Analyzing System Performance with Probabilistic Performance Annotations

Doctoral Dissertation submitted to the
Faculty of Informatics of the Università della Svizzera Italiana
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Daniele Rogora

under the supervision of
Prof. Antonio Carzaniga and Prof. Robert Soulé

January 2021

Dissertation Committee

Prof. Matthias Hauswirth	Università della Svizzera Italiana, Switzerland
Prof. Fernando Pedone	Università della Svizzera Italiana, Switzerland
Prof. Amer Diwan	Google, Mountain View, USA
Prof. Timothy Roscoe	ETH Zurich, Switzerland

Dissertation accepted on 11 January 2021

Research Advisor	Co-Advisor
Prof. Antonio Carzaniga	**Prof. Robert Soulé**

PhD Program Director

Prof. Walter Binder, Prof. Silvia Santini

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Daniele Rogora
Lugano, 11 January 2021

To Cecilia, may your journey know no limits

Abstract

Understanding the performance of software is complicated. For several performance metrics, in addition to the algorithmic complexity, one must also consider the dynamics of running a program within different combinations of hardware and software environments. Such dynamical aspects are not visible from the code alone, and any kind of static analysis falls short.

For example, in reality, the running time of a `sort` method for a list is going to be different from the expected $O(n \log n)$ complexity if the hardware does not have enough memory to hold the entire list.

Moreover, understanding software performance has become much more complex because software systems themselves continue to grow in size and complexity, and because modularity works quite well for functionality but less so for performance. In fact, the many subsystems and libraries that compose a modern software system usually guarantee well documented functional properties but rarely guarantee or even document any performance behavior. Furthermore, while functional behaviors and problems can be reasonably isolated, performance problems are often interaction problems and they are pervasive.

Performance analysts typically rely on profilers to understand the behavior of software. However, traditional profilers like *gprof* produce aggregate information in which the essential details of input or context-specific behaviors simply get lost. Some previous attempts at creating more informative performance profiles require that the analyst provide the performance models for software components. Other performance modeling tools deduce those models automatically but consider only the abstract algorithmic complexity, and therefore fail to find or even express interesting runtime performance metrics.

In this thesis, we develop the concept of *probabilistic performance annotations* to understand, debug, and predict the performance of software. We also introduce Freud, a tool that creates performance annotations *automatically* for real world C/C++ software.

A performance annotation is a textual description of the expected performance of a software component. Performance is described as a *function* of fea-

tures of the input processed by the software, as well as features of the systems on which the software is running. Performance annotations are easy to read and understand for the developer or performance analyst, thanks to the use of *concrete performance metrics* such as running time, measured in seconds, and *concrete features*, such as the real variables as defined in the source code of the program (e.g., the variable that stores the length of a list).

Freud produces performance annotations *automatically* using dynamic analysis. In particular, Freud instruments a binary program written in C/C++ and collects information about performance metrics and features from the running program. Such information is then processed to derive probabilistic performance annotations. Freud computes regressions and clusters to create regression trees and mixture models that describe complex, multi-modal performance behaviors.

We illustrate our approach to performance analysis and the use of Freud on three complex systems—the ownCloud distributed storage service; the MySQL database system; and the x264 video encoder library and application—producing non-trivial characterizations of their performance.

Acknowledgments

My sincere thanks go to my friendly advisor Antonio for his continuous guidance, teachings, and support.

I want to thank Alessandro, Gianpaolo, and Robert for guiding and encouraging me in my academic career, and Ali, Koorosh, Michele, and Fang for sharing memorable steps during my path.

My dear friends Alberto, Cristina, Fabio, Grazia, Luca, Maria Elena, and Roberto for sharing joyful moments that helped in overcoming adversities.

My beloved family, my center of gravity, for their unlimited patience and unconditional support.

Lastly, I want to thank those who think they do not deserve a spot here. Your friendship was an invaluable contribution in writing this thesis.

The achievement would have never been possible without the unique contribution of all these people. Thank you! Grazie!

Contents

Figures

Tables

Chapter 1

Introduction

In this thesis we introduce a methodology for dynamic performance analysis for complex software systems. We also introduce a set of tools, *Freud*, that implement such analysis. Freud creates *probabilistic performance annotations* for software components at different granularity, ranging from small utility methods to the main function of a program. These performance annotations describe cost *functions* that correlate some performance metrics measured on the running program with some features describing either the state of the program or the system on which the program is running.

The metrics are measurable quantities that describe the real behavior of the software system when running on real hardware. Example metrics are the running time, measured in seconds, or the dynamic memory that is allocated, measured in bytes. Other interesting metrics include lock holding or waiting time, or the number of memory page faults.

Similarly, the features are measurable values that describe the state of the program or of the system on which it is running. A feature may indicate the value of a variable, the number of threads being used by the program, or the clock frequency of the CPU.

One key aspect of the analysis is that it is almost completely automated. The performance analyst does not need a deep understanding of the system to generate rich and meaningful performance annotations. Freud automatically finds the features that affect performance, and also manages to identify even non trivial, multi-modal performance behaviors.

Performance annotations created with Freud can serve as an intuitive documentation of the performance of a system or as a performance specification. Additionally, the automatic generation of performance annotations can be triggered programmatically to accompany the development and evolution of a soft-

ware system, similar to functional tests in continuous integration. In such a context or more generally, performance annotations can be used as assertions or test oracles, for example to detect performance regressions. Also, since they define cost functions, performance annotations can extrapolate what is measured experimentally to predict the performance of a software system with new inputs or within new operating contexts.

Our performance annotation analysis improves over previous approaches to software performance analysis and modeling. Like traditional profilers, Freud analyzes the real behavior observed on a real system. At the same time, instead of computing some aggregate information about the performance like traditional profilers, Freud generates cost functions which are much more informative about the observed and *expected* behavior of software. This approach was also explored by algorithmic and input-sensitive profilers, but previous attempts failed at considering real performance metrics or at automating the analysis.

In essence, this thesis proposes a new approach to performance analysis that combines some key aspects of previous approaches to create more informative performance specifications. In addition to the approach, in this thesis we also describe Freud, a set of tools that implement such approach to create performance annotations automatically for C/C++ programs. Freud is a modularized open source project, and can be expanded to work with other programming languages.

The chapter is structured as follows: we first introduce the challenges in performance analysis (Section 1.1); then we describe the main contributions of this thesis consisting of a new methodology for performance analysis (Section 1.2), and of a concrete implementation of such methodology in a tool called Freud (Section 1.3).

1.1 Performance Analysis

To manage complexity in programs, developers use a combination of tools (e.g., [15, 14, 21, 20]) and best practices (e.g., modularization and stylized documentation). These tools and practices help with functionality but not so much with runtime dynamical aspects such as performance. Understanding performance is complicated. Admittedly, understanding functionality is not easy either. However, in principle, functional behaviors are fully determined by an algorithm, which is expressed in the program code. Even if the code is non-deterministic and therefore the resulting behavior will be stochastic in nature, that behavior would still be fully characterized by the code. In this sense, performance analysis

is fundamentally more complex.

The complexity of performance analysis is twofold: one factor is the algorithmic complexity that, as for functionality, is fully characterized by the algorithm alone. Another factor is the embedding and interaction of the algorithm within its execution environment and ultimately the real-world. Thus real-world performance depends also on the compilation process, which might also be dynamic (just-in-time), or on the dynamic allocation of computational resources among several other competing applications on the same execution platform, which might itself be virtual and/or distributed and therefore affected significantly by, for example, network traffic or power management.

Performance might refer to different metrics of the software. Some of these metrics, such as memory allocation or number of executed instructions, can be fully understood with the algorithmic complexity alone. Even for these metrics, the analysis is not easy, and generally rely on asymptotic approximations, such as $O(n)$, $\Theta(n)$, $\Omega(n)$.

Conversely, performance metrics like the running time or number of page faults incurred during the execution heavily depend on the dynamics of the system that is executing the algorithm.

The analysis of these two types of performance metrics is not fundamentally different. In the end, we always try to find correlations between a metric and features. The difference is that, for some metrics, the noise affects the measurement in possibly considerable ways, and that the dynamics of the system introduce modalities in the behavior that are not visible from the algorithm.

