Execution Time Disambiguation Profiling

by

Aniket Kumar Lata

B.E., University of Mumbai 2012

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Master of Science

Department of Electrical and Computer Engineering

2016

This thesis entitled:
Execution Time Disambiguation Profiling
written by Aniket Kumar Lata
has been approved for the Department of Electrical and Computer Engineering

_____

Prof. Pavol Černý

_____

Prof. Ashutosh Trivedi

_____

Prof. Sriram Sankaranarayanan

Date _____

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Lata, Aniket Kumar (Electrical and Computer Engineering)

Execution Time Disambiguation Profiling

Thesis directed by Prof. Pavol Černý

We consider the problem of developing an efficient cost model for Java Bytecode, where cost is determined in terms of execution time. The key insight is that the execution time of Java applications can be predicted by modeling these applications based on the method invocation counts and their linear fitting to get an accurate estimate. Cost estimates for an application can be useful in determining parameters such as worst case execution time which can form a basis for static analyses and can be useful in distributed applications.

Dedication

To all my mentors, friends and family for their support and guidance.

Acknowledgements

Contents

Chapter

Tables

Table

Figures

Figure

Chapter 1

Introduction

Accurate execution time measurement of Java bytecode instructions is an important estimate in determining the total execution time of Java applications. There are a number of factors make the execution time of Java bytecode difficult to predict. Java applications are run on a Java Virtual Machine (JVM) which translates Java bytecode to platform dependent machine code. It consists of stages that introduce optimizations in the translated machine code. A JIT Compilation further introduces non determinism in the execution of a program. Dynamic linking and loading is another factor that slows down the compilation process of Java. C/C++ code is typically compiled to an object file then multiple object files are linked together to produce a usable artifact such as an executable or dll. During the linking phase symbolic references in each object file are replaced with an actual memory address relative to the final executable. In Java this linking phase is done dynamically at runtime. When a Java class is compiled, all references to variables and methods are stored in the class's constant pool as a symbolic reference. The JVM implementation can choose when to resolve symbolic references, this can happen when the class file is verified, after being loaded, called eager or static resolution, instead this can happen when the symbolic reference is used for the first time called lazy or late resolution.

Due to these factors making Java programs less deterministic, it is useful to have a prediction mechanism that can provide a timing estimate for a block of instructions. My Master's thesis builds upon this idea of timing basic blocks and predicting the execution time of applications that

use this basic blocks. A cost model has been developed that can efficiently predict the execution time of basic blocks and methods declared within a program. Fetching costs for methods used in popular libraries can be a very useful measurement in profiling of Java programs.

The cost model provides estimates pertaining to the execution time for expensive methods in Java bytecode. Java applications are profiled to generate:

(1) Counts of method invocations and

(2) Total execution time of the application

This profiling is performed with the help of bytecode instrumentation. The data generated from these runs is fed to a Linear Regression toolbox to efficiently predict the "time per basic block" or the "time per method" metric for that specific application.

The cost model tool can help in determining the time taken for each method or basic block to execute within an application. The worst case execution time (WCET) of a program is an important parameter which helps in real time scheduling applications. WCET can be predicted with the help of cost information from the model. The costs can be fed to a static analyses framework which can check all possible path combinations to get a WCET metric for that application. Cost modeling can also be used to detect security vulnerabilities in Java applications. Security vulnerabilities can be classified broadly into Availability and Confidentiality attacks. Availability problems arise when a user-provided input controls the asymptotic complexity or termination of a program component. The cost model can flag the blocks within a program that consume more CPU time with a high cost. A section of code susceptible to availability vulnerabilities can be determined by checking cost annotations of each block and what kind of inputs would reach this block of code.

An important use of the cost model is to predict the execution time of Java applications whose

libraries are profiled with the cost model. The model data forms a training input set and the application data forms a testing input set. The execution time of each method or basic block is an output from the training data set. These estimates are applied to the testing data set to predict the execution time of this application for a specific set of inputs.

Chapter 2

Motivating Examples and the Cost Model approach

2.1 Block Timing

Java bytecode can be profiled at different points of interest to measure execution time for these sections of code. These measurements need to be consistent to ensure disambiguation of the analysis.

We start with profiling the basic blocks which are the most fundamental units of a program. Basic blocks present a straight-line code sequence making them highly amenable to analysis. Instrumentation can be added to add source code for timing the basic blocks within a program. For disambiguation profiling, we consider an example program shown below with two basic blocks.

