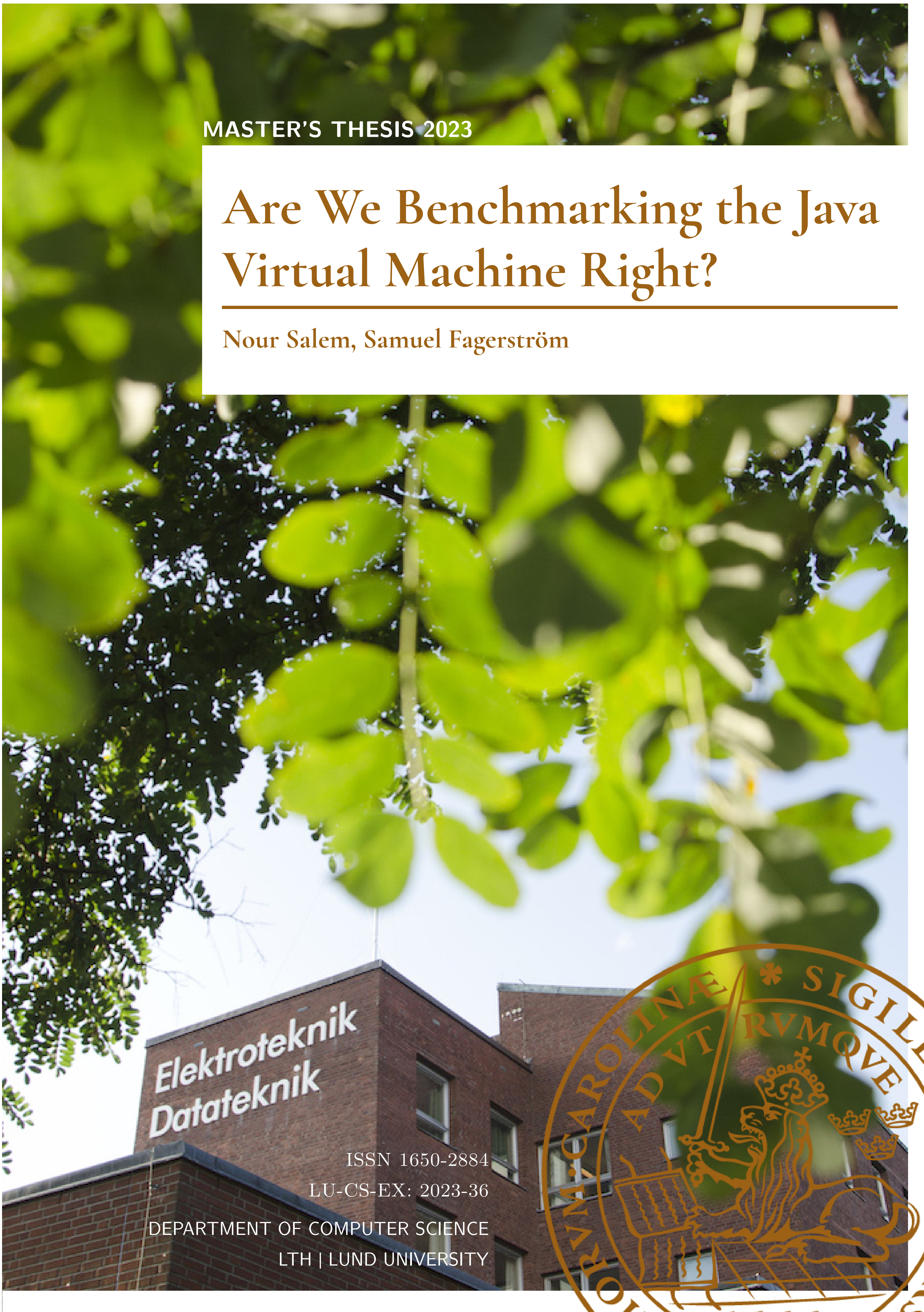


MASTER'S THESIS 2023

Are We Benchmarking the Java Virtual Machine Right?

Nour Salem, Samuel Fagerström

ISSN 1650-2884
LU-CS-EX: 2023-36

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2023-36

Are We Benchmarking the Java Virtual Machine Right?

Benchmarkar Vi Javas Virtuella Maskin Rätt?

Nour Salem, Samuel Fagerström

Are We Benchmarking the Java Virtual Machine Right?

Nour Salem
noursalem689@gmail.com

Samuel Fagerström
samuel.fagerstrom@gmail.com

September 6, 2023

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisor: Christoph Reichenbach, christoph.reichenbach@cs.lth.se

Examiner: Görel Hedin, gorel.hedin@cs.lth.se

Abstract

Benchmarks are important to developers and give them a frame of reference for how a change could affect the project they are working on. Therefore, it is important that the benchmarks developers use give a realistic representation of their environment.

This thesis aims to examine the heavily cited benchmark suite for Java called DaCapo to determine how well it represents the Java Virtual Machine. This is revealed by using the code instrumentation tool called "Did My Code Execute" developed at Ericsson. An important observation made was that parts of the Java Virtual Machine do not get exercised by any DaCapo benchmark. The answers were found by looking at the total amount of Java Virtual Machine code that was covered by the benchmark suite and by analyzing how many times specific lines of code were executed. The thesis also produced logistic regression models used to tell how important different parts of the Java Virtual Machine are to specific benchmarks.

Keywords: Java, DaCapo, Benchmark, Did My Code Execute, OpenJDK

Acknowledgements

We would like to thank our supervisor, Christoph Reichenbach, for providing us with this research topic and for his continuous guidance throughout the thesis work.

We would also like to thank Patrik Åberg, who helped us set up a tool that we needed and gave us useful insight on how to utilize the tool as well.

Lastly, we would like to thank Johan Eker for providing us with servers at Ericsson that we used to collect data.

Contents

Chapter 1
Introduction

Benchmarks are tests that can be used to measure the performance of different properties in software or hardware. Researchers typically use benchmarks to evaluate new system features and optimizations. In the case that an idea does not improve a set of relevant benchmarks, then that idea might not be accepted by the community and will probably not be explored any further [10]. Because of this, benchmarks can both encourage and discourage new ideas to be explored. Given this, it is important that the benchmarks that researchers use when developing new software evaluate as much as possible that is relevant to that software.

In contrast to other programming languages, Java programs are first statically compiled into bytecode, which is the same for all computers [12]. The Java Virtual Machine (JVM) is an abstract computing machine that has an instruction set and handles memory management at runtime. When executing a program the only information that the JVM has is a binary formatted file called a class file. The contents of a class file are instructions for the JVM (called bytecodes), a symbol table, and other ancillary information [11]. Therefore, the JVM is a vital part of the Java platform, it is the reason why Java is independent of hardware and operating systems. A widely used benchmark suite that tests the performance of Java is the DaCapo benchmark suite, which has an official release called DaCapo Bach [5]. There also exists an experimental Github repository branch with newer versions of the benchmark suites and several new benchmark suites called DaCapo Chopin [2].

1.1 Research questions

To tell what implications our measuring method has and how accurate the answers to our following research questions (**RQ:s**) are, we would first like to answer this research question and its sub questions:

- **RQ1:** *What implications does our measuring method have?*

  RQ1.1: *What are the challenges of our measuring method?*

RQ1.2: *How does the measuring affect execution speed of benchmarks?*

RQ1.3: *How stable are our measurements on different executions of the same program?*

This master's thesis work then answers the following research questions:

- **RQ2:** *How much of the JVM code is covered by state of the art benchmark suites?*
- **RQ3:** *To what degree are our state of the art benchmarking suites exposing the performance of the JVM?*

  RQ3.1: *Are there any parts of the JVM code that get executed by other Java programs that do not get hit by the benchmarks?*

  RQ3.2: *Does the distribution of JVM code executed by the benchmarks reflect that of other Java programs?*
- **RQ4:** *To what degree do the different benchmarks exercise different JVM code?*

  RQ4.1: *For each individual benchmark, what degree of importance do different parts of the JVM code have for that benchmark?*

  RQ4.2: *To what degree are different parts of the JVM code heavily exercised by all benchmarks?*

1.2 Contribution

The results of our research may be used to create more accurate benchmark suites for Java in the future or improve the existing ones. Also, all of our results and everything we have used to produce them have been uploaded to Zenodo [1], making it possible to review and use our work for continued research.

The tool Did My Code Execute (DMCE), developed and used at Ericsson, is an open source code instrumentation tool for C and C++ [3]. DMCE can see what part of the code that is executed when running a program. This work could also be used as a demonstration of how DMCE can be used to evaluate Java programs and as an example of using DMCE on a large scale project. Our findings when exploring the OpenJDK using DMCE also yielded important feedback to the developer of DMCE (Patrik Åberg) who in turn produced several patches for DMCE.

We have created logistic regression models that take DMCE data from a benchmark as input and predict what benchmark was executed to create the input data. These models can also be used to tell what parts of the JVM code are important for certain benchmarks.

When we worked with DaCapo Chopin the benchmark **h2o** did not function correctly due to the fact that it pointed to a non existing URL. Our supervisor Christoph Reichenbach created a fix for this[1].

We have worked together and helped each other with everything regarding this thesis work, including the report. However, we have written down who we consider had the most responsibility for different sections of the report in appendix A

[1]`https://github.com/creichen/dacapobench`, git hash: 1c746ae37b9a4c38b2179165635e8163a50d5e50

1.3 Related work

DaCapo and the OpenJDK have been analyzed and tested. For example, there is a paper that explores how the DaCapo benchmarks' memory allocations work [8]. There also exists a paper exploring the coverage of different Java benchmarks [9], but this paper did not analyze DaCapo. While DaCapo and the OpenJDK have been analyzed and tested, our work will explore the inner workings of the JVM and how well DaCapo covers it in a way that we believe has not been done in this manner before.

Apart from DaCapo, there are many other benchmarks. XCorpus [6] is another benchmark suite for Java that claims to have a higher branch coverage than DaCapo. Therefore, it would have been interesting to also analyze XCorpus and compare the results to that of DaCapo. We did, however, not manage to get the XCorpus benchmarks to work on our system and therefore could not analyze them.

Chapter 2
Background

This chapter will explain all the information needed to understand the following chapters of the report.

2.1 How Java works

Even though it is possible for Java to execute the bytecode using an interpreter, it is more common to first compile the bytecode to native code, which can be executed directly. This compilation is done using a Just-In-Time (JIT) compiler. An important aspect of JIT compilation is the trade off between initial performance loss and later performance gain due to the execution of the compilation and the resulting compiled code. Therefore, the JVM is selective about submitting code for JIT compilation, only methods that have been executed enough times are compiled with complex optimizations. The precise result of JIT compilation depends on numerous timing factors. Therefore, several executions of the same benchmark may not result in the same JIT behavior, meaning the same code may not always be used [12]. Oracle's Java Development Kit (JDK) contains a compiler, which compiles Java source code to bytecode, and a run-time system that implements the JVM [11].

2.2 The OpenJDK structure

OpenJDK is an open source code version of a JDK. The different OpenJDK versions can be found on the OpenJDK GitHub[1]. The OpenJDK consists of everything needed to create a working JDK as well as tests for the JDK. The source code that is used to build a JDK is located in a folder called "src", where the code for the "hotspot" and other Java functionality can be found. One aspect that turned out to be important for this project is that the "src/hotspot" folder contains the CPU specific and OS specific code. The source code of the OpenJDK

[1]`https://github.com/openjdk/`

is mainly written in C++ and C, which means that DMCE instrumentation is well suited for the OpenJDK. When the build is finished, a folder is created containing the newly built JDK. The contents of this folder can vary depending on what build options were used. For example, you can use the build option "–with-jvm-variants" to build a certain variant of the hotspot. You can choose between server, client, minimal, core, zero, and custom. The built folder should contain a "java" and "javac" application in the "jdk/bin/" folder, which can be used to run and compile code.

2.3 Did My Code Execute (DMCE)

DMCE is a source code level instrumentation tool that works off of git repositories, this means that DMCE can only be used on code that is inside a repository. DMCE works by directly editing the source code inside all eligible expressions it finds. Which can be done due to a functionality in C and C++ where an expression can be nested before another expression, for example: $(INSERTED\_CODE(x), ORIGINAL\_CODE)$. This will not modify the result of the original program in any way, except if the inserted code has any side effects.

The main idea behind DMCE is to create what we define in this report as "probes", which are essentially snippets of C code that you design to run at every relevant expression from the source code. These probes will then instrument the code and give information depending on how the probe was implemented. DMCE includes some examples of probe code, which include printing the probe number whenever a probe is executed or creating a heatmap depending on what probes are executed.

Using DMCE usually involves two steps: The first is setting up the configurations and probe behavior for DMCE. The configuration options not only allow the user to tell DMCE what folders or files to exclude or include when probing but also give the option to write regular expressions that DMCE will use to exclude insertion of probes in the code. These settings can be specified when using the "dmce-set-profile" command or by editing the DMCE configuration files. The second step is to insert the probes into the desired source code. DMCE decides where to insert the probes depending on a couple of factors. Firstly, DMCE can only insert probes around expressions, and secondly, if an eligible expression is found, it will check it against the excluded configurations that are set up. This is done automatically by DMCE, but one can change what files that are to be probed depending on git commits as well. This is useful when one wants to only probe code that was recently added.

Probing generates a "probe-reference" file that keeps track of all the generated probes. This file is important for DMCE when generating data from runs and is located in a Linux specific temporary folder that will disappear if the computer is turned off. Therefore, it is important to save this file locally if the probing takes significant time.

After the execution of the compiled modified source code, DMCE will generate data inside a binary file located in the Linux temporary folder, along with the previously mentioned probe-reference file. For heatmap probes, the binary file will keep track of all probe hits, increasing the number of hits for probes when a program containing probes is executed. Therefore, the binary file needs to be deleted to reset the probe hits count. Readable data is obtained by running a command specifying the "probe-reference" file and the binary file. This generates a file with data depending on the probe behavior.

