Java Program Execution Analysis Tool

Dongyang Zhang
University of Texas, Austin
dyz@utexas.edu

Jeremy Joachim
University of Texas, Austin
jajoachim@gmail.com

I. Abstract

Automated instrumentation of compiled code is a valuable resource for programmers who wish to analyze their programs. This paper presents one such tool called the Java Dyanmic Analyzer which is built upon the Javassist and JUNG libraries. It comes with various instrumentation options to gather data which is displayed with an interactive GUI upon the instrumented programs main function exit.

II. Introduction

It is well known that software testing and profiling is the majority component to software development. This makes automated techniques extremely valuable to software developers. Automated techniques often take the form of instrumentation because it can be loaded post-compilation and requires minimal intervention on the programmers side. There are several instrumentation systems for Java coding, however they tend to have narrow purposes. The most popular focuses are code coverage and performance evaluation. In this paper, we provide a system that focuses on covering a breadth of subjects rather than the depth of a few. The Java Dynamic Analyzer (JDA) provides data on both performance evaluation and code coverage while remaining easily configurable.

III. Previous Work

There are two notable systems that the JDA competes with. The first is called JVisualVM [**?**], which focuses on performance analysis. It provides detailed information about method execution times via instrumentation, however the actual performance hit taken from the instrumentation is surprisingly small. Additionally, The instrumentation is applied during a programs execution rather than as a precursor. JVisualVM monitors processes running on the JVM to accomplish this feat. JVisualVM also has thread monitoring capabilities for multithreaded programs. Our system will focus on single threaded profiling; the instrumentation classes and methods that we use are not thread safe.

The second notable system is called Emma [**?**], which is specific to the Eclipse IDE. Emma provides detailed source code coverage analysis for program executions with an informative yet intuitive GUI. The source code coverage analysis is very useful when used in conjunction with junit tests. However, it is important to note that source code coverage analysis is different from bytecode coverage analysis because a single source code line can contain multiple basic blocks of bytecode. For instance, the following block of code may have many dead bytecode basic blocks despite having complete source code coverage:

```
for(MyObject i(0); i.lessThan(10); i.inc()){
  if(i.foo()) i.bar();
  String n= (i.printable()) ? "yes" : "no";
}
```

IV. Java Instrumentation

Java instrumentation is done with javaagents, which are parameters given to the JVM. A javaagent is a class with 3 main methods: main, premain, and agentmain. The main function allows the agent to be launched like a normal application, the agentmain allows it to be launched during a programs execution, and the premain allows it to be launched as a precursor to the normal application. JVisualVMs GUI is implemented with a main method, whereas the actual instrumentation uses an agentmain method. The JDA capitalizes on the premain method so that our program may instrument classes are they are initially loaded for a program. As such, our program is launched as a parameter to the JVM. The following is an example of launching a java program normally.

```
java target
```

To instrument a program using the JDA, the following command is used.

```
java -javaagent:pathTo/JDA..jar[=args] target
```

The instrumentation is done by implementing the ClassFileTransformer interface in the javaagent class. This interface provides a transform() function which essentially takes a byte array input and gives a byte array output. All classes that are loaded by the ClassLoader are run through this transformation, which ultimately inserts profiling code. Finally, bytecode interpretation and instrumentation is done with the help of Javaassist [**?**], which is a bytecode instrumentation library thats known for allowing users to insert source code rather than manually constructing bytecodes. While the ease of some API calls is highly beneficial, lower level instrumentation becomes burdensome when trying to access the information created by the higher level API calls.

V. Method Instrumentation Locations

Methods are always instrumented at the beginning and end of their code blocks. The beginning of a method is defined as the entry point of that method, which is always the first line. The end of the method is defined as any return statement within the method. Instrumenting at the end of a method does usually mean instrumenting before every return statement. It

