UNIVERSITY OF MILAN
Department of Computer Science

Nicola Amadio

Java Bytecode Instrumentation for Performance Analysis

Master's Thesis

Supervisor(s): Alessandra Gorla
Antonio Carzaniga
Mattia Monga

24.02.2022

Abstract

In this work, we explain how to instrument Java applications to measure their performance, and we take the first steps into extending an open-source project named Freud [1], a tool to create probabilistic performance annotations for C/C++ programs [2].

Modern software applications are getting increasingly complex [3]. They often rely on different, interconnected and synchronized distributed systems, possibly built with different technologies and by different teams, and no single person has a comprehensive understanding of their entirety. The trends of cloud computing and microservices [4], while having democratized the possibility to scale systems, have also contributed to making it ubiquitous to incur in situations where it's difficult to understand what is happening when problems do arise, like in the case of performance regressions [5][6]. Therefore, in order to address these issues, tools, products and services have been researched and developed in the field of software monitoring and observability [7][8][9][10][11].

A common technique used to monitor small and large systems is that of software instrumentation [12], which involves adding extra code to a compiled application, to track, log and visualize some programs' behavior and which is foundational for most types of dynamic analyses of software programs.

In the context of this thesis, to instrument Java byte code, we relied on a tool named Soot [13]. Soot is a Java optimization framework that provides four intermediate representations for analyzing and transforming Java byte code. Transforming the byte code for some existing applications, adding log statements that print or write to memory the characteristics of a program at a given time - like the running time of a method, the magnitude of some variables and so on -, allows to create a documentation for the run-time behavior of such applications, which ultimately gives the user visibility of what happens underneath arbitrarily complex software.

We built a system [15][16] for instrumenting and analyzing Java applications to measure their performance, in a way that is compatible with Freud's performance analysis API [14] and thus can be used in tandem with it to annotate Java methods with probabilistic performance annotations.

Contents

1 Introduction

Monitoring the performance of software applications is an important and difficult aspect of modern software development [17][18].
The performance of a software system can in general be defined as a measure of how effective the system is with respect to time constraints and allocation of resources.
As perceived by the user, for example in the context of web applications, performance could be the speed of a website, defined as the time that takes the site to load and display content. Apparently, speed is the single most important factor affecting user experience, as shown by an experiment run by Google on its search engine [19][20][21], which showed that slowing down the search results page by 100 to 400 milliseconds had an impact of 0.2% to 0.6% fewer searches, with the impact getting even worse the longer the users are exposed to the slowed down version of the site: users exposed to a 400 ms delay since the beginning of the experiment did 0.44% fewer searches during the first three weeks, but 0.76% fewer searches during the second three weeks.

Software performance is not just critical, it can also be very complex to monitor and troubleshoot. Especially when systems are large. And if this was once only an issue for the few organisations that relied on complex software systems [5][6], today it is a much more widespread issue, given that the internet has scaled in many different ways, and that an ever growing number of companies and organizations operate software at scale, be it for the number of users their services have, the amount of data, the variety of geographical locations that the software is serving and so on [3][4].

Therefore, there is a growing need for developers to have suitable tools to easily and timely evaluate the performance of their systems, such that their ability to experiment and iterate fast on their products, which is key in modern software development, is not compromised. In particular, it would be nice to have a tool that would allow developers to rapidly understand how a new feature that they are planning to build would affect the overall application's performance, such that a trade-off between the value added by the new feature and its impact on performance can take place, for example. Or the feasibility of its implementation can be correctly evaluated.

Freud [1] is a project started by researchers at the University of Lugano, which aims to tackle exactly this problem, and to do so, it introduces a new concept: that of Probabilistic Performance Annotations [2]. This concept is about annotating software methods with a probability distribution, which models the relation between some performance metrics of the method (such as running time, memory utilization, lock holding time etc) and its input parameters. This annotation can ultimately be used as a form of software documentation or as a tool to debug performance regressions.

Freud itself is a tool, currently build entirely in C++ and applicable to C/C++ systems, that takes a target method as its input and produces performance annotations in the form of text files and/or graphical plots.
Its mechanism is organized in 3 different phases. In the first phase, it instruments the target method with the help of a tool named Pin [22]; after that, the user can run the application containing the instrumented method and this will generate some logs containing relevant information about the method's performance metrics and input parameters; the last step is to run a statistical analysis [14], implemented through the use of popular R libraries and leveraging mostly regression trees and mixture models, which ultimately produces the performance annotations.

Freud has been tested over large software systems like ownCloud [15] - a distributed storage system -, the x264 encoder library [23] and the MySQL database system [24]. For MySQL, it also managed to isolate a performance regression and identify the root cause of a performance bug.
Even if it has shown its potential on the aforementioned experiments, it is still a project in its infancy, and there are multiple ways in which it can be improved. We will dive deeper into Freud in chapter 2.

A current limitation of Freud is that it can only be used on C/C++ systems, and in this thesis' work we started the process of making it compatible with Java systems as well. To do this, we needed to build a custom instrumenter for Java byte code. The three components of Freud are isolated and independent, such that it is possible to have a system, external to Freud, that instruments Java applications and at run-time logs data that can later be used by the statistical analysis module of Freud.
We built such a system, composed of two Java applications [15][16], that works as a tool that is compatible with the aforementioned statistical analyser. The first application [15] takes care of instrumenting Java methods: the user gives as input the path to a compiled Java class and a method's name, and the system instruments the class' target method (or methods, in case there are multiple ones matching the name). It also contains a "workload generator", which is a process that creates log files of input parameters that can be given as inputs to the target methods, and a runner, which runs a set of benchmark methods with such inputs, producing log files about the methods' performance.

For this thesis, we have developed an instrumenter that only instruments methods to make them log their running time. This is different than Freud's original instrumenter, which can monitor and log different kinds of metrics - not just running time, but also lock holding time, memory consumption and others-, and which can also automatically extract interesting features from the target methods and log their values.
Therefore, we were able to run a complete analysis, that starts from instrumenting Java

byte code and finishes with probabilistic performance annotations produced by Freud, only on a class of benchmark methods that we created and provided manually to the instrumenter: for these methods, we manually store the interesting features, which in our case are the methods' input parameters, to create the log files which will be later used to create the performance annotations.
Our application, though, can easily be extended in order for it to be able to operate on any compiled Java class. We also believe that our experiments are still meaningful, since the target methods that we analyse capture interesting characteristics, for example cases where the execution time is related to the input parameters through linear or exponential relations, where the correct relations where captured by Freud's statistical analyser, as we later in chapter 4.4 show when describing our experiments and tests.

We also developed a second Java application [16], that takes care of serializing all the log files generated by the instrumented methods in a way that is compatible with Freud's API, such that it is possible to call Freud's statistical analysis component and generate performance annotations for our benchmark methods. This component is complete: it does not need to be improved or changed in the case the first component will be updated to work with any Java class.
We will describe more in detail how our instrumenter and logger works in chapter 3 of this thesis.

