Dynamic analysis tools considered difficult
(to write)

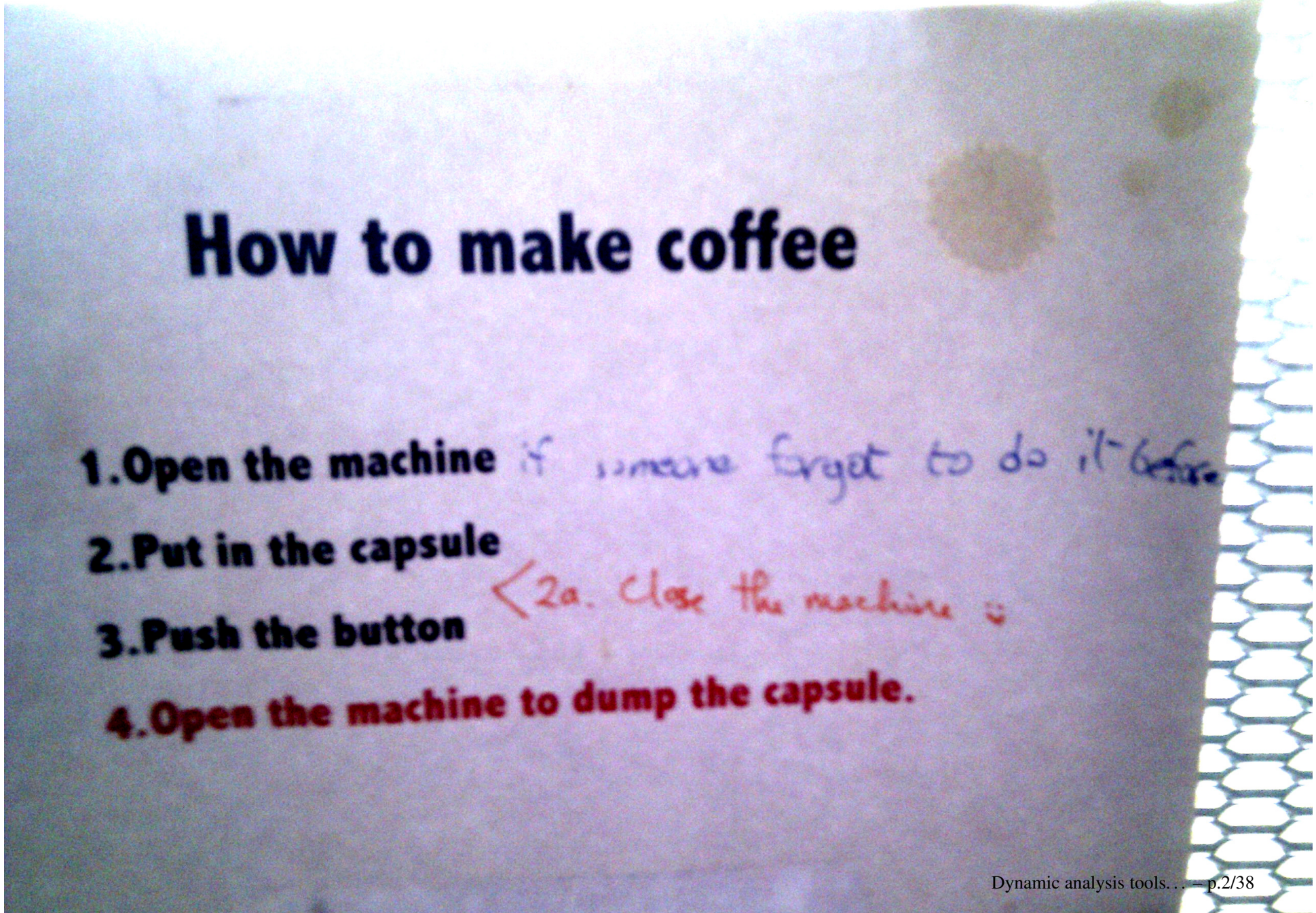
Stephen Kell

stephen.kell@usi.ch

University of Lugano

including joint work with
Danilo Ansaloni, Yudi Zheng, Walter Binder (U. Lugano)
Lubomír Bulej, Lukáš Marek, Petr Tůma (Charles University, Prague)

Programming is hard



Program analyses can help

Static analysis: analyse *all* executions

- infinitely many executions → need abstraction
- approximate statements…
- … about "the program"
- e.g. compiler reasoning, type checker, other verifiers…

Dynamic analyses: analyse a *single execution*

- precise statements…
- … about "the execution"
- e.g. profiler, debugger, Valgrind, other bugfinders…

Writing dynamic analyses

Straw poll: who here has written a dynamic analysis?

