ORACLE

jcstress
Breaking Concurrency Bad

Aleksey Shipilev
aleksey.shipilev@oracle.com, @shipilev

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Concurrency testing is hard

Problems

1. Time is the external variable
2. The tests are probabilistic at best; need many runs to catch the unlucky behaviors
3. The faster the test infrastructure has to be, the more hardcore concurrency stuff it has to use, the more error-prone it is

jcstress

Experimental harness + suite of tests:

`http://openjdk.java.net/projects/code-tools/jcstress/`

- Lots of non-covered areas
- Lots of tests already (12K+)
- Found handful of bugs at SW/HW levels

Test Sample

Volatile increment atomicity test:

```
 class MyTest implements ConcurrencyTest<State, Res> {
    void actor1(State s, Res r) { r.r1 = s.v++; }
    void actor2(State s, Res r) { r.r2 = s.v++; }

    class State {  volatile int v;  }
    State newState() { new State(); }
}
```

Can infer the behavior from observed (r1, r2) pairs

```
  State        Occurrences           Expectation
 [1, 1] (       1,360,407)    KNOWN_ACCEPTABLE
 [1, 2] (      57,137,771)            REQUIRED
 [2, 1] (      55,286,472)            REQUIRED
```

The Sweet Taste of Failure

hotspot/src/share/vm/prims/unsafe.cpp[1]

```
#define GET_FIELD_VOLATILE(obj, offset, type_name, v) \
  oop p = JNIHandles::resolve(obj); \
  type_name v =
    OrderAccess::load_acquire(
       (volatile type_name*)
       index_oop_from_field_offset_long(p, offset));
```

Unsafe_GetDoubleVolatile() compiles[2] to :

```
mov     0x18(%esp),%ebp
add     %ebp,%eax
; field offset in %eax
fldl    (%eax)
fstpl   0x18(%esp)
```

[1] not really, see next slide

[2] native GCC, targeting i586

The Sweet Taste of Failure

```
#define GET_FIELD_VOLATILE(obj, offset, type_name, v) \
  oop p = JNIHandles::resolve(obj); \
  volatile type_name v =
    OrderAccess::load_acquire(
       (volatile type_name*)
       index_oop_from_field_offset_long(p, offset));
```

`GetDoubleVolatile()` actually compiles to:

```
mov     0x18(%esp),%ebp
add     %ebp,%eax
mov     0x4(%eax),%edx
mov     (%eax),%eax
mov     %eax,0x20(%esp)
mov     %edx,0x24(%esp)
mov     0x28(%esi),%esi
fldl    0x20(%esp)
mov     0x8(%esi),%eax
mov     0x4(%esi),%ebp
fstpl   0x18(%esp)
```

The Sweet Taste of Failure

```
#define GET_FIELD_VOLATILE(obj, offset, type_name, v) \
  oop p = JNIHandles::resolve(obj); \
  volatile type_name v =
    OrderAccess::load_acquire(
       (volatile type_name*)
       index_oop_from_field_offset_long(p, offset));
```

GetDoubleVolatile() actually compiles to:

```
mov    0x18(%esp),%ebp
add    %ebp,%eax
mov    0x4(%eax),%edx
mov    (%eax),%eax
mov    %eax,0x20(%esp)
mov    %edx,0x24(%esp)
mov    0x28(%esi),%esi
fldl   0x20(%esp)
mov    0x8(%esi),%eax
mov    0x4(%esi),%ebp
fstpl  0x18(%esp)
```

Tear My Heart Apart

We know the non-volatile longs/doubles are not guaranteed to be atomic. And other types?

```
          short s = 0;
---------------------------------
s = 0xFFFF; | short r1 = s;
```

Tear My Heart Apart

We know the non-volatile longs/doubles are not guaranteed to be atomic. And other types?

```
        short s = 0;
-----------------------------
s = 0xFFFF; | short r1 = s;
```

JLS/JMM requires $r1 \in \{0x0000, 0xFFFF\}$.

Tear My Heart Apart

We know the non-volatile longs/doubles are not guaranteed to be atomic. And other types?

```
        short s = 0;
-------------------------------
s = 0xFFFF; | short r1 = s;
```

JLS/JMM requires $r1 \in \{0x0000, 0xFFFF\}$.

And it empirically is!

