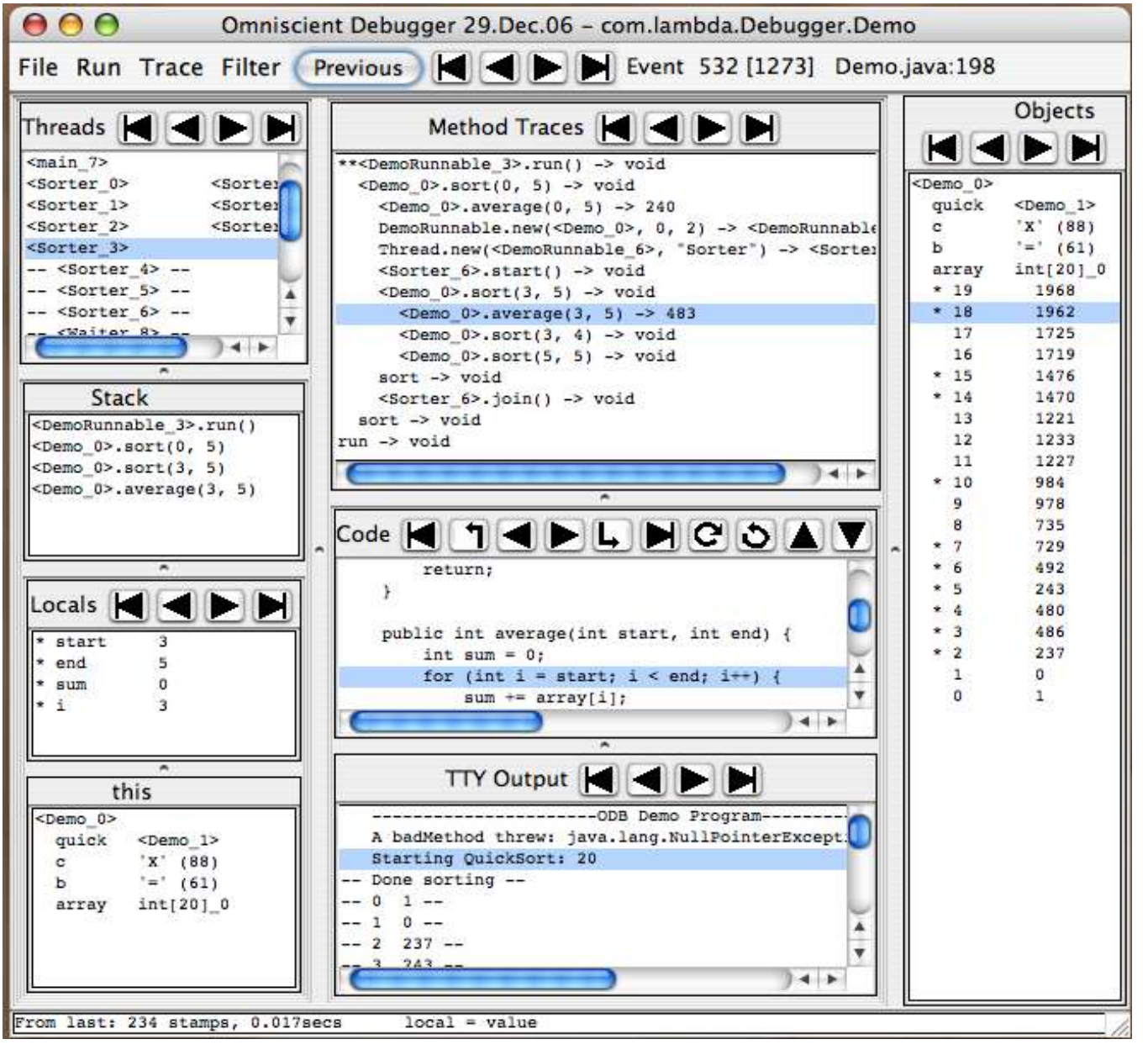
The Omniscient Debugger: Debugging Backwards in Time

“Because the Debugger Knows Everything”

Bil Lewis


Omniscient Debugger 29.Dec.06 – com.lambda.Debugger.Demo
File Run Trace Filter Previous Event 532 [1273] Demo.java:198
Threads
<main_7>
<Sorter_0>
<Sorter_1>
<Sorter_2>
<Sorter_3>
-- <Sorter_4> --
-- <Sorter_5> --
-- <Sorter_6> --
Stack
<DemoRunnable_3>.run()
<Demo_0>.sort(0, 5)
<Demo_0>.sort(3, 5)
<Demo_0>.average(3, 5)
Locals
* start 3
* end 5
* sum 0
* i 3
this
<Demo_0>
quick <Demo_1>
c 'X' (88)
b '=' (61)
array int[20]_0
Method Traces
**<DemoRunnable_3>.run() -> void
<Demo_0>.sort(0, 5) -> void
<Demo_0>.average(0, 5) -> 240
DemoRunnable.new(<Demo_0>, 0, 2) -> <DemoRunnable
Thread.new(<DemoRunnable_6>, "Sorter") -> <Sorter
<Sorter_6>.start() -> void
<Demo_0>.sort(3, 5) -> void
<Demo_0>.average(3, 5) -> 483
<Demo_0>.sort(3, 4) -> void
<Demo_0>.sort(5, 5) -> void
sort -> void
<Sorter_6>.join() -> void
sort -> void
run -> void
Code
return;
}
public int average(int start, int end) {
int sum = 0;
for (int i = start; i < end; i++) {
sum += array[i];
TTY Output
----------ODB Demo Program----------
A badMethod threw: java.lang.NullPointerExcept
Starting QuickSort: 20
-- Done sorting --
-- 0 1 --
-- 1 0 --
-- 2 237 --
Objects
<Demo_0>
quick <Demo_1>
c 'X' (88)
b '=' (61)
array int[20]_0
* 19 1968
* 18 1962
17 1725
16 1719
* 15 1476
* 14 1470
13 1221
12 1233
11 1227
* 10 984
9 978
8 735
* 7 729
* 6 492
* 5 243
* 4 480
* 3 486
* 2 237
1 0
0 1
From last: 234 stamps, 0.017secs local = value