Writing dynamic analyses

Straw poll: who here has written a dynamic analysis?

What about something like this?

```
    /* ...  */
    x = 0;
} else {
    printf("DEBUG: something happened!  x is %d\n", x);
    /* ...  */
}
```

Instrumentation

You've just (manually) instrumented your program

- collect (retain) program state
- … for further processing
- somewhat intuitive

Most dynamic analyses are implemented this way

- often preferable to modifying runtime
- but: specify instrumentation programmatically

Programmatic instrumentation

Here you go:

```
static
IRSB* ar_instrument ( VgCallbackClosure* closure,
                      IRSB* sb_in,
                      VexGuestLayout* layout,
                      VexGuestExtents* vge,
                      IRType gWordTy, IRType hWordTy )
{
        // imperatively manipulate instructions
}
```

More friendly abstractions could help

Usually we *add* code before/after certain *features*:

- function/method calls and returns
- memory access
- synchronization operations
- loop back edges
- …

Want to exploit this for more declarative instrumentation

- can we borrow any existing approaches?

Aspect-oriented programming

AOP lets us *quantify* over execution events

- “pointcuts” are expressions capturing such events
- e.g. `call(void Point.setX(int))`

Then we can insert code to do some extra work

- `before(): System.out.println(”about to...”);`
- “weaving” splices the code in at compile- or load-time

DiSL redux

Existing AOP systems aren't optimal for instrumentation

- lack of join points e.g. basic blocks, instructions
- lack of coverage, e.g. inside core libraries
- overly dynamic semantics limit performance

DiSL is an AOP-inspired instrumentation tool for the JVM

- good coverage ($\approx$ "full bytecode coverage"; more later)
- good performance
- open joint point model

Centres on a "Java-hosted" domain-specific language…

Trivial example

```
@Before(marker=BodyMarker.class, scope="Point.setX(int)")
static void mySnippet( ) {
    System.out.println( "about to ... " );
}
```

Bigger example: allocation counter

```
@AfterReturning(marker = BytecodeMarker.class, args = "new")
public static void beforeAlloc(MethodStaticContext ma,
  DynamicContext dc)
{   Analysis.instance(). onObjectInitialization (
      dc.getStackValue(0, Object.class), // allocated object
      ma.getAllocationSite()
    );
}
```

+ similar for other bytecodes newarray, multinewarray, ...

Parvum in multo

How can I build an analysis a bit like yours?

- answer: copy–paste, of course

```
From:  Nicholas Nethercote
Date:  Thu, 10 Mar 2011 14:17:26 -0800

(snip)

Really, I think the easiest way to do these things is
to just modify Memcheck.
```

Composition and decomposition

DiSL-style snippets don't compose or decompose easily

- snippet and its quantifier (annotations) are one unit
- snippet is opaque Java code – could do anything
- hands off through user-defined inteface
- no ready-made abstractions of common-case structures
- snippet-based design defeats Java inheritance
- still bad at the things Java is bad at

Let's be FRANC (1)

