Isolating Failure-Inducing Thread Schedules

by Jong-Deok Choi - IBM T.J. Watson Research Center
and Andreas Zeller - Saarland University

Marlene Hochrieser
FMV - Institute for Formal Models and Verification
Introduction to the problem

Consider a multi threaded application is running on the same input several times and fails occasionally.

- How do I reproduce a failure?
  Thread switches are non-deterministic.

- How do I isolate the error?
  Thread schedules may be composed of thousands of thread switches

- How do I find failing runs?
  Is it possible to test different thread switch sequences with the same input?
Introduction to the solution

- Record a test run
- Deterministic replay
- Generate test cases
- Isolate failure causes by using delta debugging
- Relate causes to errors
Contents

- Code Example
- Jalapeño - DejaVu
- Capturing and Playbacking
- Isolating failure with Delta Debugging
- Generating altered schedules
- Case study - Ray tracer
- Related work
- Conclusion and future work
- Literature
Code example

24 // Enqueue ELEM.
25 public void enqueue(int elem) {
26     link[elem] = 0;
27     if (head == 0)
28         head = elem;
29     else {
30         synchronized (this) {
31             link[tail] = elem;
32         }
33     }
34     tail = elem;
35 }
36
37 // Return first element of queue.
38 // No error checking.
39 public int dequeue() {
40     int elem = head;
41     if (elem == tail)
42         tail = 0;
43     synchronized (this) {
44         head = link[head];
45     }
46     return elem;
47 }
48
49 // Constructor
50 IntQueue() {
51     head = 0;
52     tail = 0;
53     for (int i = 0; i < numberOfElements; i++)
54         link[i] = 0;
55 }
56
57 // Print elements of queue
58 public void print() {
59     for (int e = head; e != 0; e = link[e])
60         System.out.print(e + " ");
61     System.out.println();
62 }
63
64 class IntQueue {
65     // The queue holds integers in the range
66     // of [1...numberOfElements - 1]
67     static final int numberOfElements = 100;
68     // link[N] is N's successor in the queue
69     int link[] = new int[numberOfElements];
70     int head; // First element of queue
71     int tail; // Last element of queue
72 }
73
Three test runs with same input - Passing / Failing

Clock - Thread A
1  enqueue(11)
2  26 link[elem] = 0; // link[11] = 0
3  28 if (head == 0) // 0 == 0
4  29 head = elem; // head = 11
5  30 tail = elem; // tail = 11

Clock - Thread B
1  dequeue()
2  42 elem = head // elem = 11
3  43 if (elem == tail) // 21 == 11
4  44 tail = 0; //
5  47 head = link[head]; // head = 0
6  50 return elem; // return 11

Clock - Thread C
1  enqueue(95)
2  26 link[elem] = 0; // link[95] = 0
3  28 if (head == 0) // 0 == 0
4  29 head = elem; // head = 95
5  30 tail = elem; // tail = 95

Clock - Thread A
1  enqueue(13)
2  26 link[elem] = 0; // link[13] = 0
3  28 if (head == 0) // 0 == 0
4  29 head = elem; // head = 11

Clock - Thread B
1  dequeue()
2  42 elem = head // elem = 11
3  43 if (elem == tail) // 21 == 11
4  44 tail = 0; //

Clock - Thread C
1  enqueue(95)
2  26 link[elem] = 0; // link[95] = 0
3  28 if (head == 0) // 0 == 0
4  29 head = elem; // head = 95
5  30 tail = elem; // tail = 95

Clock - Thread A
10 47 head = link[head]; // head = 0
11 50 return elem; // return 11
Three test runs with same input - Passing

```
Clock   Thread A                     Thread B                     Thread C
1       enqueue(11)                        
2       link[elem] = 0;     // link[11] = 0
3       if (head == 0)  // 0 == 0
4       head = elem;    // head = 11
5       enqueue()                        
6       dequeue()                     
7       elem = head       // elem = 11
8       tail = elem;     // tail = 11
9       dequeue()                     
10      if (elem == tail)  // 11 == 11
11      tail = 0;          //
12      head = link[head]; // head = 0
13      return elem; /// return 11
14      ...                        
15       enqueue(95)                     
16       link[elem] = 0;     // link[95] = 0
17       if (head == 0)  // 0 == 0
18       head = elem;    // head = 95
19       tail = elem;     // tail = 95
```
Jalapeño - DejaVu

**Jalapeño** - Research VM for Java servers, developed at the IBM T.J. Watson Research Center.

- Designed for scalability (SMP), high performance, sophisticated thread support, availability, rapid response,..