It is no surprise, then, that today developers use good testing frameworks and established practices for functional specifications (that is, the API), while performance specifications are often ignored.

Indeed, even for implementations of classic algorithms, such as sorting algorithms, the dynamics can be complex and significant to the point that asymptotic computational complexity alone would simply be a mischaracterization. Interactions with the memory subsystem, for example, can dramatically affect the performance of these algorithms and introduce modalities. Therefore, developers cannot readily understand the consequences of their code changes. If this problem exists for very basic, standard-library functions, it is even more prevalent for large systems using several layers of external libraries. For example, invoking an innocuous-sounding `getX()` method may result in RPC calls, acquiring locks, allocating memory, or performing I/O. The outcome of not fully understanding the performance of a method may be catastrophic; unintended changes in performance can significantly degrade user experience with the program or the remote service (e.g., by resulting in more timeouts for the user).

To illustrate the challenges in understanding the performance of even a simple method, Figure 1.1 shows the measured running time of the C++ standard library sort method `std::list<int>::sort()`. We measure the running time for a range of input sizes in a controlled environment.

We run the experiments on a Intel Xeon CPU E5-2670, with 8x8GB PC4-17000 ram modules (Samsung M393A1G40DB0), whose root partition is mounted on a Samsung SSD 850 PRO drive. The OS is Ubuntu 18.04.5 LTS, and we compile the program with `g++ v7.5.0-3ubuntu1 18.04`, using the flags `-g -O2`. The system uses the default Ubuntu configuration, with the default scheduler and the `intel_pstate` scaling governor. We execute the `sort` method in a program that we wrote for the purpose. The program first adds its `pid` to a */sys/fs/cgroup/memory* directory to add itself to a specific Linux Control Group, then creates a list of integers with the size that is read as a command line argument. The program creates the list dynamically pushing random integers to the back of the list, and finally calls `std::list<int>::sort()`.

To show the interactions with the memory subsystem we limit to 52MB the amount of resident memory available to the program running `sort` (using Linux Control Groups). Using much larger inputs would have the same effect, depending on the available system memory.

Despite the documented $O(n \log n)$ computational complexity of the function, Figure 1.1 shows that running time has three distinct modalities: $n \log n$ for small inputs, a linear increase with a steep slope for medium-size inputs, and a dramatic jump but lower slope for large inputs. Also, not only we can observe the algorithmic complexity, but we can also see the actual absolute values that determine precisely the running time.

As long as the system has enough memory to store the entire list, `sort` exhibits the expected $O(n \log n)$ time complexity. As soon as the list becomes too big (around 1.5M elements), some memory pages containing nodes of the list are swapped out of the main memory to the slower storage memory. When that happens, the time taken by Linux to store and fetch memory pages from the storage memory dominates the execution time of the algorithm, and the time complexity becomes linear. When the list becomes bigger than roughly 2.1M elements, the performance behavior in our environment changes again. Thanks to our analysis and tool, as we will show in Section 1.3, we can conclude that the big jump in the running time is still the result of the Linux memory management swapping memory pages out of the main memory. In fact, we see that the running time of the `sort` function exhibits a trend that is very similar to the trend seen for the number of major page faults incurred by the program.

Given this explanation of the results, we can conclude that using systems

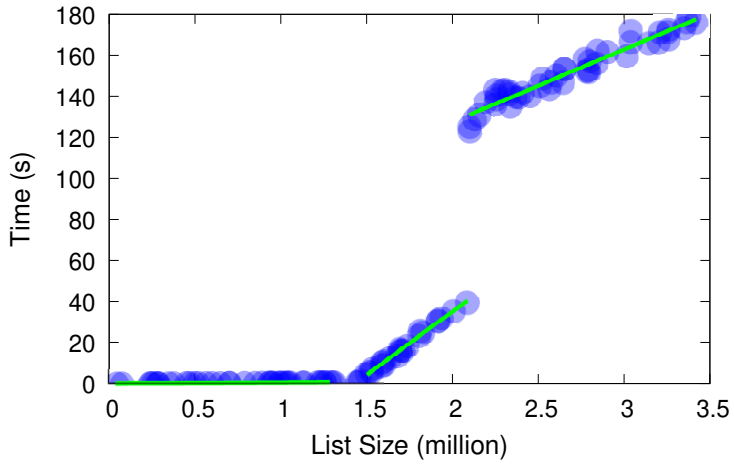


Figure 1.1. Running time for `std::list<int>::sort`

with different amounts of memory (or using different sizes for the Linux Control Group) would change the results considerably. The change from the first to the second modality in the performance would happen at different point on the x axis, shifting to the right with more memory and shifting to the left with less memory. Also, we can argue that running the same program with the same amount of memory on a different machine would also change the performance profile. For example using a CPU with a lower clock speed would probably increase the slope of the $n \log n$ part, while it would probably have little to no effect on the other parts of the graph, where the running time is mainly limited by the storage access time.

How would a programmer wanting to use this code know about these modalities and behaviors? The programmer could carefully study the code before using it. But apart from the fact that this would be very onerous and would lose much of the benefit of modularity, even this is not enough: the big jump in the running time is not obvious or even visible from the code, and instead is a consequence of the interaction of the code with the underlying kernel and memory subsystem. The contribution of every additional element in the list to the total running time depends on the hardware that executes the code; a different CPU executing the code might result in different slopes; a different standard library implementation might result in completely different algorithmic complexities.

In this thesis we present methodologies and tools to generate performance analyses as presented here in this example. We produce descriptions of performance with real metrics, real values for the features, and the real behavior that we extrapolate from the observations on a real system. They can include any

relevant feature, such as the length of a list, the value of an integer paramenter, or the clock speed of the CPU. And we apply this approach to every type of performance metric, whether it is affected by the dynamics of the system, or not.

1.2 Probabilistic Performance Annotations

With our performance analysis we combine and improve every aspect of the profilers and empirical models developed in the past, as described in Chapter 2.

At a high level, our approach works as follows. We collect performance data (e.g., CPU time, memory usage, lock holding/waiting time, etc.) along with features of the arguments to functions (e.g. value of an int, length of a string, etc.) or of the system (e.g. CPU clock speed, network bandwidth, etc.). We then use statistical analysis to build mathematical models relating input features to performance. We produce performance annotations at the method granularity. In other words, our performance annotations always refer to named software routines, which take a set of input parameters, and produce some output before finishing their execution.

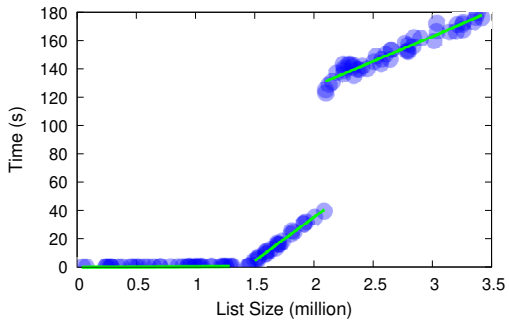
We produce cost *functions* for different performance metrics. These cost functions correlate concrete performance metrics to features of different nature. As we already discussed, cost functions are more informative than the aggregate metric data produced by traditional profilers.

Similarly to traditional profilers, we analyze *real* performance metrics, and whether they are affected by the dynamical aspects of the system, or not. This choice is at the base of two important features of our performance annotations: (1) performance annotations are *probabilistic*, to handle and represent the noise that stems from the dynamical aspects of a system, and (2) performance annotations use *scopes* to represent the multi-modal performance behaviors, whether they are due to the algorithm or to the interaction of the target program with the rest of the system.

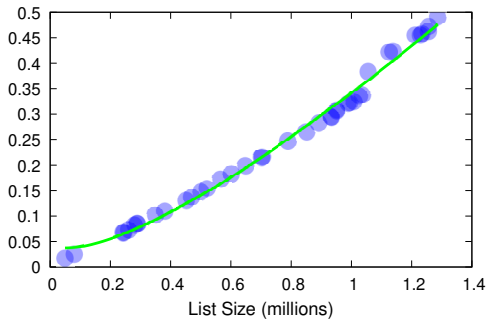
Let's see a real example of performance annotation, for the example of the `std::list<int>::sort` method shown in Section 1.1. Remember that we are artificially limiting the amount of memory to make the sort method swap memory to the main storage, and this induces different modalities in the running time of the method for different list sizes.