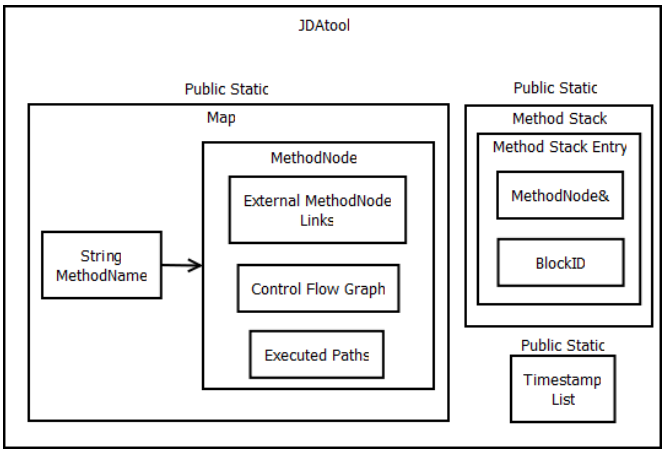

Fig. 1. JDA's Object Heirarchy

should be noted that our definition of the end of a function is not as strong as being defined as any exit point out of the method; methods can also exit via exceptions. However, exceptions are not caught by the JDA.

Depending on the JDA input options, instrumentation can also be inserted at the beginning of every bytecode basic block. Basic blocks are formally defined as one entry and one exit blocks of code, however the definition in this case is slightly broader. Method calls normally count as an exit from a basic block, but such fine -granularity is unnecessary for this profiler. Separating method calls into their own basic blocks can cause sparse CFGs that just become harder to read as well as further run-time intrusion from instrumentation. Furthermore, Javaassist provides CFG API that also considers methods as part of basic blocks, so the decision for this granularity was natural.

The instrumentation options provided by the JDA are trackTiming, trackBlocks, and trackPaths. The option trackTiming instruments at the beginning and end of methods. The option trackBlocks instruments at the beginning of every basic block. Finally, trackPaths automatically enables trackBlocks and adds additional instrumentation to the beginning and end of every block.

VI. Object Hierarchy

All of the data gathered during instrumentation needs to be stored in a global pool. Thee JDAtool is a class filled with public static members that instrumentation functions can reach from any class. All of the data organization structures reside in the JDAtool as well. The major data organization structures will be enumerated and explained in the following sections. Figure 1 shows the overall data hierarchy of the JDA.

A. TimestampList

The TimestampList is essentially a list of time blocks that do not include instrumentation at the beginning and end of methods. The small amount of instrumentation at the beginning of bytecode basic blocks, however, is too fine grain to exclude from timing. Naturally, the instrumentation for this option occurs at the beginning end of methods. The time that the code enters and exits instrumentation blocks can be subtracted from the overall method execution time to retrieve the native method run time. Afterwards, blocks of times can be summed together and the extra blocks removed to save memory. The noise, granularity, and odd JIT optimizations make the accuracy questionable for smaller methods, however large methods behave appropriately. The featuer can be turned on by setting the trackTimes option.

B. MethodNode

The core of all method information resides in the MethodNode class. Every unique method tag (which includes overloaded methods) reserves its own persistent MethodNode. The MethodNodes reside in the JDAtool in a Map. The Map allows the String typed method tags to access their respective MethodNodes.

MethodNode contains three vital pieces of information: a Control Flow Graph (CFG), a Set of ExternalLinks, and a Set of BasicBlockPaths. The CFG contains information regarding the control flow of bytecode basic blocks in the method. This is analyzed when the class is first loaded. The Set of ExternalLinks contains method invocation information within that MethodNode. They essentially keep track of the MethodNodes basic blocks in the CFG that call other MethodNodes. This allows us track method invocations of any sort, including invokedynamic calls.

The final data structure is the Set of BasicBlockPaths. A methods instrumentation can track the basic blocks that any single invocation executes. The blocks are concatenated together to form a BasicBlockPath. However, the BasicBlockPath implementation is somewhat unfinished; while it can recognize identical paths, it does not include loop detection. Large or unbounded loops within a method can become very burdensome in memory with this implemetation. As such, the option for path tracking, trackPaths, may be disabled.

C. MethodStack

The method stack is simple a list of MethodStackEntrys (MSE), where each MSE contains a MethodNode reference and an integer block ID. Every time a new bytecode basic block is executed, the block ID is updated. Every time a new method is called, that method creates a new MSE with its own MethodNode reference and adds it to the stack. With the MethodStack and block ID updates in each MSE, methods can extrapolate what other methods called them. They can then update the callers MethodNode with a now known ExternalLink. In order to track ExternalLinks, the trackBlocks option must be enabled.