2.1.1 Examples

```
public void workerFunction(int arg){
       int N = arg;
     int x = 0, a = 1, m = 0;
     if(N > 5) {
       // Basic block 1 starts
           x++;
           x = x * N;
           x &= (a << 1);
           int i = 0;
           // Basic block 1 ends
           do{
               // Basic block 2 starts
                 x += i;
```

```
                m = (x >> (N - i) & 1);
                i++;
                // Basic block 2 ends
            }while(i < N);
        }
    }
```

The worker function is instrumented to measure execution time of the two basic blocks as shown in the modified worker function:

```
public void workerFunction(int arg){
         int N = arg;
      int x = 0, a = 1, m = 0;
      if(N > 5) {
               long l1 = System.nanoTime();
        // Basic block 1 starts

            x++;
            x = x * N;
            x &= (a << 1);
            int i = 0;
                // Basic block 1 ends
                long l3 = System.nanoTime();
            long l5 = l3 - l1;
            double d = (double)l5 / 1000000D;
            System.out.println((new StringBuilder()).append("\n\rDURATION: ").
                append(d).toString());

            do{
                       long l2 = System.nanoTime();
                // Basic block 2 starts
                  x += i;
                  m = (x >> (N - i) & 1);
                  i++;
                  // Basic block 2 ends
                    long l4 = System.nanoTime();
                long l6 = l4 - l2;
                double d1 = (double)l6 / 1000000D;
                System.out.println((new StringBuilder()).append("\n\rDURATION: ").
                    append(d1).toString());
            }while(i < N);
      }
 }
```

2.1.2 Results

The basic block measurements were tested by executing the instrumented version of the code multiple times with the same input.

Command: $ java -jar Test.jar 6

Run 1:

Execution time measurements in milliseconds(ms):

Basic block 1: 0.181

Basic block 2:

Iteration 1: 0.291

Iteration 2: 0.199

Iteration 3: 0.215

Iteration 4: 0.193

Iteration 5: 0.202

Iteration 6: 0.191 **Run 2:**

Execution time measurements in milliseconds(ms):

Basic block 1: 0.231

Basic block 2:

Iteration 1: 0.3

Iteration 2: 0.403

Iteration 3: 0.182

Iteration 4: 0.257

Iteration 5: 0.194

Iteration 6: 0.287

Run 3:

Execution time measurements in milliseconds(ms):

Basic block 1: 0.432

Basic block 2:

Iteration 1: 0.354

Iteration 2: 0.349

Iteration 3: 0.301

Iteration 4: 0.291

Iteration 5: 0.324

Iteration 6: 0.289

Run 4:

Execution time measurements in milliseconds(ms):

Basic block 1: 0.286

Basic block 2:

Iteration 1: 0.487

Iteration 2: 0.387

Iteration 3: 0.421

Iteration 4: 0.393

Iteration 5: 0.331

Iteration 6: 0.372

Statistics:

Basic block 1:	Mean: 0.2825	Standard Deviation: 0.1085	Variance: 0.01177
Basic block 2:	Mean: 0.3005	Standard Deviation: 0.0850	Variance: 0.00723

2.1.3 Inferences

Disambiguation among the execution time measurements can be ensured if the timing measurements are consistent. From the above measurements, it can be seen that the variance is significant considering the fact that timing measurements require very high accuracy.

2.2 Cost model approach

The basic block timing approach does not work in disambiguation of execution time as the standard deviation of the reading is significant when multiple runs are accounted. We propose a novel approach to disambiguate execution time for Java applications. The cost model profiles a program to measure the execution time for one entire run as opposed to timing individual sections of code. Instrumentation is added to fetch block invocation counts.

The basic example would be profiled as shown in the subsequent sections to generate a cost model.

2.2.1 Execution time Profiling

```
public void workerFunction(int arg){
       // Instrumentation for exec time
       long startTime = System.nanoTime();
       int N = arg;
     int x = 0, a = 1, m = 0;
     if(N > 5) {
       // Basic block 1 starts
           x++;
           x = x * N;
           x &= (a << 1);
           int i = 0;
           // Basic block 1 ends
           do{
               // Basic block 2 starts
                 x += i;
                 m = (x >> (N - i) & 1);
                 i++;
                 // Basic block 2 ends
           }while(i < N);
     }

     long endTime = System.nanoTime();
     long elapsedTime = endTime - startTime;
     double duration = (double)elapsedTime / 1000000.0;
     System.out.println(duration);
 }
```

2.2.2 Block invocation Profiling

```
public void workerFunction(int arg){
       // Instrumentation for block invocation counts
       // counters: counter_bb1, counter_bb2
       int N = arg;
     int x = 0, a = 1, m = 0;
     int counter_bb1 = 0, counter_bb2 = 0;
     if(N > 5) {
       // Basic block 1 starts
              counter_bb1++;
           x++;
           x = x * N;
           x &= (a << 1);
           int i = 0;
           // Basic block 1 ends
           do{
               // Basic block 2 starts
                       counter_bb2++;
                 x += i;
                 m = (x >> (N - i) & 1);
                 i++;
                 // Basic block 2 ends
           }while(i < N);
     }
 }
```

Model generation requires two passes for each input: execution time pass and block invocation count pass. This data is logged on the disk for a large number of inputs.

