Program Dynamic Analysis

Overview

- Dynamic Analysis
- JVM & Java Bytecode [2]
- A Java bytecode engineering library: ASM [1]

What is dynamic analysis? [3]

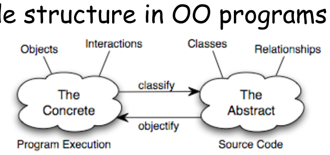
- The investigation of the properties of a running software system over one or more executions

Has anyone done dynamic analysis? [3]

- Loggers
- Debuggers
- Profilers
- …

Why dynamic analysis? [3]

- Gap between run-time structure and code structure in OO programs



Trying to understand one [structure] from the other is like trying to understand the dynamism of living ecosystems from the static taxonomy of plants and animals, and vice-versa.

-- Erich Gamma et al., Design Patterns

Why dynamic analysis?

- Collect runtime execution information
  - Resource usage, execution profiles
- Program comprehension
  - Find bugs in applications, identify hotspots
- Program transformation
  - Optimize or obfuscate programs
  - Insert debugging or monitoring code
  - Modify program behaviors on the fly

How to do dynamic analysis?

- Instrumentation
  - Modify code or runtime to monitor specific components in a system and collect data
  - Instrumentation approaches
    - Source code modification
    - Byte code modification
    - VM modification
- Data analysis

7

A Running Example

- Method call instrumentation
  - Given a program's **source code**, how do you modify the code to record which method is called by main() in what order?

```
public class Test {
    public static void main(String[] args) {
        if (args.length == 0) return;
        if (args.length % 2 == 0) printEven();
        else printOdd();
    }
    public static void printEven() {System.out.println("Even");}
    public static void printOdd() {System.out.println("Odd");}
}
```

8

Source Code Instrumentation

- Call site instrumentation
  - Call print(…) before each actual method call
- Method entry instrumentation
  - Call print(…) at entry of each method

9

Method Entry Instrumentation

```
public class Test {
    public static void main(String[] args) {
        if (args.length == 0) return;
        if (args.length % 2 == 0) printEven();
        else printOdd();
    }
    public static void printEven() {
        System.out.println("printEven() is called");
        System.out.println("Even");
    }
    public static void printOdd() {
        System.out.println("printOdd() is called");
        System.out.println("Odd");
    }
}
```

10

Call Site Instrumentation

```
public class Test {
    public static void main(String[] args) {
        if (args.length == 0) return;
        if (args.length % 2 == 0) {
            System.out.println("printEven() is called");
            printEven();
        } else {
            System.out.println("printOdd() is called");
            printOdd();
        }
    }
    public static void printEven() {System.out.println("Even");}
    public static void printOdd() {System.out.println("Odd");}
}
```

11

Method entry vs. Call site

12

Can you do instrumentation automatically?

13

People also do byte code instrumentation, because

- Source code is not needed, so transformations can be used on applications with closed source and commercial applications
- Code can be weaved in at runtime transparently to users
- Why source code?

14

Tools for Program Analysis and Transformation

- ASM
  - Class generation and transformation based on byte code
- Soot
  - Program analysis and transformation framework based on byte code
- WALA
  - Program analysis and transformation framework based on source code of Java and Javascript, and byte code of Java

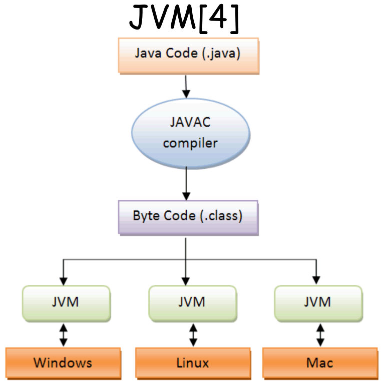
15

Java Virtual Machine (JVM)

- A "virtual" computer that resides in the "real" computer as a software process
- Java byte code is the instruction set of the JVM
- It gives Java the flexibility of platform independence