For the instrumentation itself, we relied on a tool named Soot [13], which is a tool originally developed by the Sable Research Group at McGill University to analyse and manipulate Java byte code for different purposes, such as code optimization, program static and dynamic analysis and software instrumentation and monitoring. It makes use of intermediate languages which can be translated to and from Java byte code, and for which it provides an API that enables users to modify them. We will explore the features of Soot in chapter 2 of this thesis.

The fifth and last chapter of this document is a discussion on the results of this work, explaining how it is relevant in the context of probabilistic performance annotations and performance analysis of complex systems in general, and we conlcude by outlining a set of improvements that can be made on this work and give some inputs for future explorations.

2 Freud, a tool to create Performance Annotations

Freud [1] is a project from researchers at the University of Lugano, that introduces a new concept in the field of performance analysis: probabilistic performance annotations. Probabilistic performance annotations are meant to help software developers understand, debug and predict the performance of complex software systems: they are methods' annotations, which describe a relation between a measurable performance metric, such as running time, and one or more of the methods' features, such as its input parameters, or the state of the underlying system. They aim at capturing non-trivial behaviors beyond the more classical algorithmic complexity of a function or the measures made by traditional profilers. They also represent a kind of software documentation related to performance.

2.1 Problem background

There are several factors that make the concept of probabilistic performance annotations interesting and relevant. Nowadays, to manage complexity in programs, developers use a combination of tools and best practices, such as code modularisation and documentation. These tools and practices help with functionality but not with run-time dynamical aspects like performance. Modeling and measuring the run-time performance metrics of a program is complex: in fact, even for simple functions like sorting algorithms, the running time is not always well modeled by its computational complexity. There are other factors, like interactions with the memory subsystem, which can affect how the method performs. This implies that when developers work on some code, there is no way for them to rapidly understand the consequences of their changes regarding the system's performance. Especially when operating in highly distributed scenarios, where a method may result in several RPC calls, acquire locks, perform I/O operations and so on, this becomes very complicated. And the outcome of not fully understanding the performance of a method may compromise the quality of the software application in many ways, such as with user timeouts, worse UX, etc.
To deal with and measure systems' performance, developers usually rely on profilers. A problem with profilers though, is that they aggregate resource usage and don't relate the performance of methods to their input. Also, they don't offer direct mathematical and predictive capabilities. Probabilistic performance annotations are supposed to help in this scenarios.

2.2 Probabilistic Performance Annotations

Concretely, a performance annotation is an expression that characterizes a performance metric as a random variable (the dependent variable) whose distribution is a function of zero or more features of the input or state of the method (the independent variables): a relation of a random variable Y and a relevant feature x.

In the case of Freud, they also include scope conditions to separate modalities: for example Y could behave according to some distribution when x falls in a determined range, and according to another distribution when x falls in another range.

2.3 Freud

Freud [1] is a tool developed to implement the concept of probabilistic performance annotations and make it usable for C/C++ and PHP systems.
It is composed of several components which can be used together as a single tool, to instrument and analyse the performance of a target program's function. The first thing Freud does is collect the method's performance data (CPU time, memory usage, lock holding/waiting time, etc.), along with the method's features (input arguments of the method, state of the system, etc.). Then, it provides a statistical analysis module to build mathematical models relating input features to performance.

Freud assumes that the code can be instrumented to measure the performance metrics of interest such as running time, memory consumption, lock holding time, number of RPCs, etc. Many systems already have such instrumentation; e.g., Google's RPC libraries and locking libraries for C++ and Java measure metrics on RPC calls and lock holding behavior. If the instrumentation is not there, it is possible to use binary or byte code instrumentation tools to add it.

So in total there are two phases: instrumentation and model selection. Once the target method has been instrumented, every time it gets executed it will log the desired performance metrics, and all the log data can then be used to extract relevant features of the input or of the state of the system.
Metrics are the dependent variables in performance annotations. Unlike traditional profilers, Freud doesn't only collect the running time, but it also logs other metrics such as memory consumption and lock holding/blocking time. Moreover, the collection process is extensible, allowing users to add new metrics, passed as a command line argument.
Features are the independent variable in performance annotations. Feature selection is not taken care by the user, like it is for the metrics, but it is Freud that selects features for target methods automatically, by statistically finding the values of local or global variables which may impact the execution of the method. There are three types of features in Freud: program variables, system variables, and quantities computed from program variables. The first and most directly relevant variables which are used are the parameters to the method.

What happens in detail is that instrumented methods produce logs about metrics and features, in the form of log of records. A record represents a single component in the following format: $id, y_1, ..., y_m, x_1, ..., x_n, b_{1,1}, b_{1,2}, ..., \#, b_{2,1}, b_{2,2}, ..., \#, ...$ where id is

the name of the component, followed by the performance metrics y_i, by the raw and derived features x_i, and the binary outcomes of the execution of the first branch instruction, $b_{1,1}, b_{1,2}, ...$, followed by the outcomes of the second branch instruction, $b_{2,1}, b_{2,2}, ...$, and so on.

The second phase of Freud's workflow is an offline statistical analysis. Freud processes the logs to produce probabilistic performance annotations for the chosen components and it applies the same statistical analysis to every target component for every target performance metric. The analysis is based on two statistical models: regression trees and mixture models. A regression tree defines a hierarchical partitioning of the performance records such that the records in each partition are defined by a scope condition and are modeled by a regression, whereas a mixture model is a combination of two or more sub-models each associated with an occurrence probability rather than a scope condition.

2.4 Implementation and experiments

Rogora et al [2] implemented two working instrumentation tools, one for PHP based on Runkit, and the other for C/C++ based on DWARF [25] debugging symbols and the Pin [22] instrumentation tool. These tools are designed to work with any program written in those programming languages.

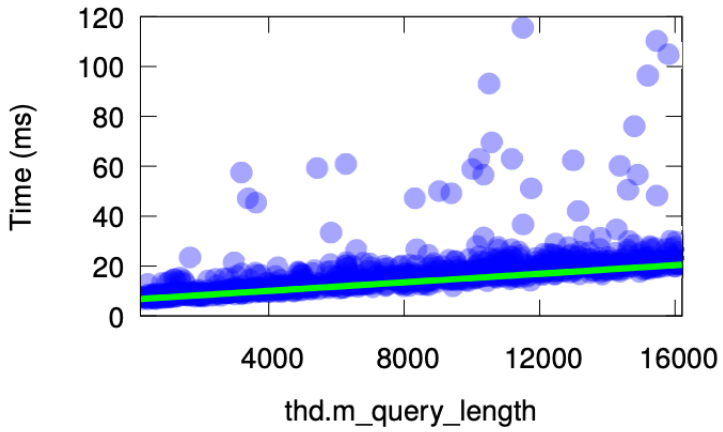
This is how the process goes, as described by the authors [2]: «We use the type information to log relevant features. We collect the values of variables of the C primitive data types: float, double, signed and unsigned char, short, int, long, and long long as well as size-t and bool. For fixed length arrays, we collect the size and optionally an aggregate value, like the sum of all the elements of the array. For char pointers, we try to find the string terminator to compute the string length. We compute the difference between variables named {start, begin, first} and {stop, end, last}, and log them as a derived feature representing a size or time span».