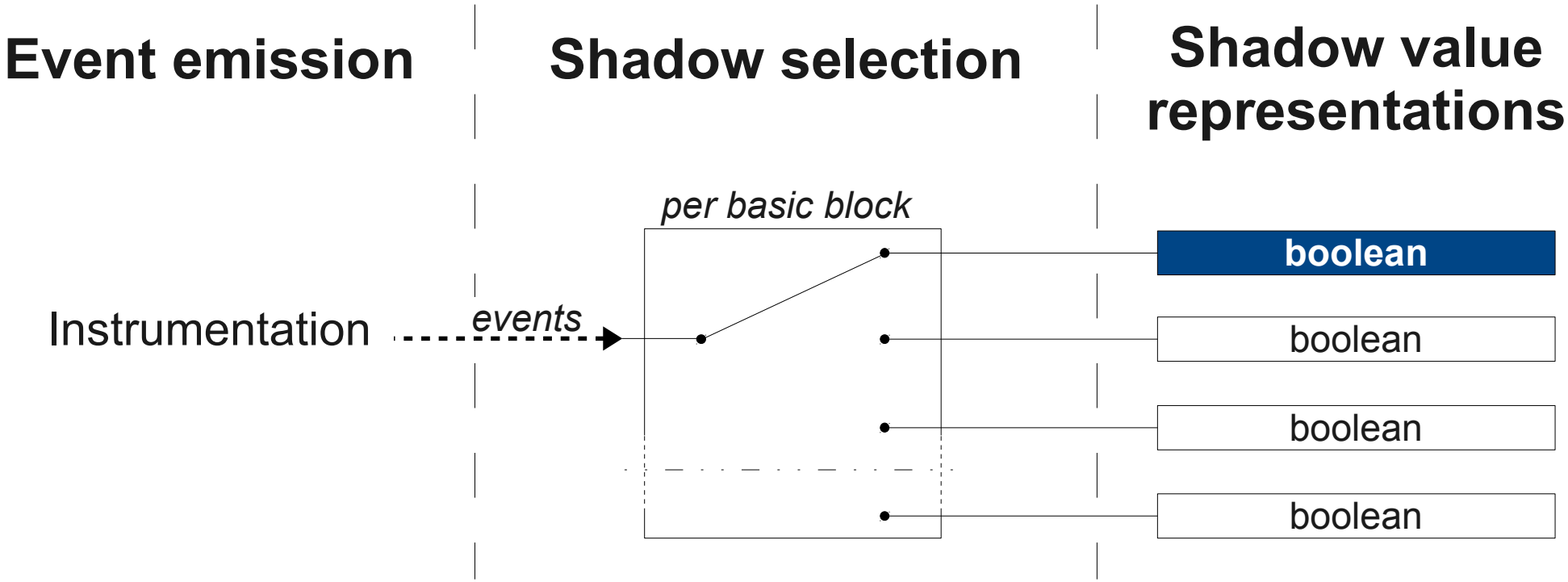
FRANC is a system for analysis *composition*

- observation: analyses update *state* in reaction to *events*
- let's build abstractions at this level, instead of snippets!

FRANC decomposes analyses using the following equation.

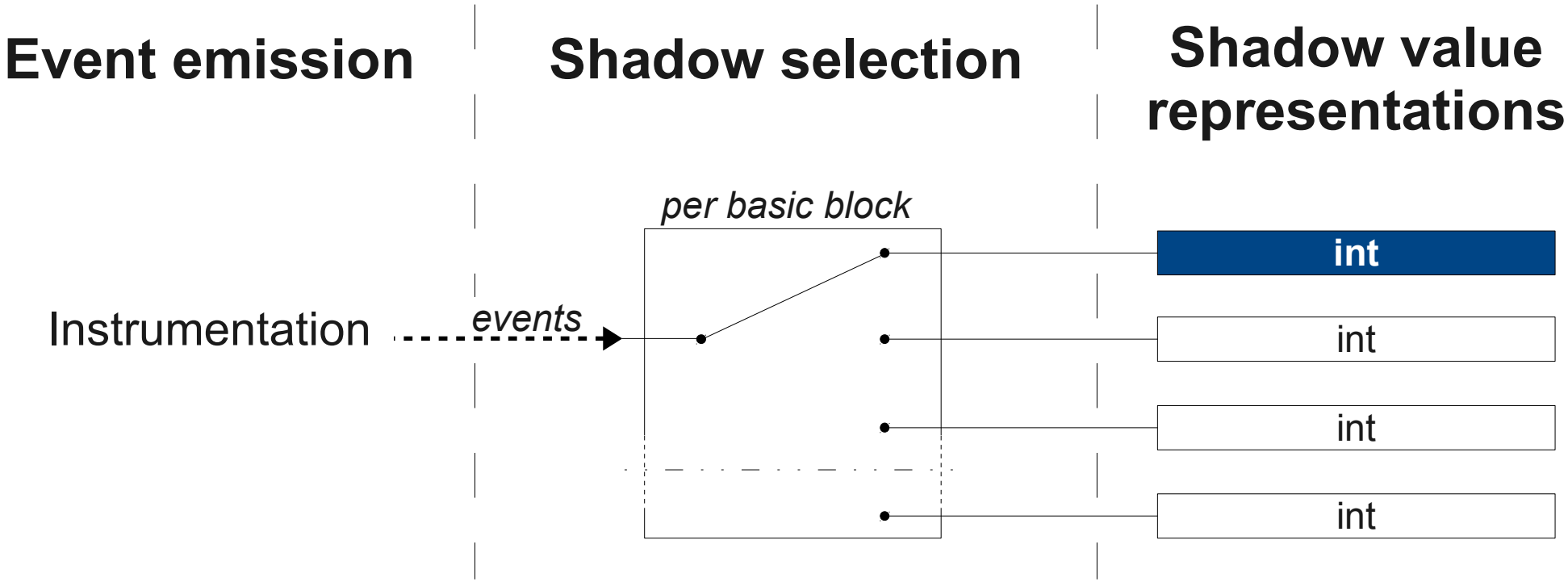
$$Analysis = Instrumentation + ShadowMapping + ShadowValues$$