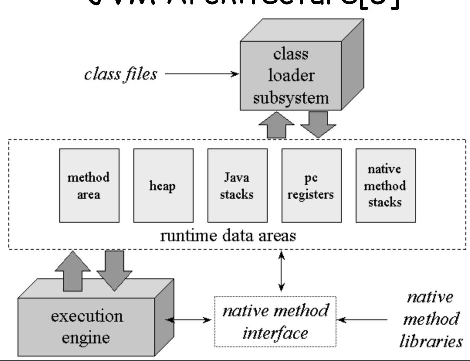
16

JVM[4]



17

JVM Architecture[5]



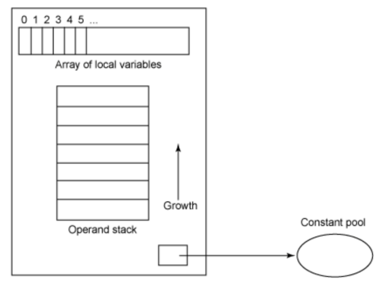
18

Java Stack

- JVM is a stack-based machine
  - Each thread has a JVM stack which stores frames
  - A frame is created each time a method is invoked, including
    - an operand stack,
    - an array of local variables, and
    - a reference to the runtime constant pool
  - Operations are carried out by popping data from the stack, processing them, and pushing back the results

19

Frame Structure



20

Method Area

- This is the area where byte code resides
- The program counter (PC) points to some byte in the method area
- It always keeps tracks of the current instruction which is being executed (interpreted)
- After execution of an instruction, the JVM sets the PC to next instruction
- Method area is shared among all threads of the process

21

Garbage-collected Heap

- It is where the objects in Java programs are stored
- Java does not have free operator to free any previously allocated memory
- Java frees useless memory using Garbage collection mechanism

22

Execution Engine

- Execute byte code directly or indirectly
  - Interpreter
    - Interpret/read the code and execute accordingly
    - Start fast without compilation
  - Just-in-time (JIT) compilers
    - Translate to machine code and then execute
    - Start slow due to compilation

23

Execution Engine

- Adaptive optimization
  - Performs dynamic recompilation of portions of a program based on the current execution profile
  - Make a trade-off between just-in-time compilation and interpreting instructions
    - E.g., method inlining

24

Java Byte Code

- Each instruction consists of a one-byte opcode followed by zero or more operands
  - "iadd": receives two integers as operands and adds them together.

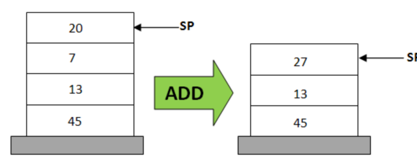
Seven Types of Instructions

1. Load and store
   - aload_0, istore
2. Arithmetic and logic
   - ladd, fcmpl
3. Type conversion
   - i2b, d2i
4. Object creation and manipulation
   - new, putfield

Seven Types of Instructions

5. Operand stack management
   - swap, dup2
6. Control transfer
   - ifeq, goto
7. Method invocation and return
   - invokespecial, areturn

Example: iadd



Instrumentation in byte code

- System.out.println("printEven() is called")

```
getstatic     #16     //Field java/lang/System/out:Ljava/io/PrintStream;
ldc           #22     //Load String "printEven() is called"
invokevirtual #24     //Method java/io/PrintStream.println: (Ljava/lang/
                      String;)V
```


How to manipulate byte code with ASM?

- Using ClassReader to read from a class file
- Using ClassWriter to write to a class file
- Put new declared ClassVisitor(s) between them to rewrite bytecode as needed

Interface ClassVisitor

- A visitor to visit a Java class
- The visit methods are invoked in the following order:
  - visit [visitSource] [visitOuterClass] (visitAnnotation | visitAttribute)* (visitInnerClass | visitField | visitMethod)* visitEnd.