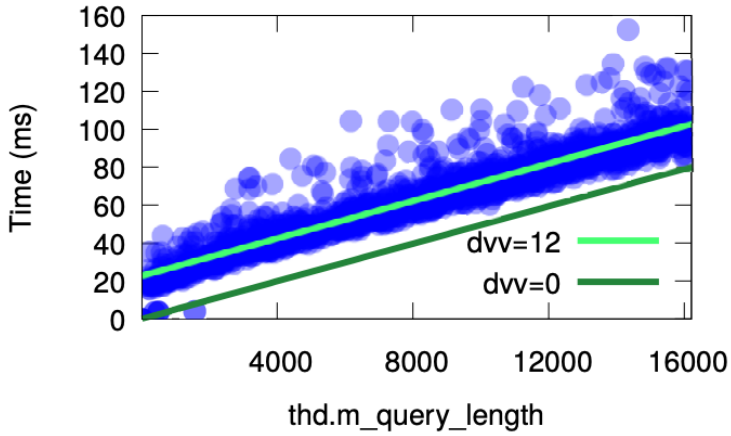
Since instrumentation necessarily introduces overhead, even if that is considered as an acceptable cost as the benefits of collecting the data outweigh the performance degradation, the authors have tried to minimize the run-time overhead of the instrumentation, by using sampling: in particular, they use a reservoir sampling algorithm to collect a fixed number of observations for every method or function for every run.

The analysis component is implemented as an off-line C++ program. The program uses the R statistical package library to compute regressions, mainly using the lm function (fitting linear models), and other basic functions to compute statistical properties of data sets.

Rogora et al [2] also tested Freud on three real-world, complex systems: ownCloud, a remote storage web application, similar to the more widely known Dropbox and written in PHP; the MySQL DBMS, written in C/C++; the x264 library and application for encoding video streams, written in C. They derived performance metrics including running time, dynamic memory allocation, and lock holding/waiting time. In the case of MySQL, they were also able to isolate a performance regression.

The following pictures present two graphs produced by Freud when it was tested over MySQL [2]. They illustrate the running time of a series of INSERT operations with respect to the query size (number of rows to insert). The first picture refers to a version of MySQL which was precedent to a performance bug, and the second to a version with such bug.





As we can see, Freud managed to understand that there was a linear relation between the size of the query and the running time, as both figures show. It also showed that, for the version with the performance bug, this linear coefficient for the relation between running time and the query size is bigger: this gives us more information about the bug, offering an input to the developer about where to explore when debugging the issue.

2.5 Use cases

To summarize, Freud is a tool aimed at modeling the performance of complex software systems in a mathematical and probabilistic way, with the goal of having mathematical functions and probability distributions that describe how a part of a system behaves performance-wise. This type of abstraction and modelling would provide a lot of value to engineers working on very large-scale systems, such as data centers, where parallelism and distribution is particularly present and where it is essential to reliably predict the behaviour of such systems, as many other services and applications rely on them, but it could also help any developer working on an a system with non-trivial scale, to gain a more profound understanding of the system's performance.

3 Soot, a Java optimization framework

To extend the scope of applicability of Freud, we moved the first steps in building an instrumenter for Java systems: such that we could instrument Java classes in a way that, when those were run, they would log performance metrics and features in a way that is compatible with Freud's statistical analysis component.
To build this instrumenter, we relied on a tool named Soot [13]. Soot is usually described as a Java optimization framework, which provides four intermediate representations for analyzing and transforming Java byte code. It can be used as a stand alone tool to optimize or inspect class files, as well as a framework to develop optimizations or transformations on Java byte code. In our case, we mainly used it to transform Java byte code for performance analyses, but we also built some proof of concept applications to show what Soot can do and how the intermediate representations look like, focusing on the representation called "Jimple".

3.1 Class analysis using Soot

Static program analysis is the analysis of computer software performed without executing any program, in contrast with dynamic analysis, which is performed on programs during their execution. The term is usually applied to analysis performed by an automated tool or software.
In the rest of this section, we will show how Soot performs static analyses. To give a practical understanding, we will show a classical example, a simple Java class, implementing the FizzBuzz function:

```
public class FizzBuzz {

    public void printFizzBuzz(int k){
        if (k%15==0)
            System.out.println("FizzBuzz");
        else if (k%5==0)
            System.out.println("Buzz");
        else if (k%3==0)
            System.out.println("Fizz");
        else
            System.out.println(k);
    }

    public static void main(String[] args){
  FizzBuzz fb = new FizzBuzz();
        for (int i=1; i<=100; i++)
            fb.printFizzBuzz(i);
    }
}
```

FizzBuzz is a simple program that prints each number from 1 to 100, and if the number is divisible to 3, 5 or 15 it will print Fizz, Buzz or FizzBuzz respectively.

We will take the above class' byte code, and use Soot to analyze it. In particular, we will use Soot to convert our FizzBuzz class into one of the intermediate representations used by Soot: we will convert it to Jimple format, which is the default intermediate representation for Soot.

The following is the Jimple file for the above FizzBuzz class. In the rest of this paragraph we will explain how to get the Jimple file of a byte code file and how to interpret it.

```
r0 := @this: FizzBuzz
i0 := @parameter0: int
$i1 = i0 % 15
if $i1 != 0 goto $i2 = i0 % 5
$r4 = <java.lang.System: java.io.PrintStream out>
virtualinvoke $r4.<java.io.PrintStream: void println(java.lang.String
    )>("FizzBuzz")
goto [?= return]
$i2 = i0 % 5
if $i2 != 0 goto $i3 = i0 % 3
$r3 = <java.lang.System: java.io.PrintStream out>
virtualinvoke $r3.<java.io.PrintStream: void println(java.lang.String
    )>("Buzz")
goto [?= return]
$i3 = i0 % 3
if $i3 != 0 goto $r1 = <java.lang.System: java.io.PrintStream out>
$r2 = <java.lang.System: java.io.PrintStream out>
virtualinvoke $r2.<java.io.PrintStream: void println(java.lang.String
    )>("Fizz")
goto [?= return]
$r1 = <java.lang.System: java.io.PrintStream out>
virtualinvoke $r1.<java.io.PrintStream: void println(int)>(i0)
return
```

Jimple is the principal representation in Soot. The Jimple representation is a typed, 3-address, statement based intermediate representation. Jimple representations can be created directly in Soot or based on Java source code and Java byte code/Java class files.

In a Jimple file, an analysis only has to handle the 15 statements in the Jimple representation compared to the more than 200 possible instructions in Java bytecode.
In Jimple, statements correspond to Soot Units and can be used as such.

