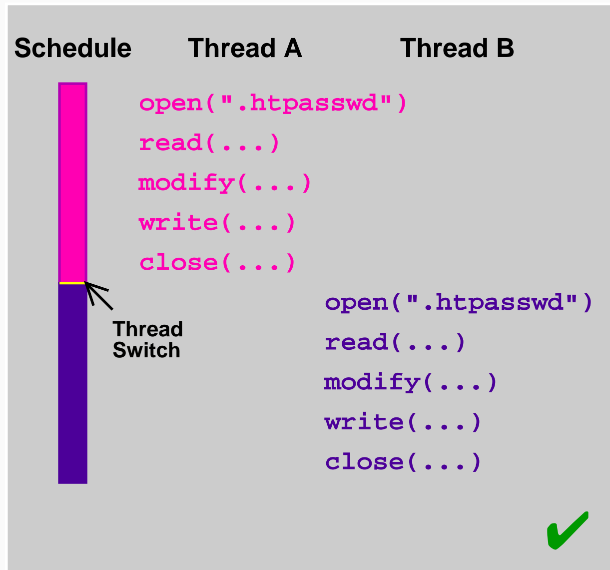
How Thread Schedules Induce Failures

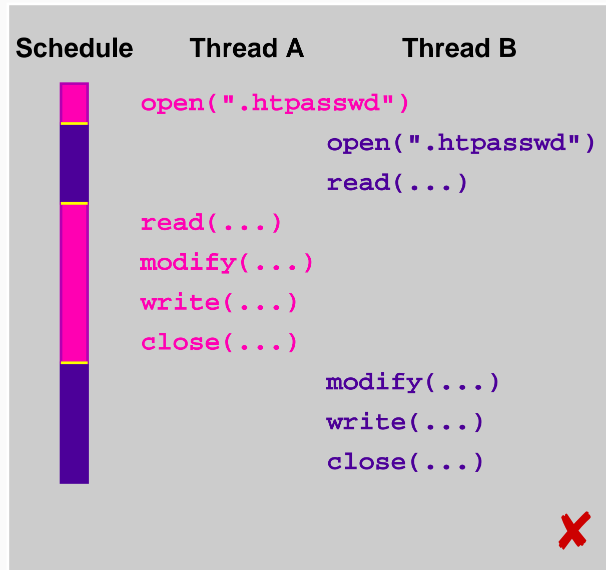
The behavior of a multi-threaded program can depend on the *thread schedule:*



How Thread Schedules Induce Failures

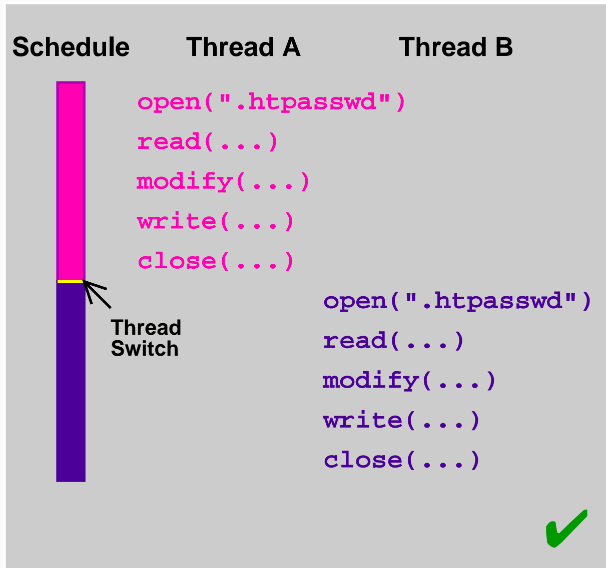
The behavior of a multi-threaded program can depend on the *thread schedule:*

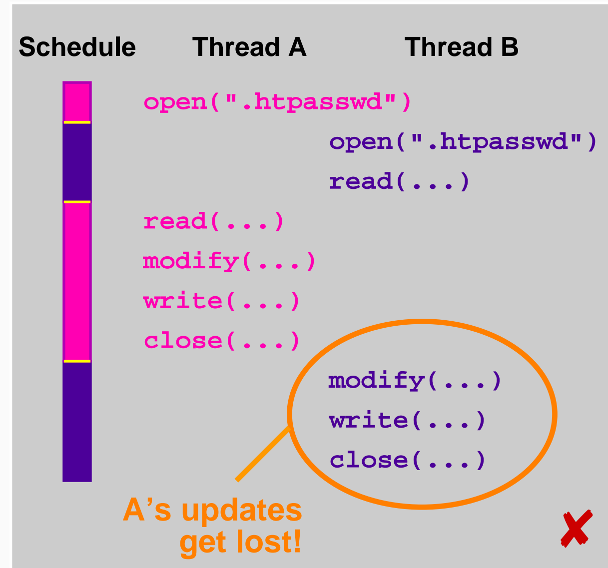




How Thread Schedules Induce Failures

The behavior of a multi-threaded program can depend on the *thread schedule:*





Thread switches and schedules are *nondeterministic:* Bugs are *hard to reproduce* and *hard to isolate!*

