The JVM is not observable enough
(and what to do about it)

Stephen Kell

stephen.kell@usi.ch

“University of Lugano”

joint work with: Danilo Ansaloni, Walter Binder, Lukáš Marek
This is a talk about Java bytecode instrumentation

- the Java platform’s *de facto* standard mechanism
- … for *observing* programs in execution
- (non-interactively, usually)
What

- profilers (JP2, ...)
- data race detectors (FastTrack, ...)
- white-box / active testing (jCUTE, ...)
- security monitors (TaintDroid, ...)
- memory / GC analyses (ElephantTracks, ...)
- ...

The JVM is... – p.3/20
How

Rewrite the bytecode, adding analysis “snippets”

■ on e.g. method entries, object allocations, locking, …

Can use libraries that help to munge bytecode

■ ASM, BCEL, Javassist, Soot, …

Or, some systems abstract the problem a bit more

■ Chord, DiSL, BTrace, RoadRunner, …
An “innocuous” example (using DiSL)

```java
public class TargetClass {
    public static void main(String[] args) {
        System.err.println("MAIN");
    }
}

public class DiSLClass {
    @Before(marker = BodyMarker.class, scope = "java.lang.Object.*")
    public static void onMethodExit(MethodStaticContext msc) {
        System.err.print(".");
    }
}
```
‘Typically, these alterations are to add “events” to the code of a method — for example, to add, at the beginning of a method, a call to MyProfiler.methodEntered(). Since the changes are purely additive, they do not modify application state or behavior.’

*Purely additive?*
Wishful thinking

Some questions:

- what problems occur writing tools this way?
- can we avoid them?
- what would be a better observation mechanism?

Answers: several; not really; let’s talk about it...
A summary of the difficulties

In the paper:

- deadlock between instrumentation and program
- state corruption by non-reentrant code
- method calls: unsafe but unavoidable
- “my instrumentation crashes the VM”
- instrumented bytecode that doesn’t verify
- coverage underapproximation (initializers, startup)
- coverage overapproximation (shared threads)
Deadlock

Thread-3
org.dacapo..$XalanWorker.run
org.apache.xml.dtm.DTManager.<init>
ch.usi.dag.disl.test.fia.....onFieldWrite
<locked ...MyWeakKeyIdentityHashMap:
java.lang.ClassLoader.getDeclaredFields
java.lang.ClassLoader.findClass
java.net.URLStreamHandler.getHostAddress
<wait to lock sun.net.www.....Handler:

Thread-2
org.dacapo..$XalanWorker.run
org.apache.xml.serializer.....run
java.net.URL.openConnection
<locked sun.net.www.....Handler:
sun.net.www.MessageHeader.grow
ch.usi.dag.disl.test.fia.....onFieldRead
ch.usi.dag.disl.test.fia.....registerIfNeeded
<wait to lock ...MyWeakKeyIdentityHashMap:

...MyWeakKeyIdentityHashMap
@0x....8950

sun.net.www.....Handler
@0x....2450

The JVM is... – p.9/20
Q. Can’t we just never share mutable state? (→ no locking)

A. Good idea. But

- this implies calling no methods
- … not even static ones
- does your analysis do I/O? (hint: yes)
Reentrancy example

```java
public class TargetClass {
    public static void main(String[] args) {
        System.err.println("MAIN");
    }
}
```

```java
public class DiSLClass {
    @Before(marker = BodyMarker.class, scope = "java.lang.Object.*")
    public static void onMethodExit(MethodStaticContext msc) {
        System.err.print("");
    }
}
```

Any guesses about the output?
The output

MAIN.MAIN
.
....
Non-reentrant code now called reentrantly

```java
package java.io;
class PrintStream {
    // ...
    void println () {
```
Non-reentrant code now called reentrantly

```java
package java.io;

class PrintStream {
   // ...

   void println () {
      // ...
      try {
         this.state = PENDING;
      }
   }
}
```
Non-reentrant code now called reentrantly

```java
package java.io;
class PrintStream {
    // ...

    void println () {
        // ...
        try {
            this.state = PENDING;

            while (pos != len) pos = copySome(in, out, pos, len);
        }
    }
}
```
Non-reentrant code now called reentrantly

```java
package java.io;

class PrintStream {
    // ...

    void println () {
        // ...
        try {
            this.state = PENDING;
            while (pos != len) pos = copySome(in, out, pos, len);
        } finally {
            assert this.state == PENDING; // FAILS following reentrant call!
            this.state = CLEAR;
        }
    }
}
```

The JVM is... – p.13/20
Q. Maybe just do your analysis in native code?

A. Okay, but

- (I thought you liked Java?)
- any library method might be implemented natively...
- and might call back into [instrumented] Java
- so sharing can still happen, unbeknownst to analysis

Less likely perhaps, but how to be safe?
A known approach we *could* borrow…

Valgrind, Pin, DynamoRIO et al:

- share neither state nor code with the observed program
- → private libraries (duplicate libc, etc.)
- → avoid signal handling, `wait()`, shared fds, …

We can do the same, at least from native code…

- maybe from Java too?
- … if can replicate down to `Object`, `ClassLoader` etc.

Problem: lost expressiveness!
Expressiveness lost

If we’re avoiding shared state, we can’t call any Java APIs:

- no reflection
- can’t call getters (→ field access instead)
- can’t observe even basic semantics (e.g. `equals()`)
- → can’t aggregate data using equality
- can’t synchronise

One consequence: can’t analyse *user-defined abstractions*

- including library-defined abstractions!
Aiming at something better

Wanted: keep the abstraction, but *add* isolation

- bytecode instrumentation (BCI) is an abstraction
- so far, we have made it “safe” by throwing it away
What’s the design space of observation?

- isolation: in-process (soft) versus out-of-process (hard)
- abstraction: VM-level (fixed) versus user-level (flexible)
- synchrony...

We have a weird asymmetric isolation requirement.

- observed is *not* influenced by observer
- observer *is* influenced by observed!
Isolated bytecode abstractions

Existing systems we can take inspiration from

- debugger expression eval (VM-style)
- debugger expression eval (native-style)
- Unix `fork()`
- shared memory (is asymmetric...)
- isolates, SIPs (MVM, Singularity)
- async assertions (Aftandilian & al, OOPSLA ’11)
- JIT purity analysis

Can we share the work with expression eval in debuggers?
Conclusions

Currently, bytecode instrumentors risk

- deadlock, reentrancy-derived corruption, …
- more in the paper!

We can only do things safely by

- trapping to a sharing-free environment ASAP
- avoid interacting with user-defined abstractions

This limits our expressiveness. Real solution:

- an asymmetric “isolated bytecode” abstraction
- might unify/replace a subset of JDWP too! (ask me)

Thanks for listening. Questions?