Let's be FRANC (2)



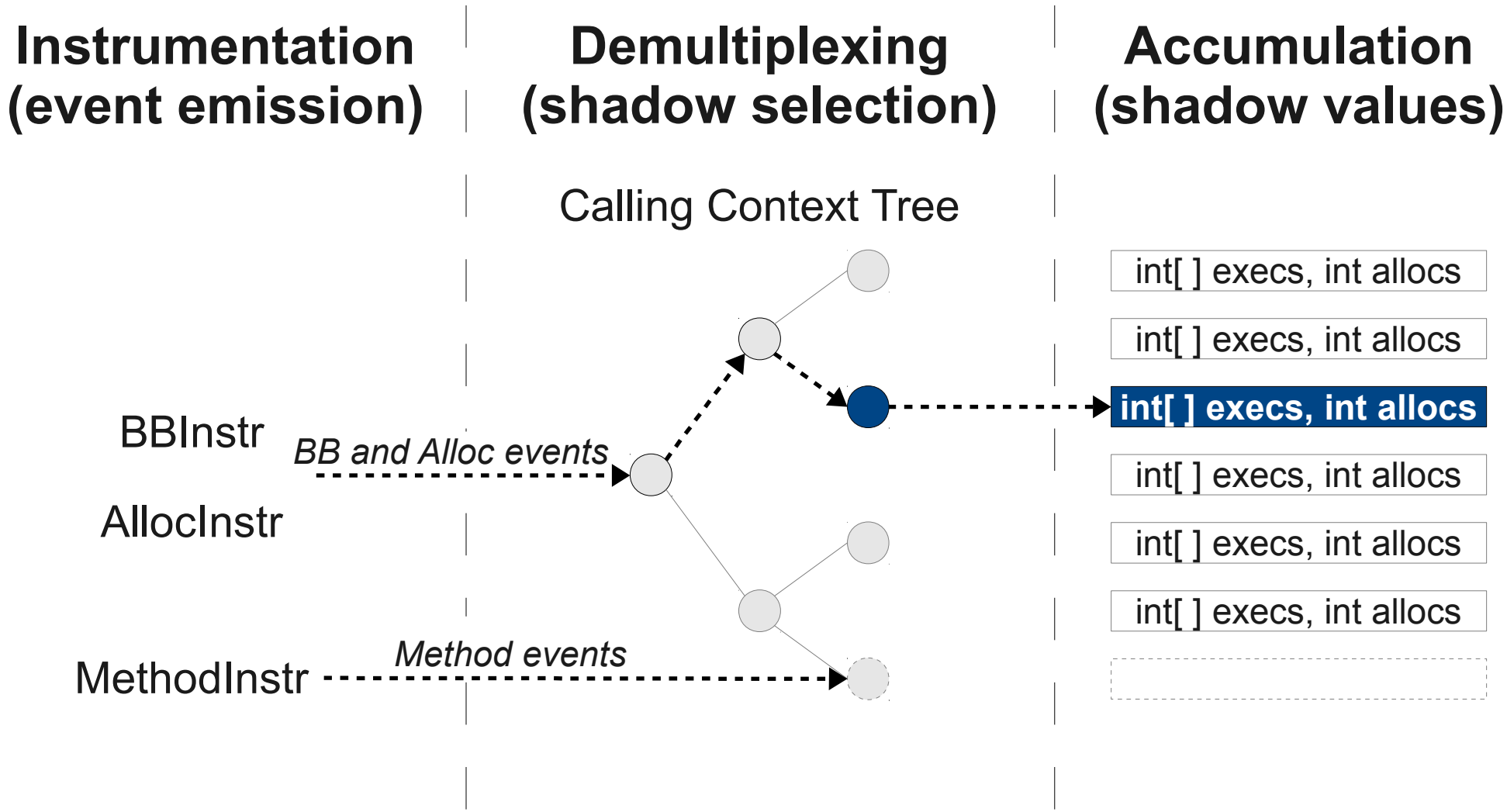
This is a basic block coverage tool.