Recording and Replaying Runs

DEJAVU captures and replays program runs deterministically:

recorded schedule

record

```
x = 45
y = 39
z = 67
```

replay

```
x = 45
y = 39
z = 67
```

```
x = 45
y = 39
z = 67
```

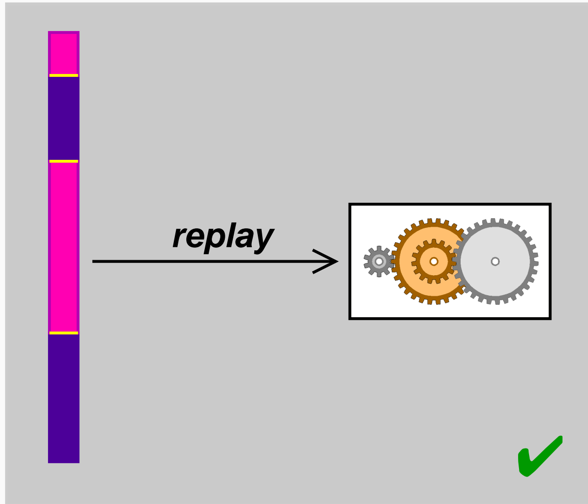
```
x = 45
y = 39
z = 67
```

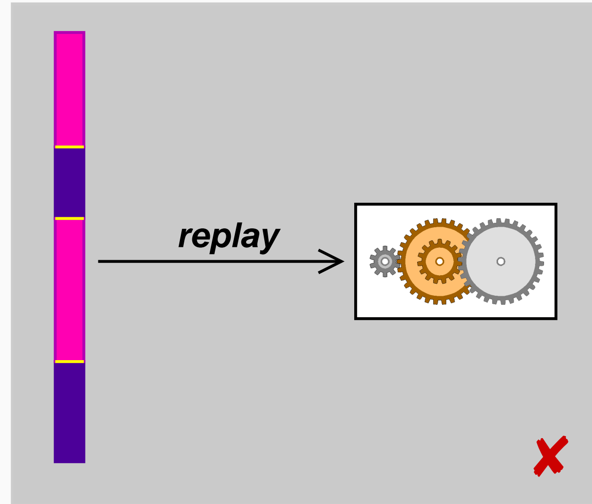
DEJAVU

Allows simple *reproduction* of schedules and induced failures

Differences between Schedules

Using DEJAVU, we can consider the schedule as an *input* which determines whether the program passes or fails.



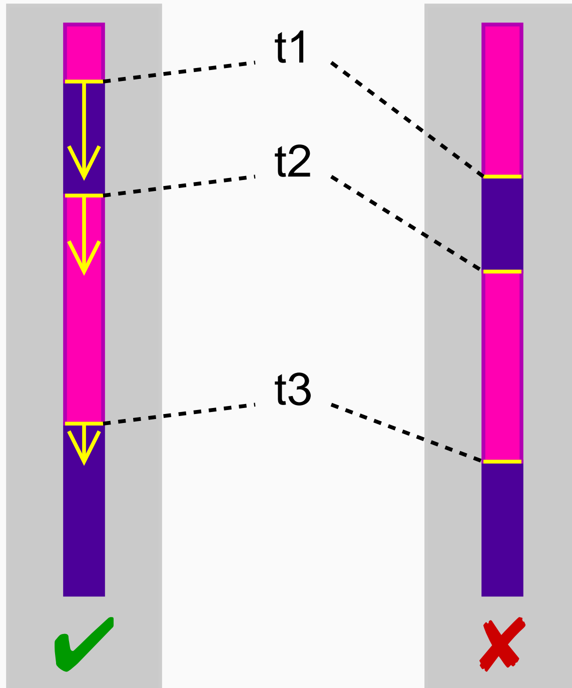


Differences between Schedules

Using DEJAVU, we can consider the schedule as an *input* which determines whether the program passes or fails.

The *difference* between schedules is relevant for the failure:
A *small* difference can pinpoint the failure cause