We learned this information about DMCE by reading in the DMCE Github repository

[3], but also by exploring the tool ourselves

2.4 The DaCapo benchmark suite

The DaCapo benchmarks are a set of open source Java benchmarks that were made to improve the way Java was benchmarked at the time [10]. In the paper, which is heavily cited, the benchmarks are described to be general purpose, realistic, and freely available applications for Java to evaluate how well Java performs on different computers. DaCapo was made specifically for Java 8, but still works for later releases of Java, although not all benchmarks are guaranteed to function. The latest stable release, as of this paper, of DaCapo is called DaCapo Bach, released in 2009. There is also an experimental branch on the DaCapo git called DaCapo Chopin, which contains several new benchmarks as well as updates to the old ones from DaCapo Bach. DaCapo Chopin has received continuous updates up until the release of this report.

The way DaCapo operates is quite simple. Any of its benchmark can be executed by running the DaCapo jar-file, the file we used was called dacapo-9.12-MR1-bach.jar. To run a benchmark, you execute the command: **java -jar dacapo-9.12-MR1-bach.jar 'benchmark'**. Here 'java' can be replaced by the path to any Java executable, which can be an executable produced when building an OpenJDK, and 'benchmark' is replaced by the benchmark you want to run. You can also add options such as -n, which tells DaCapo how many iterations of the benchmark to run. To be able to run a DaCapo Chopin, you have to build it from source code, which can be found here [2]. The benchmark **h2o** did not build successfully, therefore, we used a fix for this, see 1.2. When DaCapo Chopin is built, you can run a jar-file in the same way as the jar-file for DaCapo Bach.

2.4.1 DaCapo Bach and its benchmarks that works

On our setup using Java 17, 10 out of the 15 benchmarks in DaCapo Bach work. The benchmarks that do not work get runtime errors when we execute them. Therefore, in this paper, only benchmarks that work on our setup with Java 17 have been used to collect data on. Here is a table that shows which DaCapo Bach benchmarks that work on our setup:

avrora	*batik*	*eclipse*	*fop*	*h2*	*jython*	*luindex*	*lusearch*
x			x	x	x	x	x
lusearch-fix	*pmd*	*sunflow*	*tomcat*	*tradebeans*	*tradesoap*	*xalan*	
x	x	x				x	

2.4.2 DaCapo Chopin and its benchmarks that works

DaCapo Chopin includes all previous Dacapo Bach benchmarks and more. DaCapo Chopin also fixes some of the old benchmarks when using later Java versions. In total, DaCapo Chopin contains 22 benchmarks, of which 18 work on our setup. The 4 benchmarks that do not work on our setup receive runtime errors and will therefore not be used to collect data on. Here is a table that shows which DaCapo Chopin benchmarks that work on our setup:

avrora	*batik*	*biojava*	*cassandra*	*eclipse*	*fop*	*graphchi*	*h2*
x	x	x		x	x	x	x
h2o	*jme*	*jython*	*kafka*	*luindex*	*lusearch*	*pmd*	
x	x	x		x	x	x	
spring	*sunflow*	*tomcat*	*tradebeans*	*tradesoap*	*xalan*	*zxing*	
x	x	x			x	x	

2.5 Logistic regression models

Logistic regression can be used to tell how one aspect affects the probability of a certain outcome. In our case, given a probe's number of invocations received after running a benchmark, we will use a logistic regression model to see how that number affects the probability of the benchmark that was executed to have been a certain benchmark.

The logistic regression model is based on the odds of a 2-level outcome of interest. For one event of interest, the odds of that event happening is the probability of the event happening divided by the probability of that event not happening. This means that if the odds = 1, the event will happen half of the time. The logistic regression model uses the natural logarithm of the odds as a regression function [14]. scikit-learn's logistic regression model contains a function fit, which takes a matrix X and a vector y as input data [4]. Where row number i in the X matrix corresponds to the input values that produced the value of the element at index i in y. After running this fit function with X and y values, you can predict output values given inputs using the predict function. If you want to know how important different inputs are to produce different outputs, you can check the .coef_ attribute on the Logistic Regression model. This attribute contains a matrix where each row represents a possible output value and each column represents a certain input value. To give an understanding of what the coefficients in this matrix mean, here is an equation:

$$result = c_{00} * iv_{00} + c_{00} * iv_{01} + ... + c_{0max} * iv_{0max} \tag{2.1}$$

What "result" in 2.1 tells us is how likely the output is to be zero, given max number of input values, iv, and max number of coefficients, c, where the coefficients are received from the first row in the .coef_ matrix.

Chapter 3
Approach

This chapter explains the approach we took for each research question and answers we found along the way.

3.1 Instrumenting the OpenJDK

We used DMCE to probe the source code of an OpenJDK with heatmap probes. After this, we built the OpenJDK using the default build options, resulting in an OpenJDK with the name “linux-x86_64-server-release” containing 192244 probes. This means that our built OpenJDK has a server variant of the Hotspot and is built for the operating system Linux and the CPU architecture x86_64. With this probed OpenJDK we could measure the impact benchmarks have on the execution of JVM code. One important thing to mention is that we excluded all folders with OS and CPU versions our system was not running on in order to not have probes in code that will not be used by our system. Another aspect that is important to mention is that our probed OpenJDK has probes inside code that is specific to the compiler and will therefore never be invoked when we execute a benchmark or another Java program. Ideally, we would have liked to exclude code that only regards the compiler from being probed, but we do not know what parts of the source code in the OpenJDK that only regards the compiler. Since the number of probes for the OpenJDK may vary depending on DMCE setup, version of OpenJDK, the machine the probes are inserted on, etc., we used this same probed OpenJDK, built with OpenJDK version 17.0.6 and DMCE version 1.7.4, for all our data collection, meaning 192244 is a constant number for all our calculations. Then, using this probed OpenJDK, we ran the benchmarks and collected "heatmap data", i.e. data on what probes were executed and how many times. One advantage with this approach is that we got all of the relevant code instrumented automatically without having to insert any code manually. Another advantage was that it was easy to collect the data using existing DMCE commands.

3.2 Measuring performance

In order to answer RQ1.2(*How does the measuring affect execution speed of benchmarks?*) we also built the same version of the OpenJDK without any probes. We then ran all the DaCapo Bach and DaCapo Chopin benchmarks using both the OpenJDK with probes and the OpenJDK without probes and collected all the average execution times.

3.3 Collecting DMCE heatmap data

RQ1.3(*How stable are our measurements on different executions of the same program?*), RQ2(*How much of the JVM code is covered by state of the art benchmark suites?*), RQ3(*To what degree are our state of the art benchmarking suites exposing the performance of the JVM?*), and RQ4(*To what degree do the different benchmarks exercise different JVM code?*) all require heatmap data to be answered. The heatmap data that we needed to collect was data for both individual benchmarks being run and the full benchmark suites, i.e. running all benchmarks consecutively in DaCapo Bach or DaCapo Chopin and then saving the heatmap data. When running the benchmarks, we always used the option -n 20, which means that each benchmark is executed 19 times and then one final time, on which the performance is measured. We did this in order to make the JVM able to use JIT compilation on all methods in the benchmarks that have been executed enough times during the first 19 iterations of the program. This in turn means that we will have invoked probes in the OpenJDK source code regarding JIT compilation that we would not have had, had we only used the default options. In order to collect data regarding the JIT compilation, we collected DMCE data for all the 20 iterations. This is done automatically by DMCE, as the binary file will add the number of probe invocations during each iteration of the benchmark. In the rest of the report, when we mention a benchmark run or an execution of a benchmark, we mean running a benchmark with the -n 20 option, meaning, for example, that when we say we used data for 20 different benchmark runs, we mean that a benchmark has been executed with the -n 20 option 20 times. We also collected heatmap data from running our own Java programs, Lambda.java, SealedClasses.java, and SwingComponents.java (see Appendix B). We created Lambda.java as it contains features added in Java 8 [13], SealedClasses.java as it contains features added in Java 17 [7] and SwingComponents.java as it uses a library that is common in Java programs. All of these programs had a for loop iterating the entire program 20 times to achieve the same effect as using -n 20 when running the DaCapo benchmarks. In order to collect all this data, we wrote different bash scripts that we used.

3.4 Measurement variations

To answer RQ1.3 (*How stable are our measurements on different executions of the same program?*) we analyzed heatmap data from different runs of all DaCapo Bach and DaCapo Chopin benchmarks. We first wrote Python code that, for each benchmark, checked how much each probe's number of invocations would vary for different runs of that benchmark. To get an estimate of how much the JVM code that gets executed varies for different benchmarks, we added all these numbers for all probes for each benchmark. Then, for each benchmark, we calculated the average number of total probe invocations based on the same runs as the variations were calculated in order to be able to compare against the variation. We also wanted to know a bit more about how much different probes' numbers of invocations varied on average. To do this, we created a plot for each benchmark showing how many probes had a certain percentage of probe variation using Python.

3.5 Evaluating the DMCE heatmap data

Our approach to answer RQ2(*How much of the JVM code is covered by state of the art benchmark suites?*) was to analyze heatmap data received from running all DaCapo Bach and DaCapo Chopin benchmarks. For both DaCapo Bach and DaCapo Chopin, the amount of probes invoked at least once was collected for several runs of all individual benchmarks, then for each benchmark, the minimum and maximum value were collected. Furthermore, for both DaCapo Bach and DaCapo Chopin, the amount of probes invoked at least once after running every single benchmark was collected, and this was repeated several times, and minimum and maximum values were kept. Lastly, based on mean values, percentages of the coverage were calculated by dividing the mean values with 192244, the total number of probes in our OpenJDK.

To answer RQ3.1(*Are there any parts of the JVM code that get executed by other Java programs that do not get hit by the benchmarks?*) we wrote a Python program which checked if any probes were invoked by our Java programs but not by any DaCapo Bach or DaCapo Chopin benchmarks.

We answered RQ3.2(*Does the distribution of JVM code executed by the benchmarks reflect that of other Java programs?*) by comparing heatmap data for our own Java programs and the full DaCapo Bach and DaCapo Chopin benchmark suites. We did this by generating an Excel document showing probe invocations for all folders using Python.

To answer RQ4(*To what degree do the different benchmarks exercise different JVM code?*) we started with writing Python code generating two Excel Documents, one for DaCapo Bach and one for DaCapo Chopin. These documents show the average number of probe invocations each directory had for all the different benchmarks.

3.6 Creating a model for our heatmap data

We decided to use scikit-learn's logistic regression model [4], in order to answer RQ4.1(*For each individual benchmark, what degree of importance do different parts of the JVM code have for that benchmark?*). We used two separate logistic regression models, one for DaCapo Bach and one for DaCapo Chopin. The models take the number of probe invocations for all probes as input and predict what benchmark has been executed given these numbers. To train the DaCapo Bach model, we used the fit function with heatmap data from 360 DaCapo Bach benchmark runs (40 runs per benchmark), where data from runs 180 (20 runs for each benchmark) were collected by two machines, see 4.1 for details on these machines. We also noted that lusearch-fix contains a bugfix for lusearch [5]. This resulted in the model having a hard time telling the difference between lusearch and lusearch-fix because of their similarities. Therefore, we decided to exclude lusearch from the data used to train the model. In order to train the DaCapo

Chopin model, we used the fit function with heatmap data from 360 DaCapo Chopin runs (20 runs per benchmark), all this data was collected on the same machine. We transformed the heatmap data from the benchmark runs into arrays, where element number j represents the number of times probe number j was invoked. These arrays were combined to produce the $\boldsymbol{X}$-matrix, where each row, i represented a benchmark run. We also gave each benchmark a number and kept track of what benchmarks were run for all runs and stored the number for benchmark run number i in the y array as element number i. You could train on this data directly and get a logistic regression model that can predict, given heatmap data, what benchmark had been run. However, you can also preprocess the data in $\boldsymbol{X}$ in several ways to get different predictions. One way we used to check how good different ways of preprocessing the data were was to check how many correct predictions we would get using test sets we produced. For DaCapo Bach, the test set contained heatmap data from the same number of runs, collected on the same machines, as we used to train our models, but the runs were separate from what we used to train our models. The test set for DaCapo Chopin contained heatmap data for the same number of runs as the training set, but it was collected on the other machine. The test set was also preprocessed in the same way as the data we had used to train our model had been preprocessed. Then, for each run in the test set, we predicted what benchmark had been executed using our models and compared it against our validation set, containing the correct answer to what benchmark had been executed. The only way of preprocessing that we tried that got 100% correct predictions for all DaCapo Bach and DaCapo Chopin benchmark runs in the test set was first adding 1 to each element in the $\boldsymbol{X}$-matrix and then using its 10 logarithmic value instead, and lastly normalizing each row in the matrix using sklearn.preprocessing.normalize with norm=max. The other ways of preprocessing data also had a high prediction rate, and not preprocessing the data at all had 99.4% correct predictions. We believe that first transforming the data to logarithmic values is good as it makes differences in smaller values have a higher importance than for larger values, for example, the difference from 1 to 100 will be valued higher than the difference from 1000000 to 2000000 when using logarithmic values, which we believe is good for our case. Normalizing the data is also good, as it removes any possible data redundancy and minimizes possible data modification errors. With all this in mind, we decided to use this way of preprocessing $\boldsymbol{X}$. Given this, we used the coefficient matrix received by using the coef_ attribute in the logistic regression models to tell how each probe's number of invocations affect the likelihood of all all different benchmarks being predicted.

3.7 Identifying important parts of the JVM code

To answer RQ4.1 (*For each individual benchmark, what degree of importance do different parts of the JVM code have for that benchmark?*) we first have to answer the question: what makes parts of the JVM code important for a certain benchmark? To answer this question we start by dividing the JVM Code into lines represented by the probes. We consider a specific line to be important for a benchmark if that line has significantly more invocations for that benchmark than for others. The coefficients in the coefficient matrices, received from our logistic regression models, tells us this exactly, the probe with the highest coefficient will be the probe that has the highest number of probe invocations in relation to the number of probe invocations for all other benchmarks.

In order to answer RQ4.2(*To what degree are different parts of the JVM code heavily exercised by all benchmarks?*) we first analyzed which probes had the same number of invocations for all benchmarks for both DaCapo Bach and DaCapo Chopin. Then we aggregated these probes' number of invocations to their corresponding directories. This gave us a result that told us how many invocations, only counting probes with an equal number of invocations, each directory had.

