Java Bytecode Timing Cost Models

Guillermo Román Díez

Abstract

Estimating execution time, during compilation process, can be very useful in software development. We are going to analyze possible relationships between executed Bytecode instructions and execution time. Using these relationships, the main aim of this report is to detail and test several cost models to estimate, as accurate as possible, a Java program execution time. Estimation may be a powerful tool in some scenarios like distributed systems, p.e. during tasks assignment. Nowadays Java Virtual Machine provides profiling tools but lacks any tool to estimate execution times.

The final objective is to develop a tool able to estimate execution time of a program without run it. With this objective in mind, this article proposes several cost models for Java Bytecode to analyze possible relationships between Bytecode instructions and execution times. Bytecode is very similar to assembler code and it’s closer to machine instructions than Java source code. Furthermore, Bytecode can be executed by every Java Virtual Machine and its compiler independent.

Several cost models are proposed and described in detail, comparing them in this report. Cost models need a tool to count executed Bytecode instructions. Prolog Profiler executes symbolically a program and counts executed Bytecode and has been used in this work.

1 Brief Description

Java is a predominant programming language in professional environments and Java Virtual Machine (JVM) is Java’s heart. JVM lacks any tool to predict or estimate the execution cost of a program. Most Java developers resort to the wall clock time in order to benchmark their algorithms. However, this approach usually gives imprecise results, due to the measurement granularity.

Having information about execution time of a program are really useful. Usually different programmers implements different solutions to solve the same problem. However, to know which are the best solution is not easy and having estimated execution time may help us to select the quicker solution. Furthermore, this information may be very useful in the context mobile phones, where users usually only have a .class file o a .jar file containing several .class
files and an estimation tool may be very useful to compare different solutions.

Java has an intermediate language, Bytecode, which has its own instruction set. Bytecode is distributed in Class files which contains code and JVM runs this code. This Bytecode will help us to calibrate and estimate the execution time of a Java program, because it is closer to machine instructions than Java source code. Thus, all models described in document uses Bytecode to be calibrated and to test estimated times.

As regards Java Bytecode instructions set, proposed solution does not support several Bytecode instructions like floating point arithmetic (long or double instructions), instructions that uses floating point types, class attributes access or dynamic class loading. Even so, Bytecode instructions subset is enough significant to write complex programs to calibrate and test the described cost models.

This document is structured as follows: First section describes the details for each cost model, the next section explain the experiment steps, the programs and the tools used. After that, experiments results and conclusions are detailed.

Java's heart is the Java Virtual Machine, which allows users to run Bytecode in every platform. Java Virtual Machine is an abstract computer and its specification defines certain features which every Java virtual machine must implement, but leaves many choices to the designers of each JVM implementation. Even so, JVM specification is flexible enough to allow a Java Virtual Machine to be implemented either completely in software or to varying degrees in hardware. JVM is a middleware between operating system and a Java application.

Java Bytecode is universal language for Java Virtual Machine. Bytecode is a low-level object-oriented programming language, similar than assembler languages and, as some assembler languages, JVM uses a operand stack to hold operations results. Bytecode is stored in a class file which may contain several methods. Loading and running binary class files are the main job in a JVM. Actually, only a class files is necessary to run a Java program. The Bytecode are loaded and executed in JVM execution engine. JVM and its execution engine may be implemented using hardware or software and uses Bytecode as instructions set. Bytecode is very similar than assembler code, but uses a stack and a heap for memory allocation.

Java Bytecode has been related to cost model in several previous studies, like Binder and Hulaas whose article describes how to use Bytecode instructions counting as CPU consumption metric. Hermenegildo, Puebla and others have also written about this topic several papers about static analysis and estimation, or experiments about cost analysis in Java.

In this study we will try to establish a relationship between Java Bytecode and the execution time. Using symbolic execution to get executed Bytecode instructions. Prolog Profiler, a tool used in experiments implementation, uses Java Bytecode as input data, and Class files are loaded by this tool. Prolog Profiler executes a method symbolically and get executed Bytecode in this “execution”. Using Java Bytecode as input for cost models are very useful since Bytecode may be generated using all Java compilers. Java Bytecode allow us not considering possible compilers optimizations o differences between compilers.

The experiment will test and compare the estimation results of the described models in Cost Models section. All analyzed models try to define the relationship between execution time and executed Bytecode instructions. All described models try to define a way to estimate the execution time of a Java program.
using Bytecode instructions.

All described experiments need four steps:

1. **Calibrating the model:** In this step, using the calibration Java programs and Prolog Profiler, will be calculated all necessary parameters and constants of the model. Each model needs different constants and all models describe the way to get these constants.

2. **Estimating programs:** The second step estimates the execution time for Java testing programs using the constants gotten in the first step and Prolog Profiler to get executed Bytecode instructions.

3. **Running testing programs:** In this step, we run test programs in order to get real execution time for test programs.

4. **Comparing the estimation and the real execution time:** In last step we compare estimated execution time to real execution time and calculate the deviation between them.

All models calibration constants are based in execution time. One of the most important aspect is measured as accurately as possible this execution time. The execution time must be large enough to significantly exceed the measurement granularity. Java API provides methods to measure milliseconds in wall time. This granularity is not enough to measure a single Bytecode instruction runtime. To solve the the measurement granularity calibration and estimating methods must be executed in a loop, to increase execution time and solve the granularity problem.

We have proposed a way to estimate execution time for a program in a JVM. Proposed solution combines static analyses and symbolic execution (Prolog Profiler) with calibration programs execution timing. Experiments results presented in Results section show that the proposed cost model can be useful in Java programs estimations. Deviations are still significant, but improving calibration programs and calibration groups obtained results can be improved. A good feature is that this environment only need run calibration programs in the final environment (physical machine and JVM) to allow us to estimate execution time in this machine. Afterwards, another machine can estimate execution time from this program in final machine.

As we commented, estimation tool can be very useful in distributed environments where each platform have different capabilities, or in situations where we only have binary code, and we need to evaluate the resources consumption. Supposing a distributed environment, we can calibrate each machine and distribute each information in all machines. Once having the calibration results for each machine in each machine, one node can select best machine to run a program and optimize its execution in distributed environment.

As regards future work, some improvements can be evaluated. Such as, described models only take into account Bytecode in a static way. This models don't take into account anything about dynamic operations, like method dynamic loading, seekings in Classpath. This operations may cause deviations in calibration or execution times. Measuring times is another important aspect because it depends by workload on the benchmarking machine. Moreover, using wall time to measure execution time may be imprecise.
Another aspect to improve is the isolation of Bytecode instructions in calibration programs, because some models need single Bytecode instructions absolutely isolated in order to calibrate as well as possible execution time of these instructions. A lot of mathematical instructions need to accede to memory and generate iload or istore during compilation process. This instructions avoid us to get a single Bytecode instruction execution time without noise caused by memory access instructions.