2.3 Bytecode disambiguation

The cost model approach can be applied at the bytecode level to disambiguate different types of bytecode instructions and their execution times on a target platform. In this example, the disambiguation of bytecode instructions has been performed for an arithmetic intensive program.

2.3.1 Example

```
public class TestBinaryOp {
       public static void main(String[] args) {
              int x = Integer.parseInt(args[0]);
```

```
                int y = Integer.parseInt(args[1]);
                int numAdd = Integer.parseInt(args[2]);
                int numSub = Integer.parseInt(args[3]);
                int numMul = Integer.parseInt(args[4]);
                int numDiv = Integer.parseInt(args[5]);
                int result;

                for( int i = 0; i < numAdd; i++ ) {
                        result = x + y;
                }
                for( int i = 0; i < numSub; i++ ) {
                        result = x - y;
                }
                for( int i = 0; i < numMul; i++ ) {
                        result = x * y;
                }
                for( int i = 0; i < numDiv; i++ ) {
                        result = x / y;
                }
        }
}
```

2.3.2 Results

The bytecode instructions have been divided into different classes depending upon the correlation with one another.

ARITHMETIC - ADD,SUB,MUL,DIV,LOAD,STORE,INC,AND,OR,XOR,SHL,SHR,NEG

MEMORYALLOCATIONS - NEW,DUP,GETFIELD,PUTFIELD,PUSH,POP,GETSTATIC,LDC

IOCALLS - java.io.*

UTILCALLS - java.util.*

MATHCALLS - java.lang.Math.*

Below are the estimates for the expensive bytecode categories of instructions:

	Estimate	Std. Error
ArithmeticOp	1.098e-06	1.841e-08
MemoryAlloc	3.842e-03	7.779e-04

It can be seen here that the cost for arithmetic instructions and memory allocations has been estimated by the model. However, at the lowest level this disambiguation can tend to have some

deviation depending on the interference from other instructions to the measurement of arithmetic and related bytecode costs. We move one level higher to groups of bytecode instructions - precisely basic blocks and methods to achieve better disambiguation.

2.4 Method disambiguation

We have seen that timing basic blocks for the same input produces inconsistent results. Our approach relies on timing the entire application as opposed to timing each fundamental unit of the program. With the cost model, we measure the execution time of the application and log method invocation counts for a number of inputs. Linear regression is used to produce the estimates for each method. The example given below illustrates our approach.

```
public class Functions {

	public void function1() throws InterruptedException{
		java.lang.Thread.sleep(1);
	}

	public void function2() throws InterruptedException{
		java.lang.Thread.sleep(10);
	}

	public void function3() throws InterruptedException{
		java.lang.Thread.sleep(500);
	}

	public void function4() throws InterruptedException{
		java.lang.Thread.sleep(100);
	}

	public void function5(){
		return;
	}

}
```

The "Functions" library consists of five functions and each one takes different amounts of time to execute. Our goal here is to disambiguate these functions by accurately identifying the amount of

time for which they run.

The cost model approach instruments the Functions library to fetch method invocation counts and total execution time for the application. The cost model is explained in detail in the later sections. A test application has been written that calls the methods in the library with different call distributions. As it can be seen in the test class below, methods are invoked in different order for different inputs received from the users. This exercise simulates a good coverage of the functions being called in random distributions.