Chapter 4
Evaluation

In this chapter, we will discuss the results we got from the approach we took and also the details of the machines that gathered the results. The main goal of the evaluation is to present our results, then discuss how we can answer our research questions with them and the threats to validity we found.

4.1 Experimental setup

The experimental setup involved two identical virtual machines provided by Ericsson. All the heatmap data was collected after running benchmarks or our own Java programs on these machines.

For the DMCE setup, clang-check15 and DMCE release 1.7.4 is used to gather as many probes as possible. The OpenJDK used is the OpenJDK 17.0.6+10 release. The virtual machines have Ubuntu 22.04.2 LTS (GNU/Linux 5.15.0-25-generic x86_64) as operating system. They are running on machines with the CPU model Intel(R) Xeon(R) CPU E5-2670, which have a frequency of 2.60 GHz and 16 cores. The virtual machines have access to at most 4 of these cores at a time and have 15.6 GB of RAM available in total. The JVM in our built OpenJDK has a maximum heap size of 3.9 GB on the virtual machines.

4.2 Results

To answer RQ1.2(*How does the measuring affect execution speed of benchmarks?*) we ran every DaCapo Bach and DaCapo Chopin benchmarks 20 times each using both the probed and unprobed OpenJDK and collected the execution times for several runs on the same machine. We then used these times to calculate an average execution time for each benchmark and 95% confidence interval for the execution times, this we did for both the probed and unprobed OpenJDK. See 4.1 and 4.2.

Table 4.1: Execution times of DaCapo Bach benchmarks for both probed and unprobed OpenJDK, and the percentile slowdown for the probed version (negative if the probed version is faster). The table also includes the confidence interval with a confidence level of 95%, the maximum and minimum slowdown based on the confidence intervals is also shown. (*The slowdown use the geometric average of the other benchmarks)

	avrora	**fop**	**h2**	**jython**	**luindex**	**lusearch**
Average Unprobed Times (ms)	22121.7	604.0	6722.2	2830.3	1230.1	1215.0
Average Probed Times (ms)	27045.1	657.5	6325.3	3357.9	1354.8	1826.9
Average Percentage Slowdown	22.26%	8.85%	−5.90%	18.64%	10.14%	50.36%
Unprobed Confidence Interval (ms)	21161.0	563.8	6500.4	2785.3	1160.9	1192.4
	23082.3	644.2	6943.9	2875.3	1299.2	1237.6
Probed Confidence Interval (ms)	26042.3	638.1	6081.8	3289.5	1259.7	1765.9
	28047.8	676.8	6568.8	3426.4	1449.9	1887.8
Min-Max Percentage Slowdown	32.55%	20.04%	1.05%	23.02%	24.90%	58.32%
	12.82%	−0.95%	−12.41%	14.41%	−3.04%	42.68%

	lusearch-fix	**pmd**	**sunflow**	**xalan**	**average**
Average unprobed Times (ms)	1188.8	2154.3	2447.2	1253.0	4176.6
Average Probed Times (ms)	1769.8	2765.6	3093.6	1447.8	4964.4
Average Percentage Slowdown	48.87%	28.37%	26.42%	15.55%	*21.25%
Unprobed Confidence Interval (ms)	1167.0	2081.4	2368.4	1203.9	4018.4
	1210.6	2227.2	2526	1302.1	4334.9
Probed Confidence Interval (ms)	1739.7	2684.0	3033.5	1396.9	4793.1
	1799.9	2847.1	3154	1498.7	5135.7
Min-Max Percentage Slowdown	54.24%	36.79%	33.17%	24.49%	*29.90%
	43.70%	20.51%	20.09%	7.28%	*13.20%

Table 4.2: Execution times of DaCapo Chopin benchmarks for both probed and unprobed OpenJDK, and the percentile slowdown for the probed version. The table also includes the confidence interval with a confidence level of 95%, the maximum and minimum slowdown based on the confidence intervals is also shown. (*The slowdown use the geometric average of the other benchmarks)

	avrora	**batik**	**biojava**	**eclipse**	**fop**
Unprobed Times (ms)	35757.6	3008.5	23891.0	33049.3	1595.2
Probed Times (ms)	48198.0	3623.0	24045.3	35715.1	2042.3
Percentage Slowdown	34.79%	20.42%	0.65%	8.07%	28.03%
Unprobed Confidence Interval (ms)	33884.0	2920.8	23712.8	32463.2	1571.2
	37631.1	3096.2	24069.2	33635.4	1619.2
Probed Confidence Interval (ms)	47376.9	3584.5	23967.2	35458.2	2031.8
	49019.1	3661.4	24123.4	35971.9	2052.8
Min-Max Percentage Slowdown	44.67%	25.35%	1.73%	10.81%	30.66%
	25.90%	15.77%	−0.42%	5.42%	25.48%

	graphchi	**h2**	**h2o**	**jme**	**jython**
Unprobed Times (ms)	15182.1	6399.1	12602.8	7256.9	7642.2
Probed Times (ms)	19411.6	6813.3	12301.1	7712.7	9671.4
Percentage Slowdown	27.86%	6.47%	−2.39%	6.28%	26.55%
Unprobed Confidence Interval (ms)	14971.5	6111.5	12142.7	7219.1	7507.0
	15392.6	6686.6	13062.8	7294.6	7777.4
Probed Confidence Interval (ms)	19319.3	6687.3	12099.4	7696.1	9612.1
	19503.9	6939.3	12502.7	7729.2	9730.6
Min-Max Percentage Slowdown	30.27%	13.54%	2.96%	7.07%	29.62%
	25.51%	0.01%	−7.38%	5.50%	23.59%

	luindex	**lusearch**	**pmd**	**spring**	**sunflow**
Unprobed Times (ms)	11965.8	19096.5	5591.8	14740.5	19267.5
Probed Times (ms)	11580.3	19147.2	8126.7	16062.9	23330.7
Percentage Slowdown	−3.22%	0.27%	45.33%	8.97%	21.09%
Unprobed Confidence Interval (ms)	11727.0	18690.1	5370.0	14465.0	18693.3
	12204.5	19502.9	5813.5	15015.9	19841.7
Probed Confidence Interval (ms)	11475.6	18969.1	8029.5	15942.2	23079.0
	11684.9	19325.3	8223.9	16183.6	23582.4
Min-Max Percentage Slowdown	−0.36%	3.40%	53.15%	11.88%	26.15%
	−5.57%	−2.74%	38.12%	6.17%	16.32%

	tomcat	**xalan**	**zxing**	**average**	
Unprobed Times (ms)	38706.2	6874.6	7385.5	15000.7	
Probed Times (ms)	41395.0	7440.1	9834.0	17025.0	
Percentage Slowdown	6.95%	8.23%	33.15%	*14.58%	
Unprobed Confidence Interval (ms)	38396.7	6690.4	7149.8	14649.2	
	39015.6	7058.7	7621.2	15352.2	
Probed Confidence Interval (ms)	41259.3	7359.3	9730.7	16871.0	
	41530.6	7520.8	9937.3	17179.1	
Min-Max Percentage Slowdown	8.16%	12.41%	38.99%	*18.51%	
	5.75%	4.26%	27.68%	*10.85%	

To answer RQ1.3 (*How stable are our measurements on different executions of the same program?*) we analyzed heatmap data from 20 different runs of all DaCapo Bach and DaCapo Chopin benchmarks. First, we wrote Python code that, for each benchmark, checked how much each probe's number of invocations would vary for different runs. This was done by comparing the number of probe invocations for all possible pairs of runs for a benchmark, where a possible pair is, for example, run number 1 and run number 2. Given 20 runs which we had for each benchmark, we could create $\binom{20}{2} = 190$ number of pairs for each benchmark. For each probe, we added the difference of probe invocations for all possible pairs, and then divided by the number of pairs, 190. By doing this, we got an estimate of how much each probe's number of invocations would vary on average. To get an estimate of how much the JVM code that gets executed varies for different benchmarks we added all these numbers for all probes for each benchmark. Then, for each benchmark, we calculated the average number of total probe invocations based on the same 20 runs as the variations were calculated, in order to be able to compare against the variation. The results from this can be seen in 4.3 and 4.4. All the data used to produce these tables was collected on the same machine.

Table 4.3: The average total number of probe invocations and the average variations for the number of invocations for DaCapo Bach. The percentages of the average number of probe invocations that varies is also shown

	avrora	**fop**	**h2**	**jython**
Average probe invocations	3181923199.5	5481397634.1	12629513326.4	15898056286.3
Average probe variation	145192479.1	249019260.4	756858448.0	1093660699.0
Percentage	4.56%	4.54%	5.99%	6.88%
	luindex	**lusearch**	**lusearch-fix**	**pmd**
Average probe invocations	3344329049.2	4351301627.1	4424622432.4	15587484381.6
Average probe variation	219457749.7	418370144.4	383476702.1	932686630.6
Percentage	6.56%	9.61%	8.67%	5.98%
	sunflow	**xalan**		
Average probe invocations	5554413737.4	9056625827.4		
Average probe variation	262010965.2	711095627.1		
Percentage	4.72%	7.85%		

Table 4.4: The average total number of probe invocations and the average variations for the number of invocations for DaCapo Chopin. The percentages of the average number of probe invocations that varies is also shown

	avrora	**batik**	**biojava**	**eclipse**
Average probe invocations	4765682213.9	9463672238.2	3713944553.4	81275756431.5
Average probe variation	193704804.8	951911642.8	188593948.0	3419337444.1
Percentage	4.06%	10.06%	5.08%	4.21%
	fop	**graphchi**	**h2**	**h2o**
Average probe invocations	15832968792.5	14311784501.4	19368821546.2	22003813066.9
Average probe variation	563707616.4	244697791.0	1506900734.4	1711119381.1
Percentage	3.56%	1.71%	7.78%	7.78%
	jme	**jython**	**luindex**	**lusearch**
Average probe invocations	5047796547.7	35312533978.4	7872287266.6	8688996106.1
Average probe variation	227699083.2	1396043582.39	524513142.1	699090156.3
Percentage	4.51%	3.95%	6.66%	8.05%
	pmd	**spring**	**sunflow**	**tomcat**
Average probe invocations	42155531491.9	60630586543.4	34978523971.4	50509264373.8
Average probe variation	1304857786.7	2601593864.0	1169258297.5	2505911317.6
Percentage	3.10%	4.29%	3.34%	4.96%
	xalan	**zxing**		
Average probe invocations	10307077601.7	20776158719.7		
Average probe variation	578644499.4	804979306.0		
Percentage	5.61%	3.87%		

In order to further answer RQ1.3 (*How stable are our measurements on different executions of the same program?*) we also wanted to know a bit more about how different probes' number of invocations varied on average. To do this, using the same heatmap data used to produce table 4.3 and 4.4, we created a plot for each benchmark. On the y-axis, we had the average percentage invocation variance for a probe, and on the x-axis, we had the percentage of all probes that had at least one invocation in any run of the benchmark. We decided to exclude the probes that only had zero invocations for all runs because they only represent code that never gets executed by that benchmark, and we are interested in how the number of probe invocations varies for code that gets executed by the benchmark. To explain the meaning of a plot, say, for example, that a plot would pass through (10, 25), that would mean that for that benchmark, 10% of the probes invoked at least once have an average percentage invocation variance of 25% or more. The resulting plots are found in figure 4.1 and 4.2.

Figure 4.1: Plots what percentage of probes have a certain number of invocation variation percentage on average for all DaCapo Bach benchmarks

