Java Program Execution Analysis Tool

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

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

March 10, 2013

1 Introduction

Software testing has always been a fickle yet vital task for product assurance. As a result of growing software complexity, software testing has become a very expensive process for any programming language. Java is one such language, and many profiling hooks to the Java Virtual Machine's (JVM) inner workings are made readily available for debugging tools to exploit. Furthermore, the Java language has many constraints not found in complex langauges like C++ that allow code analysis to be more concise and straightforward.

2 Problem Statement

We aim to build a Java program execution profiling tool that instruments the Java bytecode using the ASM library. The tool's main purpose will be to help view code coverage and map the code to a node graph. With the help of a GUI tool, users will be able to track a Java program's execution and see both static and dynamic statistics.

What we want is to apply the software testing knowledge in building the tool such that it would facilitate test cases generation and provide relevant coverage suggestions in test cases. Another advantage is that our tool could easily be morphed into a hotline fault detecting tool with test inputs.

3 Objectives

Within the two main types of program analysis, static and dynamic, we wish to complete a small list of objectives. In static analysis, we wish to display a basic block node map of the program. Static analysis is mostly for visualization.

Dynamic analysis is the main goal of this project. The primary dynamic analysis objective in this project is to track method calls, timings, and inter-method paths. Additionally, the program will introduce a unique tool to display the difference between two executions of the same map. If time persists, we

would also like to perform CPU usage and memory footprint analysis on classes and methods within a program. The following sections will describe our current plans for each implementation in both static and dynamic analyses.

3.1 Static Analysis

The profiler will parse all classes within a program and create a node map. Each node will represent a basic block, but method calls will be in separate nodes. Each node will be able to be expanded into bytecode instructions. In the case of method call nodes, they will either jump to the method's node tree or simply display the bytecode that calls the method.

At this point, we could also offer synatictic pathfinding, however the importance of such a function seems limited. It could help show possible paths between methods, however semantic paths from execution are generally much more important. We will only implement this functionality if time allows us.

3.2 Dynamic Analysis

Each method will keep a set of simple paths executed with each method call. Knowing the amount of times each path was execution also shows how many times each node and edge is executed. However, loops and repeated nodes will have to be handled in a special manner that has not yet been decided. We are currently thinking of keeping them as a special node within a path that represets a subset nodes.

Method timing will be implemented with two 64 bit counters. One counter accumulates the total time spent in the method during execution (we will aim for nanosecond accuracy). The other counter keeps track of how many calls to the method there are. When the program needs to display the average time spent in a method, it only has to divide the two numbers. Methods with recursive calls may yield skewed results. Additionally, very short methods that will have runtimes in the timer's noise range will be excluded. Their timings can be extrapolated from methods that call these smaller ones.

Finally, we will offer a display showing the difference between two executions of a program. This will be implemented by first normalizing all of the timings of each method in both executions. Next, a subtraction is performed between the two programs' method timings. The difference in normalized timings will show the difference in program hotspots in two different executions.

CPU utilization and memory footprints are a bit more complicated to track; these are low level statistics that are not as readily available from the JVM. There hooks that allow some limited functionality, but multithreaded applications will cause additional complexity. We will only implement this functionality if time allows us.

4 Related tools

Java profiling tool has been an object of development for some time. Jmap [1] developed by Oracle focused on printing shared object memory maps or heap memory details. It is able to show object counts, increases of java class with little delay even for very large JVM heaps. The same for other memory analyzer tool e.g. [2]. Another tool is the VisualVM [3] built in the JVM. the creators described it as a visual tool integrating several command line JDK tools and lightweigth profiling capabilities. It enables gathering statistics throughout normal application execution rather than sampling information at intervals. The downside is the profiler is brute-force and it redefine most of the classes and methods. BTrace [4] can let the user specify what type of information they want to gather by scripts. The benefit is BTrace script is just a normal Java class containing some special annotations to specify where and how BTrace will instrument the application. Then BTrace Scripts are compiled into standard . class files by its compiler. The drawback is the design makes it difficult to navigate output format to better process the data. Later tools e.g. EurekaJ [5] provides better visualization and parsing of the statistics. ther commercial tools e.g. JProfiler [6] support more extensive profiling including hihger level profiling data, database profiling, analysis of memory leaks, QA capabilities as well as integrated thread profiler.

5 Conclusion

Conclusion goes here. It will conclude things. Concluding things is good. That concludes this paper.

References

[1] `http://download.oracle.com/javase/1,5.0/docs/tooldocs/share/jmap.html`.

[2] `http://eclipse.org/mat/`.

[3] `http://http://visualvm.java.net/`.

[4] `http://http://kenai.com/projects/btrace`.

[5] `http://eurekaj.haagen.name/`.

[6] `http://www.ej-technologies.com/products/jprofiler/overview.html`.