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

```
class IntQueue {
    // The queue holds integers in the range
    // of [1..numberOfElements - 1]
    static final int numberOfElements = 100;

    // link[N] is N's successor in the queue
    int link[] = new int[numberOfElements];

    int head;  // First element of queue
    int tail;  // Last element of queue

    // Constructor
    IntQueue() {
        head = 0;
        tail = 0;
        for (int i = 0; i < numberOfElements;
             i++) {
            link[i] = 0;
        }
    }

    // Enqueue ELEM.
    public void enqueue(int elem) {
        link[elem] = 0;

        if (head == 0)
            head = elem;
        else {
            synchronized (this) {
                link[tail] = elem;
            }
        }

        tail = elem;
    }

    // Return first element of queue.
    // No error checking.
    public int dequeue() {
        int elem = head;
        if (elem == tail)
            tail = 0;

        synchronized (this) {
            head = link[head];
        }

        return elem;
    }

    // Print elements of queue
    public void print() {
        for (int e = head; e != 0; e = link[e])
            System.out.print(e + " ");
        System.out.println();
    }
}
```

Three test runs with same input - Passing / Failing

Clock	Thread A		Thread B		Thread C
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	36 tail = elem; // tail = 11				
6		1 →	**dequeue()**		
7			42 elem = head // elem = 11		
8			43 if (elem == tail) // 11 == 11		
9			44 tail = 0; //		
10			47 head = link[head]; // head = 0		
11			50 return elem; // return 11		
12				2 →	**enqueue(95)**
13					26 link[elem] = 0; // link[95] = 0
14					28 if (head == 0) // 0 == 0
15					29 head = elem; // head = 95
16					36 tail = elem; // tail = 95

Clock	Thread A		Thread B		Thread C
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		1 →	**dequeue()**		
6			42 elem = head // elem = 11		
7	36 tail = elem; // tail = 11	← 2			
8		3 →	43 if (elem == tail) // 11 == 11		
9			44 tail = 0; //		
10				4 →	**enqueue(95)**
11					26 link[elem] = 0; // link[95] = 0
12					28 if (head == 0) // 11 == 0
13					32 link[tail] = elem; // link[0] = 95
14					36 tail = elem; // tail = 95
15			47 head = link[head]; // head = 0	← 5	
16			50 return elem; // return 11		

Three test runs with same input - Passing

Clock	Thread A		Thread B		Thread C
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		→1	**`dequeue()`**		
6			42 `elem = head` `// elem = 11`		
7	36 `tail = elem;` `// tail = 11`	←2			
8		→3	43 `if (elem == tail)` `// 11 == 11`		
9			44 `tail = 0;` `//`		
10			47 `head = link[head];` `// head = 0`		
11			50 `return elem;` `// return 11`		
12			...		
13				→4	**`enqueue(95)`**
14					26 `link[elem] = 0;` `// link[95] = 0`
15					28 `if (head == 0)` `// 11 == 0`
16					29 `head = elem;` `// head = 95`
17					36 `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 - **De**terministic **Jav**a **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

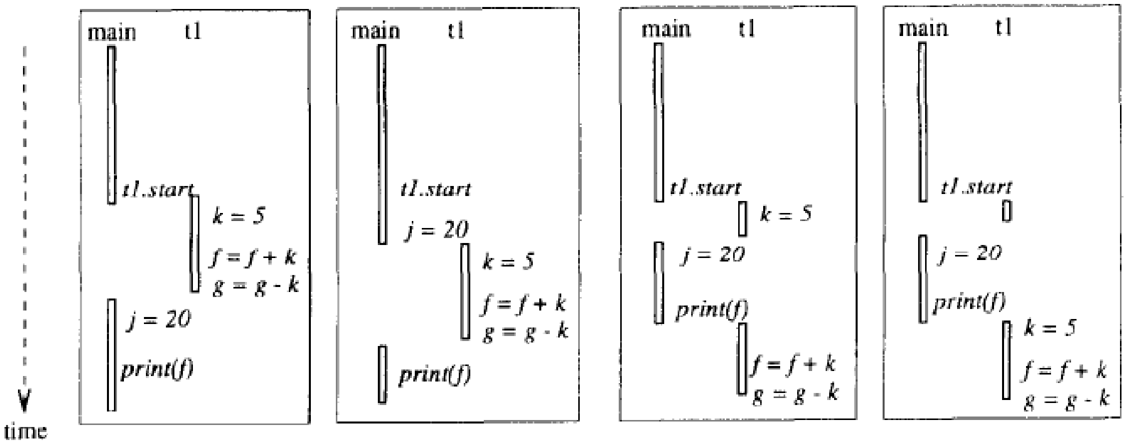
- *Dejavu* will use them to force a thread switch

- Can be used for test generation

Capturing thread schedule - another approach [2]

```
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
      + `` j = `` + j);
  }
}

class MyThread extends Thread {
  public void run() {
    int k; // local variable
    k = 5;
    Test.f = Test.f + k;
    Test.g = Test.g - k;
  }
}
```



(a) (b) (c) (d)

- 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 ✔ and T ✘ are defined as
$\delta : \tau \rightarrow \tau$ with $\delta(T_{✔}) = (T_{✘})$