- Papers are available at http://jikesrvm.org/ (redirected from IBM's research page)

- Overview of *Jalapeño*: see [1]

**DejaVu** - **D**eterministic **Java** **U**tility is a tool to *capture, alter*, and *replay* Thread schedules.

- Running as part of *Jalapeño* VM

- Making failures reproducible by capturing a non-deterministic run and replaying it deterministically.
Jalapeño and Scheduling

Using implementation of the garbage collector to do scheduling.

Quasi-preemptive scheduling

**Safe point** - is a program location where the compiler that created the method body is able to describe where all the live references exit.

**Yield point** - is a safe point located at a method prologue (such as function invocation or at a loop back-edge).

Thread switching takes place only when a running thread has reached a *yield point*. 
Capturing and playbacking

- *Dejavu* is using the *yield points* as global clock values
- During recording *Dejavu* stores global clock values of thread switches
- During playback a thread switch is forced at every *yield point*
- Different sequences of global clock values can be generated
- *Dejavu* will use them to force a thread switch
- Can be used for test generation
Capturing thread schedule - another approach [2]

```java
class Test {
    static public volatile int f = 0;
    // shared variable
    static public volatile int g = 20;
    // shared variable
    static public void main(String argv[]) {
        int j; // local variable
        MyThread t1 = new MyThread();
        t1.start();
        j = 20;
        System.out.println("f = f + f + j = j + j");
    }
}
class MyThread extends Thread {
    public void run() {
        int k; // local variable
        k = 0;
        Test.f = Test.f + k;
        Test.g = Test.g - k;
    }
}
```

- Thread schedule is a sequence of time intervals (time slices), containing thread schedule information

- Logical thread schedules: Generate equivalence classes for different order of access to shared variable

- Use logical thread schedule information with physical thread schedule
Isolating failure with Delta Debugging

A thread schedule is defined as a list of \( n \) clock times: \( t_1, ... t_n \)

*Padded* form of example results:

\(< 6, 12, 17, 17, 17 >\) - passing thread schedule in the *padded* form
\(< 5, 7, 8, 10, 15 >\) - failure-inducing thread schedule
\(< 5, 7, 8, 13, 17 >\) - passing thread schedule

\( stest \) is defined to return:
- ✔ if the queue holds the value 95
- ✗ if the queue is empty
- ? otherwise
Difference decomposition

- Difference of two schedules $T_\triangledown$ and $T_\star$ are defined as $\delta : \tau \rightarrow \tau$ with $\delta(T_\triangledown) = (T_\star)$

- Set of all differences: $C' = \tau^\tau$

- Decompose $\delta$ in number of thread switch changes $\delta_i$

- A schedule difference $\delta$ between two schedules $T_\triangledown = < t_{\triangledown 1}, ..., t_{\triangledown n} >$ and $T_\star = < t_{\star 1}, ..., t_{\star n} >$ is defined as $\delta = \delta_1 \circ \delta_2 \circ ... \circ \delta_n$ where each $\delta_i : \tau \rightarrow \tau$ maps $t_{\triangledown i}$ to $t_{\star i}$; that is $\delta_i(T_\triangledown) = < t_{1\triangledown}, ..., t_{i-1\triangledown}, t_{\star i}, t_{\triangledown i+1}, ..., t_{\triangledown i+n} >$