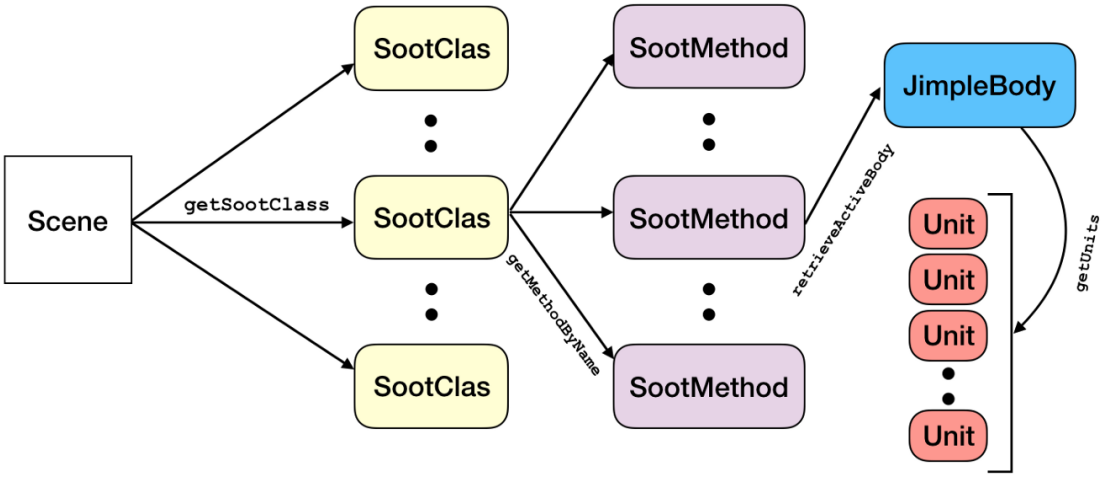
Jimple has 15 statements, the core statements are: NopStmt, IdentityStmt and AssignStmt. Statements for intraprocedural control-flow: IfStmt, GotoStmt, TableSwitchStmt

(corresponds to the JVM tableswitch instruction) and LookupSwitchStmt (corresponds to the JVM lookupswitch instruction). Statements for interprocedural control-flow: InvokeStmt, ReturnStmt and ReturnVoidStmt. Monitor statements: EnterMonitorStmt and ExitMonitorStmt. The last two are: ThrowStmt, RetStmt (return from a JSR, not created when making Jimple from byte code).

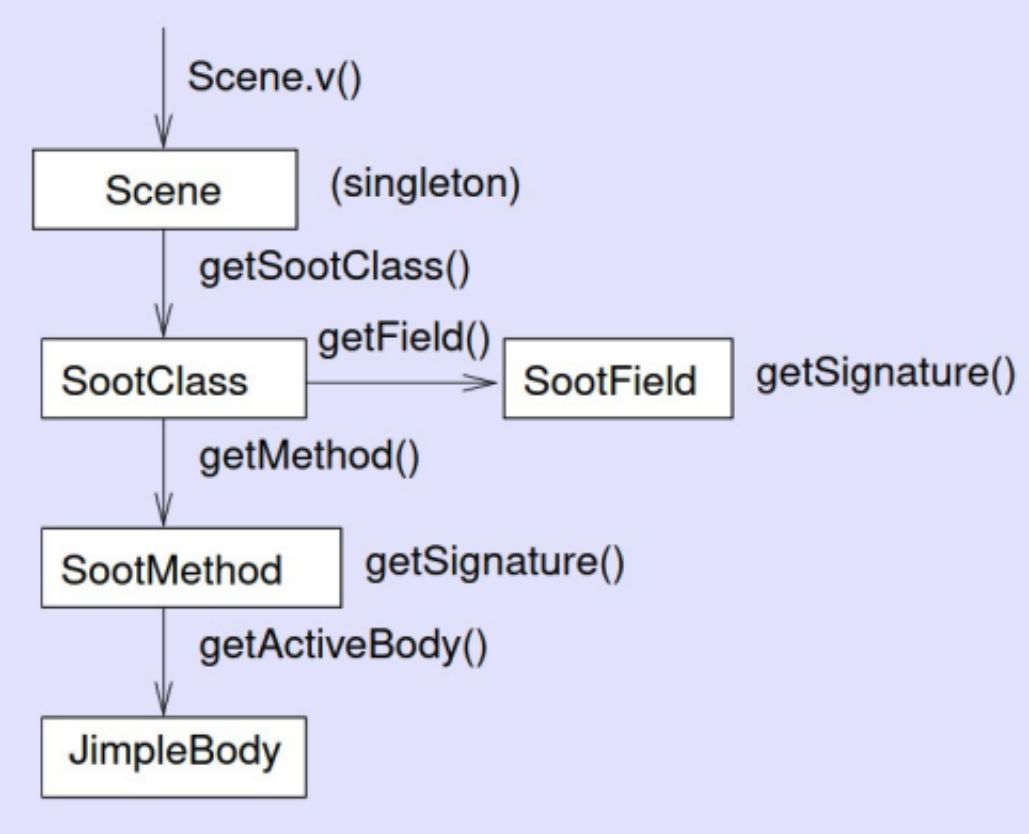
Jimple is a hybrid between Java source code and Java byte code. We recognize the statement-based structure from Java with declarations of local variables and assignments, but the control flow and method invocation style is similar to the one in Java byte code. The local variables which start with a $ sign represent stack positions and not local variables in the original program whereas those without $ represent real local variables.

Nothing is implicit in Jimple. For example, 'this' is represented as r0 which is a Local object (the data structure of variables in Soot). Or the argument of the function is explicitly defined in i0 and its type is int. Each line represents a Unit, which corresponds to a Stmt in the case of Jimple files.

The following image graphically represents the structure of a Jimple file for the byte code of a Java class, including the Soot API methods to access the structures the user is interested in (classes, methods, units, etc.).



The following is a more in-depth view on the actual Java API used to retrieve the different components and structures that we see in the previous image.



Once we have the JimpleBody, we can retrieve the single Units through the Body.getUnits() function. Then if we are considering a Jimple file, the Units will be statements.

The following is a somewhat comprehensive classification of the types of statements that are available in Jimple.

- **Core statements:**
  - NopStmt
  - DefinitionStmt
    - IdentityStmt
    - AssignStmt
- **Intraprocedural control flow:**
  - IfStmt
  - GotoStmt
  - TableSwitchStmt
  - LookupSwitchStmt
- **Interprocedural control-flow:**
  - InvokeStmtReturnStmt,
  - ReturnVoidStmt
- **ThrowStmt**
- **RetStmt**
- **Monitor:**
  - MonitorStmt:
    - EnterMonitorStmt
    - ExitMonitorStmt
- **IdentityStmt**

3.2 Using Soot to instrument Java byte code

The following is a basic example of code that received a JimpleBody as argument and gets a chain of Units, which is conceptually similar to a normal Java Iterable data structure.

```
protected void internalTransform(Body body, String phase, Map options) {
    SootMethod method = body.getMethod();

    // get body's unit as a chain
    Chain units = body.getUnits();
```

Once we have this Chain, we can create statements in Jimple format and insert them in the original Chain of statement. This is how we instrument the Jimple representation of Java compiled classes. Afterwards, we will use Soot to re-transform the Jimple into byte code and the result will be an instrumented version of our original compiled Java class.