Recording Events

Record “time stamps” for “interesting events” in the program:

- **State Changes:**
  - **Local Variables**
  - **Instance Variables**
  - **Static Variables**
  - **Array Elements**
- **Method Calls:**
  - **“Virtual” Methods**
  - **Static Methods**
  - **Constructors**
  - **Super Methods**

Interesting

Debugging is easier if you can go backwards:

It eliminates the worst problems with breakpoints:

- **No “guessing” where to put breakpoints**
- **No “extra steps” to debugging:**
  - **Set breakpoints, start, examine, continue…**
- **No “fatal” mistakes (no “Whoops, I went too far”)**
- **No non-deterministic problems**

It gives the programmer a unique view of the program

All data is serializable:

- **It can be saved to a file**
- **Customers can email a debugging session to developers**

It’s Not A Bug…

The ODB divides bugs into two groups:

Snakes in the grass whose tails you can see

Snakes in the grass whose tails you can’t see

If a program outputs bad data, then we can see the tail. With the ODB, we can always find the head. We don’t even have to know the program!

If We Can't See The Tail...

If a program fails to output expected data, then we can't see the tail. Now we have to know the program and search for where the output *should* have happened.

This means we want to do a complex search over a large set of events representing the execution of the program...

If We Can't See The Tail...

An effective method of doing this is to use an event analysis engine... which is a familiar problem!

I use the prolog-style event search interface of M. Ducasée:

port = call & arg0 = 0 & methodName = "sort"

With Breakpoint Debuggers

It’s not a snake, it’s more like a lizard…

The ODB

An implementation of the Omniscient Debugging Concept in Java.

The ODB:

- **Instruments the class' byte code**
- **Runs in the same process**
- **Uses a single lock**
  - **(all events in all threads are linearly ordered)**

I know how to do the same for C, C++, Eiffel, etc. (It's just harder!)

Print Strings

I have chosen these formats:

```
<MyObj_123 Bob>
int[20]_2

MyObj
"Random string"
1234, 2.4
‘X’ (88)
true, false
--                          (No value yet)
```

Value Display

The contents of objects will be displayed with their values current to the selected time stamp:

```
<Person_2 Lovi>
  name          "Lovi"
  age           23
  home          <Address_3>
  friend        <Person_3 Tarvi>

int[3]_2
  0           33
  1           66
  2           77

<Person_2 Lovi>            (previous to creation)
  name          --
  age           --
  home          --
  friend        --
```

Method Traces

```
<Obj_1>.frob(<Obj_6> , 156, "frobbing") -> true
  <Obj_6>.twiddle() -> 5
    <Obj_6>.spin() -> void
    <Obj_6>.spin() -> void
    <Obj_6>.spin() -> void
  twiddle -> 5
frob -> true
```

Let’s Take a Tour…

Performance

Answer #1:

My objective is to show that Omniscient Debugging is an effective technique, irrespective of performance.

Answer #2:

There are bugs which are not amenable to this technique.

I have never seen one.

Answer #3

The ODB is a naive implementation which does NO optimization.

I know how to make it much faster, but answer #2 still applies.

Answer #3

In 31-bit address space I have recorded:

100 million events

1us/event

(caveat, caveat, caveat…)

Answer #3

In a 64-bit address space, you should get:

2 Quadrapule-Mega-Gazillion events*

1us/event => 20,000 years ??!

If it takes 1,000 years for your bug to manifest itself,
would you be willing to wait 10,000 years for the debugger?

* **8E+17**

Answer #3

The slowdown for an individual program varies:

```
for (int i=0; i<MAX; i++)    sum+=smallArray[i];          300x
for (int i=0; i<MAX; i++)    x=x*x+x;                     100x
for (int i=0; i<MAX; i++)    sum+=bigArray[i];             30x
for (int i=0; i<MAX; i++)    s="Item"+i;                    2x
```

```
Debugging ODB back-end                          300x
Debugging ODB display                       10x
Debugging Ant                             7x
```

Performance

Reduction Techniques:

- **Start/Stop Collection:**
  - **Manually via “Start/Stop” button**
  - **Automatically via event matching**
- **Garbage Collection**
- **Don’t record all methods:**
  - **Skip “uninteresting” methods (e.g., java.lang)**
  - **Skip “recomputable” methods**

Conclusion

Answer #2:

There are bugs which are not amenable to this technique.

I have never seen one.