2.4.1 Basic example

```
public class MainFunc {

	public static void main(String args[]) throws InterruptedException{

		int N = Integer.parseInt(args[0]);
		Functions func = new Functions();

		switch(N){
			case 1:
				func.function1();
				func.function3();
				func.function3();
				func.function5();
				func.function3();
				break;
			case 2:
				func.function5();
				func.function2();
				func.function2();
				func.function4();
				func.function4();
				break;
			case 3:
				func.function3();
				func.function4();
				func.function4();
				func.function5();
				func.function4();
```

```
                        break;
                case 4:
                        func.function1();
                        func.function1();
                        func.function5();
                        func.function4();
                        func.function2();
                        break;
                case 5:
                        func.function1();
                        func.function1();
                        func.function1();
                        func.function2();
                        func.function1();
                        break;
                case 6:
                        func.function5();
                        func.function4();
                        func.function3();
                        func.function2();
                        func.function1();
                        break;
                }
        }
}
```

2.4.2 Initial Results

Multiple linear regression is used to generate the best fitting estimates that can disambiguate the time taken by each method. The dependent variable is "execution time" for a specific run and the independent variables are "method names" for that same run. The estimates are generated for each independent variable which are the "functions" in the above example. The estimates generated using our cost model tool for 200 inputs are as shown below:

	Estimate
Function1	1.131e+00
Function2	1.014e+01
Function3	5.002e+02
Function4	1.002e+02

Function5 2.807e-02

As it can be seen, the estimates generated are quite accurate. We can compare the estimates to the sleep call parameters within each function as the functions are not doing any other work. These estimates become more and more precise as the input coverage gets better and the number of test inputs increases.

Thus, it can be inferred from this experiment that the method disambiguation experiment worked well using the cost model for the basic example illustrated.

2.5 Problems with method disambiguation

Method disambiguation worked well in the example demonstrated in section 2.2.1. The methods used in the library did not have any control flows. If the method is a sequential block of statements, for a single input the same section of code will be executed. However with control flows being a part of the method, different branches are executed for differing inputs to the program. This can be illustrated well by modifying the basic example presented earlier.

Control flows introduce path imbalances in the program execution. In such cases, it becomes difficult to provide an estimate for the execution time of such methods if the execution time of paths taken for an input is noticeably different.

2.5.1 Control flow example

```
public void function1() throws InterruptedException{
       java.lang.Thread.sleep(20);
}

public void function2(int n) throws InterruptedException{
       java.lang.Thread.sleep(5);
       if(n > 6)
              function4();
}
```

```
public void function3() throws InterruptedException{
       java.lang.Thread.sleep(500);
}

public void function4() throws InterruptedException{
       java.lang.Thread.sleep(100);
}

public void function5(){
       return;
}
```

2.5.2 Results

The control flow example in section 2.3.1 is a modification of the basic example shown with methods. We have a branch in function2() which changes the predicted estimates significantly. It can be seen that function5() gets a negative estimate and the estimate for function4() is incorrect. function2() has a branch and it's estimate will vary depending upon the value of 'n'. It is incorrect to have an estimate for functions where the control flow determines the execution time of the program.

	Estimate
function1	21.323
function2	39.599
function3	500.569
function4	5.361
function5	-0.907

The aim of this example was to illustrate the amount of inaccuracy that a branch can introduce within the cost measurements. Real life programs will have a large number of branches and if these branches introduce paths that are significantly different from one another in terms of execution time, it is necessary to account mitigate the effects of control flows within a program.

Chapter 3

Model

Our cost model needs to predict the value of the dependent variable – execution time, using a linear function of the independent variables – method counts.

A naive approach to bucketing would be accounting for each method used in the application as a separate bucket. The standard error for coefficients in multiple regression can be high if we don't account for the correct subsets of independent variables in the model. The variance of coefficient for a variable could increase if there is high correlation among with another variable. The mean squared error for all the variables gives a good indication of the estimate provided by the cost model. Even though a model shows an ideal coefficient of determination, it may not be sufficient to give an accurate estimate for an independent test application using the same independent variables i.e. methods in our case. There is a need to refine the cost model to address the standard error among variables within a model and the mean squared error for accurate estimation of test applications.

Model refinement addresses two specific problems: **Subset selection** and **Estimation Accuracy**.

3.1 Subset Selection

An important decision to be made is the bucketing (categorization) of methods in a Java application to perform multiple linear regression. This is termed as "subset selection" for the model. The naive approach considers all independent variables in the data for prediction. There

are several reasons why this could be undesirable:

- Estimates of regression coefficients are likely to be unstable due to multi-collinearity in models with many variables.
- We get better insights into the influence of regressors from models with fewer variables as the coefficients are more stable for parsimonious models.
- It can be shown that using independent variables that are uncorrelated with the dependent variable will increase the variance of predictions.
- It can be shown that dropping independent variables that have small (non-zero) coefficients can reduce the average error of predictions.

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_n X_n \tag{3.1}$$

$$MSE = \frac{1}{N} \sum_{i=0}^{N} (y_i - \hat{y}_i)^2 \tag{3.2}$$

Chapter 4

Algorithm

Chapter 5

Implementation

Chapter 6

Experiments

Chapter 7

Conclusion