The next picture illustrates how to create Jimple statements in Java. The code is taken by a proof of concept application that we used for experimenting and which is available at this repository. In this example, we aimed at instrumenting a class with a counter that would increment every time the instrumented class is executed, so we add an increase method to increase the counter and a report method to log the total number of times the class' method was executed.
So, first of all we create Jimple objects for our counter class and methods.

```
Options.v().set_soot_classpath(sourceDirectory);

counterClass    = Scene.v().loadClassAndSupport( className: "MyCounter");
increaseCounter = counterClass.getMethod( subsignature: "void increase(int)");
reportCounter   = counterClass.getMethod( subsignature: "void report()");
```

Then, while we loop through a copy of the Units Chain of the method that we wish to instrument, we create a Jimple statement out of the increase method of our counter, and then we insert it in the chain. In this case we insert it before a particular statement (variable stmt), which we might select based on our use case. A possibility for example is to iterate through the Units of a method's Chain, and then we do a conditional check to see if a statement is an IF-statement, and if so, we insert the counter's method before that statement (this would result in counting how many times we call IF statements in the instrumented method).

```
// 1. first, make a new invoke expression
InvokeExpr incExpr= Jimple.v().newStaticInvokeExpr(increaseCounter.makeRef(),
IntConstant.v(1));
// 2. then, make a invoke statement
Stmt incStmt = Jimple.v().newInvokeStmt(incExpr);

// 3. insert new statement into the chain
//    (we are mutating the unit chain).
units.insertBefore(incStmt, stmt);
```

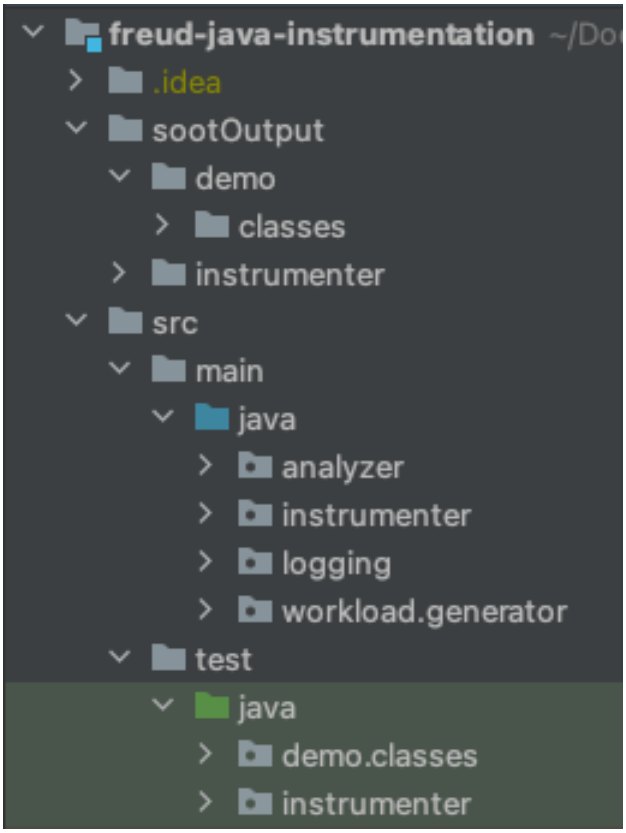
4 Our custom built Java Instrumenter

The main contribution of this thesis is a tool [15] to instrument Java byte code, given the names of target methods as input through the CLI, in such a way that it will log the running time of instrumented methods, and, combining such logs with those of the methods' input data, it will serialize [16] all of it in a way that is compatible with the statistical analysis component of Freud [1], in order to produce probabilistic performance annotations of the Java methods. A limit of our solution is that our instrumenter only logs the running-time, and not the input features. This is different than the C/C++ instrumenter of Freud, and in order to use the statistical analysis component of Freud we ourselves need to store and provide the input parameters of the analysed methods. Therefore we only demonstrated and tested our applications on a MicroBenchmark class that we wrote ourselves, and not on software from the real-world.
That said, the most important components of the instrumenter have been implemented, and, in order for it to be used as a standalone tool that can fully extend the capabilities of Freud to Java systems, it only needs to be completed in some peripheral parts, without the need for it to be re-designed.

4.1 Instrumenter - Architecture Overview

We will now explore the architecture of our instrumenter [15], explaining how each component fits into the overall application. As illustrated by the next picture, we have four main packages: analyzer, instrumenter, logging, and workload.generator. We will first provide a high-level view of what each component does, and then dive deeper into showing more details.

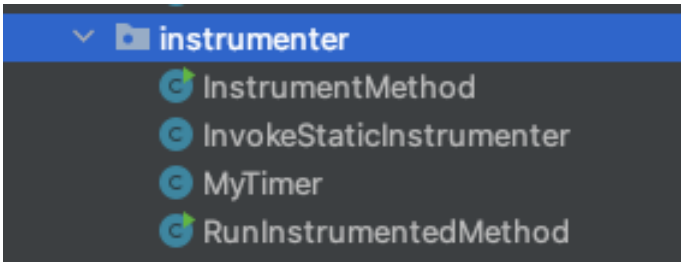
The core component is of course the instrumenter, which contains the classes and methods to actually instrument classes, and also to run them easily.
The analyzer is a an additional tool which just contains the code which needs has to be used to retrieve methods' features, and in practice in this case it contains a method that, given a method as input, prints the input features of that method in Jimple format.
The logging component is a just a tool that packages log files about input features and performance metrics in JSON format. This can be used to visualise those logs in a compacted way, and which could be used in the future in case the statistical analysis component of Freud would update its API to accept also language-independent format such as JSON or Protocol Buffers.
The last component is the workload.generator, which serves in our case to generate and log input features, which we feed our instrumented methods as input at run-time.



The demo.classes package contains our MicroBenchmark file, and a couple of other test classes that are used by the test.java.instrumenter to test that the application works.
The sootOutput folder contains the MicroBenchmark class after it has been instrumented, along with other classes which are useful for the execution of instrumented methods, this folder basically contains all the files that are needed to run the instrumented methods and generate performance logs.

4.2 Component In-Depth Analysis

We will now look at each component more in detail. We will not dive deep into exactly each one of them, but we will focus on the more important ones.



The instrumenter package displayed above is the core component of our application. The most important classes here are the InvokeStaticInstrumenter and the MyTimer class.