Interface MethodVisitor

- A visitor to visit a Java method
- The visit methods are invoked in the following order:
  - [visitAnnotationDefault] (visitAnnotation | visitParameterAnnotation | visitAttribute)* [visitCode (visit*X*Insn | visitLabel | visitTryCatchBlock | visitLocalVariable | visitLineNumber)* visitMaxs] visitEnd.

Class File Instrumentation

```
public class Instrumenter {
    public static void main(final String args[]) throws Exception {
        FileInputStream is = new FileInputStream(args[0]);
        byte[] b;
        ClassReader cr = new ClassReader(is);
        ClassWriter cw = new
ClassWriter(ClassWriter.COMPUTE_FRAMES);
        ClassVisitor cv = new ClassAdapter(cw);
        cr.accept(cv, 0);
        b = cw.toByteArray();
        FileOutputStream fos = new FileOutputStream(args[1]);
        fos.write(b);
        fos.close();
    }
}
```


Class Rewriting

```
class ClassAdapter extends ClassVisitor implements Opcodes {

    public ClassAdapter(final ClassVisitor cv) {
        super(ASM5, cv);
    }

    @Override
    public MethodVisitor visitMethod(final int access, final String name,
            final String desc, final String signature, final String[] exceptions) {
        MethodVisitor mv = cv.visitMethod(access, name, desc, signature,
exceptions);
        return mv == null? null: new MethodAdapter(mv, name);
    }
}
```


Method Rewriting – Method Entry

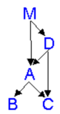
```
class MethodAdapter extends MethodVisitor implements Opcodes {
    String name;
    public MethodAdapter(final MethodVisitor mv, String name) {
        super(ASM5, mv);
        this.name = name;
    }
    @Override
    public void visitCode() {
        mv.visitFieldInsn(GETSTATIC, "java/lang/System", "out",
"Ljava/io/PrintStream;");
        mv.visitLdcInsn(name + " is called");
        mv.visitMethodInsn(INVOKEVIRTUAL, "java/io/
PrintStream", "println", "(Ljava/lang/String;)V", false);
        mv.visitCode();
    }
}
```


Method Rewriting - CallSite

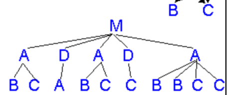
```
@Override
public void visitMethodInsn(int opcode, String owner, String name,
String desc, boolean itf) {
    mv.visitFieldInsn(GETSTATIC, "java/lang/System", "err",
"Ljava/io/PrintStream;");
    mv.visitLdcInsn(name + " is called");
    mv.visitMethodInsn(INVOKEVIRTUAL, "java/io/PrintStream",
"println", "(Ljava/lang/String;)V", false);
    mv.visitMethodInsn(opcode, owner, name, desc, itf);
}
```

With a method call trace, we can create

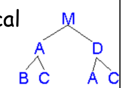
- Call graph
  - Each method corresponds to a node
  - No context sensitivity



- Call tree
  - Context sensitivity



- Calling context tree
  - Collapse nodes with same hierarchical context



37

With instrumentation, we can collect more information…

- Execution path
- Statement coverage
- Method input/output values
- Read/write access of variables

38

Reference

[1] Eric Bruneton, ASM 4.0
A Java bytecode engineering library,
http://download.forge.objectweb.org/asm/asm4-guide.pdf
[2] Instrumenting Java Bytecode with ASM,
http://web.cs.ucla.edu/~msb/cs239-tutorial/
[3] Orla Greevy & Adrian Lienhard, Analyzing Dynamic Behavior
https://www.iam.unibe.ch/scg/svn_repos/Lectures/OORPT/12DynamicAnalysis.ppt.
[4] Viral Patel, Java Virtual Machine, An inside story!!,
http://viralpatel.net/blogs/java-virtual-machine-an-inside-story/
[5] Bill Venners, The Java Virtual Machine, http://www.artima.com/insidejvm/ed2/jvm2.html

39