Let's be FRANC (3)



Now it's a basic block hotness profiler.

Let's be FRANC (4)



Now it's a context-sensitive hotness and allocation profiler…

FRANC example: counting field accesses

```
Map<String, AtomicLong> fieldAccesses = new ShadowMap<>(...);

class FieldMapper extends ThreadLocal<String>
  implements AfterCompletion<FieldAccess> {
    public void afterCompletion(FieldAccess codeRegion)
    { set(FieldAccessContext.getFullFieldName(codeRegion)); }
}
FieldMapper currentField = new FieldMapper();

Analysis<FieldAccess> updater = new PostIncrement<>(fieldAccesses, c

FRANC.deploy(FRANC.complete(currentField, updater));
```

FRANC design summary

- event-based programming model
- instrumentation produces events
- “mappers” group events spatially
- shadow value *updaters* aggregate events over time

Both mappers and updaters consume events

- e.g. CCT consumes method call/return events…
- separately, shadow values consume BB events
- CCT “routes” BB events to the relevant counter
- wart: processing order still specified manually

FRANC results

Improvements:

- FRANC allows library-based analysis development
-
- event sources, mappers, shadow values

Performance redux:

- a bit slower than manual DiSL, but not too much
- typically 25–30% additional overhead

More detail, case studies etc. in forthcoming ECOOP paper

The trouble with Java

Java is a simple language, but JVM is *very* complex.

Remember this guy?

```
@AfterReturning(marker = BytecodeMarker.class, args = "new")
public static void beforeInitialization (MethodStaticContext ma,
  DynamicContext dc)
{ /* ... */ }
```

To record all the memory allocations, you need to

- add two more snippets (each with subtleties)
- implement JVMTI's VMObjectAlloc hook
- add some JNI function interposition

... and even then, your picture is incomplete

An “innocuous” example (using DiSL)

```
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(".");
}   }
```

A choice quotation

(from http://docs.oracle.com/javase/6/docs/technotes/guides/jvmti/)

> ‘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

We would instrument *all* the bytecode in our program, but:

- bootstrapping problems
- interference problems

Can we avoid them?

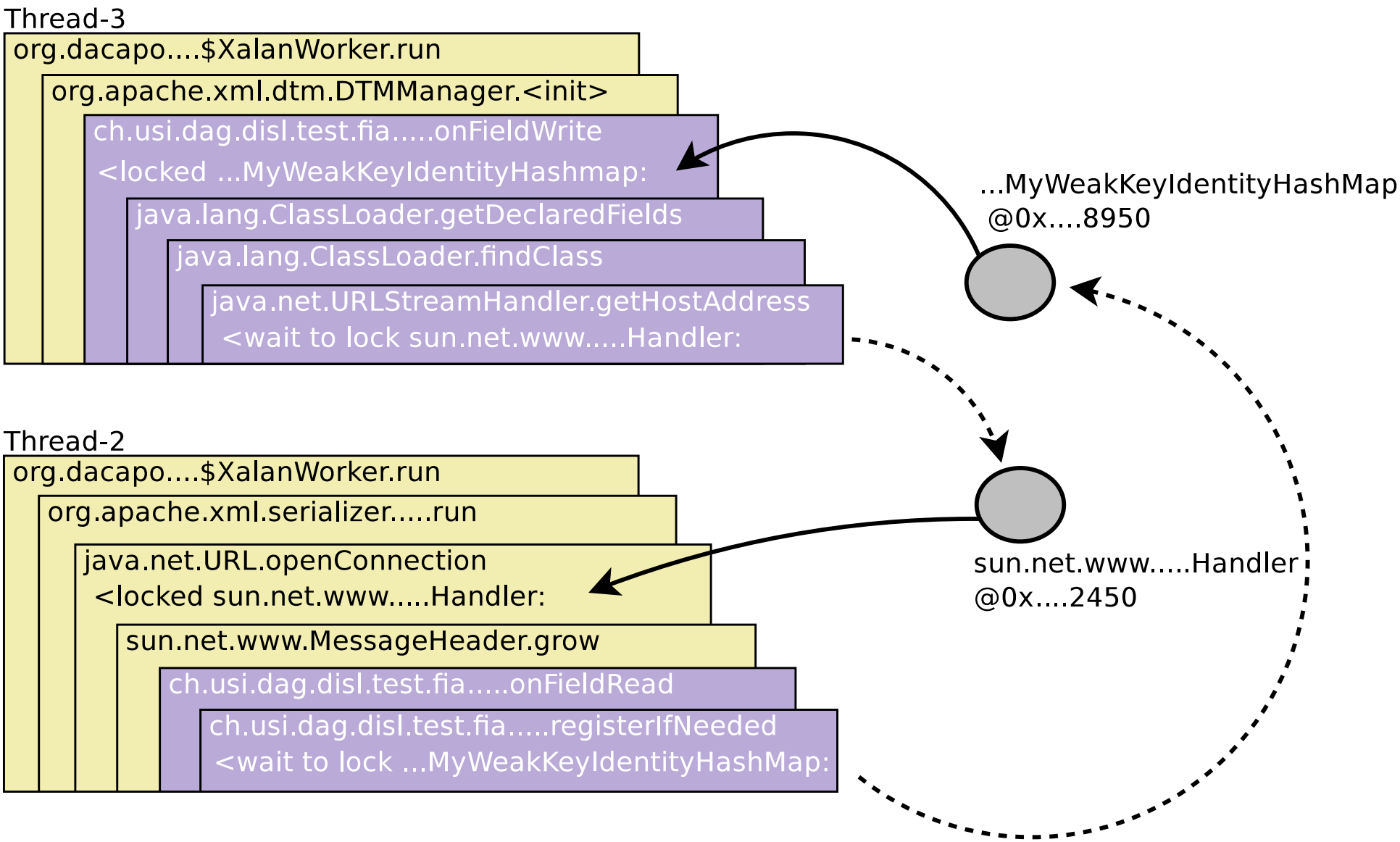
If not, what would be a better observation mechanism…