VII. GUI

The GUI is launched at the end of the main function of the instrumented program, which ensures that the launch of the GUI will not interfere with the instrumention or add jitter to the runtime statistics. The main purpose of the GUI is to enable better interpretation of the statistics JDA generates. In fact,

almost all information JDA analyzes are related to Control Flow Graph vertices and edges. Thus, building a graphical interface with control flow graph is worthwhile. On one hand, a lot of information can be shown in a macroscopic manner immediately after the rendering of the control flow graph. e.g. active nodes, code coverage, edge weight etc. On the other hand, more detailed information is accessable by text area output in the GUI invoked by mouse listeners, which facilitates the search for certain statistics the user cares about.

Based on the methodology we do the intrumentation and build the JDA tool. We design the GUI to be multiple instance and for every methodnode the GUI will provide related data and interface to the basicblock and paths between them in a JFrame. The main chanllenge in the GUI is to provide a stable layout for the control flow graph. When it comes to graph layout, there exisits numerous layout algorithms to fulfill various constraints. e.g KK layout, FR layout etc.

But in our application, the graph we are looking at is the control flow graph with potentially complex structures. Thus, the name of the game becomes devising an algorithm that produce efficient and clean CFG layout. The requirements are four folded: unique layout for specific input, no overlapping in nodes or edges, space efficient and the flow goes in top-down direction.

Algorithm 1 is the algorithm we use to position the control flow graph. It can be easily proved that our algorithm generates a non-overlapping layout in $O(V + E)$ time complexity. It is a BFS based algorithm. When designing the layout algorithm, we make the decision that only expand the graph in width to the right, which lowers space consumption but is prone to more overlapping. In our algorithm we solve this problem by expanding the ansestors when there exisits branches in successors. The caveat is if we always use the indentation as the summation of successors, the graph will easily expand to a large size in width given a branchy input. The insight is we can reset the indentation as we see a vertex that has only one child, e.g. the vertex 4 in the graph as shown in Figure 2. Vertex 4 here has successor 6 that is of 3 children. But as long as there is no other branch within two levels, i.e. vertex 6 is the only child of vertex 4, we can reset the indentation vertex 4 returned to upper level.

Figure 3 shows the comparison between our algorithm and other main stream solution to CFG layout. All four CFGs are of same structure and the only difference is our algorithm doesn't make the start block and return block a separate one, and we forced the block with return statement to the bottom with more distance to upper levels. Comparison shows our algorithm saves more space in both width and height even if the input problem size is just medium. Meanwhile, our algorithm doesn't compromise the clear interpretation of the control flow.

VIII. JUNG Library

We searched and investigated different Java graph libraries and decide to use JUNG as the framework to build our GUI. JUNG, the Java Universal Network/Graph Framework

Algorithm 1 CFG layout

Require: $G = (V, A)$
$\exists!\ start \in V,\ s.t.\ Indegree(start) = 0,$
$\exists\ E \in V,\ s.t.\ \forall\ end \in E,\ Outdegree(end) = 0$
Ensure: $Postion(G),\ \forall\ v \in V,\ \nexists\ v,\ v' \in V,\ v \neq v'$
$s.t.\ Position(v) = Position(v')$
1: $start \Leftarrow initpos(initx, inity)$
2: $Visited \cup \Leftarrow start$
3: $N \Leftarrow SortedSuccessors(start)$
4: $maxWidth,\ maxDepth\ \Leftarrow$
$SuccessorLayout(N,\ initx,\ inity + \Delta y,\ Visited)$
5: **for all** $end \in E$ **do**
6: $end \Leftarrow endpos(initx, inity + maxDepth + \Delta y)$
7: $Next$
8: $initx \Leftarrow initx + \Delta x$
9: **end for**
10: **return**

$SuccessorLayout$
Require: $N,\ width,\ depth,\ Visited$
Ensure: $maxWidth,\ maxDepth$
1: **if** $N = \{\}$ **then**
2: **return** $width,\ depth$
3: **else**
4: $newdepth \Leftarrow depth$
5: $successorCount \Leftarrow 0$
6: **for all** $n \in N$ **do**
7: **if** $n \notin Visited$ **then**
8: $continue$
9: **end if**
10: $Visited \cup \Leftarrow n$
11: $n \Leftarrow initpos(width, depth)$
12: $maxWidth,\ maxDepth\ \Leftarrow$
$SuccessorLayout(N,\ width,\ depth + \Delta y,$
$Visited)$
13: $Next$
14: $newdepth \Leftarrow$
$MAX(newdepth, maxDepth)$
15: $width \Leftarrow$
$width + \Delta x * (OutDegree(n) + maxWidth)$
16: $successorCount \Leftarrow$
$successorCount + OutDegree(n) - 1$
17: **end for**
18: **if** $size(N) = 1$ **then**
19: $maxWidth \Leftarrow 0$
20: **else**
21: $maxWidth \Leftarrow successorCount$
22: **end if**
23: $maxDepth \Leftarrow newDepth$
24: **return** $maxWidth,\ maxDepth$
25: **end if**