The MyTymer class is a simple Java class which contains methods that create a log file and write on it the execution time of a function, once such function is instrumented.

```java
package instrumenter;

import java.io.*;

public class MyTimer {
    private static long time;
    private static String PATHNAME = "time-logs.txt";
    private static FileWriter writer;

    public MyTimer() throws IOException {
        File out = new File(PATHNAME);
        out.createNewFile();
    }

    public static synchronized void startTiming() { time = System.currentTimeMillis(); }

    public static synchronized void reportDuration() throws IOException {
        writer = new FileWriter(PATHNAME,  append: true);
        writer.write( str: (System.currentTimeMillis() - time) + "\n");
        writer.close();
        System.out.println("You have stayed " +
                (System.currentTimeMillis() - time) + " milliseconds in " +
                "your target method.");
    }

}
```

These methods will be transformed into Jimple statements and then get inserted in a Jimple Units Chain by the InvokeStaticInstrumenter class.

The InvokeStaticInstrumenter class is the core piece of logic of our entire application, and it contains a static component which takes care of creating Jimple methods of our logging logic contained in MyTimer, and an internalTransform method, which is used by Soot to take a target method converted in Jimple format, go through its content, and insert Jimple statements where it sees fit: in this case, after checking that we are analysing the correct method, we insert a time counter at the beginning of the method, and a report time statement at the end of the method, which simply computes the execution time of the function and writes down the result in a separate text file.

```
public class InvokeStaticInstrumenter extends BodyTransformer{

    static SootClass timerClass;
    static SootMethod startTiming, reportDuration;
    static {
        String sourceDirectory = System.getProperty("user.dir") +
                File.separator + "target" + File.separator + "classes";
        Options.v().set_soot_classpath(sourceDirectory);
        timerClass = Scene.v().loadClassAndSupport( className: "instrumenter.MyTimer");
        startTiming = timerClass.getMethod( subsignature: "void startTiming()");
        reportDuration = timerClass.getMethod( subsignature: "void reportDuration()");
    }
    final String targetMethod;

    public InvokeStaticInstrumenter(String targetMethod) { this.targetMethod = targetMethod; }

    /* internalTransform goes through a method body and inserts */
    protected void internalTransform(Body body, String phase, Map options) {
        SootMethod method = body.getMethod();
        if (method.getName().equals(targetMethod)) {
            System.out.println("instrumenting method : " + method.getSignature());
            InvokeExpr incExpr = Jimple.v().newStaticInvokeExpr(startTiming.makeRef());
            Stmt incStmt = Jimple.v().newInvokeStmt(incExpr);
            Chain units = body.getUnits();

            //  insert new statement into the chain (we are mutating the unit chain).
            units.insertBefore(incStmt, ((JimpleBody) body).getFirstNonIdentityStmt());

            Iterator stmtIt = units.snapshotIterator();
            Stmt stmt;
            while (stmtIt.hasNext()) {
                stmt = (Stmt) stmtIt.next();
                if ((stmt instanceof ReturnStmt) || (stmt instanceof ReturnVoidStmt)) {
                    InvokeExpr reportExpr = Jimple.v().newStaticInvokeExpr(reportDuration.makeRef());
                    Stmt reportStmt = Jimple.v().newInvokeStmt(reportExpr);
                    units.insertBefore(reportStmt, stmt);
                }
            }
        }
    }
}
```

The next image illustrates the content of our MicroBenchmark. As we can see, it contains trivial but significant methods, with clearly identifiable and visualisable time complexities. These are the methods that we instrument to test our application, and as we will see in the rest of this thesis, Freud will be able to recognize the type of time complexities that these function represent, thus inferring correct probability distributions, graphs and probabilistic performance annotations.

```java
package demo.classes;

public class MicroBenchmark {

    public static void main(String[] args) throws InterruptedException {
        testQuad(Integer.parseInt(args[0]));
    }

    /* numeric */

    private static void testLinear(int t) throws InterruptedException {
        System.out.println(t);
        for (int i = 0; i < t; i++) {
            Thread.sleep( millis: 250);
        }
    }

    private static void testQuad(int t) throws InterruptedException {
        for (int i = 0; i < t; i++) {
            for (int j = 0; j < t; j++) {
                Thread.sleep( millis: 250);
            }
        }
    }

    /* strings */

    private static void testLinearStr(String str) throws InterruptedException {
        for (int i = 0; i < str.length(); i++) {
            Thread.sleep( millis: 100);
        }
    }
```

The last thing we want to show in this section is a JSON-formatted example, created through our logging package, of the log files generated after running an instrumented version of the testQuad function of the MicroBenchmark class.

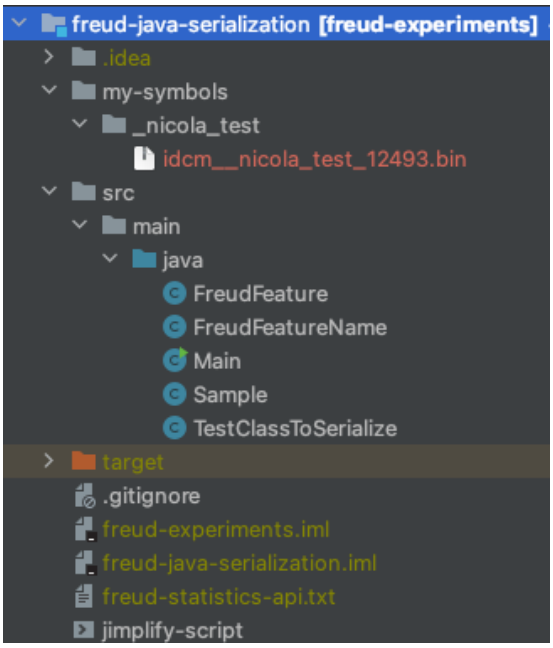

```
"Samples" : [ {
  "inputParam" : 1,
  "measuredTime" : 251
}, {
  "inputParam" : 2,
  "measuredTime" : 1011
}, {
  "inputParam" : 3,
  "measuredTime" : 2272
}, {
  "inputParam" : 4,
  "measuredTime" : 4043
}, {
  "inputParam" : 5,
  "measuredTime" : 6324
}, {
  "inputParam" : 6,
  "measuredTime" : 9095
}, {
  "inputParam" : 7,
  "measuredTime" : 12380
}, {
  "inputParam" : 8,
  "measuredTime" : 16159
}, {
  "inputParam" : 9,
  "measuredTime" : 20473
}, {
  "inputParam" : 10,
  "measuredTime" : 25267
```


As we can see, this is as simple as a series of tuples of input and output of the target function. Where the input in this case is input integer parameter, and the output is the running time of the function with that input. Next thing to to is to generate a mathematical relation based on these data points, through a statistical analysis.

4.3 Serializer Application

Before being able to use Freud's statistical analyser, we need to format and serialise our log file in a way that it is compatible with Freud's API, since as of today, its API only takes binary files serialised in a particular way. Therefore, to solve this we had to look at how these binaries were read inside Freud's statistical analysis component, and reverse engineer the problem, creating a serializer [16] that was compatible with that API.

The following is the application we have built to address this problem.



This application is quite straightforward: the main java folder contains some POJOs which represent some data structures used in Freud to package the performance logs, the my-simbols folder contains the binary files after the serialization.