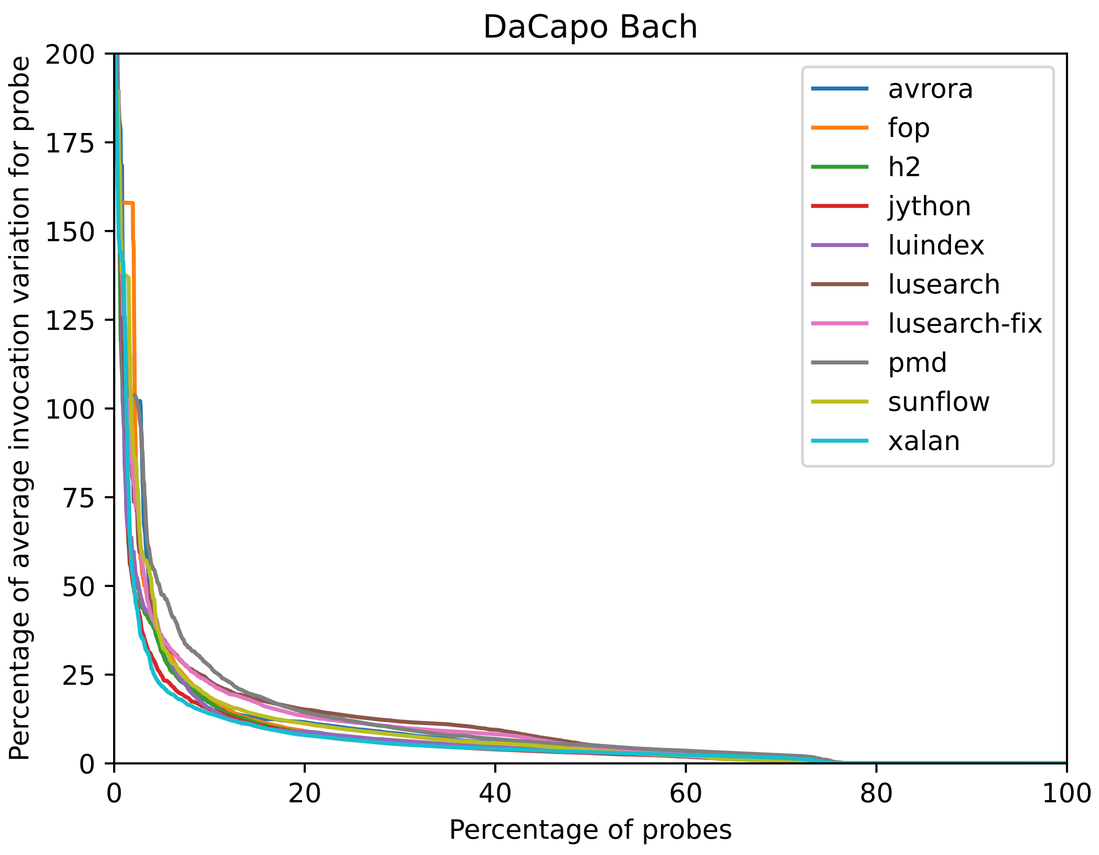
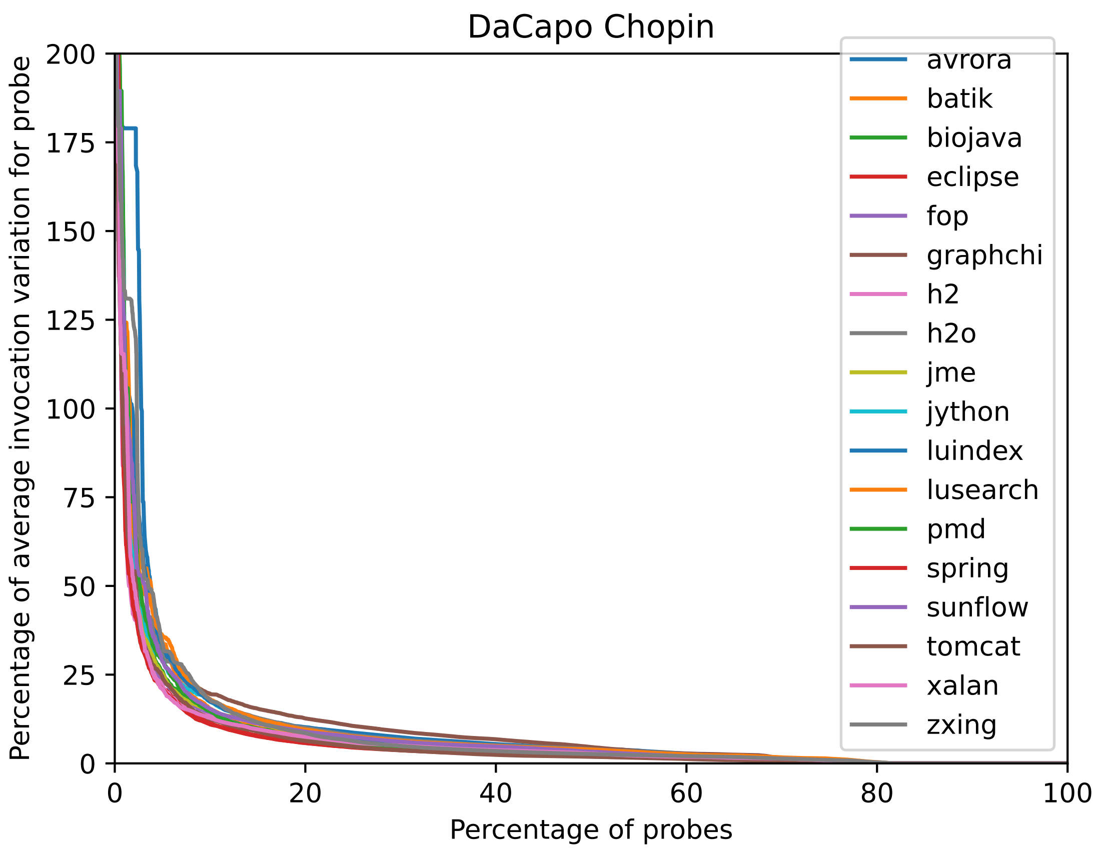


Figure 4.2: Plots what percentage of probes have a certain number of invocation variation percentage on average for all DaCapo Chopin benchmarks



To answer RQ2(*How much of the JVM code is covered by state of the art benchmark suites?*) we collected the number of unique probes invoked for 80 runs of every single benchmark for both DaCapo Bach and DaCapo Chopin and calculated the averages. Data from 40 of the 80 runs were collected on one machine and 40 from the other for both DaCapo Bach and DaCapo Chopin. We also, for both DaCapo Bach and DaCapo Chopin, collected the amount of unique probes invoked after running every benchmark consecutively and repeated this 20 times. All of this data collection was done on one machine. From these 20 values, we collected the minimum and maximum amount of unique probes invoked for both DaCapo Bach and DaCapo Chopin. Lastly, for all the benchmarks' values and the values for running all benchmarks in DaCapo Bach and DaCapo Chopin, averages were calculated and used to calculate what percentage of probes were invoked at least once, see table 4.5 for DaCapo Bach and table 4.6 for DaCapo Chopin.

Table 4.5: Coverage data from heatmap data for DaCapo Bach benchmarks, where the maximum and minimum of probes invoked at least once is shown for every benchmark. The average coverage shows the mean number of probes invocated at least on for all runs of the benchmarks divided by the total number of probes, 192244, resulting in average coverage percentages for all benchmarks. The "all benchmarks" column refers to runs where all the benchmarks has been run consecutively

	avrora	**fop**	**h2**	**jython**	**luindex**	**lusearch**
maximum	42884	44815	44619	45701	44366	43469
minimum	41954	43930	44425	45424	43407	43192
average coverage(%)	21.97	23.00	23.17	23.71	22.98	22.54
	lusearch-fix	**pmd**	**sunflow**	**xalan**	**all benchmarks**	
maximum	43485	45214	42691	43880	48620	
minimum	43134	44235	41964	43740	48482	
average coverage(%)	22.53	23.43	21.99	22.79	25.25	

Table 4.6: Coverage data from heatmap data for DaCapo Chopin benchmarks, where the maximum and minimum of probes invoked at least once is shown for every benchmark. The average coverage shows the mean number of probes invocated at least on for all runs of the benchmarks divided by the total number of probes, 192244, resulting in average coverage percentages for all benchmarks. The "all benchmarks" column refers to runs where all the benchmarks has been run consecutively

	avrora	**batik**	**biojava**	**eclipse**	**fop**
maximum	42859	45882	44848	48178	47219
minimum	41886	45627	43948	47948	46808
average coverage(%)	21.88	23.81	23.20	25.0	24.48
	graphchi	**h2**	**h2o**	**jme**	**jython**
maximum	44679	44453	48953	45881	46903
minimum	44325	44263	48489	45546	46502
average coverage(%)	23.12	23.07	25.35	23.82	24.29
	luindex	**lusearch**	**pmd**	**spring**	**sunflow**
maximum	44833	45790	47665	49068	42228
minimum	44021	45374	47122	48909	41815
average coverage(%)	23.27	23.72	24.65	25.46	21.86
	tomcat	**xalan**	**zxing**	**all benchmarks**	
maximum	48388	44378	46624	54092	
minimum	48199	44183	45743	53985	
average coverage(%)	25.12	23.05	24.05	28.10	

To answer RQ3.1(*Are there any parts of the JVM code that get executed by other Java programs that do not get hit by the benchmarks?*) we started with writing a Python program which checks if any probes were hit by our Java programs but not by any DaCapo Bach or DaCapo Chopin benchmarks. This analysis was performed on the average values of 20 runs for each program and 20 full DaCapo Bach and DaCapo Chopin runs, the result can be found in table 4.7 and 4.8. To get an understanding of where these probes are located, using Python, we created lists of what files the probes were located in, resulting in: 4.9 and 4.10. We also wanted to know which of these files that had no probes in them invoked by any benchmark, we wrote Python code that calculated this and marked these files with an asterisk in the tables. All the data used to create these tables was collected on the same machine.

Table 4.7: Unique number of probes and total probe invocations for all probes hit by program but not by any DaCapo Bach benchmarks

Program	Unique Number Probes	Total Average Probe Invocations
Lambda.java	90	105.25
SealedClasses.java	90	105.25
SwingComponents.java	313	2402.90

Table 4.8: Unique number of probes and total probe invocations for all probes hit by program but not by any DaCapo Chopin benchmarks

Program	Unique Number Probes	Total Average Probe Invocations
Lambda.java	2	2
SealedClasses.java	2	2
SwingComponents.java	141	2096

Table 4.9: Files containing probes hit by Java programs but not by any DaCapo Bach benchmark (*This file was exclusively hit by the program)

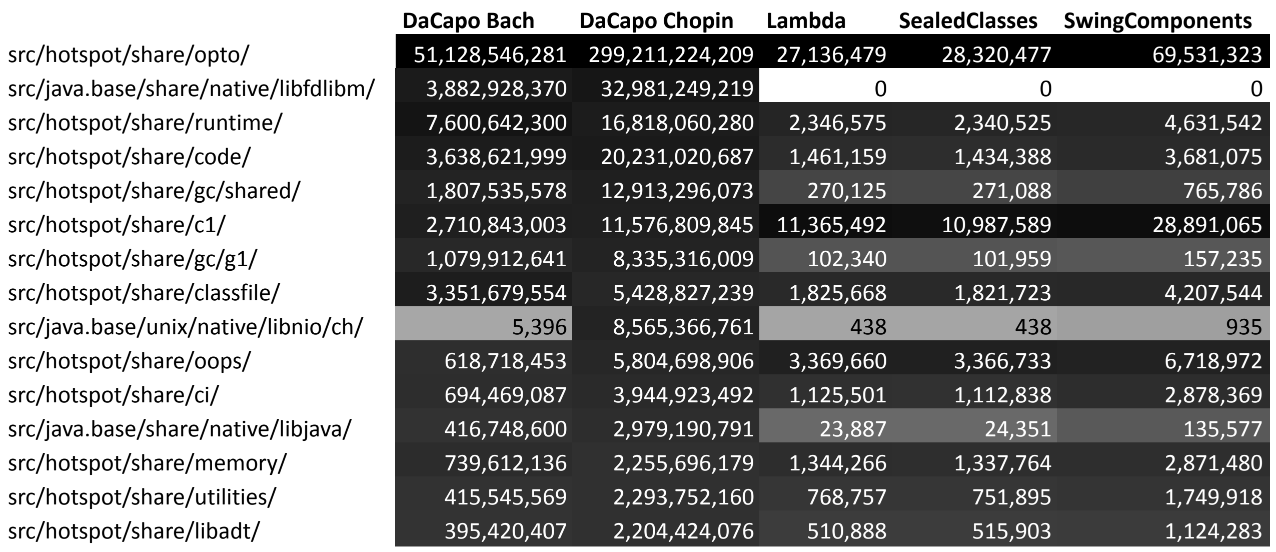
	Lambda	SealedClasses	SwingComponents
src/hotspot/share/ci/ciInstance.cpp			x
src/hotspot/share/ci/ciMethodData.cpp			x
src/hotspot/share/memory/universe.cpp	x	x	x
src/hotspot/share/oops/accessBackend.cpp			x
src/hotspot/share/prims/jni.cpp	x	x	x
src/hotspot/share/runtime/synchronizer.cpp	x	x	x
src/hotspot/share/runtime/thread.cpp	x	x	x
src/hotspot/share/runtime/vmThread.cpp	x	x	x
src/hotspot/share/services/threadService.cpp	x	x	x
src/java.base/share/native/libjli/java.c	x	x	x
src/java.base/share/native/libnet/net_util.c			x
src/java.base/unix/native/libjli/java_md.c	x	x	x
src/java.base/unix/native/libnet/Inet6AddressImpl.c *			x
src/java.desktop/share/native/libfontmanager/DrawGlyphList.c *			x
src/java.desktop/share/native/libfontmanager/freetypeScaler.c *			x
src/java.desktop/share/native/libfontmanager/sunFont.c *			x
src/java.desktop/unix/native/common/awt/fontpath.c *			x

Table 4.10: Files containing probes hit by Java programs but not by any DaCapo Chopin benchmark (*This file was exclusively hit by the program

	Lambda	SealedClasses	SwingComponents
src/hotspot/share/oops/accessBackend.cpp			x
src/hotspot/share/prims/jni.cpp			x
src/java.base/share/native/libjli/java.c	x	x	x
src/java.desktop/share/native/libfontmanager/freetypeScaler.c *			x
src/java.desktop/share/native/libfontmanager/sunFont.c			x
src/java.desktop/unix/native/common/awt/fontpath.c			x

To answer RQ3.2(*Does the distribution of JVM code executed by the benchmarks reflect that of other Java programs?*) we compared the heatmap data for our own Java programs, the full DaCapo Bach and DaCapo Chopin benchmark suites. We did this using Python code that generated an Excel document showing how many probe invocations each directory had on average for 20 full DaCapo Bach runs, 20 full DaCapo Chopin runs, 20 Lambda.java runs, 20 SealedClasses.java runs, and 20 SwingComponents.java runs. The cells were color coded in a way such that for each benchmark suite or program, the directory with the maximum number of probe invocations is black and directories with zero probe invocations are white, and all other directories are colored using a logarithmic scale. We sorted each row in the document based on the total number of probe invocations for the rows starting with the most invocations, the full document can be found published here [1]. We created an excerpt showing the first 15 rows from this document. These rows contain 97.6% of the total number of probe invocations, see 4.3. All the data used to create this figure was collected on the same machine.