Figure 1.2 shows the concrete output of our analysis for the running time. It consists of a textual representation (Fig. 1.2f), and a graphical one (Fig. 1.2e).

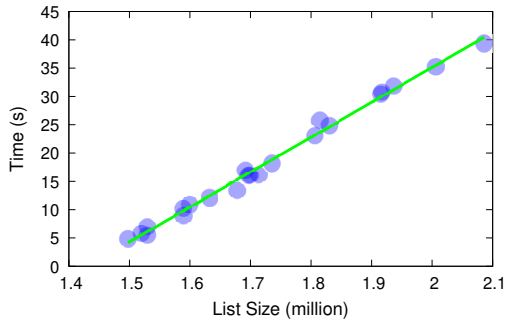
The performance annotation shows that the *running time* metric is mainly affected by the size of the list. The cost functions show that the performance



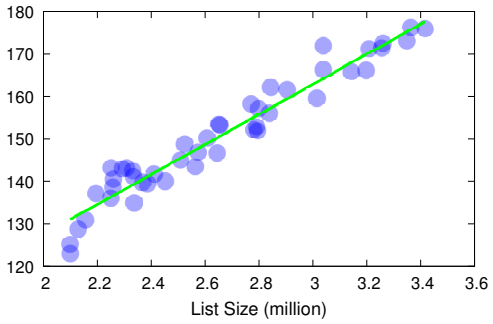
(a) Overview



(b) $O(n \log n)$, small inputs



(c) $O(n)$, medium inputs



(d) $O(n)$, large inputs

(e) Graph representation

```
1 std::list<int>::sort().time {
2 features:
3  int s = *(this->_M_impl._M_node._M_storage._M_storage);
4
5 annotations:
6  [s < 1450341]
7  Norm(53350.31 - 2.10*s + 0.12*s*log(s), 12463.88);
8
9  [s > 1589482 && s < 2085480]
10 Norm(-90901042.29 + 63.11*s, 899547.29);
11
12 [s > 2098759]
13 Norm(56712024.50 + 35.38*s, 3379580.27);
14 }
```

(f) Textual representation

Figure 1.2. Performance annotation for the running time of `list<int>::sort`

complexity is $O(n \log n)$ for lists of less than 1450341 elements, while it becomes $O(n)$ for bigger inputs. Since the correlation is not perfect and there is some noise around the expected performance cost, our cost functions contain random variables. In this case the random variable has a normal distribution with a known mean that depends on the *size* feature, and a known variance.

We believe that any developer or analyst with some domain knowledge would be able to read and interpret the performance annotation easily, without any in-depth knowledge of our annotations language.

Notice how the main feature, the list size, is expressed with the exact name that the feature has in the program that is being analyzed. Thus the meaning of the feature should be immediately clear to the developer.

The graph contains the visual representation of the performance annotation, that is, the set of cost functions, and the set of observations that were used to infer such function. To the human analyst, the graph is even more immediate to interpret than the formal performance annotation,

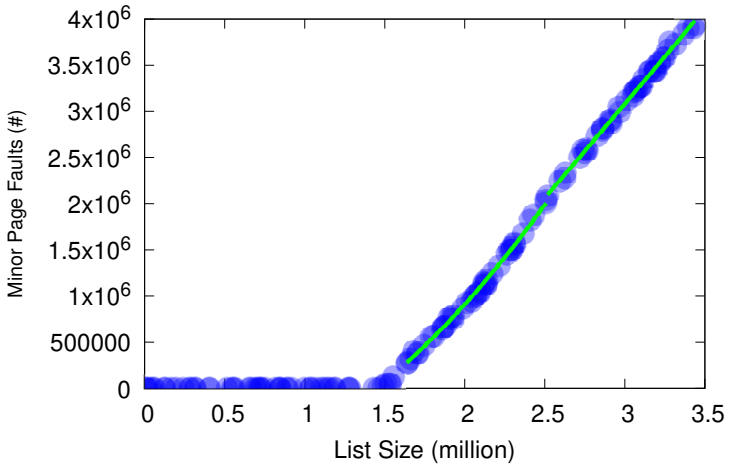
The graphical representation is very useful to human users and are a very effective documentation of the performance behavior, while the textual representation can be efficiently parsed by computers, and make the ideal base for assertion checking.

In order to have a better understanding of the nature of the different modalities in the running time, we produce the performance annotations for `sort` also for two additional performance metrics: minor page faults (Figure 1.3), and major page faults (Figure 1.4).

These performance annotations confirm that indeed there are zero major page faults when the list fits entirely in memory. Also, when the memory is not enough to store the entire list, the number of minor page faults grows almost linearly with the size of the list. It is very interesting to see that the growth is slightly more than linear when the list has around 2^{20} elements.

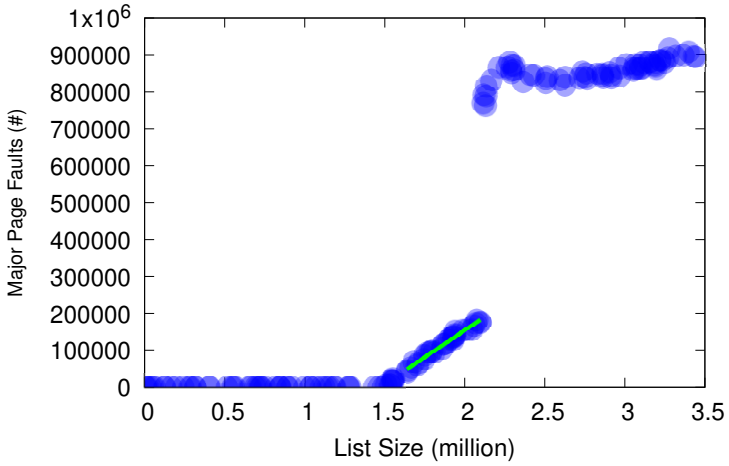
At the same time it is clear that it is the number of major page faults that affects the running time of the `sort` function the most. Guided from the performance annotations, we run more experiments manually to see that the big jump in the major page faults (and in the running time) happens exactly when the list contains $2^{21}+1$ elements. Observing Figure 1.2b more carefully, we can notice a smaller jump in the running time when the list contains exactly $2^{20}+1$ elements. These observations lead us to think that the big jump in the number of memory page faults is not only related to the amount of resident memory available to the program, but also to other factors, like the size of the cache memory of the CPU, or the effect of the algorithm on the memory pages.

While the analysis for this `sort` example might appear trivial even for the




```
1 std::list<int>::sort().pfaults {
2 features:
3  int s = *(this->_M_impl._M_node._M_storage._M_storage);
4
5 annotations:
6  [s > 1637857 && s < 2515031]
7  Norm(-1003849.81 + 4.79e-7 * s^2, 290017652);
8
9  [s > 2515059 && s < 3439237]
10 Norm(-3235264.47 + 2.05 * s, 356324125);
11 }
```

Figure 1.3. Performance annotation for minor page faults for `sort`




```
1 std::list<int>::sort().Pfaults {
2 features:
3  int s = *(this->_M_impl._M_node._M_storage._M_storage);
4
5 annotations:
6  [s > 1637857 && s < 2094257]
7  Norm(-431212.95 + 0.23 * s, 19540310);
8 }
```

Figure 1.4. Performance annotation for major page faults for `sort`

human performance analyst, our analysis can be applied to much more complex cases with similar results, as we will show in the experimental evaluation presented later in Chapter 6.

Again, one of the key advantages of our analysis is that it can be performed automatically, minimizing the input requested to the user. This avoids human error, makes the adoption of the performance analysis more likely, and opens new possibilities in terms of automation and integration of performance analysis in common development practices. Indeed, the graphs and textual annotations showed in Figures 1.2-1.4 were generated automatically by our tool, Freud.