In this process, we have also created a text file, namely 'freud-statistic-api.txt', which documents the API required by Freud's statistical analysis component.
As we can see in the following picture, the log files are structured like this: method name for the instrumented and analysed function, features list of state variables of any kind, a list of metrics samples (in our case, we are only giving values for the execution time), to which is associated a list of features (like the method's input parameters) which contributed to the generation of such metrics samples.

```
// METHOD NAME
name_len: uint32_t
fname: sizeof(char) * name_len

// FEATURE NAMES
feature_names_count: uint32_t
    vnlen: uint16_t
    vname: sizeof(char) * vnlen

// TYPE NAMES
type_names_count: uint32_t
    vnlen: uint16_t
    tname: sizeof(char) * vnlen

// SAMPLES
samples_count: uint32_t
    uid_r: uint32_t
    time: uint64_t
    mem: uint64_t
    lock_holding_time: uint64_t
    waiting_time: uint64_t
    minor_page_faults: uint64_t
    major_page_faults: uint64_t

    // FEATURES
    num_of_features: uint32_t
        name_offset: uint64_t
        type_offset: uint64_t
        value: int64_t

    // BRANCHES
    num_of_branches: uint32_t
        branch_id: uint16_t
        num_of_executions: uint32_t
            taken: bool

    // CHILDREN
    num_of_children: uint32_t
        c_id: uint32_t
```

We believe this simple text file improves the documentation of Freud's API. For instance, it could also be used by anyone interested in building an instrumenter for a new language (Python, JavaScript, Go, Rust and so on) to create their own Freud's compatible serializer.
An interesting alternative would be to change Freud's statistical analysis component to make it accept log files in a language-agnostic format. The main reason why Freud's current API only accepts log files in binary format is that the log reading process needs to be very performant. JSON files could become a problem if the files to read are huge, which is a common scenario for large-scale systems if they are run for a non-trivial amount of time, since they would generate huge performance log files. That said, there exist other compiled, language-agnostic and efficient formats, like Protocol Buffers, which could be tested, and which could be a viable solution.

4.4 Experiments and Tests

In this section we show the results of a complete experimentation of our applications over the MicroBenchmark class, instrumenting and deriving probabilistic performance annotations for two methods testLinear and testQuad, one with linear time complexity with respects to the input variable and the other with quadratic time complexity.

These are the implementations for the two methods.

```
private static void testLinear(int t) throws InterruptedException {
    System.out.println(t);
    for (int i = 0; i < t; i++) {
        Thread.sleep( millis: 250);
    }
}
```

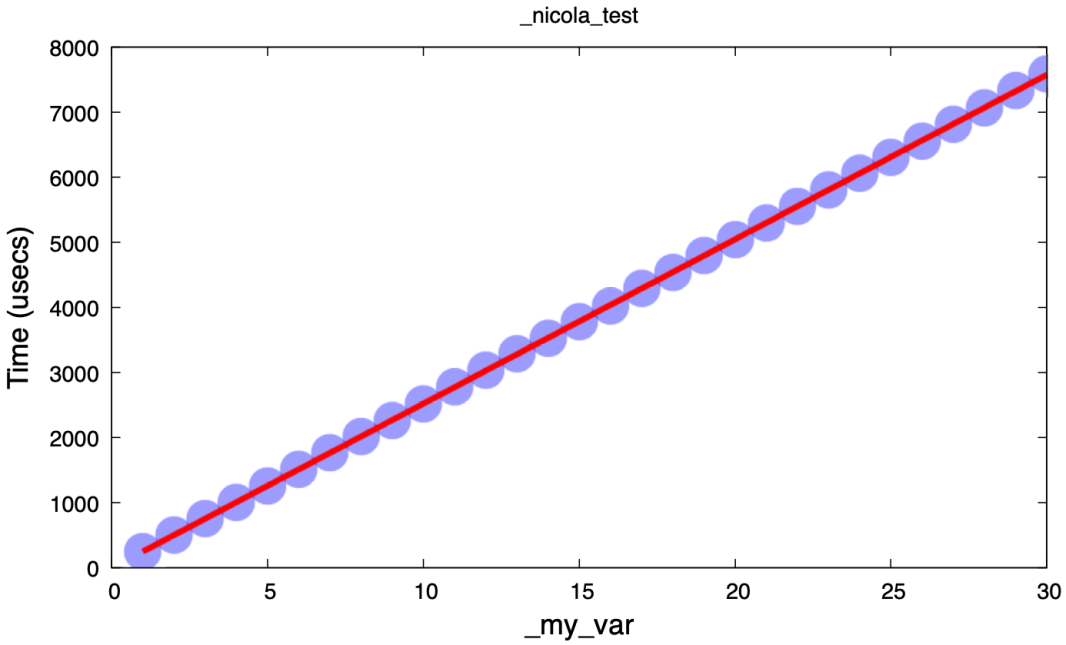
```
private static void testQuad(int t) throws InterruptedException {
    for (int i = 0; i < t; i++) {
        for (int j = 0; j < t; j++) {
            Thread.sleep( millis: 250);
        }
    }
}
```

The following are the results of statistical analyses made with Freud over the performance logs produced by running the instrumented versions of the above two methods. The fact that the points follow almost perfectly the inferred distributions is to be expected, because our instrumented methods are a synthetic simulation of a program with such time complexities. Of course, when the instrumented method is taken from a real-world application, this would not happen and the inference in that case would seem less obvious.

Annotation for testLinear

```
R C [ * ] T ~ [det= 9.999914720924512057 1e-01 ] -1.188505747126032741 1e+00 +2.525906562847608256 6e+02*_my_var
```