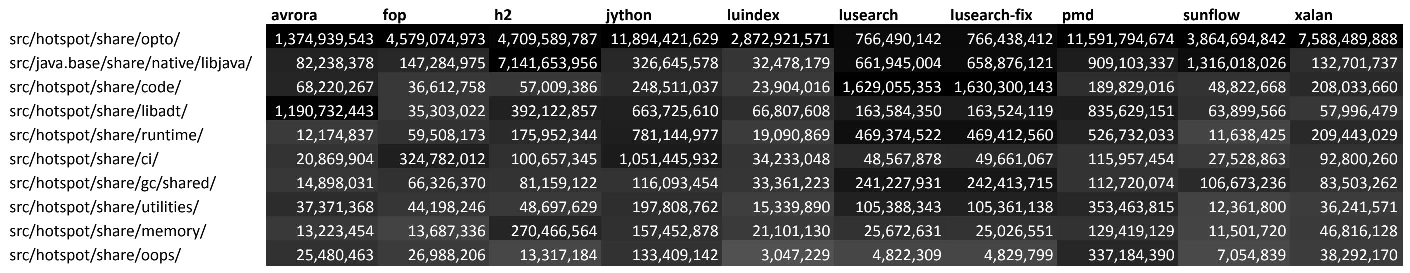
Figure 4.3: An excerpt of the Excel document showing average number of probe invocations for directories after running the DaCapo benchmark suites or our Java programs



	DaCapo Bach	DaCapo Chopin	Lambda	SealedClasses	SwingComponents
src/hotspot/share/opto/	51,128,546,281	299,211,224,209	27,136,479	28,320,477	69,531,323
src/java.base/share/native/libfdlibm/	3,882,928,370	32,981,249,219	0	0	0
src/hotspot/share/runtime/	7,600,642,300	16,818,060,280	2,346,575	2,340,525	4,631,542
src/hotspot/share/code/	3,638,621,999	20,231,020,687	1,461,159	1,434,388	3,681,075
src/hotspot/share/gc/shared/	1,807,535,578	12,913,296,073	270,125	271,088	765,786
src/hotspot/share/c1/	2,710,843,003	11,576,809,845	11,365,492	10,987,589	28,891,065
src/hotspot/share/gc/g1/	1,079,912,641	8,335,316,009	102,340	101,959	157,235
src/hotspot/share/classfile/	3,351,679,554	5,428,827,239	1,825,668	1,821,723	4,207,544
src/java.base/unix/native/libnio/ch/	5,396	8,565,366,761	438	438	935
src/hotspot/share/oops/	618,718,453	5,804,698,906	3,369,660	3,366,733	6,718,972
src/hotspot/share/ci/	694,469,087	3,944,923,492	1,125,501	1,112,838	2,878,369
src/java.base/share/native/libjava/	416,748,600	2,979,190,791	23,887	24,351	135,577
src/hotspot/share/memory/	739,612,136	2,255,696,179	1,344,266	1,337,764	2,871,480
src/hotspot/share/utilities/	415,545,569	2,293,752,160	768,757	751,895	1,749,918
src/hotspot/share/libadt/	395,420,407	2,204,424,076	510,888	515,903	1,124,283

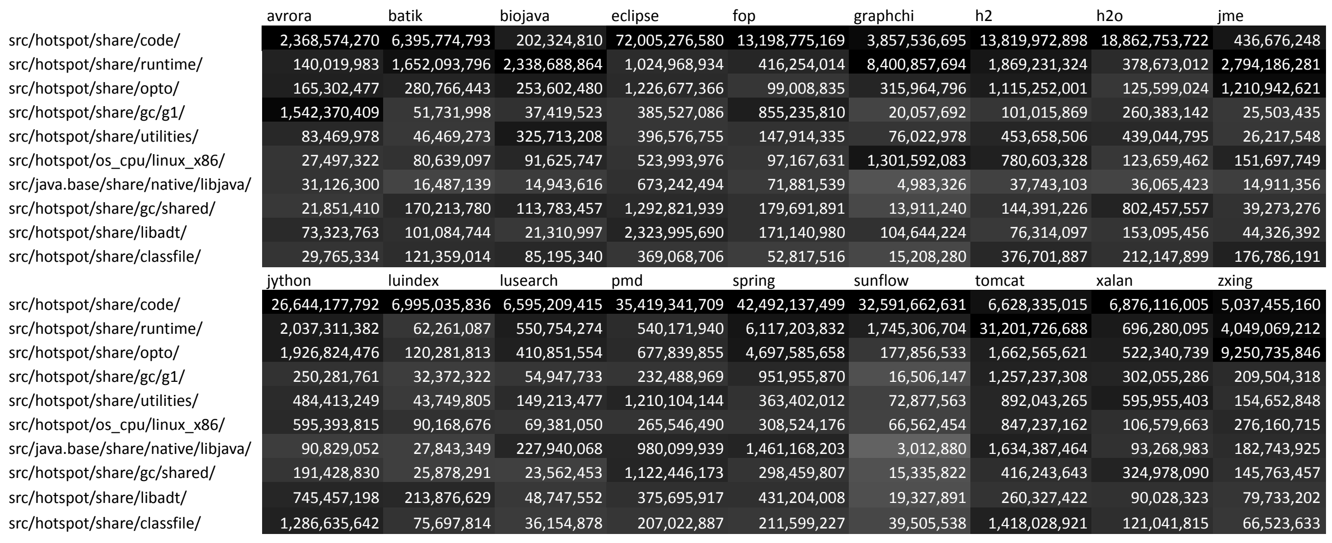
For RQ4(*To what degree do the different benchmarks exercise different JVM code?*) we compiled the information we got from 80 DaCapo Bach runs and 80 Chopin runs to create Excel tables that show how many invocations each folder in the OpenJDK gets on average for the benchmarks. Data for 40 of these 80 runs were collected on one machine and the other 40 on the other machine, for both DaCapo Bach and DaCapo Chopin. The cells are also colored using the same coloring method we used for 4.3. Using Excel, we also sorted the benchmarks depending on the total invocations for the folders. The data for DaCapo Bach can be seen in 4.4 and this shows an excerpt of the whole Excel document. For DaCapo Chopin, see 4.5, in this case, DaCapo Chopin had too many columns to present them all in one row, hence the repeating folders. The figure shows the top ten folders in regards to the highest amount of invocations for the folder across all benchmark runs. For DaCapo Bach, the invocations shown represent 96.3% of all invocations, while the top ten in DaCapo Chopin represent 96.4% of all its invocations. The full Excel table and calculations can be found published here [1].

Figure 4.4: An excerpt of the Excel document that shows the number of invocations for every folder in the OpenJDK depending on the benchmark from DaCapo Bach



	avrora	**fop**	**h2**	**jython**	**luindex**	**lusearch**	**lusearch-fix**	**pmd**	**sunflow**	**xalan**
src/hotspot/share/opto/	1,374,939,543	4,579,074,973	4,709,589,787	11,894,421,629	2,872,921,571	766,490,142	766,438,412	11,591,794,674	3,864,694,842	7,588,489,888
src/java.base/share/native/libjava/	82,238,378	147,284,975	7,141,653,956	326,645,578	32,478,179	661,945,004	658,876,121	909,103,337	1,316,018,026	132,701,737
src/hotspot/share/code/	68,220,267	36,612,758	57,009,386	248,511,037	23,904,016	1,629,055,353	1,630,300,143	189,829,016	48,822,668	208,033,660
src/hotspot/share/libadt/	1,190,732,443	35,303,022	392,122,857	663,725,610	66,807,608	163,584,350	163,524,119	835,629,151	63,899,566	57,996,479
src/hotspot/share/runtime/	12,174,837	59,508,173	175,952,344	781,144,977	19,090,869	469,374,522	469,412,560	526,732,033	11,638,425	209,443,029
src/hotspot/share/ci/	20,869,904	324,782,012	100,657,345	1,051,445,932	34,233,048	48,567,878	49,661,067	115,957,454	27,528,863	92,800,260
src/hotspot/share/gc/shared/	14,898,031	66,326,370	81,159,122	116,093,454	33,361,223	241,227,931	242,413,715	112,720,074	106,673,236	83,503,262
src/hotspot/share/utilities/	37,371,368	44,198,246	48,697,629	197,808,762	15,339,890	105,388,343	105,361,138	353,463,815	12,361,800	36,241,571
src/hotspot/share/memory/	13,223,454	13,687,336	270,466,564	157,452,878	21,101,130	25,672,631	25,026,551	129,419,129	11,501,720	46,816,128
src/hotspot/share/oops/	25,480,463	26,988,206	13,317,184	133,409,142	3,047,229	4,822,309	4,829,799	337,184,390	7,054,839	38,292,170

Figure 4.5: An excerpt of the Excel document that shows the number of invocations for every folder in the OpenJDK depending on the benchmark from DaCapo Chopin



	avrora	batik	biojava	eclipse	fop	graphchi	h2	h2o	jme
src/hotspot/share/code/	2,368,574,270	6,395,774,793	202,324,810	72,005,276,580	13,198,775,169	3,857,536,695	13,819,972,898	18,862,753,722	436,676,248
src/hotspot/share/runtime/	140,019,983	1,652,093,796	2,338,688,864	1,024,968,934	416,254,014	8,400,857,694	1,869,231,324	378,673,012	2,794,186,281
src/hotspot/share/opto/	165,302,477	280,766,443	253,602,480	1,226,677,366	99,008,835	315,964,796	1,115,252,001	125,599,024	1,210,942,621
src/hotspot/share/gc/g1/	1,542,370,409	51,731,998	37,419,523	385,527,086	855,235,810	20,057,692	101,015,869	260,383,142	25,503,435
src/hotspot/share/utilities/	83,469,978	46,469,273	325,713,208	396,576,755	147,914,335	76,022,978	453,658,506	439,044,795	26,217,548
src/hotspot/os_cpu/linux_x86/	27,497,322	80,639,097	91,625,747	523,993,976	97,167,631	1,301,592,083	780,603,328	123,659,462	151,697,749
src/java.base/share/native/libjava/	31,126,300	16,487,139	14,943,616	673,242,494	71,881,539	4,983,326	37,743,103	36,065,423	14,911,356
src/hotspot/share/gc/shared/	21,851,410	170,213,780	113,783,457	1,292,821,939	179,691,891	13,911,240	144,391,226	802,457,557	39,273,276
src/hotspot/share/libadt/	73,323,763	101,084,744	21,310,997	2,323,995,690	171,140,980	104,644,224	76,314,097	153,095,456	44,326,392
src/hotspot/share/classfile/	29,765,334	121,359,014	85,195,340	369,068,706	52,817,516	15,208,280	376,701,887	212,147,899	176,786,191

	jython	luindex	lusearch	pmd	spring	sunflow	tomcat	xalan	zxing
src/hotspot/share/code/	26,644,177,792	6,995,035,836	6,595,209,415	35,419,341,709	42,492,137,499	32,591,662,631	6,628,335,015	6,876,116,005	5,037,455,160
src/hotspot/share/runtime/	2,037,311,382	62,261,087	550,754,274	540,171,940	6,117,203,832	1,745,306,704	31,201,726,688	696,280,095	4,049,069,212
src/hotspot/share/opto/	1,926,824,476	120,281,813	410,851,554	677,839,855	4,697,585,658	177,856,533	1,662,565,621	522,340,739	9,250,735,846
src/hotspot/share/gc/g1/	250,281,761	32,372,322	54,947,733	232,488,969	951,955,870	16,506,147	1,257,237,308	302,055,286	209,504,318
src/hotspot/share/utilities/	484,413,249	43,749,805	149,213,477	1,210,104,144	363,402,012	72,877,563	892,043,265	595,955,403	154,652,848
src/hotspot/os_cpu/linux_x86/	595,393,815	90,168,676	69,381,050	265,546,490	308,524,176	66,562,454	847,237,162	106,579,663	276,160,715
src/java.base/share/native/libjava/	90,829,052	27,843,349	227,940,068	980,099,939	1,461,168,203	3,012,880	1,634,387,464	93,268,983	182,743,925
src/hotspot/share/gc/shared/	191,428,830	25,878,291	23,562,453	1,122,446,173	298,459,807	15,335,822	416,243,643	324,978,090	145,763,457
src/hotspot/share/libadt/	745,457,198	213,876,629	48,747,552	375,695,917	431,204,008	19,327,891	260,327,422	90,028,323	79,733,202
src/hotspot/share/classfile/	1,286,635,642	75,697,814	36,154,878	207,022,887	211,599,227	39,505,538	1,418,028,921	121,041,815	66,523,633

To answer RQ4.1 (*For each individual benchmark, what degree of importance do different parts of the JVM code have for that benchmark?*) we wrote Python code that analyzed the coefficient matrices from the logistic regression models for both DaCapo Bach and DaCapo Chopin. This code created a text file for each benchmark where all lines of code, represented by a probe, and their coefficients were printed in order of the coefficients, with the highest coefficient first. We took the top three lines of code and their coefficients from each of these files and put them into 4.11, 4.12, and 4.13.