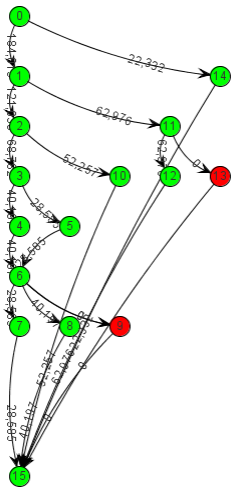

Fig. 2. Indentation of our algorithm

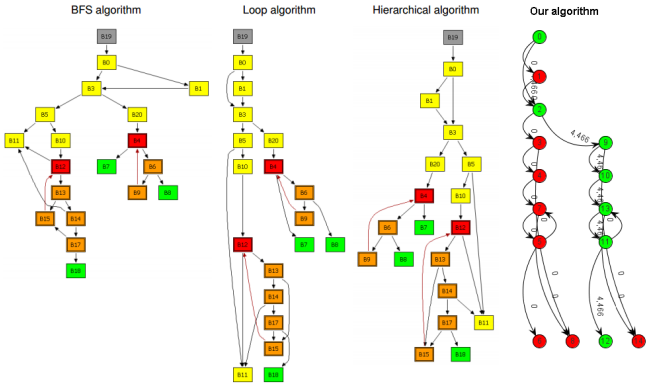

Fig. 3. Comparison of different layout algorithms

is a software library that provides a common and extendible language for the modeling, analysis, and visualization of data that can be represented as a graph or network. It is written in Java, which allows JUNG-based applications to make use of the extensive built-in capabilities of the Java API, as well as those of other existing third-party Java libraries. The JUNG library provides powerful graph representation and easily customized graph APIs. In our GUI, we used the mouse listener, vertex/edges render/transformer along with Java build-in javax.swing package to realize most of the functionalities.

IX. Features

As shown in Figure 4, our GUI pops out after the instrumentation program finishes. It will show the user with the CFG of each method in the class with different colors indicating the activeness and size indicating whether there is method call in the basic block. edges are filled with number of executions of all the path that pass through this edge. The mouse listeners has two modes, transforming and picking. In transforming mode, users can drag, rotate, zoom in/out the whole graph. In picking mode, users can click on a certain node to see the statistics in the textarea and see the path that go through that

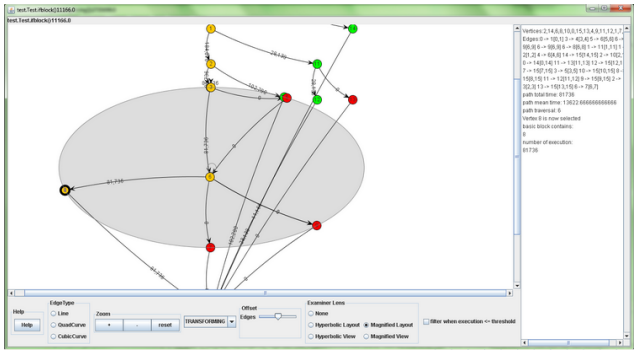

Fig. 4. GUI window

nodes one by one. On the control panel, the user can change the type and offset of the edges to render the graph, reset the graph, choose mouse modes as well as filter out infrequent nodes/path by input a threshold. In case of more A examiner lens is also provided to see the details of the graph when there are great amount of basic blocks in a method.

X. Future Work

There is a good deal of improvement that can be made to the JDAs path tracking features. Namely, the implementation of loop detection in paths. Ideally, paths should be pruned at the exit of methods by compressing loops into special subpaths. Loops can be simply made up of single paths or contain numerous branch statements which makes the control flow analysis hard to sort. There are various ways to hash loops, and thus the hashing method should be configurable. The concept of the JDA was more focused around tracking virtual method invocations, however the baseline instrumentation is already in place for more complex analysis.

XI. Conclusion

We made this tool that provides information.

References