

Using Soot to instrument a class file

Feng Qian

December 17, 2004

1 Goals

The purpose of this tutorial is to let you know:

1. how to inspect a class file by using Soot, and
2. how to profile a program by instrumenting the class file.

2 Illustration by examples

I am going to show an example first. From the example, you can get some feelings about how to modify a class file using Soot. Then I will explain internal representations of classes, methods, and statements in Soot.

In this tutorial, I am making an example slightly different from the example `GotoInstrumenter` on the web¹. You should also check the old tutorial to learn how to add local variables and fields, etc...

Task : count how many `InvokeStatic` instructions executed in a run of a tiny benchmark `TestInvoke.java` :

```
class TestInvoke {
    private static int calls=0;
    public static void main(String[] args) {

        for (int i=0; i<10; i++) {
            foo();
        }

        System.out.println("I made "+calls+" static calls");
    }
}
```

¹<http://www.sable.mcgill.ca/soot/tutorial/profiler/>

```

private static void foo(){
    calls++;
    bar();
}

private static void bar(){
    calls++;
}
}

```

In order to record counters, I wrote a helper class called `MyCounter` `MyCounter.java`

```

/* The counter class */
public class MyCounter {
    /* the counter, initialize to zero */
    private static int c = 0;

    /**
     * increases the counter by <pre>howmany</pre>
     * @param howmany, the increment of the counter.
     */
    public static synchronized void increase(int howmany) {
        c += howmany;
    }

    /**
     * reports the counter content.
     */
    public static synchronized void report() {
        System.err.println("counter : " + c);
    }
}

```

Now, I am creating a wrapper class to add a phase in Soot for inserting profiling instructions, then call `soot.Main.main()`. The main method of this driver class adds a transformation phase named ‘‘jtp.instrumenter’’ to Soot’s ‘‘jtp’’ pack. The `PackManager` class the various Packs of phases of Soot. When `MainDriver` makes a call to `soot.Main.main`, Soot will know from `PackManager` that a new phase was registered, and the `internalTransform` method of new phase will be called by Soot. `MainDriver.java` :

```

1  /* Usage: java MainDriver [soot-options] appClass
2     */
3

```

```

4  /* import necessary soot packages */
5  import soot.*;
6
7  public class MainDriver {
8      public static void main(String[] args) {
9
10         /* check the arguments */
11         if (args.length == 0) {
12             System.err.println("Usage: java MainDriver [options] classname");
13             System.exit(0);
14         }
15
16         /* add a phase to transformer pack by call Pack.add */
17         Pack jtp = PackManager.v().getPack("jtp");
18         jtp.add(new Transform("jtp.instrumenter",
19                               new InvokeStaticInstrumenter()));
20
21         /* Give control to Soot to process all options,
22          * InvokeStaticInstrumenter.internalTransform will get called.
23          */
24         soot.Main.main(args);
25     }
26 }

```

The real implementation of instrumenter extends an abstract class `BodyTransformer`. It implements the `internalTransform` method which takes a method body (instructions) and some options. The main operations happen in this method. Depends on your command line options, Soot builds a list of classes (which means a list of methods also) and calls *`InvokeStaticInstrumenter.internalTransform`* by passing in the body of each method. *`InvokeStaticInstrumenter.java`* :

```

1  /*
2   * InvokeStaticInstrumenter inserts count instructions before
3   * INVOKESTATIC bytecode in a program. The instrumented program will
4   * report how many static invocations happen in a run.
5   *
6   * Goal:
7   *   Insert counter instruction before static invocation instruction.
8   *   Report counters before program's normal exit point.
9   *
10  * Approach:
11  *   1. Create a counter class which has a counter field, and
12  *      a reporting method.
13  *   2. Take each method body, go through each instruction, and

```

```

14  *      insert count instructions before INVOKESTATIC.
15  *      3. Make a call of reporting method of the counter class.
16  *
17  * Things to learn from this example:
18  *      1. How to use Soot to examine a Java class.
19  *      2. How to insert profiling instructions in a class.
20  */
21
22  /* InvokeStaticInstrumenter extends the abstract class BodyTransformer,
23  * and implements <pre>internalTransform</pre> method.
24  */
25  import soot.*;
26  import soot.jimple.*;
27  import soot.util.*;
28  import java.util.*;
29
30  public class InvokeStaticInstrumenter extends BodyTransformer{
31
32      /* some internal fields */
33      static SootClass counterClass;
34      static SootMethod increaseCounter, reportCounter;
35
36      static {
37          counterClass    = Scene.v().loadClassAndSupport("MyCounter");
38          increaseCounter = counterClass.getMethod("void increase(int)");
39          reportCounter   = counterClass.getMethod("void report()");
40      }
41
42      /* internalTransform goes through a method body and inserts
43      * counter instructions before an INVOKESTATIC instruction
44      */
45      protected void internalTransform(Body body, String phase, Map options) {
46          // body's method
47          SootMethod method = body.getMethod();
48
49          // debugging
50          System.out.println("instrumenting method : " + method.getSignature());
51
52          // get body's unit as a chain
53          Chain units = body.getUnits();
54
55          // get a snapshot iterator of the unit since we are going to