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

- Decompose δ in number of thread switch changes δ_i

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

Atomic decomposition

$\delta_i = \delta_{i,1} \circ \delta_{i,2} \circ .... \circ \delta_{i,|t_{\checkmark i} - t_{\times i}|}$ where each $\delta_{i,j}$ is defined as $\delta_{i,j}(T_{\checkmark}) = \delta_{i,j}(< t_{\checkmark 1}, t_{\checkmark 2}, ..., t_{\checkmark n} >) =$
$=< t_{\checkmark 1}, t_{\checkmark 2}, ..., t_{\checkmark i-1}, t_{\checkmark}{}'_i, t_{\checkmark i+1}, ...t_{\checkmark n} >$

where $t_{\checkmark}{}'_i$ is the value altered by $\delta_{i,j}$; that is
$\rightarrow t_{\checkmark i} + 1$ if $t_{\checkmark i} < t_{\times i}$
$\rightarrow t_{\checkmark i} - 1$ if $t_{\checkmark i} > t_{\times i}$

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
$T_{\checkmark}$	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	⟨6, 12, 17, 17, 17⟩	✔
$T_{\times}$	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	⟨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																						□				

Generating altered schedules with fuzzy approach

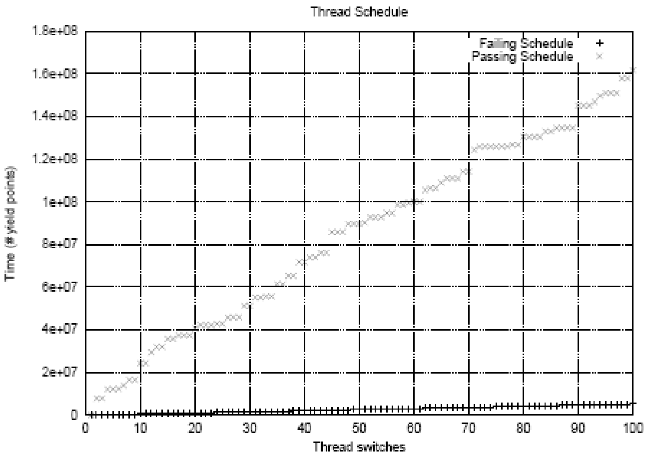
- Failing test run $\rightarrow$ passing test run
- Passing test run $\rightarrow$ failing test run

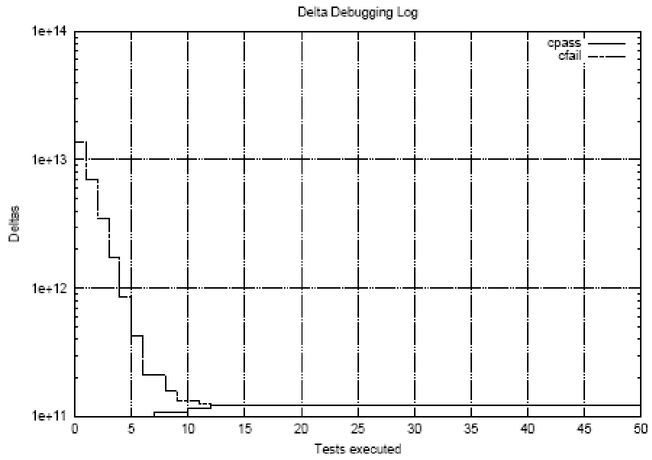
- Start from existing schedule
- Generate similar test cases to optimize determining differences
- Use simple Gaussian distribution centered around t:

From given schedule $T = < t_1, t_2, ..., t_n >$ generate $T = < f(t_1), f(t_2), ..., f(t_n) >$, where $f(t)$ is a perturbation function. - Widen distribution until an altered schedule is found

Case study - Ray tracer

```
public class Scene { ...
    private static int ScenesLoaded = 0;
    (more methods...)
    private
    int LoadScene(String filename) {
        int OldScenesLoaded = ScenesLoaded;
        (more initializations...)
        infile = new DataInputStream(...);
        (more code...)
        ScenesLoaded = OldScenesLoaded + 1;
        System.out.println("" +
            ScenesLoaded + " scenes loaded.");
        ...
    }
    ...
}
```





- Each ray-tracing thread calls *LoadScene* → race condition → no update for *ScenesLoaded*
- Record of failing schedule T ✘ 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 ✔

- 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

[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 jalapeno virtual machine. IBM Systems Journal, Vol 39, No 1, 2000.

[2] Jong-Deok Choi and Harini Srinivasan. Deterministic replay of java multithreaded applications. In SPDT 98: Proceedings of the SIGMETRICS symposium on Parallel and distributed tools, pages 4859, New York, NY, USA, 1998. ACM Press.

[3] Jong-Deok Choi and Andreas Zeller. Isolating failure-inducing thread schedules. In ISSTA 02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis, pages 210220, New York, NY, USA, 2002. ACM Press.

[4] Y. Nir G.Ratsaby O. Edelstein, E. Farchi and S. Ur. Multithreaded java program test generation. IBM Systems Journal, 41(1):111-125, Februar 2002.

[5] J. M. Stone. Debugging concurrent processes: A case study., June 1988.