1.3 Freud

Freud performs the analysis automatically. The user is not required to have any knowledge of the program under analysis, to create semantically meaningful performance annotations. The only input taken by Freud are (1) the binary of the target program compiled with debugging symbols, (2) the names of the methods (or "symbols") that shall be analyzed, and (3) the performance metrics of interest. Also, if necessary, the workload that is processed by the target program has to be provided. We do not provide any recipe for the creation of workloads triggering interesting performance behavior.

As we will detail in Chapter 5, Freud uses different approaches to infer the different modalities in the performance behavior automatically, regardless of the origin of the multi-modal behavior.

Concretely, Freud consists of a set of tools. On the one side there are instrumentation modules, that find and extract data from a running program. On the other side there is the analysis module, which performs a statistical analysis on the data collected by the instrumentation.

The statistical analysis is generic. As longs as the data collected by the instrumentation is provided in the correct format, the analysis can work with software written in any language.

The instrumentation part is necessarily language specific. We have implemented a prototype for C/C++ code, that uses Intel Pin [35] to dynamically instrument the target program. The choice of the C/C++ programming language is motivated by several factors.

C++ is the language used for most of the performance critical software. It is extensively used by the industry in many different areas. Also, C is probably one of the most difficult programming language to analyze for our purpose of creating performance specifications. In particular, C has a type system that en-

forces only a weak form of type safety that can in any case be circumvented. In other words, we give a reference implementation for a difficult language, and therefore argue that creating instrumentation tools for other, stricter languages should be relatively easy.

Concretely, to create a performance annotation for the `std::list::sort` function, the performance analyst would:

1. compile the program that uses the `sort` function with debugging symbols (usually passing the -g flag to the compiler)
2. run the Freud binary analyzer (*freud-dwarf*) on the binary file that contains the compiled program
3. compile the PinTool using the instrumentation code generated in step 2
4. run the instrumented program multiple times with Intel Pin (*freud-pin*), executing the sort function with diverse inputs
5. run the Freud statistical analyzer (*freud-statistics*) on the logs collected in step 4

We developed Freud with the ideal goal of making it reasonable efficient to be used in a production environment. In this way Freud could continuously collect new data, to react to unexpected performance behaviors as soon as they happen. For this purpose, we tried different approaches to minimize the overhead introduced by Freud.

1.4 Contribution and Structure of the Thesis

In essence, this thesis describes *systems* work. We develop a new model for performance analysis with a very practical approach. Indeed the requirements and the development of the model have been driven by real-world cases of complex performance analysis. We also present a real, open-source prototype that performs performance analysis of complex software using most of the features of our performance modeling language.

In summary, this thesis makes the following contributions:

- We compare our approach and tool for performance analysis to the models and tools developed in the past (Chapter 2);
- We develop a performance model in the form of probabilistic performance annotations. In Chapter 3 we detail the features and the language used for our model formally;

- We describe a methodology to derive performance annotations through dynamic analysis. In particular, our techniques can automatically identify relevant features of the input, and relate those features through a synthetic statistical model to a performance metric of interest (Chapter 4);
- We describe a concrete implementation of the methodology in a tool named Freud, which automatically produces performance annotations for C/C++ programs (Chapter 5);
- We evaluate Freud through controlled experiments, and analyze two complex systems: MySQL (written in C++), and the x264 video encoder (written in C). We also show how the same analysis can be applied with promising results also to the ownCloud storage service, which is a web application written in PHP (Chapter 6).

Chapter 2

Related Work

A lot of research spanning several decades has focused on performance analysis for software systems. Indeed, optimizing algorithms and their implementations to reduce resource usage is one of the driving factors in software development and computer science generally, both for performance-critical applications and not. Hardware and software profilers already existed in the early 70s, allowing the programmer to identify hot spots and bottlenecks in their programs. For example, the Unix *prof* command already appears in the 4th edition of the Unix Programmer's Guide, dated 1973.

Given the number of publications and authors working on performance analysis in the past, there are different possible ways of classifying the previous approaches. Examples are the surveys by Balsamo et al. [2] and Koziolek [26].

The first distinction that we can make between different approaches to performance modeling, is that of *static* models as opposed to *empirical* models.

Static models (e.g. [5, 13, 47, 18, 4]) are designed to be used without any concrete implementation of the program. The main advantage is that they can be employed in the early design phases of software. These models typically represent software as a network of boxes, either extending UML diagrams or using ad-hoc modeling languages. Each box represents a software component, and is characterized by a user provided performance profile. These models typically provide tools to compose the behavior of different components in the system (e.g., the runtime needed to access the database) to infer and predict the overall performance of the complete software system (e.g., the end-to-end latency to serve a user request).

Conversely, empirical models (e.g. [30, 29, 33]) are created through observation of real software. These models typically instrument or augment existing software to extract performance information while the software runs. It goes

without saying that these models need a concrete implementation of the software, but have many advantages over static approaches. The amount and quality of information that can be collected is much higher, and the analysis can be automated, reducing the need for any user input, which can be costly and is anyway subject to human errors.

We now focus exclusively on empirical approaches to software modeling. Indeed, this is the approach that we take with our performance annotations. In the following sections we describe part of the research efforts in the field of software performance. Besides traditional profilers (Section 2.1), software engineering research has developed several ideas and tools to assert (Section 2.2) or infer (Section 2.3) software performance. Most of these efforts resulted in some prototype implementations of the ideas. Finally, Section 2.4 and Section 2.5 show examples of research papers that discuss two of the immediate extensions of this work, namely distributed instrumentation and performance prediction on different hardware. In Section 2.6 we briefly compare all these implementations to Freud, our tool to generate performance annotations.

2.1 Traditional Profilers

Today, developers rely on profilers like *gprof* [17] or *JProfiler* [24] to understand the performance of their code.

Traditional profilers measure the execution *cost* (e.g., running time, executed instructions, cache misses) of a piece of code. The profiler would typically access information about function calls and periodically sample the program counter to observe which function of the program is being executed. Knowing the total running time for the program, the profiler computes aggregate statistics on functions, indicating for example that function f was called a total of 1000 times, that 10% of program execution time was spent in function f, and that invocations of function f took 4ms on average.

The information the gathered and summarized by the profiler might be insufficient in some scenarios. Consider the `sort` example above: not only a traditional profiler would have no information about the different modalities, but the output produced would probably be misleading. The aggregate information about the runtime would show only the total time spent in the sort method during the execution of the program, and the number of calls, indicating an average execution time but not its variance.

Since the variance in the running time is large (it ranges from 0 to 180 seconds), the aggregate information would fail completely in describing the real

performance, suggesting an expected running time much higher than what it is in reality.

Traditional profilers do not relate the performance of a method to its input and offer no predictive capabilities. Also, traditional profilers generate information that is specific not only to a particular executable program, but also to a specific run (one or more) of the program. This means that a profiler shows information that is potentially affected by a specific context (e.g. the number of CPUs, the size of the input passed to the program, ...), without collecting any information about such context. This means that the data collected with profilers is useful only to the user of the profiler who ran the experiments, since the results depend on information that is usually not provided with the analysis.

DTrace and *Perfplotter* are more advanced tools that extend the capabilities of traditional profilers in different directions.

DTrace [8] represents one of the first attempts to instrument production systems, with zero overhead on non-probed software components. DTrace introduces a new language, called `D`, to let users write their own instrumentation code, which can access information in the user-space or in the kernel space. With the `D` language, performance analysts can either produce aggregate information about a finite set of executions of a given method, or produce specific information about every single execution. On the other hand, both the selection of the information to collect and the analysis of the raw data are a responsibility of the users of DTrace. While DTrace was originally written for Solaris, several ports exist for different operating systems, including Linux. On the other hand, DTrace relies on its own kernel module to work, which is not in the Linux mainline repository, possibly limiting the adoption of the tool.

Perfplotter [9] uses probabilistic symbolic execution to compute performance distributions for Java programs. Perfplotter, which takes as input Java Bytecode, a usage profile, and outputs performance distributions. Perfplotter uses symbolic execution, in conjunction with the usage profile, to explore all the most common execution paths in the Java byte-code. Every unique path is executed a number of times with varying input parameters, and the average execution time is measured and associated with that path. The output distribution shows the average execution time and the probability of being executed for each path.