Table 4.11: Top three lines of codes and their coefficients, based on their coefficients received from the coefficient matrix in the logistic regression model for DaCapo Bach

avrora	line of code	coefficient
	src/hotspot/share/runtime/objectMonitor.cpp:1714	0.03970
	src/hotspot/share/runtime/objectMonitor.cpp:1713	0.03970
	src/hotspot/share/runtime/objectMonitor.cpp:1710	0.03970
fop	line of code	coefficient
	src/hotspot/share/oops/method.cpp:1854	0.05493
	src/hotspot/share/oops/method.cpp:1864	0.05472
	src/hotspot/share/oops/method.cpp:1855	0.05472
h2	line of code	coefficient
	src/hotspot/os/posix/os_posix.cpp:1470	0.03590
	src/hotspot/os/posix/os_posix.cpp:1468	0.03590
	src/hotspot/share/gc/g1/sparsePRT.cpp:320	0.03255
jython	line of code	coefficient
	src/hotspot/share/oops/methodData.cpp:425	0.04591
	src/hotspot/share/oops/methodData.cpp:426	0.04469
	src/hotspot/share/oops/methodData.cpp:427	0.04469
luindex	line of code	coefficient
	src/java.base/unix/native/libjava/UnixFileSystem_md.c:265	0.03913
	src/java.base/share/native/libjava/io_util.c:219	0.03633
	src/java.base/share/native/libjava/io_util.c:186	0.03633
lusearch-fix	line of code	coefficient
	src/hotspot/share/opto/runtime.cpp:1419	0.04791
	src/hotspot/share/opto/runtime.cpp:1423	0.04791
	src/hotspot/share/opto/runtime.cpp:1427	0.04791
pmd	line of code	coefficient
	src/java.base/share/native/libjava/System.c:76	0.03697
	src/hotspot/share/classfile/systemDictionary.cpp:1276	0.03546
	src/hotspot/share/gc/g1/g1RemSet.cpp:1816	0.03365
sunflow	line of code	coefficient
	src/java.base/share/native/libfdlibm/e_sqrt.c:188	0.03961
	src/java.base/share/native/libfdlibm/e_sqrt.c:200	0.03960
	src/java.base/share/native/libfdlibm/e_sqrt.c:201	0.03959
xalan	line of code	coefficient
	src/hotspot/share/runtime/synchronizer.cpp:705	0.04558
	src/hotspot/share/runtime/synchronizer.cpp:704	0.04558
	src/hotspot/share/runtime/synchronizer.cpp:936	0.04444

Table 4.12: Top three lines of codes and their coefficients, based on their coefficients received from the coefficient matrix in the logistic regression model for DaCapo Chopin

avrora	line of code	coefficient
	src/hotspot/share/runtime/objectMonitor.cpp:1714	0.03669
	src/hotspot/share/runtime/objectMonitor.cpp:1713	0.03669
	src/hotspot/share/runtime/objectMonitor.cpp:1710	0.03669
batik	line of code	coefficient
	src/hotspot/share/gc/shared/referenceProcessor.cpp:1064	0.02557
	src/java.desktop/share/native/libawt/java2d/loops /GraphicsPrimitiveMgr.c:521	0.02507
	src/java.desktop/share/native/libawt/java2d/loops/MaskFill.c:113	0.02507
biojava	line of code	coefficient
	src/java.base/share/native/libjava/io_util.c:159	0.03858
	src/java.base/share/native/libjava/io_util.c:157	0.03858
	src/java.base/share/native/libjava/io_util.c:150	0.03858
eclipse	line of code	coefficient
	src/java.base/share/native/libjava/io_util.c:68	0.03034
	src/java.base/unix/native/libnio/fs/UnixNativeDispatcher.c:893	0.02595
	src/java.base/share/native/libjava/io_util.c:62	0.02582
fop	line of code	coefficient
	src/hotspot/share/oops/method.cpp:1854	0.03160
	src/hotspot/share/oops/method.cpp:1864	0.03152
	src/hotspot/share/oops/method.cpp:1855	0.03152
graphchi	line of code	coefficient
	src/hotspot/share/oops/typeArrayKlass.cpp:120	0.03824
	src/hotspot/share/oops/objArrayKlass.cpp:202	0.03624
	src/hotspot/share/oops/objArrayKlass.cpp:204	0.03621
h2	line of code	coefficient
	src/hotspot/os/posix/os_posix.cpp:1478	0.04460
	src/hotspot/os/posix/os_posix.cpp:1479	0.04459
	src/hotspot/os/posix/os_posix.cpp:1476	0.04459
h2o	line of code	coefficient
	src/java.base/share/native/libfdlibm/s_log1p.c:201	0.05676
	src/java.base/share/native/libfdlibm/s_log1p.c:170	0.05675
	src/java.base/share/native/libfdlibm/s_log1p.c:166	0.05674
jme	line of code	coefficient
	src/hotspot/share/utilities/copy.cpp:280	0.02886
	src/hotspot/share/utilities/copy.cpp:279	0.02886
	src/java.desktop/share/native/libjavajpeg/jdcoefct.c:308	0.02722

Table 4.13: Top three lines of codes and their coefficients, based on their coefficients received from the coefficient matrix in the logistic regression model for DaCapo Chopin

jython	line of code	coefficient
	src/hotspot/share/gc/shared/referenceProcessor.cpp:436	0.03602
	src/hotspot/share/gc/shared/referenceProcessor.cpp:434	0.03601
	src/hotspot/share/gc/shared/referenceProcessor.cpp:444	0.03586
luindex	line of code	coefficient
	src/java.base/unix/native/libnio/ch/FileDispatcherImpl.c:137	0.04396
	src/java.base/unix/native/libnio/ch/FileDispatcherImpl.c:134	0.04396
	src/java.base/unix/native/libnio/fs/UnixNativeDispatcher.c:893	0.02543
lusearch	line of code	coefficient
	src/java.base/unix/native/libnio/ch/FileChannelImpl.c:86	0.03853
	src/java.base/unix/native/libnio/ch/FileChannelImpl.c:85	0.03853
	src/java.base/unix/native/libnio/ch/FileChannelImpl.c:166	0.03853
pmd	line of code	coefficient
	src/hotspot/share/runtime/reflection.cpp:388	0.03767
	src/hotspot/share/classfile/javaClasses.cpp:2954	0.03687
	src/hotspot/share/classfile/javaClasses.cpp:2949	0.03687
spring	line of code	coefficient
	src/hotspot/share/runtime/reflection.cpp:247	0.04704
	src/hotspot/share/runtime/reflection.cpp:251	0.04704
	src/hotspot/share/runtime/reflection.cpp:252	0.04704
sunflow	line of code	coefficient
	src/java.base/share/native/libfdlibm/e_sqrt.c:200	0.04206
	src/java.base/share/native/libfdlibm/e_sqrt.c:201	0.04205
	src/java.base/share/native/libfdlibm/e_sqrt.c:188	0.04205
tomcat	line of code	coefficient
	src/hotspot/share/interpreter/bytecodeUtils.cpp:410	0.03917
	src/hotspot/share/interpreter/bytecodeUtils.cpp:385	0.03906
	src/hotspot/share/interpreter/bytecodeUtils.cpp:417	0.03851
xalan	line of code	coefficient
	src/java.base/share/native/libjava/io_util.c:69	0.03287
	src/java.base/share/native/libjava/io_util.c:72	0.03287
	src/java.base/share/native/libjava/io_util.c:62	0.03257
zxing	line of code	coefficient
	src/java.desktop/share/native/libawt/java2d/loops/TransformHelper.c:498	0.03200
	src/java.base/share/native/libjava/RandomAccessFile.c:108	0.03128
	src/java.desktop/share/native/libawt/java2d/loops/TransformHelper.c:225	0.03057

Looking at the text files from which 4.11, 4.12, and 4.13 were created, we could conclude that most of the lines having the highest importance for a benchmark were located in the same files. This made us come to the conclusion that we would like to aggregate the importance for each line of code based on the file that line is in. The way we did this was that for each benchmark, we would go through all probes and check what file that probe was located in. If the coefficient had a positive value for that probe and benchmark, we would add the coefficient to a value that is unique for that file and benchmark. We ignored negative coefficients, a negative coefficient implies that the probe having a high number of invocations will decrease the chances of predicting a certain benchmark, this does not, however, mean that the probe decreases the importance of a file in regards to that benchmark. Using these unique values for each benchmark and all the files, we created text files for each benchmark containing the files and their aggregated coefficients, sorted by the aggregated coefficients. From each of these text files we took the top three files and created tables: 4.14, 4.15, and 4.16.

Table 4.14: Top three files and their aggregated coefficients, based on their aggregated coefficients for DaCapo Bach

avrora	file	aggregated coefficient
	src/hotspot/share/runtime/objectMonitor.cpp	4.42895
	src/java.base/share/native/libverify/check_code.c	2.06105
	src/hotspot/share/opto/superword.cpp	1.15139
fop	file	aggregated coefficient
	src/java.base/share/native/libverify/check_code.c	3.98178
	src/hotspot/share/c1/c1_GraphBuilder.cpp	2.67638
	src/hotspot/share/c1/c1_LinearScan.cpp	2.32996
h2	file	aggregated coefficient
	src/hotspot/share/opto/stringopts.cpp	2.83842
	src/hotspot/share/runtime/objectMonitor.cpp	2.62528
	src/hotspot/share/opto/parse2.cpp	1.96905
jython	file	aggregated coefficient
	src/hotspot/share/opto/library_call.cpp	2.74016
	src/java.base/share/native/libverify/check_code.c	2.64448
	src/hotspot/share/opto/parse2.cpp	1.77986
luindex	file	aggregated coefficient
	src/hotspot/share/opto/superword.cpp	3.15878
	src/hotspot/share/gc/g1/g1ConcurrentMark.cpp	2.29592
	src/hotspot/share/opto/loopopts.cpp	2.19105
lusearch-fix	file	aggregated coefficient
	src/hotspot/share/gc/shared/referenceProcessor.cpp	2.04942
	src/hotspot/share/c1/c1_GraphBuilder.cpp	1.69383
	src/hotspot/share/opto/library_call.cpp	1.50145
pmd	file	aggregated coefficient
	src/hotspot/share/gc/g1/g1ConcurrentMark.cpp	2.48371
	src/hotspot/share/opto/parse2.cpp	2.1456
	src/hotspot/share/opto/library_call.cpp	1.8409
sunflow	file	aggregated coefficient
	src/java.base/share/native/libfdlibm/e_sqrt.c	1.76309
	src/hotspot/share/opto/superword.cpp	1.57896
	src/hotspot/share/c1/c1_LinearScan.cpp	1.02869
xalan	file	aggregated coefficient
	src/hotspot/share/opto/library_call.cpp	2.42359
	src/hotspot/share/opto/loopopts.cpp	2.01325
	src/hotspot/share/opto/escape.cpp	1.99174

Table 4.15: Top three files and their aggregated coefficients, based on their aggregated coefficients for DaCapo Chopin

avrora	file	aggregated coefficient
	src/hotspot/share/runtime/objectMonitor.cpp	4.58394
	src/hotspot/share/runtime/thread.cpp	0.77141
	src/hotspot/share/c1/c1_LinearScan.cpp	0.72302
batik	file	aggregated coefficient
	src/java.desktop/share/native/libawt/java2d/pipe/ShapeSpanIterator.c	2.1215
	src/hotspot/share/gc/g1/g1ConcurrentMark.cpp	1.82246
	src/hotspot/share/opto/parse2.cpp	1.41514
biojava	file	aggregated coefficient
	src/hotspot/share/opto/library_call.cpp	1.29996
	src/hotspot/share/gc/g1/g1CollectedHeap.cpp	1.09466
	src/hotspot/share/gc/g1/g1RemSet.cpp	1.03669
eclipse	file	aggregated coefficient
	src/hotspot/share/opto/library_call.cpp	2.54365
	src/hotspot/share/opto/parse2.cpp	1.9177
	src/hotspot/share/opto/loopnode.cpp	1.77933
fop	file	aggregated coefficient
	src/java.desktop/share/native/liblcms/cmsio0.c	3.80062
	src/java.base/share/native/libverify/check_code.c	3.31272
	src/hotspot/share/c1/c1_GraphBuilder.cpp	2.2944
graphchi	file	aggregated coefficient
	src/java.base/share/native/libverify/check_code.c	4.06274
	src/hotspot/share/gc/g1/g1ConcurrentMark.cpp	3.35629
	src/hotspot/share/gc/g1/g1CollectedHeap.cpp	1.91452
h2	file	aggregated coefficient
	src/hotspot/share/opto/library_call.cpp	1.52125
	src/hotspot/share/opto/escape.cpp	1.51818
	src/hotspot/share/opto/loopnode.cpp	1.2969
h2o	file	aggregated coefficient
	src/java.base/share/native/libverify/check_code.c	5.44986
	src/hotspot/share/opto/superword.cpp	2.86773
	src/hotspot/share/opto/library_call.cpp	2.01223
jme	file	aggregated coefficient
	src/java.desktop/share/native/libjavajpeg/jidctint.c	1.90426
	src/java.desktop/share/native/libjavajpeg/jdcoefct.c	1.88517
	src/java.desktop/share/native/libjavajpeg/imageioJPEG.c	0.97515

Table 4.16: Top three files and their aggregated coefficients, based on their aggregated coefficients for DaCapo Chopin

jython	file	aggregated coefficient
	src/java.base/share/native/libverify/check_code.c	5.20012
	src/hotspot/share/opto/escape.cpp	1.31626
	src/java.base/unix/native/libjava/ProcessImpl_md.c	1.11529
luindex	file	aggregated coefficient
	src/hotspot/share/opto/superword.cpp	5.56558
	src/hotspot/share/opto/library_call.cpp	1.62586
	src/hotspot/share/opto/loopopts.cpp	1.57152
lusearch	file	aggregated coefficient
	src/hotspot/share/opto/superword.cpp	2.67127
	src/hotspot/share/opto/library_call.cpp	2.28366
	src/hotspot/share/gc/g1/g1CollectedHeap.cpp	1.76041
pmd	file	aggregated coefficient
	src/java.base/share/native/libverify/check_code.c	5.08736
	src/hotspot/share/opto/library_call.cpp	3.26336
	src/hotspot/share/opto/stringopts.cpp	1.97258
spring	file	aggregated coefficient
	src/hotspot/share/opto/library_call.cpp	3.80886
	src/hotspot/CPU/x86/c2_MacroAssembler_x86.cpp	2.19304
	src/java.base/share/native/libverify/check_code.c	2.13373
sunflow	file	aggregated coefficient
	src/java.base/share/native/libfdlibm/e_sqrt.c	1.90518
	src/hotspot/share/gc/g1/g1CollectedHeap.cpp	1.36276
	src/hotspot/share/c1/c1_LinearScan.cpp	0.98259
tomcat	file	aggregated coefficient
	src/hotspot/share/interpreter/bytecodeUtils.cpp	4.03587
	src/hotspot/share/classfile/javaClasses.cpp	1.24428
	src/java.base/unix/native/libnio/ch/Net.c	1.10328
xalan	file	aggregated coefficient
	src/java.base/share/native/libverify/check_code.c	7.37195
	src/hotspot/share/c1/c1_GraphBuilder.cpp	0.93505
	src/hotspot/share/ci/ciTypeFlow.cpp	0.92353
zxing	file	aggregated coefficient
	src/hotspot/share/opto/superword.cpp	2.89393
	src/java.desktop/share/native/libawt/java2d/loops/TransformHelper.c	2.01205
	src/hotspot/share/opto/loopTransform.cpp	1.39555

In order to answer RQ4.2(*To what degree are different parts of the JVM code heavily exercised by all benchmarks?*) we wrote Python code that first checked how many average probe invocations each probe would have for each benchmark based on 20 runs of that benchmark. For each probe, we then checked the aggregated difference of average invocations for each pair of benchmarks. If this value was zero, we added that probes average count to the average count for its directory. This gave us a result showing how many probe invocations each directory had for probes with equal number of invocations between benchmarks. We took the top ten directories from this for both DaCapo Bach and DaCapo Chopin, and the result for this can be seen in 4.17 and 4.18. Note that DaCapo Chopin only had invocations for the top ten directories. The data used to produce these tables was collected on the same machine.

