# Java Instrumentation for Dynamic Analysis

Jeff Perkins and Michael Ernst

MIT CSAIL

# Java Instrumentation Approaches

- Instrument source files
- Java Debug Interface (JDI)
- Instrument class files

# Instrument source files

- Advantages
  - Instrumentation code is easy to create and understand
  - Easy to debug
- Disadvantages
  - Source is complex to parse (inner classes, generics)
  - Source format changes often (e.g, 1.1 - 1.5)
  - Complex to use (user must manage multiple copies of source and class files)

# Java Debug Interface

- Advantages
  - Easy to implement
  - Doesn’t change target
  - Easy to use
- Disadvantages
  - Very slow.
  - Interface has changed often

# Instrument class files

- Advantages
  - Easy to use
  - Class file format is stable
  - Good editing tools exist
  - Efficient
- Disadvantages
  - Building instrumentation in bytecodes is cumbersome

# Java Virtual Machine (JVM) Instruction Set

- Opcodes implement a straightforward stack machine
- Code is not optimized (1-1 relationship with source)
- Each method has a stack frame.
- Each parameter/local has a slot in the stack frame
- Example instructions
  - load/store
  - put/getfield
  - invoke
  - etc.

# Class File format

- Constant pool
  - Symbolic information is stored in the constant pool
  - Instructions reference items in the pool
- Each Method in the class
  - Code
  - Local Variable Table (optional)
  - Line Number table (optional)
- Inner classes are in separate files

# Instrumentation Mechanisms

- Static translation
  - Class files on the disk are translated
  - Doesn't work with class file that are not available
  - Somewhat more difficult to use
- Class loader
  - New class loader delegates to the original class loader to get the original class file
  - Target program class loaders require special support
- Pre-agent
  - Instrumentation code is passed each class as it is loaded
  - New classes cannot be created

# Tools

- ASM (http://asm.objectweb.org/)
  - Small and fast
  - Supports Java 1.5
- BCEL (http://jakarta.apache.org/bcel/)
  - Works well
  - No longer under active development
- SERP (http://serp.sourceforge.net/)
  - Not designed for speed
  - Somewhat memory intensive
- Soot (http://www.sable.mcgill.ca/soot/)
  - Supports source
  - Designed for optimizations

# Example source and instructions

- Source

```
public static int sum (int x, int y) {    return (x + y);  }
```

- Bytecodes

```
iload 0    // load first argument  iload 1    // load second argument  iadd       // Add operands on top of stack  iret       // return the sum
```

# Example entry trace (source)

```
public static int sum (int x, int y) {
  trace ("sum", new Object[] {x, y});
  return (x + y);
}
```

# Example entry trace (bytecodes)

- trace ("sum", new Object[] {x, y});
- Bytecodes

```
ldc "sum"          // push name of method  iconst_2          // push number of args  anewarray [java.lang.obj]  // create an array for the args  dup          // du...
```

# Example exit trace

- Source

```
int result = x+y;  trace ("sum", result);  return (result);
```

- Bytecodes

```
... ldc "sum"            // push name of method  istore 2            // store result in new local  new [java.lang.Integer]    // Create wrapper  dup                // dup w...
```

# Verification

- JVM checks each class as it is loaded
- Checks include
  - Classfile format must be valid
  - Jumps must have valid targets
  - References to constant pool must be valid
  - Each instruction requires appropriate types on the stack and in the local variable array
  - Only initialized values can be accessed
  - Stack must not underflow or overflow
- JVM verifier doesn't detail the exact location of errors
- BCEL verifier gives more information, but complains about valid code

# JDK Issues

- Instrumenting the JDK is often required
  - Heap Analysis
  - Profiling
  - Comparability
- JDK instrumentation problems
  - The instrumentation itself should not use instrumented classes
    - If a heap analysis used an instrumented version of HashMap it would recursivly call itself
  - Some classes (200) are preloaded and can't be instrumented dynamically.
- Avoid when possible

# JDK Approaches

- New rt.jar file (static)
  - Can’t be changed per target program
  - Calls or classes must be doubled to avoid instrumenting the instrumentation
- Twin Class Hierarchy
  - Duplicates all JDK classes
  - Difficult to implement
  - doesn’t handle native calls well.

# Instrumenting rt.jar

- Some classes cannot be modified
  - Object cannot be changed at all
  - Fields in String, Class, and others cannot be modified
- Methods can be added
- Difficult to debug
  - JVM exits with machine level dump
  - Slow turnaround

# Doubling Methods Approach

- Algorithm
  - Create a new copy of each method
  - Add a distinguishing parameter to the copy
  - Instrument the copy
  - Change user code to reference the instrumented copies
- Instrumentation uses original code
- Handle fields by storing information externally
- Native calls work well
- Parameter can cause problems with reflection
- Class initializers cannot use this mechanism

# Twin Class Hierarchy

- Factor, Schuster, and Shagin -- OOPSLA 2004
- Parallel isomorphic instrumented class hierarchy

```
Object     TCH.Object      TCH.String
```

- User code is modified to reference the new classes
- Many complex issues
  - reflection
  - arrays
  - serialization
  - root class (Object)
- Native calls require hand written solutions

# Hints

- Do as little as possible as direct instrumentation. Call a java method instead.
- Reflection calls are surprisingly fast
- Pay attention to the different sizes of types
- Locals are easy to create and use. They help with stack level instrumentation
- `javap` dumps class file contents
- Dump original classes and new versions when instrumenting

# Conclusion

- Java instrumentation tools are powerful
- JDK problems can be resolved
- Java instrumentation is surprisingly easy to do