2.2 Performance Assertion Specification

PSpec [34] introduces the idea of *assertions* that specify performance properties. PSpec is a language for specifying performance expectations as automatically

checkable assertions.

PSpec uses a trace of events produced by an application or system (e.g., a server log). Developers modify their programs to produce event traces in the correct form. These traces amount to a sequential list of typed events. Each event type has a unique name and a user-defined set of attributes. Basic attributes include the *thread_id*, the *processor_id*, and the *timestamp*. Other attributes might include the *size* of an input, the *name* of a file, and so on.

Developers manually specify performance assertions in the PSpec language. They first define an interval, identified by a pair (*start_evt*, *end_evt*). For each interval they can specify a metric computed as a function of some attributes of some events. For example, a simple metric would be running time, computed as $timestamp(end\_evt) - timestamp(start\_evt)$. For each interval, users of PSpec can write assertions using the attributes of the events defining the interval. The assertions can include unknown variables that can be assigned by the *checker* tool provided by PSpec. PSpec also provides the *where* keyword, which can be used in assertions to filter the intervals for which the assertion is checked.

Finally, PSpec provides the *checker* tool, that processes logs and the performance specifications to validate the assertions.

Vetter and Worley [45] use assertions that are added to the source code of the target program by the performance analyst. Assertions apply to specific code segments identified by the keywords `pa_start` and `pa_end`, and can access a predefined list of hardware performance counters, in addition to any user-defined feature that is passed to the performance assertion explicitly in the code. Vetter and Worley develop a runtime system that applies the optimal instrumentation to extract the metrics requested by the user in the most efficient way. Assertions are evaluated at runtime while the target program is executing.

Since performance assertions integrate natively with the source code of the target program, developers can use the result of the assertions directly in their programs, reacting to the outcome of the assertions. This approach requires a tight integration with the specific programming language of the target program. The authors implemented the runtime as a library for C, but argue that integrating their code in the compiler might be a viable alternative.

Both of the above approaches allow developers to specify fixed bounds for the values of chosen metrics in their assertions (e.g., on running time), but do not provide a statistical approach based on the *distribution* of those metrics. Also, the approaches require a manual specification of the performance models/assertions. Finally, the approaches described in this section require the availability of the source code of the target program, so that the program can be augmented and recompiled with the required instrumentation specifications.

2.3 Deriving Models of Code Performance

Trend Profiling [16], Algorithmic Profiling [49] and Input-Sensitive Profiling [11, 12] represent a new form of profiling: instead of only measuring the execution *cost*, these profilers characterize a specific *cost function,* namely a relationship between input size and execution cost. They produce profiles such as: function f takes $5 + 3i + 2i^2$ ms, where i is the length of the input array.

Besides the common goal of inferring performance cost models, these approaches have a number of differences:

- *Granularity:* Trend Profiling and Algorithmic Profiling analyze sub-routine code blocks, while Input-Sensitive profiling considers complete routines.

- *Feature discovery:* Trend Profiling does not make any attempt at finding relevant features, and relies completely on the input provided by the user. Algorithmic Profiling and Input-Sensitive profiling, instead, try to find the relevant feature automatically.

- *Feature types:* While the definition of a feature is left to the user of Trend Profiling, both Algorithmic and Input-Sensitive profiling use a notion of size as the only feature affecting performance. Algorithmic Profiling computes the size of the data structures that are used either as input or output by the code block under analysis, while Input-Sensitive profiling considers the number of memory cells accessed. None of these profilers account for multiple feature when computing cost models.

All these algorithmic profilers consider the number of executions of basic blocks or loop iterations as the performance metric. While this performance metric has the advantage of not being subject to noise in the measurement, and thus being good for analyzing the asymptotic performance behavior of software, it also has serious limitations.

Again, this kind of analysis would be limiting in our sort example. We would only see the $O(n \log n)$ algorithmic complexity, but we would have no signs of the different modalities, since the interaction with the memory subsystem would be completely ignored. Moreover, we would have no idea of the actual time spent executing the function. Our work complements algorithmic profilers in that, rather than using an abstract notion of complexity, it considers real performance metrics and also the interaction of the code with the underlying hardware and software.

2.4 Distributed Profiling

While we do not consider the problem of distributed instrumentation directly in this dissertation, extending our analysis to distributed systems is clearly a possible future extension. In fact, using distributed instrumentation to be able to collect more potential features would greatly extend the possible uses and applicability of our analysis.

Distributed profiling allows instrumenting distributed systems to correlate events happening on physically or logically different nodes of the distributed system. Recent architectures running cloud services often run in big data centers, where interactive software components run on virtually and physically different machines. Still, it is usually essential for the developers and performance analysts to be able to correlate events observed on one server to events observed on a different server.

The first documented system in order of time that has this precise goal is Magpie ([3]). The authors wanted to measure the end-to-end performance of a web service from the user's perspective. To do that, Magpie tags incoming requests with a unique id, and then propagates this id throughout the processing steps required to fulfill the request.

Dapper [38] takes a very similar approach. Dapper is a low-overhead instrumentation system developed and used internally by Google within their production data centers. Dapper provides precious functional and performance debugging information to developers by continuously and transparently recording traces for requests entering a data center. In particular, Dapper instruments a small set of flow-control Python libraries that run at a low level of the software stack and that are used by most of the higher level software components across the Google data centers. Similar to Magpie, the instrumented version of the libraries appends a trace id to every RPC call that is then shared and propagated for all the follow up requests related to the original transaction. In order to limit the overhead brought by the instrumentation, Dapper allows sampling of the traces. Interestingly, researchers at Google claim that analyzing one out of a thousand requests is sufficient to characterize precisely the workflow on their systems.

More distributed instrumentation tools that implement the same ideas are Twitter's Zipkin [43], Uber's Jaeger, and OpenTracing.

Mace at al. combine dynamic instrumentation techniques with distributed instrumentation in their tool Pivot Tracing [28]. This tool uses the established distributed profiling technique of associating requests entering the system with uniquely identifying metadata tags, which are propagated along with the request in the distributed system. Performance analysts, who must have a good knowl-

edge of the system, define *tracepoints*. A tracepoint is a specification containing information on where, in the distributed system, to add jumps to the instrumentation code. Such instrumentation code is created by Pivot Tracing from the user-defined specification of which input parameters to collect from software method being instrumented. Pivot Tracing enables dynamic instrumentation of a distributed systems, allowing performance analysts to collect requested information, optionally including causal relationships between different events, from the system.

2.5 Hardware Aware Performance Specifications

Finally, while we do not experiment directly with hardware with different architectures in this thesis, we think that it is possible to consider hardware specifications as additional features that Freud can analyze to document and predict performance. Some recent research papers explored similar ideas, and proved effective.

Valov et al. [44] show that it is possible to build linear transfer models that allow to predict with good accuracy the performance metrics from one specific hardware configuration to a different one. The approach is still limited in many ways. It considers only the running time as a performance metric. Moreover, the transfer models it produces are effective only over similar hardware, for which linear transformations are appropriate. This is limiting because, in reality, specific hardware architectures could be used to speed up the execution of specific tasks with super-linear improvements. One example is the use of GPUs to encode a video stream to h264.

Thereska et al. [41] introduce the notion of *performance signatures*. A performance signature is a collection of key-value pairs that describe a specific system in a specific state. For example, a performance signature may state that an instance of Microsoft Excel is saving a 100KB file to the filesystem, on a 2-cores x86 machine, with 2GB of RAM. These performance signatures can be generated and collected by Microsoft software that have specific debugging flags activated by the user. Once the authors have collected metrics from thousand of deployment around the world, they make predictions picking the signature that is the closest to the signature of the hypothesis. The distance between two signatures is computed as a summation of the distances of specific (static) features of the signatures, where each feature is assigned a weight chosen by the analyst.