- …than plain old instrumentation?

A summary of the difficulties

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

Deadlock


Thread-3
org.dacapo....$XalanWorker.run
org.apache.xml.dtm.DTMManager.<init>
ch.usi.dag.disl.test.fia.....onFieldWrite
<locked ...MyWeakKeyIdentityHashmap:
java.lang.ClassLoader.getDeclaredFields
java.lang.ClassLoader.findClass
java.net.URLStreamHandler.getHostAddress
<wait to lock sun.net.www.....Handler:
...MyWeakKeyIdentityHashMap
@0x....8950
Thread-2
org.dacapo....$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:
sun.net.www.....Handler
@0x....2450

Attempted escape (1): share no mutable state!

Q. Can't we just *share no mutable state*? ($\approx$ avoid locking)

A. Good idea. But

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

(Reminder: this would be okay …
if we weren't instrumenting libraries too.)

Reentrancy example

```
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(".");
}   }
```

Any guesses about the output?

The output

```
.....................................................
MAIN.MAIN

.

....
```

Non-reentrant code now called reentrantly

```
package java.io;
class PrintStream {
//  ...
void  println ()  {
```

Non-reentrant code now called reentrantly

```
package java.io;
class PrintStream {
//  ...
void  println ()  {
   //  ...
   try  {
      this.state  = COPYING;
```

Non-reentrant code now called reentrantly

```
package java.io;
class PrintStream {
//  ...
void  println ()  {
   //  ...
   try  {
      this.state  = COPYING;
     while (pos != len)  pos = copySome(in, out, pos, len);
```

Non-reentrant code now called reentrantly

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

Attempted escape (2): use native code?

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 can have unwitting sharing in native libraries!

Less likely perhaps…

- (ask me about some ongoing work on this theme)

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 ($\rightarrow$ field access instead)
- can't observe even basic semantics (e.g. `equals()`)
- $\rightarrow$ can't aggregate data using equality
- can't synchronise

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

- including library-defined abstractions!

Bootstrap coverage

Can we observe JVM execution from the first bytecode?

- short answer: no (we currently believe)
- longer: Object and Class are special-cased
- ...

Contrast: "early injection" for native instrumentation:

- can successfully trap before first instruction!
- Valgrind: write your own loader
- Pin: clever use of fork() and ptrace()

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?

(To be continued. . .)

Conclusions

Writing dynamic analyses is hard on a number of levels:

- inadequate programming abstractions
- inadequate infrastructure

But all is not lost!

- higher-level programming models
- (hypothetically) tweaked infrastructure

Thanks for listening. Questions?