Table 4.17: Number of probe invocations for directories only counting probes invoked an equal number of times across all DaCapo Bach benchmarks

directory	number probe invocations
src/hotspot/share/gc/g1/	84259
src/hotspot/share/interpreter/	18243
src/hotspot/CPU/x86/	17281
src/hotspot/share/classfile/	15567
src/hotspot/share/runtime/	11123
src/hotspot/share/logging/	7813
src/hotspot/share/memory/	6600
src/java.base/linux/native/libnet/	4111
src/hotspot/share/services/	3125
src/hotspot/share/runtime/flags/	2902

Table 4.18: Number of probe invocations for directories only counting probes invoked an equal number of times across all DaCapo Chopin benchmarks

directory	number probe invocations
src/java.base/share/native/libjli/	707
src/hotspot/share/c1/	155
src/hotspot/share/classfile/	111
src/hotspot/share/gc/g1/	34
src/hotspot/share/runtime/	22
src/hotspot/share/oops/	5
src/java.base/unix/native/libjava/	2
src/hotspot/os/linux/	2
src/hotspot/os/posix/	1
src/hotspot/share/compiler/	1

4.3 Discussion

In this section, we discuss answers to our research questions and propose possible methods to be used based on our experience.

4.3.1 RQ1

We will answer RQ1(*What implications does our measuring method have?*) by answering its subquestions that will explain the challenges, performance, and variation of our measurement method.

A challenge with our measuring method(RQ1.1(*What are the challenges of our measuring method?*)) was to get the OpenJDK to build after being probed.

First, we encountered an error in the build process caused by the fact that DMCE sometimes declares a function that is never called. The C compiler used in the build process creates a warning message for this, and by default, this warning results in the build process terminating with an error. However, this behavior can be turned off by editing an OpenJDK build file.

Secondly, parts of the source code of the OpenJDK were written in a way that caused DMCE to inject code into it that caused compile errors. This was due to the fact of macros used in a way that caused DMCE to inject probes, resulting in mismatched parentheses. For example, the macro "CHECK" defined as "THREAD); if (HAS_PENDING_EXCEPTION) return ; (void)(0" was used in the following way "get_user_name(vmid, &snpid, CHECK)". We solved this problem by giving the developer of DMCE input on these compile errors, and he fixed most of them to the release of DMCE 1.7.4. The only ones that were not fixed were compile errors that were received when DMCE probed the macros: CHECK, STORE_PARAM, AWT, and JP. To avoid getting compile time errors, DMCE was configured to not probe these macros. When this was done, we had an OpenJDK inserted with probes by DMCE 1.7.4 that would build successfully.

To answer RQ1.2(*How does the measuring affect execution speed of benchmarks?*) we analyze table 4.1 for the DaCapo Bach benchmarks and table 4.2 for the DaCapo Chopin benchmarks. From these tables, we can see that the probed OpenJDK often leads to a decrease in average performance, but this does not always hold, as seen in the **h2** benchmark for DaCapo Bach, which actually provided an increase in average performance. In the Min-Max Percentage Slowdown row, which is based on the 95% confidence interval, we can see that there are many cases in which the minimum percentage slowdown is negative. This means that there are many cases where we cannot, with 95% certainty, be sure that the instrumentation done to the OpenJDK causes the benchmark execution to be slower. The geometric means tell us that both benchmark suites have an average performance slowdown using the probed OpenJDK. For DaCapo Bach, the slowdown is between 13.20% and 29.90% and for DaCapo Chopin, the slowdown is a little lower, with a slowdown between 10.85% and 18.51%. It is important to note that we did not make any effort in probing the OpenJDK in a way which would minimize the slowdown, in fact, we only focused on probing it in i way that would give us valuable heatmap data, but we still found it interesting to see how our way of using DMCE affected the execution speed of the benchmarks. DMCE has many different settings and configurations you can use to make the probed code faster, and the developer of DMCE told us that there are many cases where code probed by DMCE does not slow down the execution at all.

When answering RQ1.3(*How stable are our measurements on different executions of the same program?*), we start by analyzing table 4.3 and table 4.4. The percentage of the average number of probe variation is between 1.71% to 10.06% for the different benchmarks, and when we look at the number of average probe variations for each benchmark, we can see that the number of probe invocations that varies in between different runs of the same benchmark is a large number for all benchmarks. This means that the variation between different runs is significant, which can at least partly be attributed to the variation of code used by the JIT compiler, see 2.1. As we do not know what parts of the JVM code regards the JIT compiler, we cannot tell if this is the sole contributor to the variation in probe invocations. To try and get a better understanding of the percentage of probes that vary a certain amount, we can look at figure 4.1 and figure 4.2. Looking at these figures, we can see that the plots are similar for all benchmarks with small variations, which means that the variation percentage for different probes are similar for all benchmarks. Furthermore, given the steep downhill in the beginning of the plots, we can conclude that a small number of probes have a high percentage variation and that the majority of probes have a small percentage variation. With all this in mind, we know that we have a small but significant percentage of variation for all benchmarks and that the variation distribution across probes is similar for all benchmarks. This makes us believe that our results in this report are trustworthy, but it still raises the threat to validity that we might have needed more data to base our results on, given the variation of probe invocations.

4.3.2 RQ2

To answer RQ2(*How much of the JVM code is covered by state of the art benchmark suites?*), we look at the table that shows the coverage of the different benchmarks for DaCapo Bach and DaCapo Chopin (table 4.5 and table 4.6). This shows us that about a fourth of the OpenJDK source code gets exercised when running a full DaCapo benchmark suite. This number seems low, and we have no definitive answer to why that is, but we have to note that some of the probes in the OpenJDK source code are specific to the compiler and therefore will never be invoked when only running Java programs. Another reason for why the coverage might be low is that we only used one build of the OpenJDK with the default build options. There exist many different build options that might result in builds using other parts of the OpenJDK source code that our build does not use. Also, the coverage of JVM source code might not be a good measurement for how well the benchmarks measure Java's performance. Furthermore, when we look at the average coverage after running every single benchmark in DaCapo Bach, 25.25%, and in DaCapo Chopin, 28.10%, we can then conclude that DaCapo Chopin has a higher overall coverage of JVM source code than that of DaCapo Bach. This means that running the full DaCapo Chopin benchmark suite tests more of the JVM than running the full DaCapo Bach benchmark suite.

We then look at individual benchmarks that exist and work on our setup in both DaCapo Bach and DaCapo Chopin, and we can see that a majority of them have a higher coverage in DaCapo Chopin when compared to Bach. Some of these benchmarks also have a significantly higher coverage in DaCapo Chopin, **fop** for example, has 24.48% average coverage for DaCapo Chopin and 23.00% for DaCapo Bach. There are however three benchmarks that have higher coverage for DaCapo Bach than for DaCapo Chopin, these are **avrora**, **h2** and **sunflow**, see table 4.5 and 4.6. The differences in coverage for these three benchmarks are, however, rather small. When we compare them to all other benchmarks that have increased coverage from DaCapo Bach to DaCapo Chopin, we can see that the smallest percentage unit increase in coverage was **xalan** that had 22.79% for DaCapo Bach and 23.05% coverage for DaCapo Chopin. This makes us believe that **avrora**, **h2** and **sunflow** are very similar for

DaCapo Bach and DaCapo Chopin. We have no definite answer to why these three benchmarks decrease in coverage, but we speculate that this is due to how DaCapo Chopin is an experimental branch that we built ourselves, and there could be some additional Java code that always runs when executing a DaCapo Bach benchmark that the DaCapo Chopin benchmarks do not contain. There is also the possibility that the decrease in coverage for these three benchmarks is coincidental.

Then we analyzed the benchmarks that have either been added to DaCapo Chopin that did not exist in DaCapo Bach or work in DaCapo Chopin but not in DaCapo Bach (**batik, biojava, eclipse, graphchi, h2o, jme, spring, tomcat, zxing**). Looking at these benchmarks, we can see that the coverage varies from on average 23.12% to 25.46%. This is a higher interval than the intervals for the benchmarks working in both DaCapo Bach and DaCapo Chopin, which are 21.97% to 23.43% for DaCapo Bach and 21.88% to 24.65% for DaCapo Chopin. With this interval comparison, we can see that the benchmarks unique to DaCapo Chopin have a higher average coverage overall than the other benchmarks, compared to both DaCapo Bach and DaCapo Chopin.

4.3.3 RQ3

RQ3(*To what degree are our state of the art benchmarking suites exposing the performance of the JVM?*) will be answered through its relevant subquestions: RQ3.1(*Are there any parts of the JVM code that get executed by other Java programs that do not get hit by the benchmarks?*) and RQ3.2(*Does the distribution of JVM code executed by the benchmarks reflect that of other Java programs?*).

For RQ3.1(*Are there any parts of the JVM code that get executed by other Java programs that do not get hit by the benchmarks?*) we used our Java programs and then compared their coverage data against the benchmarks. Table 4.7 and table 4.8 tell us that there is a small amount of probes that our Java programs invoke that does not get invoked by any benchmarks. The amount of probes are very small in comparison to the total amount of probes invoked during any benchmark run, see table 4.5, 4.6, but we can see that there are significantly more probe invocations by our programs for probes not invoked by the DaCapo Bach benchmark suite compared to the DaCapo Chopin benchmark suite. To get more of an overview of what parts of the JVM code that had invoked probes by our programs, not invoked by any benchmark, we can look at table 4.9 and 4.10. From table 4.9, we can conclude that there are several files containing probes invoked by our programs but not by any DaCapo Bach benchmark, furthermore, we can see that SwingComponents.java have invoked probes in five files that does not have any probes invoked by any DaCapo Bach benchmark. Looking at table 4.10 we can see that there are fewer files containing probes invoked by our programs not invoked by any DaCapo Chopin benchmark compared to DaCapo Bach. We can also see that only one file, "src/java.desktop/share/native/libfontmanager/freetypeScaler.c", have invoked probes by our Java programs that does not contain any invoked probes by any DaCapo Chopin benchmark. We will not go into further detail about what parts of the JVM code that gets executed by our programs that does not get executed by any benchmarks, but we can draw the conclusion that it is possible to write Java programs that executes JVM code that does not get executed by any DaCapo Bach or DaCapo Chopin benchmarks, meaning new benchmarks could be developed to increase the JVM code coverage.

To answer RQ3.2(*Does the distribution of JVM code executed by the benchmarks reflect that of other Java programs?*) we can analyze figure 4.3. This figure shows us that the directory "src/java.base/share/native/libfdlibm/" gets a lot of probe invocations for both the DaCapo Bach and DaCapo Chopin benchmark suites but zero invocations for our Java programs. Other than that, looking at the log scale color coding, our programs seems to have a similar distribution to that of DaCapo Bach and DaCapo Chopin, with the exception of "src/java.base/unix/native/libnio/ch/", which has a much darker color for DaCapo Chopin compared to both DaCapo Bach and our Java programs. Because of this, our hypothesis is that the distribution of JVM code executed by the benchmarks reflects other Java programs well and that DaCapo Bach does this to a higher degree than DaCapo Chopin. Which would in turn suggest that DaCapo, and specifically DaCapo Bach's performance measurement, represent our Java programs well.

4.3.4 RQ4

When trying to answer RQ4(*To what degree do the different benchmarks exercise different JVM code?*) we start with looking at figure 4.4 and figure 4.5 to see how much each directory is exercised by each benchmark. We want to analyze differences in color between the benchmarks to try and figure out if a directory has a higher percentage of probe invocations for a certain benchmark. There are some differences, for example, in figure 4.4, we can see that "src/java.base/share/native/libjava/" has a darker color for **h2, lusearch, lusearch-fix**, and **sunflow**, compared to the other benchmarks. This means that **h2, lusearch, lusearch-fix,** and **sunflow** have a higher percentage of probe invocations in the directory "src/java.base/share/native/libjava/" compared to other benchmarks. However, it is hard to find major differences using these figures, and we can therefore conclude that the percentage of probe invocations for each directory is rather similar for all benchmarks.

We answer RQ4.1(For each individual benchmark, what degree of importance do different parts of the JVM code have for that benchmark?) by analyzing the data from table 4.11 to table 4.16. We see in the tables that the most important lines of code for all of the benchmarks in DaCapo Bach and DaCapo Chopin are different, which means that there are significant differences in the amount of invocations for the logistic regression model to draw these conclusions. The same statement cannot be said about the files's aggregated coefficients, where the three most important files for each benchmark contain the same files for several different benchmarks. We can see an example of this in table 4.15, where **biojava** and **eclipse** both have "src/hotspot/share/opto/library_call.cpp" as the most important file. If we look at the aggregated coefficients, however, we can see that **eclipse** has a higher value for this file than **biojava**, meaning "src/hotspot/share/opto/library_call.cpp" have a higher importance to **eclipse** than **biojava**. We will not look into the difference in importance of parts of the JVM code for different benchmarks further in this report, but we can draw the conclusion that this data, which can be found in full here [1], is sufficient to tell us how important different parts of the JVM code are to the different benchmarks, and it could be used in the future to explore the importance parts of the JVM code have on different benchmarks in more detail.

To answer RQ4.2(*To what degree are different parts of the JVM code heavily exercised by all benchmarks?*) we look at the tables 4.17 and 4.18. From this, we can see that when running DaCapo Bach, the directory "src/hotspot/share/gc/g1/" has a large number of probe invocations from all probes that are invoked an equal amount of times across all DaCapo Bach benchmarks. However, when we compare DaCapo Bach's and DaCapo Chopin's numbers, we can see that DaCapo Chopin has very few probes that are invoked an equal number of times across all DaCapo Chopin benchmarks. This means that we can draw the conclusion that DaCapo Bach exercises large parts of the JVM code with all benchmarks, specifically "src/hotspot/share/gc/g1/", while the DaCapo Chopin benchmarks do not exercise the same parts of the JVM code an equal amount with all its benchmarks.