2.6 Comparison with Freud

Table 2.1 summarizes the main differences between Freud and the previous approaches to performance analysis.

	no source required	**cost function**	**real metrics**	**automatic**	**scopes**
gprof	✓		✓		
PSpec			✓		✓
Perf. Assertions					✓
trend-prof	?	✓			
Input-sensitive Prof.	?	✓		✓	
Algorithmic Prof.	?	✓		✓	
Freud	✓	✓	✓	✓	✓

Table 2.1. Comparison of different approaches to performance analysis

It is clear from the table that we are trying to combine all the good features of the previous performance models into one method and tool. The reason that this has not been done before is that these features introduce competing challenges. For example, computing cost functions conflicts with using real metrics, for the reasons that we discussed already (noise and modalities). Similarly, the automation of the analysis conflicts with the description of scopes in the performance behavior, since multi-modal behaviors make the analysis much more difficult.

To recap, Freud expands on the previous ideas in four ways: (1) Unlike algorithmic profiling, which measures cost in terms of platform-independent iteration counts, we measure real performance metrics. (2) The domain of the cost function in algorithmic profiling is the size of a data structure. In input-sensitive profiling, it is the number of distinct accessed memory locations. In contrast, our performance annotations can include arbitrary features of the executing program, or of the system on which it is running. (3) Algorithmic and input-sensitive profiling produce cost functions, but the specific approach for fitting a cost function to the measurements is outside the scope of that work. Freud automatically infers cost functions from the measured data points, producing complete formal performance annotations. (4) Code often exhibits different performance modes (e.g., slow and fast paths), and Freud is able to automatically partition the measurements and to model them as sets of scoped cost functions; prior work produces a single cost function.

Also, many performance models use static features, such as calling context [7], application configuration [19], OS version [42], or hardware platform [22].

In contrast, Freud uses the dynamic state of the running system, i.e., the features that most directly affect computational complexity and are most relevant for scalability.

Finally, despite the semantic richness of the performance annotations created, Freud does not need the source code for C/C++ programs to perform its analysis. All the information needed are extracted from the debugging symbols integrated with the binary of the target program.

Chapter 3

Performance Annotations

In this chapter we describe the notion, the language, and the uses of probabilistic performance annotations. In Section 3.1 we present the syntax of the performance annotation language, and we show through a series of examples what can be expressed with performance annotations. Section 3.2 then shows how performance annotations can be used in different scenarios and for different purposes, such as documentation and performance assertion checking.

While in this thesis we introduce both a language for performance specifications and a tool to automatically generate such specifications, in this chapter we describe the language with all the features we developed for it to describe the performance behavior of any software. Some of these features, such as the composition of different performance annotations, are not yet used (i.e., not implemented) by our tool.

3.1 Performance Annotations Language

We design a language for performance annotations intended for both human engineers and automatic parsing and processing. The language defines concrete metrics, such as run-time or allocated memory, to characterize the performance of a module or function. An annotation expresses one or more relations between one such metric and features of the input or state of the module or function. A typical input feature is a parameter of the module or function being annotated.

The performance annotation language must include parts of the language of the program under analysis. The metrics and the expressions that characterize them are independent of the program, but the identification of the function or module, as well as their input or state features are expressed in the language of the program.

3.1.1 Structure

Performance annotations are composed of three distinct parts: (1) signature, (2) features, and (3) annotations.

The signature introduces the annotation and uniquely identifies a method or function as it appears in the instrumented program, typically with the method name and the list of formal parameters. The signature also indicates which performance metric the annotation describes.

The second part lists all the features that are used in the performance annotation. Each feature is defined by a type, a name, and an initialization expression:

type name = *definition-expression*;

The *type* can be `bool`, `int`, `float`, or `string`; *name* is an identifier used to refer to that feature in the performance annotation; and *definition-expression* defines the feature in terms of the program under analysis, typically in terms of input parameters and other variables accessible within the annotated method or function. Therefore, the definition expression is written is the programming language of the target program, and should be easy to read and interpret by the performance analyst and the developer of the program.

The third and last block describes the behavior of the annotated method or function with a list of expressions. This is the central part of a performance annotation. Each expression characterizes the indicated performance metric as a random variable (the dependent variable) whose distribution is a function of zero or more of the listed features (the independent variables). Thus, in essence, a performance annotation is an expressions like the following:

$$Y \sim expr(X)$$

which is read as: Y is a random variable distributed like $expr(X)$, where X is the set of relevant features. Expressions with zero input features describe behaviors that are independent of the input or for which no relevant features have been observed.

The characterization of each performance metric amounts to the total cost of an execution of the annotated method or function, including all the costs incurred by other methods or functions that are directly or indirectly within the annotated method. Using the terminology of the *gprof* profiler, we describe the *total* cost of the method, as opposed to the *self* cost.

We now present concrete examples of performance annotations to illustrate all the features the language. We will use performance annotations that we created for real software chosen from well known software libraries or programs.

3.1.2 Basics

Figure 3.1 shows a first basic example of a performance annotation for the `sort()` method of the `list<int>` class of the C++ Standard Library (*libstdc++* v6.0.24). The annotation describes the running time of `std::list<int>::sort()` with workloads that impose no memory constraints and that require no memory swapping.

```
1 std::list<int>::sort().time {
2 features:
3  int s = *(this->_M_impl._M_node._M_storage._M_storage);
4
5 annotations:
6  Norm(0.15*s*log(s), 72124.40)
7 }
```

Figure 3.1. A performance annotation with regressions

Line 1 introduces the annotation with the signature of the target method and the performance metric under analysis denoted by the *time* keyword. The unit of measure for a performance metric is implicitly defined by the metric. In this case for running time the unit is $1\mu s$ (microseconds). Table 3.1 shows the metrics that we consider in the current version of the annotation language together with the corresponding keywords and units of measure.

Line 3 defines a feature *s* of type *int* used in this performance annotation. The definition of the feature `*(this->_M_impl._M_node._M_storage._M_storage)` is written in the language of the target program and uses the exact names of variables and struct members that refer to the relevant feature as they would be interpreted in the program and in particular in the scope of the `sort` method.

metric	keyword	units
running time	`time`	microseconds
lock waiting	`wait`	microseconds
lock holding	`hold`	microseconds
memory	`mem`	bytes
minor page faults	`pfaults`	number
major page faults	`Pfaults`	number

Table 3.1. Currently supported metrics and units of measurements

Specifically, since we are analyzing a C++ program and since `sort` is a non-static method of the `std::list` class, `this` represents a pointer to the `std::list` object on which `sort` is called. In particular, the definition of the feature is an *r-value* expression that identifies an *int* object that can be read within the instrumented target program at the time of the execution of the annotated method.

In essence, the annotation states that the running time of `list<int>::sort` depends on a state variable of the list object on which it is called, and as it turns out that state variable represents the number of elements in the list.

Finally, the expression on line 6 of Figure 3.1 characterizes the indicated performance metric with the given feature. This is an example of the general expression $Y \sim expr(X)$ introduced above. In this case, the annotation states that the running time is a random variable with a normal distribution whose mean grows as $s \log s$ and whose variance is known, where s is the size of the list. Notice that the annotation expression does not only define the asymptotic, big-O complexity of the method, but it also defines an actual coefficient for the expected growth rate, which corresponds to the concrete values of the performance metric (time) in its predefined unit of measure (microseconds).

Notice that in this example and in the rest of this document we represent floating-point numbers with a few decimal digits. This is solely for presentation purposes, and is not a limit of the language or its implementation.

3.1.3 Modalities and Scopes

Oftentimes software functions exhibit different modalities in their performance behavior. There are two different sources for such variability: (1) the algorithm takes different paths in the execution depending on the input it reads, or (2) the interaction of the algorithm with the rest of the system on which it is running. More about the sources of multi-modal behaviors will be in Chapter. 4. In this section we will discuss the features of the language that allow to account for and describe this variability, regardless of its nature.

Performance annotations account for multi-modal performance behaviors by means of scope conditions. With scope conditions we want to express cases in which the occurrence of a specific distribution for the performance behavior is tied to the validity of a specific condition (the scope). For example, we have:

$$Y \sim [C_1]expr_1(X_1);\ [C_2]expr_2(X_2); \ldots ; [C_n]expr_n(X_n);$$

This means that the performance metric Y follows the distribution $expr_1(X_1)$ if condition C_1 holds, $expr_2(X_2)$ if condition C_2 holds, and so on.