```

```

56      // mutate the chain when iterating over it.
57      //
58      Iterator stmtIt = units.snapshotIterator();
59
60      // typical while loop for iterating over each statement
61      while (stmtIt.hasNext()) {
62
63          // cast back to a statement.
64          Stmt stmt = (Stmt)stmtIt.next();
65
66          // there are many kinds of statements, here we are only
67          // interested in statements containing InvokeStatic
68          // NOTE: there are two kinds of statements may contain
69          //      invoke expression: InvokeStmt, and AssignStmt
70          if (!stmt.containsInvokeExpr()) {
71              continue;
72          }
73
74          // take out the invoke expression
75          InvokeExpr expr = (InvokeExpr)stmt.getInvokeExpr();
76
77          // now skip non-static invocations
78          if (!(expr instanceof StaticInvokeExpr)) {
79              continue;
80          }
81
82          // now we reach the real instruction
83          // call Chain.insertBefore() to insert instructions
84          //
85          // 1. first, make a new invoke expression
86          InvokeExpr incExpr= Jimple.v().newStaticInvokeExpr(increaseCounter,
87                                                              IntConstant.v(1));
88          // 2. then, make a invoke statement
89          Stmt incStmt = Jimple.v().newInvokeStmt(incExpr);
90
91          // 3. insert new statement into the chain
92          //      (we are mutating the unit chain).
93          units.insertBefore(incStmt, stmt);
94      }
95
96
97      // Do not forget to insert instructions to report the counter

```

```

98      // this only happens before the exit points of main method.
99
100     // 1. check if this is the main method by checking signature
101     String signature = method.getSubSignature();
102     boolean isMain = signature.equals("void main(java.lang.String[])");
103
104     // re-iterate the body to look for return statement
105     if (isMain) {
106         stmtIt = units.snapshotIterator();
107
108         while (stmtIt.hasNext()) {
109             Stmt stmt = (Stmt)stmtIt.next();
110
111             // check if the instruction is a return with/without value
112             if ((stmt instanceof ReturnStmt)
113                 ||(stmt instanceof ReturnVoidStmt)) {
114                 // 1. make invoke expression of MyCounter.report()
115                 InvokeExpr reportExpr= Jimple.v().newStaticInvokeExpr(reportCounter
116
117                 // 2. then, make a invoke statement
118                 Stmt reportStmt = Jimple.v().newInvokeStmt(reportExpr);
119
120                 // 3. insert new statement into the chain
121                 //      (we are mutating the unit chain).
122                 units.insertBefore(reportStmt, stmt);
123             }
124         }
125     }
126 }
127 }

```

Now, test the instrumenter, before instrumentation:

```

[cochin] [621tutorial] java TestInvoke
I made 20 static calls

```

Run the instrumenter:

```

[cochin] [621tutorial] java MainDriver TestInvoke
Soot started on Tue Feb 12 21:22:59 EST 2002
Transforming TestInvoke... instrumenting method : <TestInvoke: void <init>()>
instrumenting method : <TestInvoke: void main(java.lang.String[])>
instrumenting method : <TestInvoke: void foo()>
instrumenting method : <TestInvoke: void bar()>

```

instrumenting method : <TestInvoke: void <clinit>()>

Soot finished on Tue Feb 12 21:23:02 EST 2002

Soot has run for 0 min. 3 sec.

Run the benchmark again:

[cochin] [621tutorial] java TestInvoke

I made 20 static calls

counter : 20

Compare the JIMPLE code before and after instrumentation:

BEFORE :

```

1  class TestInvoke extends java.lang.Object
2  {
    .....
14      public static void main(java.lang.String[] )
15      {
    .....
26      label0:
27          staticinvoke <TestInvoke: void foo()>();
28          i0 = i0 + 1;
    .....
42      return;
43  }
44
45      private static void foo()
46      {
    .....
52          staticinvoke <TestInvoke: void bar()>();
53          return;
54      }
55
56      private static void bar()
57      {
    .....
63          return;
64      }
    .....
71  }
```

AFTER:

```

1  class TestInvoke extends java.lang.Object
2  {
    .....
14  public static void main(java.lang.String[] )
15  {
    .....
26  label0:
27      staticinvoke <MyCounter: void increase(int)>(1);
28      staticinvoke <TestInvoke: void foo()>();
29      i0 = i0 + 1;
    .....
43      staticinvoke <MyCounter: void report()>();
44      return;
45  }
46
47  private static void foo()
48  {
    .....
54      staticinvoke <MyCounter: void increase(int)>(1);
55      staticinvoke <TestInvoke: void bar()>();
56      return;
57  }
58
59  private static void bar()
60  {
    .....
66      return;
67  }
    .....
74  }

```

We see that a method call to `MyCounter.increase(1)` was added before each `staticinvoke` instruction, and a call to `MyCounter.report()` was inserted before the `return` instruction in `main` method.

3 More on soot tutorial web page

<http://www.sable.mcgill.ca/soot/tutorial/>