4.3.5 Proposed methods

With this work done, we wanted to propose a couple of methods that we figured might be useful based on what we learned. The first method we propose is how to use DMCE on large scale projects that, in our case, use macros that need to be excluded and special build flags that need changes. How we did this is explained briefly in Approach(3.1). This method could help when wants to use DMCE or another code injection tool on a large or old project in need of measurement.

Heatmap data collected from a probed OpenJDK could be collected from running any Java program on a probed OpenJDK, not only benchmarks. This means that one could collect heatmap data for newly developed Java programs and analyze them in any way that gives some useful measurement of whether the program behaves as desired.

Another method that we propose is how to analyze and categorize the heatmap data one gets from running code probed by DMCE using logistic regression. This method involves using the data gathered to train a logistic regression model that will predict specific runs of the code depending on the heatmap data. To find interesting parts of the code from the model, you analyze the coefficient matrix from the model and categorize the probes accordingly, as explained in the Approach(3.7).

4.4 Threats to validity

In this section, we will discuss possible threats to the validity of our results.

4.4.1 Varying data

Given our analysis when answering RQ1.3(*How stable are our measurements on different executions of the same program?*) we can see that the heatmap data have significant variation for different runs of the same benchmark. To be sure that this is accounted for, a sufficient amount of data needs to be used to compute our results. We believe that we have enough data as a basis of our results, but it is a threat to validity that we might not have enough data as the basis of our results.

4.4.2 Only one setup

All of our results are based on heatmap data as well as execution times for the benchmarks. Heatmap data will vary depending on the system it is collected on, for example, the OpenJDK has unique code for different CPU:s and OS:s. Keeping this in mind, all of our results can only be trusted for the setup that we used to collect our heatmap data. The execution speed of the benchmarks varies very much depending on what setup you have. This means that we can only say that the performance data is accurate for the setup we used to collect it.

4.4.3 Not able to test all benchmarks

As described in the background (section 2.4.1 and 2.4.2) we did not manage to make all the benchmarks run successfully on our system. This means that all the data we have collected ignores all these benchmarks, which might work on other Java versions or systems. Because of this, we are missing some potential data that might have changed the output of our results, where the coverage of, for example, table 4.5 might have been higher.

4.4.4 Way of preprocessing input data to logistic regression model

We only preprocessed the data for training our logistic regression models in one way, by first adding 1 to each element, then calculating its 10 logarithmic value, and normalizing it. We did this both because it turned out to create the logistic regression models with 100% correct predictions, which the other ways we tried of preprocessing data did not get, but also because we believe it to be a good way of preprocessing the data. However, we did not try all possible ways of preprocessing data, and another way of preprocessing data might also create models that have 100% correct prediction rate and, in fact, be better suited for our situation. With this in mind, while we believe our way of preprocessing data to be good, it might not create the best model and, in turn, the best coefficient matrix. This means that if there is a better way to preprocess the data, we would get better results to answer RQ4(*To what degree do the different benchmarks exercise different JVM code?*).

4.4.5 When measuring performance, other things might have been executed in the background on the machine

When we collected data for performance on one virtual machine, we first ran every single benchmark 20 times each using the probed OpenJDK and then repeated this using the unprobed OpenJDK. This means that if something was running in the background on the hardware machine our virtual machine was running on, or in the background on our virtual machine when either the probed or unprobed OpenJDK were running it would interfere with the results.

4.4.6 Human error

While we were careful with writing Python code that we understood and manually made sure that the results we gained from different analyses made sense. We have analyzed a lot of data in many different ways, making it easy to make mistakes. We believe that this is not the case, but we cannot be 100% certain that we have not made any mistakes. This is one reason why we have decided to publish our results and code on Zenodo [1].

Chapter 5
Conclusion

This chapter summarizes the findings of this master's thesis and suggests potential future work.

5.1 Summary of findings

Here we summarize the findings from our work

- We used the Ericsson instrumentation tool called Did My Code Execute (DMCE) to instrument the Java Virtual Machine (JVM) of an OpenJDK, and from that JVM we ran the state of the art benchmark suite, DaCapo, as well as a modern, more experimental branch of DaCapo.
- The instrumented OpenJDK revealed what code is run from the OpenJDK when executing the benchmark suites, which gave us interesting data that we could gather and analyze.
- While analyzing the data, we looked at five main aspects:

  How much does the execution of JVM code vary for several runs of the same benchmark?

  How much of the JVM does the DaCapo benchmarks cover?

  Are there parts of the JVM code that do not get exercised by any benchmark?

  How different are the DaCapo benchmarks from other Java programs?

  How do the different DaCapo benchmarks differ in comparison to each other?
- We found that the data for the same benchmark executed several times has an average variation between 1.71% and 10.06%.

- The average total coverage for the regular DaCapo benchmark suite was 25.25% while the experimental branch of the DaCapo benchmark suite had an average coverage of 28.10%.
- There is JVM code which gets executed by other Java programs that does net get executed by any benchmark, which implies that new benchmarks can be created to increase the JVM code coverage.
- We found that the DaCapo benchmarks have a similar distribution of probe invocations to other Java programs.
- The benchmarks themselves did not differ much from each other, but we could still find individual differences when looking closely at what code has been run using a logistic regression model.

5.2 Future work

The data that we have collected and analyzed stems from our modified OpenJDK, which has probed most of the source code in the OpenJDK. This includes the source code for the compiler. Somebody that has more knowledge than us in the OpenJDK source code structure could create their own probed version of an OpenJDK, not probing source code that only regards the compiler. Given an OpenJDK probed in such a way, our analysis could be recreated with more accurate results.

We have only analyzed the DaCapo benchmark suites in this paper. There exists many other benchmark suites used to test the performance of Java, for example, XCorpus. It would be interesting to perform similar analysis to what we have done on other benchmark suites as well.

All heatmap data we have collected has been collected on the two virtual machines provided by Ericsson. This means that our conclusions are only valid for this setup, and we cannot tell for sure if one could draw the same conclusions if the heatmap data was collected on, for example, another Linux distribution instead. Therefore, it could be interesting to perform similar experiments as we have done using other setups to see if one can draw the same conclusions for those systems as well. It would also be interesting to see what the results would be if similar experiments were performed on different OpenJDK versions.

In this master's thesis work, the OpenJDK have always executed Java programs with the default options. This means that new data could be collected where special options have been applied to the JVM. These options include, but are not limited to: Setting the initial size of the heap, specifying which garbage collector to use, or disabling the JIT compiler. Testing how these different options affect the heatmap data would be the basis for a new research question, and another research question could be how the new heatmap data would compare to the data obtained in this thesis.

References

[1] Are we benchmarking the java virtual machine right? `https://doi.org/10.5281/zenodo.8169042` [Online; accessed: 2023-07-20].

[2] Dacapo experimental branch, chopin. `https://github.com/dacapobench/dacapobench/tree/dev-chopin` [Online; accessed: 2023-07-10].

[3] Did my code execute. `https://github.com/PatrikAAberg/dmce` [Online; accessed: 2023-07-10].

[4] Logistic regression model. `https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html` [Online; accessed: 2023-07-10].

[5] the dacapo benchmark suite. `https://www.dacapobench.org/` [Online; accessed: 2023-07-10].

[6] JB Dietrich et al. Xcorpus–an executable corpus of java programs. *The Journal of Object Technology*, 16(4):1–24, 2017.

[7] Michael Kölling et al. Java: What's new and how might it change our teaching? In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 2*, pages 1182–1182, 2022.

[8] Philipp Lengauer et al. A comprehensive java benchmark study on memory and garbage collection behavior of dacapo, dacapo scala, and specjvm2008. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, pages 3–14, 2017.

[9] Stephen Brown et al. A coverage analysis of java benchmark suites. In *The IASTED International Conference on Software Engineering*, pages 1–9, 2005. Note: This is a slightly extended version of the paper that appears in The IASTED International Conference on Software Engineering, 2005.

[10] Stephen M Blackburn et al. The dacapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on*

Object-oriented programming systems, languages, and applications, volume 41 of *ACM SIGPLAN Notices*, Oregon, Portland, USA, 2006. Association for Computing Machinery.

[11] Tim Lindholm et al. The java® virtual machine specification java se 17 edition. 2021.

[12] Vojtěch Horký et al. Dos and don'ts of conducting performance measurements in java. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, pages 337–340, 2015.

[13] Walter Lucas et al. Does the introduction of lambda expressions improve the comprehension of java programs? In *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*, pages 187–196, 2019.

[14] Michael P. LaValley. Logistic regression. *Circulation*, 117(18):2395–2399, 2008.

Appendices

Appendix A
Contribution table

This appendix contains a table stating what author had the most responsibility for different parts of the report and a Github repository with a fix.

Table A.1: Sections of this report and which author had the most responsibility for each section

Section	**Nour**	**Samuel**
Introduction		X
Research Questions	X	X
Contribution	X	
Background	X	
Approach		X
Evaluation	X	
Experimental Setup	X	
Results		X
Discussion	X	
Threats to Validity		X
Conclusion	X	
Summary	X	
Future Work		X

As the research questions is a vital part of this report and we both were heavily invested in developing them, there are two X:s for that section.

Appendix B
Our example programs

In this appendix, we present the Java code whose coverage of the Java Virtual Machine code we used to check against the DaCapo benchmarks.

B.1 Lambda.java

The first Java program exercises Java's lambda function by iterating over a list with 500 elements and doubles the value of each element.

```
import java.util.ArrayList;

class Lambda{
    public static void main(String[] args){
        for (int i = 0; i < 20; i++){
            ArrayList<Integer> numbers = new ArrayList<Integer>();
            for (int j = 0; j < 500; j++){
                numbers.add(j);
            }
            numbers.forEach((n) -> {numbers.set(n, n * 2); });
        }
    }
}
```

B.2 SealedClasses.java

Our second Java program utilized a new Java feature called Sealed Classes.

```
import java.util.ArrayList;

abstract sealed class Animal
permits Dog, Cat, Rabbit {
    private int height;
    Animal(int height){
        this.height = height;
    }
    int getHeight(){
        return this.height;
    }
}
final class Dog extends Animal{
    Dog(){
        super(3);
    }
}
final class Cat extends Animal{
    Cat(){
        super(2);
    }
}
final class Rabbit extends Animal{
    Rabbit(){
        super(1);
    }
}
class SealedClasses{
    public static void main(String[] args){
        for (int i = 0; i < 20; i++){
            ArrayList<Animal> animals = new ArrayList<Animal>();
            for (int j = 0; j < 100; j++){
                animals.add(new Dog());
                animals.add(new Cat());
                animals.add(new Rabbit());
            }
            int totalHeight = 0;
            for (int j = 0; j < 300; j++){
                totalHeight += animals.get(j).getHeight();
            }
        }
    }
}
```

B.3 SwingComponents.java

The third Java program is a simple Swing program that initializes Swing components before terminating.

```
import javax.swing.*;

public class SwingComponents{
    public static void main(String[] args){
        for (int i = 0; i < 20; i++){
            JButton b = new JButton("Hello");
            b.setBounds(130,100,100,40);
            JLabel l = new JLabel("Hi");
            JTextField tf = new JTextField("World");
        }
    }
}
```

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2023-08-15

EXAMENSARBETE Are We Benchmarking the Java Virtual Machine Right?
STUDENTER Nour Salem, Samuel Fagerström
HANDLEDARE Christoph Reichenbach (LTH)
EXAMINATOR Görel Hedin (LTH)

Hur väl testar man programmeringspråket Java egentligen?

POPULÄRVETENSKAPLIG SAMMANFATTNING **Nour Salem, Samuel Fagerström**

För att testa Javas prestanda finns det populära Javaprogram alla kan köra, som är speciellt utvecklade för att testa prestanda. Vi har i detta arbete undersökt hur bra några av dessa tester, kallade DaCapo, är. Vi har bland annat kommit fram till att hela Java faktiskt inte testas.

För att mäta prestandan av olika versioner av Java på olika system, t.ex. en Macbook eller en stationär dator med Windows, finns det så kallade benchmarks som man kan använda. Ska man kunna lita på resultaten från benchmarks är det viktigt att veta hur bra dessa benchmarks är ur flera perspektiv. Därför analyserade vi en populär samling av benchmarks som heter DaCapo vars benchmarks är Javaprogram som testar Javas prestanda.

Först har vi kommit fram till att dessa benchmarks inte lyckas testa prestandan för hela Java. Detta betyder att om man kör ett program skrivet i Java är det inte säkert att allt som detta program gör har fått sin prestanda testad av något av dessa benchmarks. Med detta resultat kan man därmed se att det finns förbättringspotential för DaCapo, man skulle kunna göra fler benchmarks som testar mer av Java.

Sedan har vi även tittat på vilka delar av Java som använts, och hur mycket de har använts. Detta gjorde vi både för DaCapo benchmarks och för våra egna självskrivna Javaprogram. Det vi kom fram till när vi gjorde detta var att procentfördelningen för användningen av olika delar i Java när vi körde DaCapos benchmarks är likt procentfördelning när vi körde våra egna Javaprogram. Detta visar på att om ett benchmark har fått ett bra prestandaresultat är det troligt att även ett annat Javaprogram som körs på samma Java version och system kommer ha bra prestanda.

Under analysen av benchmarks tittade vi även på hur stor del av Java som körs lika mycket av alla benchmarks. Vi kom fram till att den officiella versionen av DaCapo, som kallas Bach, kör en stor del av Java lika mycket för alla sina benchmarks. Däremot finns det en senare experimentell version av DaCapo, kallad Chopin, där endast en liten del av Java körs lika mycket av alla benchmarks. Detta betyder att om man kör alla DaCapo Bach benchmarks så slösas resurser på att testa samma sak i alla benchmarks.

För att kunna genomföra våra analyser har vi använt verktyget "Did My Code Execute", DMCE. Detta verktyg gjorde det möjligt för oss att se hur mycket olika delar Java använts av olika benchmarks och våra egna Javaprogram. Vårt arbete skulle kunna användas som inspiration till hur DMCE och liknande verktyg kan användas i framtida arbeten.