Tear My Heart Apart, #2

```
                        short s = 0;
--------------------------------------------------------
s = 0xFFFF; |             short t = s;
            | byte r1 = ((t >> 0) & 0xFF);
            | byte r2 = ((t >> 8) & 0xFF);
```

Tear My Heart Apart, #2

```
                   short s = 0;
-----------------------------------------------
s = 0xFFFF; |         short t = s;
            | byte r1 = ((t >> 0) & 0xFF);
            | byte r2 = ((t >> 8) & 0xFF);
```

Intuitively:

$$(r1, r2) \in \{(0x00, 0x00), (0xFF, 0xFF)\}$$

Tear My Heart Apart, #2

```
                   short s = 0;
---------------------------------------------------
s = 0xFFFF;  |          short t = s;
             | byte r1 = ((t >> 0) & 0xFF);
             | byte r2 = ((t >> 8) & 0xFF);
```

Intuitively:

$$(r1, r2) \in \{(0x00, 0x00), (0xFF, 0xFF)\}$$

Empirically:

$$(r1, r2) \in \{..., (0x00, 0xFF), (0xFF, 0x00)\}$$

Tear My Heart Apart, #3

```
                  short s = 0;
---------------------------------------------------
s = 0xFFFF; |          short t = s;
            | byte r1 = ((t >> 0) & 0xFF);
            | byte r2 = ((t >> 8) & 0xFF);
```

Tear My Heart Apart, #3

```
                       short s = 0;
------------------------------------------------------
s = 0xFFFF; |              short t = s;
            | byte r1 = ((t >> 0) & 0xFF);
            | byte r2 = ((t >> 8) & 0xFF);
```

- C1 is unaffected, C2 is failing reliably
- the same result for byte/char/short fields

Tear My Heart Apart, #3

```
                     short s = 0;
--------------------------------------------------
s = 0xFFFF; |           short t = s;
            | byte r1 = ((t >> 0) & 0xFF);
            | byte r2 = ((t >> 8) & 0xFF);
```

- C1 is unaffected, C2 is failing reliably
- the same result for byte/char/short fields
- `volatile` s is not helping

Tear My Heart Apart, #4

```
short t = short_load(s.x);
r.r1 = byte_store(and(shift(t, 0), 0xFF)));
r.r2 = byte_store(and(shift(t, 8), 0xFF)));
```

Tear My Heart Apart, #4

```
short t = short_load(s.x);
r.r1 = byte_store(and(shift(t, 0), 0xFF)));
r.r2 = byte_store(and(shift(t, 8), 0xFF)));
```

...transforms to:

```
short t = short_load(s.x);
r.r1 = byte_store(t);
r.r2 = byte_store(shift(t, 8));
```

Tear My Heart Apart, #4

```
short t = short_load(s.x);
r.r1 = byte_store(and(shift(t, 0), 0xFF)));
r.r2 = byte_store(and(shift(t, 8), 0xFF)));
```

...transforms to:

```
short t = short_load(s.x);
r.r1 = byte_store(t);
r.r2 = byte_store(shift(t, 8));
```

...transforms to:

```
r.r1 = byte_store(unsigned_load(s.x));
r.r2 = byte_store(shift(signed_load(s.x), 8));
```

Tear My Heart Apart, #5

```
short t = s.x;
r.r1 = (byte) ((t >> 0) & 0xFF);
r.r2 = (byte) ((t >> 8) & 0xFF);
```

...compiles to:

```
; references: %rdx = $s; %rcx = $r
 movzwl 0xc(%rdx),%r11d       ; read s.x
 mov    %r11b,0xc(%rcx)       ; store r.r1
 movswl 0xc(%rdx),%r10d       ; read s.x again!
 shr    $0x8,%r10d            ; shift
 mov    %r10b,0xd(%rcx)       ; store r.r2
```

Tear My Heart Apart, #5

```
short t = s.x;
r.r1 = (byte) ((t >> 0) & 0xFF);
r.r2 = (byte) ((t >> 8) & 0xFF);
```

...compiles to:

```
; references: %rdx = $s; %rcx = $r
 movzwl 0xc(%rdx),%r11d       ; read s.x
 mov    %r11b,0xc(%rcx)       ; store r.r1
 movswl 0xc(%rdx),%r10d       ; read s.x again!
 shr    $0x8,%r10d            ; shift
 mov    %r10b,0xd(%rcx)       ; store r.r2
```

Kiss the atomicity bye-bye!

jcstress:

Try it. Use it. Break it.

Thanks!