- $\circ : C \times C \rightarrow C$ is defined as $(\delta \circ \delta)(T') = \delta_i(\delta_j(T'))$
Atomic decomposition

\[ \delta_i = \delta_{i,1} \circ \delta_{i,2} \circ \ldots \circ \delta_{i,|t_n-t_i|} \] where each \( \delta_{i,j} \) is defined as:

\[ \delta_{i,j}(T_{\nu}) = \delta_{i,j}(< t_{\nu_1}, t_{\nu_2}, \ldots, t_{\nu_n} >) =
\]

\[ = < t_{\nu_1}, t_{\nu_2}, \ldots, t_{\nu_{i-1}}, t'_{\nu_i}, t_{\nu_{i+1}}, \ldots t_{\nu_n} > \]

where \( t'_{\nu_i} \) is the value altered by \( \delta_{i,j} \); that is

\[ \rightarrow t_{\nu_i} + 1 \text{ if } t_{\nu_i} < t_{\nu_i} \]

\[ \rightarrow t_{\nu_i} - 1 \text{ if } t_{\nu_i} > t_{\nu_i} \]

<table>
<thead>
<tr>
<th>Tests</th>
<th>( \delta_{1,1} )</th>
<th>( \delta_{2,1}\delta_{2,2}\delta_{2,3}\delta_{2,4}\delta_{2,5} )</th>
<th>( \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} )</th>
<th>( \delta_{4,1}\delta_{4,2}\delta_{4,3}\delta_{4,4}\delta_{4,5}\delta_{4,6}\delta_{4,7} )</th>
<th>( \delta_{5,1}\delta_{5,2} )</th>
<th>Schedule</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T_{\nu} )</td>
<td>.</td>
<td>. . . . . . .</td>
<td>. . . . . . . .</td>
<td>. . . . . .</td>
<td>.</td>
<td>( 6, 12, 17, 17, 17 )</td>
<td>✓</td>
</tr>
<tr>
<td>( T_{\nu} )</td>
<td>.</td>
<td>. .</td>
<td>. . . . . .</td>
<td>. . . . . .</td>
<td>.</td>
<td>( 5, 7, 10, 15 )</td>
<td>✗</td>
</tr>
<tr>
<td>(1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( 5, 7, 17, 17 )</td>
<td>✓</td>
</tr>
<tr>
<td>(2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( 5, 7, 10, 17 )</td>
<td>✗</td>
</tr>
<tr>
<td>(3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( 5, 7, 13, 17 )</td>
<td>✓</td>
</tr>
<tr>
<td>(4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( 5, 7, 11, 17 )</td>
<td>✓</td>
</tr>
<tr>
<td>Result</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\( t'_{\nu_i} \) is the value altered by \( \delta_{i,j} \); that is
Generating altered schedules with fuzzy approach

- Failing test run → passing test run
- Passing test run → failing test run

- Start from existing schedule
- Generate similar test cases to optimize determining differences
- Use simple Gaussian distribution centered around \( t \):

From given schedule \( T = \langle t_1, t_2, ..., t_n \rangle \) generate \( T = \langle f(t_1), f(t_2), ..., f(t_n) \rangle \), where \( f(t) \) is a perturbation function. - Widen distribution until an altered schedule is found
Case study - Ray tracer

- Each ray-tracing thread calls `LoadScene` → race condition → no update for `ScenesLoaded`
- Record of failing schedule $T_x$ contained 3770 thread switches
- Using fuzzy approach it took 66 test to generate a passing schedule
- More than a million `yield points` → 3,842,577,240 atomic deltas have been applied
- Outcome: Amount of time was larger for $T_x$

- Error isolated at `yield point` 59,772,127 → back tracing for a given set of `yield points` has been implemented
Related work

Article about *Debugging concurrent processes*, presented by [5]:

- Alter thread schedules manually
- Main idea of the paper *Isolating Failure-Inducing Thread Schedules*, but solution for automated testing.

Testing alternate schedules by [4]:

- Idea is to manipulate in order to get different schedules
- Using sleep or priorities
- Focus on coverage rather than on isolating failure causes
Conclusion and future work

- Method that automatically isolates the failure-inducing difference(s) between a passing and a failing schedule

- Purely experimental / analysis of the program in question is not required

- Use capturing, replaying, and isolating thread schedule as integrated part of testing and debugging of concurrent applications

- Use delta debugging

- Focus on cause-effect chains and other circumstances

- Integration of static analysis, dynamic analysis, and automated experiments

- More case studies
Literature