Going back to the `std::list<int>::sort` function, we limit the amount of resident memory that the program can use. Again, we deliberately set a limit (of 36MB) to cause memory swapping. We make the sort function behave as we showed in Figure 1.1, in Chapter 1. The performance annotation becomes the following:

```
1 std::list<int>::sort().time {
2 features:
3  int s = *(this->_M_impl._M_node._M_storage._M_storage);
4
5 annotations:
6  [s < 1450341]
7  Norm(53350.31 - 2.10*s + 0.12*s*log(s), 12463.88);
8
9  [s > 1589482 && s < 2085480]
10 Norm(-90901042.29 + 63.11*s, 899547.29);
11
12 [s > 2098759]
13 Norm(56712024.50 + 35.38*s, 3379580.27);
14 }
```

Figure 3.2. A performance annotation with regressions

We see that the relevant feature for the running time is still the length of the list. This time, though, we observe three different modalities in the behavior of the method. Each modality is described by a different expression, and a *scope*. Each scope describes the condition that must evaluate to true for the expressions in the following lines to be representative of the expected behavior. As example, line 6 states that if the length of the list (`s`) is smaller than 1450341, then the expected running time for the sort method is the one expressed at line 7. This is exactly what we expect: if the list fits entirely in memory, the running time for `sort` grows as $n \log n$ with the length of the list.

Similarly, at line 9 we have the condition in which the second modality is observed (linear increase, high slope, line 10). Finally at line 12 we have the condition in which the third modality is observed (linear increase, lower slope, line 13).

In all three cases, the performance annotation uses a normal distribution. In the first case, ignoring the constants, we see that the mean value is $n \log n$ where n is the length of the list being sorted, which conforms to the expected $O(n \log n)$

complexity. In the second and third cases, the performance is linear with the length of the input list but with different constant values.

When a specific feature is not present in a scope condition, its value is irrelevant for the evaluation of the condition. The union of all the scopes in one performance annotation covers the entire input domain of the function, while their intersection is empty. In other words, scopes represent a partitioning of the input domain of the function that is described by the performance annotation. As we will see in the following Chapter, the annotation describes a type of classification tree where the scope conditions indicate partitions.

While in Figure 3.2 the scope conditions contain only one variable/feature, there is no such limit in general. For example, one scope condition of a performance annotation could state `[ s < free_mem ]`.

3.1.4 Mixture Models

Expressions with zero input features describe behaviors that are independent of the input or for which no relevant features have been observed.

In the previous examples we always had performance annotations using at least one feature, used in the performance expressions to characterize the expected performance behavior.

In other cases we might have performance annotations that do not use any feature. While such annotations carry less information about the trends in the performance behavior of functions, they might still describe different modalities. In such cases, performance annotations resort to probabilities of occurrence to describe the different modalities. More formally, performance annotations allow expressions like the following:

$$Y \sim \{p_1\}expr_1; \{p_2\}expr_2; \ldots; \{p_n\}expr_n;$$

where each p_i represents the probability that the performance metric is distributed like $expr_i$. When a performance annotation specifies probabilities, it must cover the entire space of behaviors. In other words, $\sum_{i=1}^{n} p_i = 1$.

Here is a concrete example of a performance annotation that does not use any features, but still describes a multi-modal behavior through probabilities:

```
1 emit_file_hooks_pre($exists, $path, &$run).time {
2 features:
3
4 annotations:
5  {0.77}Norm(0.13, 0.00075);
6  {0.23}Norm(0.24, 0.001);
7 }
```

Figure 3.3. A performance annotation with clusters

This performance annotation shows a bi-modal behavior, in which in 77% of the cases the function executes in $0.13\mu s$, while in 23% of the cases is executes in $0.24\mu s$.

In addition, performance annotations can also describe models that combine probabilistic annotations within some given scoping conditions, as follows:

$$Y \sim [C_1]expr_1(x);\ [C_2]\{\{p_2\}expr_2(x);\ \{p_3\}expr_3(x)\}$$

In this case, it still holds true that $p_2 + p_3 = 1$, covering the entirety of the observations in which C_2 is true.

3.1.5 References to Other Annotations

Software written with a good modular design usually breaks the computation in different smaller components to maximize the reuse of such basic components in different parts of the program.

Performance annotations can describe this modularization, and therefore can themselves be modular. This way, performance descriptions are easier to read for programmers, and they are also reusable. In particular, higher level methods can be described by composing the performance annotations of their basic components.

Concretely, performance annotations might refer to other performance annotations to describe the behavior of methods that execute other methods. For example, performance annotations can express models like the following:

$$Y \sim expr(X) + f_1 + f_2 + \cdots + f_n$$

Here we are stating that the expected performance (Y) for some metric, is equal to the sum of the expected performances, for the same metric, of methods $f_1, f_2, \ldots, f_n$, plus some other component that depends on the feature set X. Each

f_i is another method, which may or may not be executed on the same machine, and for which we may or may not already have a performance annotation.

Let's see a concrete example based on the `sort_two` C++ function listed in Figure 3.4:

```
1 void sort_two(std::list<int> * l1, std::list<int> * l2,
2           unsigned int l2_size) {
3   for (unsigned int i = 0; i < l2_size; i++) {
4      l2->push_back(random());
5   }
6   l1->sort();
7   l2->sort();
8 }
```

Figure 3.4. The C++ source code of sort_two

On lines 6–7, the `sort_two` method calls `std::list<int>::sort`, for which we already have a performance annotation.

From a quick analysis of the code, the expected running time of `sort_two` can be expressed as a linear function of the running times of the two `sort` methods, plus some linear complexity given by the size of the list `B`. We can be even more precise, and state the size of the list on which the two `sort` methods are going to be executed. In other words, $Y = a \times sort(l1) + b \times sort(l2) + c \times l2\_size$.

We can create two different performance annotations for the method above: in one (Figure 3.5) we consider the method alone, ignoring the fact that it executes other known methods.

```
1 sort_two(std::list<int> &l1, int l2_size).time {
2 features:
3  int s1 = *(l1->_M_impl._M_node._M_storage._M_storage);
4  int s2 = l2_size;
5
6 annotations:
7  Norm(3.35 * s1 + 3.55 * s2, 94594.71);
8 }
```

Figure 3.5. A performance annotation without references

Conversely, in the other performance annotation (Figure 3.7) we use references to the performance annotations of other (known) methods, such as `sort`.

```
1 sort_two(std::list<int> &l1, int l2_size).time {
2 features:
3  int s1 = *(l1->_M_impl._M_node._M_storage._M_storage);
4  int s2 = l2_size;
5
6 annotations:
7  Norm(0.58*s2
8       + @std::list<int>::sort[s:=s1]
9       + @std::list<int>::sort[s:=s2]
10      , 48911.38
11     );
12 }
```

Figure 3.6. A performance annotation with references

This performance annotation uses references to the performance annotation of the `std::list<int>::sort` to express the mean of the distribution.

The keyword `@` (lines 8–9) denotes a reference to another performance annotation. The performance metric that is considered is the same for all the annotations referenced in the performance annotation.

Moreover, performance annotations also document the values of the relevant features of the referenced performance annotations. In the example, the performance of the first `sort` method must be evaluated with a list size `s` equal to the value of `s1`, which represents the size of the list `l1` passed as input to `sort_two`. Similarly, the second reference to the performance annotation of `sort` must be evaluated with a list size of `s2`, which is equal to the value of the `l2_size` parameter.

With the performance annotations for `sort` and `sort_two`, it is sufficient to know the input values passed to `sort_two` to predict not only its performance, but also how the time is spent in each of the methods that are relevant for the global performance.

3.1.6 Analysis Heuristics

One important requirement for performance annotations is that they should be created automatically, possibly without any manual input from the performance analyst. Our way of creating performance annotations automatically is through statistical analysis, and to make the statistical analysis more robust in the presence of considerable noise in the measurements, we introduce some heuristics.