Finding Differences



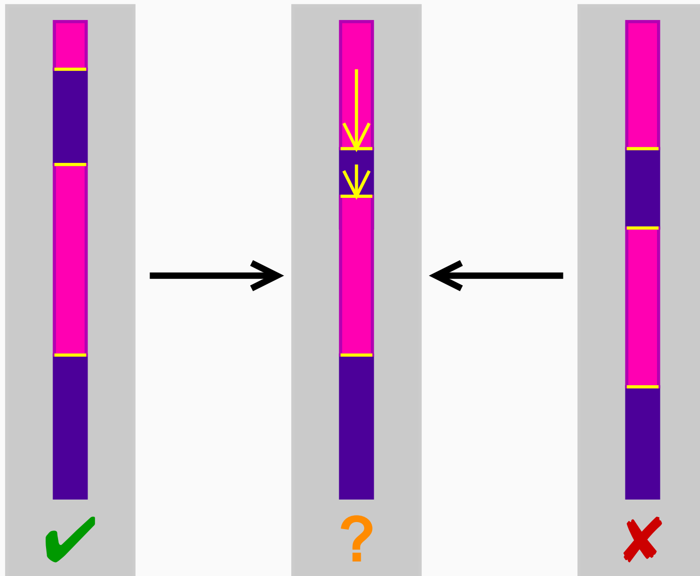
- We start with runs ✔ and ✘
- We determine the differences Δ_i between thread switches t_i:
  - t_1 occurs in ✔ at "time" 254
  - t_1 occurs in ✘ at "time" 278
  - The difference $\Delta_1 = |278 - 254|$ induces a *statement interval:* the code executed between "time" 254 and 278
  - Same applies to t_2, t_3, etc.

Our goal: *Narrow down* the difference such that only a small *relevant difference* remains, pinpointing the root cause

Isolating Relevant Differences

We use *Delta Debugging* to isolate the relevant differences

Delta Debugging applies *subsets* of differences to ✔:



- The *entire* difference Δ_1 is applied
- Half of the difference Δ_2 is applied
- Δ_3 is not applied at all

DEJAVU executes the debuggee under this *generated* schedule; an automated test checks if the failure occurs

The Isolation Process

Delta Debugging systematically narrows down the difference

✔ ? ✘

Dejavu replays the generated schedule

✔ ✘

Test outcome

A Real Program

We examine Test #205 of the SPEC JVM98 Java test suite: a raytracer program depicting a dinosaur

Program is single-threaded—the multi-threaded code is commented out

A Real Program

We examine Test #205 of the SPEC JVM98 Java test suite: a raytracer program depicting a dinosaur

Program is single-threaded—the multi-threaded code is commented out

To test our approach,

- we make the raytracer program *multi-threaded* again
- we introduce a simple *race condition*
- we implement an *automated test* that would check whether the failure occurs or not
- we generate *random schedules* until we obtain both a passing schedule (✔) and a failing schedule (✘)

Passing and Failing Schedule

We obtain two schedules with 3,842,577,240 differences, each moving a thread switch by ± 1 "time" unit

Thread Schedules

Failing Schedule
Passing Schedule

Time (# yield points)

1.8e+08
1.6e+08
1.4e+08
1.2e+08
1e+08
8e+07
6e+07
4e+07
2e+07
0

0 10 20 30 40 50 60 70 80 90 100

Thread switches

Narrowing Down the Failure Cause

Delta Debugging isolates one single difference after 50 tests:

The Root Cause of the Failure

```
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 }
```

Lessons Learned

Delta Debugging is efficient even when applied to very large thread schedules

Programs are “mostly correct” w.r.t. the thread schedule
⇒ Delta Debugging works like a binary search

Lessons Learned

Delta Debugging is efficient even when applied to very large thread schedules

Programs are “mostly correct” w.r.t. the thread schedule
$\Rightarrow$ Delta Debugging works like a binary search

No analysis is required as Delta Debugging relies on experiments alone

Only the schedule was observed and altered

Failure-inducing thread switch is easily associated with code

Lessons Learned

Delta Debugging is efficient even when applied to very large thread schedules

Programs are “mostly correct” w.r.t. the thread schedule
$\Rightarrow$ Delta Debugging works like a binary search

No analysis is required as Delta Debugging relies on experiments alone

Only the schedule was observed and altered

Failure-inducing thread switch is easily associated with code

Alternate runs can be obtained automatically by generating random schedules

Only one initial run (✔ or ✘) is required

Lessons Learned

Delta Debugging is efficient even when applied to very large thread schedules

Programs are “mostly correct” w.r.t. the thread schedule
⇒ Delta Debugging works like a binary search

No analysis is required as Delta Debugging relies on experiments alone

Only the schedule was observed and altered

Failure-inducing thread switch is easily associated with code

Alternate runs can be obtained automatically by generating random schedules

Only one initial run (✔ or ✘) is required

The whole approach is annoyingly simple in comparison to many other ideas we initially had

Conclusion

Debugging multi-threaded applications is easy:

- Record/Replay tools like DEJAVU reproduce runs
- Delta Debugging pinpoints the root cause of the failure

Debugging can do without analysis:

- It suffices to execute the debuggee under changing circumstances

There is still much work to do:

- More *case studies* (as soon as DEJAVU can handle GUIs)
- Using *program analysis* to guide the narrowing process
- Isolating *cause-effect chain* from root cause to failure

http://www.st.cs.uni-sb.de/dd/
http://www.research.ibm.com/dejavu/