Regression plot for testLinear

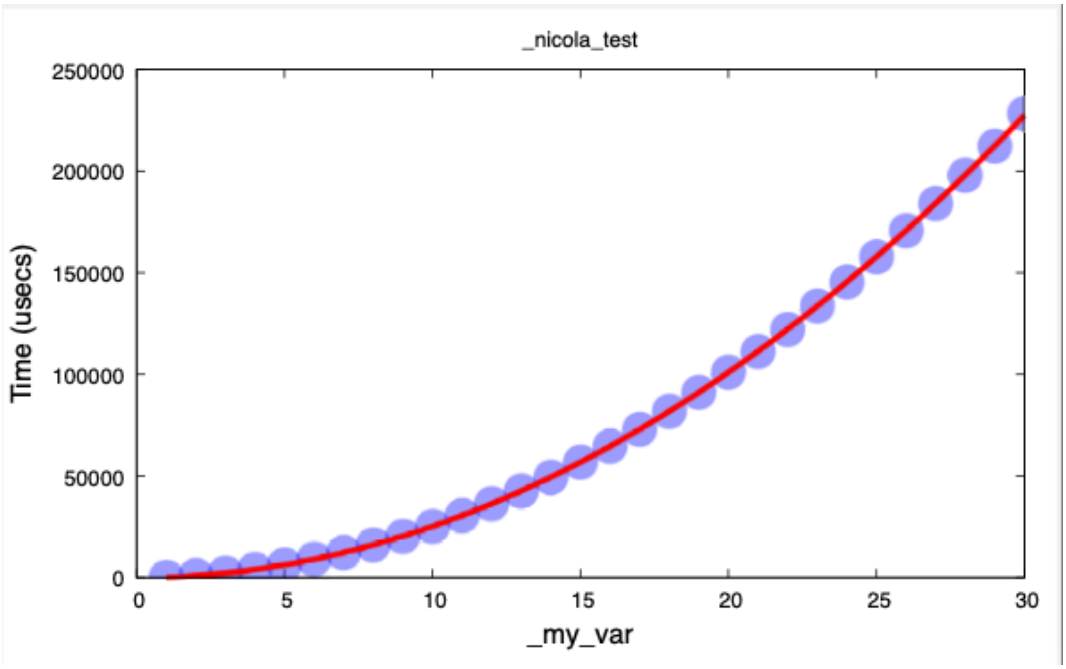


As we can see from the annotation and the plot, Freud, after having analyzed the performance log data generated through our instrumented method testLinear, decided that the performance of such method behaved according to a linear probability distribution, which makes sense given that the function implementation is a simple loop, with some wait time inside each cycle.

Annotation for testQuad

```
R C [ * ] T ~ [det= 9.999928554532315772 0e-01 ] -3.370149745844702238 2e+01 +2.528346953911955097 3e+02*_my_var^2
```

Regression plot for testQuad



Also for the testQuad method, which is implemented through two indented loop, inside which we have some little waiting time, Freud managed to understand and extract a correct probability distribution: in this case, quadratic.

As we can see from the annotations and the plots, Freud managed to understand the overall complexity of these two methods, based on real-world data (and not just on theoretical/static analyses over the structure of the methods).
This is the potential of Freud: being able to conduct complexity analyses of software, without ignoring aspects which are, in practice, very influential and which are overlooked by normal complexity analyses. It also manages to do this automatically, without requiring developers to go through lines of code in trying to analyse it. Moreover, the format in which these analyses are produced, also allows to turn them into methods' annotations and therefore into software documentation, helping guide the developer into their journey of software development.

5 Conclusions and Future Work

In this thesis, we explored a new topic in the areas of performance analysis and developers' tooling: probabilistic performance annotations. In particular, we described what this new concept is about, we talked about Freud, a research project implementing the ideas of such concept and making them viable for C/C++ systems, and we described our concrete contribution to Freud's implementation, which consists of a prototype of an instrumenter for Java systems, compatible with Freud. Last but not least, we also described Soot, the tool we relied on to implement our instrumenter.

5.1 Results

The result of this work is twofold. On one hand we documented in a succinct yet exhaustive way what Freud is about, and what to do in order to use it and extend it to work with systems built with languages that are not covered by the current version of Freud. On the other hand we started building a full-fledged instrumenter for Java systems, and, even if currently it can't be used as a standalone tool on real-world system, with some intervention it could achieve that, enabling Freud to be used on Java systems.

5.2 Future Work

There are many things which can be done in the future to improve the current results. First of all, our Java instrumenter can be improved, by enabling it to log the value of interesting features along with the desired metrics. This consists of extending the existing tool to make it reach the level of the Freud's instrumenter for C/C++ systems.

Another way to achieve new results starting from the current ones, is to improve Freud itself. In fact, there are many ways in which Freud can be further developed. In this thesis we cited the possibility of making Freud's statistical analysis component more user-friendly and more inter-compatible by not strictly requiring binary files as input.

But there also other directions of research in this field. We will go over them in the following list.

5.2.1 Extending Java Instrumentation

Already mentioned: extend our work done on these repositories: instrumenter, serializer.

5.2.2 Improving Freud's API

Already mentioned: make the statistical analysis component of Freud accept not just binary files but also intermediate representation languages such as Protocol Buffers.

5.2.3 Workloads for Performance Analysis

Generating workloads that expose a broad range of behaviors for the analysis of some target methods.

5.2.4 Compositional Annotations Using Static Analysis

Obtaining annotations of higher-level methods by analyzing their code together with the annotation of lower-level (called) methods. Ideas: probabilistic symbolic execution; probabilistic programming; Monte Carlo analysis.

5.2.5 Distributed Instrumentation

Augmenting the instrumentation so as to be able to linking the instrumentation data collected over distributed components using causal relations.

5.2.6 Dynamic, feedback-directed, optimized instrumentation for production environments

Dynamically controlling the instrumentation so as to avoid collecting data that turns out to be insignificant or too noisy for the performance analysis (metrics, branch decisions, features). The goal would be to reduce the overhead of the instrumentation to the point that Freud can be safely used with systems running in production environments.

5.2.7 Adding more experiments

Freud was tested over three major systems. More experiments on other C++ systems can be done. Or, once the Java instrumentation component is complete, tests on relevant Java systems can also be carried out: Hadoop could be a great example.

References

[1] *Freud, a tool to generate performance annotations.* https://github.com/usi-systems/freud.

[2] D. Rogora, A. Carzaniga, A. Diwan, M. Hauswirth, R. Soulé. *"Analyzing System Performance with Probabilistic Performance Annotations"*. In: Proceedings of the Fifteenth European Conference on Computer Systems. EuroSys '20. Heraklion, Greece: Association for Computing Machinery, 2020.

[3] https://lightstep.com/deep-systems/

[4] https://go.lightstep.com/global-microservices-trends-report-2018.html

[5] A. Diwan. *Life lessons and datacenter performance analysis* In: 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pages 147-147, 2014.

[6] D. Ardelean, A. Diwan and C. Erdman. *Performance Analysis of Cloud Applications* In: 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18), pages 405-417, 2018.

[7] https://www.datadoghq.com/

[8] https://aws.amazon.com/cloudwatch/

[9] https://lightstep.com/

[10] https://prometheus.io/

[11] https://grafana.com/

[12] https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470050118.ecse386

[13] http://soot-oss.github.io/soot/

[14] https://github.com/usi-systems/freud/tree/master/freud-statistics

[15] https://github.com/amadionix/freud-java-instrumenter

[16] https://github.com/amadionix/freud-java-serialization

[17] https://www.youtube.com/watch?v=mLjxXPHuIJo

[18] https://www.awwwards.com/brain-food-perceived-performance/

[19] https://ai.googleblog.com/2009/06/speed-matters.html

[20] https://nanopdf.com/download/performance-related-changes-and-their-user-impact_pdf

[21] https://www.youtube.com/watch?v=bQSE51-gr2s

[22] *Pin - A Dynamic Binary Instrumentation Tool* https://software.intel.com/enus/articles/pintool

[23] https://www.videolan.org/developers/x264.html

[24] *MySQL Server 2019. MySQL Server.* https://github.com/mysql/mysqlserver

[25] http://dwarfstd.org