As we will see in Chapter 4, we apply two heuristics to support the statistical analysis in these cases. These heuristics filter the data points that are used by the analysis, therefore the results cannot be extended to the entire data set, and it becomes necessary to report this information in the performance annotation.

In the current implementation of the statistical analysis, we have two heuristics, which we will describe in details in Chapter 4: in one we *remove additive noise*, while in the other we consider only the *main trend* in the data points.

To inform the performance analyst that some results in the performance annotations are produced thanks to these heuristics, we introduce specific keywords in the language, just before the distributions: *R* indicates that the analysis removed the additive noise, while *M* indicates that the analysis considered only the main trend.

Here we have one example:

```
1 mysql_execute_command(THD *thd, bool first_level).time {
2 features:
3  int len = thd->m_query_string.len;
4
5 annotations:
6  R Norm(6630.19 + 0.86*len, 15.78);
7 }
```

Figure 3.7. A performance annotation with heuristics

3.1.7 Grammar

In Figure 3.8 we formalize the grammar of performance annotations using the extended BNF notation. Notice that the grammar for performance annotations must include the grammar for declarations and expressions of the implementation language. For simplicity we omit that portion of the grammar, and instead simply indicate the entry points of that grammar with the *method-signature* and *native-expression* symbols. We also limit the grammar to the type of distributions (Normal), regressions (constant, linear, quadratic, and $n \log n$), and more generally the mathematical expressions that are currently supported by our Freud prototype and that are displayed in this thesis. Nevertheless, the grammar can be easily extended to include more specific expressions.

```
⟨A⟩ ::= ⟨method-signature⟩.⟨metric⟩ {
        features:
        ⟨f⟩
        annotations:
        ⟨a⟩
        }
⟨metric⟩ ::= time | mem | hold | wait | pfaults | Pfaults
⟨cmp⟩ ::= > | >= | < | <= | == | !=
⟨feature⟩ ::= ((a–z) | (A–Z))((a–z) | (A–Z) | (0–9))*
⟨num⟩ ::= -?(0–9)+(.(0–9)+)?
⟨type⟩ ::= char | int | long | float | string
⟨f⟩ ::= ε | ⟨type⟩ ⟨feature⟩ = ⟨native-expression⟩ ; ⟨f⟩
⟨a⟩ ::= ε | ⟨scope⟩ ⟨heuristic⟩ ⟨distr⟩ ; ⟨a⟩
⟨scope⟩ ::= ε | [⟨cond⟩] | [⟨cond⟩] {⟨num⟩}
⟨cond⟩ ::= ⟨cond⟩ && ⟨cond⟩ | ⟨feature⟩ ⟨cmp⟩ ⟨num⟩
        | !⟨feature⟩ | ⟨feature⟩
⟨heuristic⟩ ::= ε | R | M
⟨distr⟩ ::= Norm(⟨expr⟩,⟨expr⟩)
⟨expr⟩ ::= ⟨num⟩ | ⟨feature⟩ | log(⟨feature⟩) | ⟨feature⟩^⟨num⟩
        | ⟨expr⟩ + ⟨expr⟩ | ⟨expr⟩ - ⟨expr⟩ | ⟨expr⟩ * ⟨expr⟩
```

Figure 3.8. Grammar of the performance annotations language

3.2 Uses

Performance annotations carry much information about the performance behavior of software methods that is valuable for different uses. For example, the precise documentation of the behaviors is really valuable to developers who wish to use documented software methods in their programs. Another potential use of performance annotations is to assert that a given method is performing as expected. Also, in some cases, performance annotations can be used to *predict* the performance of a software method when executing on systems and with parameters never observed before.

In the following sections we will expand on these use cases, discussing the advantages and limitations of performance annotations in each scenario.

3.2.1 Documentation and Prediction

The first and most immediate use of performance annotations is for documenting the expected performance behavior of software methods. When considering this use case, it is important to remind that performance annotations are built through a runtime observation of a running system. This means that performance annotations always describe what has actually been observed, and what can actually be extrapolated from that. This is true for all the types of models that performance annotations can express, whether they are based on regressions or clusters.

When performance annotations contain expressions that use features, they allow for prediction of the expected behavior of the method for any value that the specific feature can have in its domain. Take the sort example with the annotation of Figure 3.1: it is easy to predict how long the method is going to take to sort a list of 10M integers, given that the memory is enough to store the entire list without swapping, even if the biggest list that was observed was around 3M integers.

But, as we have seen, if the program runs out of memory the running time changes dramatically. Performance annotations would not be able to represent and predict this different behavior without first observing the different modalities, as it happens in the experiments with limited amounts of memory.

In the same way, notice how all the performance annotations presented in this chapter do not depend on system features that a performance analyst might expect to be important. For example, it is reasonable to expect that the running time of `std::list<int>::sort` also depends on the CPU speed of the computer that is running the program. That is indeed exactly what the performance annotation would say if the annotations were derived from enough observations showing variability in the *cpu_speed* feature.

However, all the performance annotations presented in this chapter were generated through experiments on a real system with a specific and fixed hardware/software combination. So, even if *cpu_clock_speed* were a relevant feature to predict the running time for `sort`, that feature would not be included in the performance annotations shown in this chapter. We emphasize that this is not a limitation of performance annotations or our tool Freud. Rather, it is a consequence of the fact that the tests (input) used to identify all the performance models were not exhaustive in covering the feature space. In the evaluation presented in Chapter 6, we show examples in which we account for system features.

Similarly, the boundaries in the scope conditions do not represent an absolute truth. The specific numbers in the performance annotations are those that

we observe in the logs on which the statistical analysis is performed. On the other hand, if the information reported by the performance annotation is correct, having more samples to analyze can only reduce the difference between the boundaries in the annotations and the reality.

In other words, performance annotations do not represent a ground truth but rather a model of what was observed. They do not necessarily describe the performance behavior of any software method *completely* for any relevant feature. The information contained in performance annotations is what can be extrapolated from what has been observed in real world experiments. Nevertheless, it is possible at least in principle to identify *all* the relevant features that affect performance, and therefore also make a performance annotation completely system/environment aware.

Still, the idea is that performance annotations be created with experiments that represent *typical* workloads and usage patterns for the software under analysis. So, rather than documenting all the features, it makes more sense to document all the features that really matter.

3.2.2 Assertions

Performance annotations can also be used as assertions. In other words, they can be used to compare the actual behavior of some software methods to their expected behaviors, in order to trigger specific actions when the observations deviate significantly from the expected behavior. For example, this could be useful to catch performance regressions when a new version of a software is released and deployed, or more specifically to kill or isolate a misbehaving process in a data center.

This use of performance annotations requires the deployment of our automatic instrumentation software on productions systems, posing a challenging constraint on the overhead introduced by the instrumentation. We discuss these aspects in Chapter 4.

3.2.3 Prediction

Performance annotations can be used as the basis for predicting the performance of methods in scenarios that were never observed. Indeed, with some knowledge about the code and how it uses methods for which we have performance annotations, it is possible to infer the performance of new methods. We do not cover this topic in this thesis, but it is an interesting direction for future work.

Chapter 4

Automatic Derivation of Performance Annotations

In Chapter 3 we introduced a language to describe the performance of software through probabilistic performance annotations. The language allows for the representation of the typical performance behaviors commonly observed in software. As discussed in Chapter 1, we believe that a key feature of our approach to performance analysis is the ability to derive performance annotations automatically. We now describe in detail how we accomplish that.

Our analysis is *dynamic:* we observe real software, compiled and configured, running on a real system with specific inputs, and we collect measurements about some performance metrics as well as values representing the state of the program and of the system. This approach has several advantages compared to static analysis: we can observe real performance metrics that also depend on the hardware; we can observe the average-case performance behavior defined by typical input that goes beyond the algorithmic complexity that might be visible from the code alone; and finally, we can analyze real-world complex software beyond what is currently possible with static analysis, which is often intractable for even small but real code bases.

The target of the analysis are uniquely identifiable software components. We choose to consider software components at the *method* granularity. We consider methods as defined in any imperative programming language: a named finite set of instructions that takes as input a set of parameters and state, and produces an output in the form of return values and/or changes to the state. Every method has a specific entry point, and one or more exit points.

This choice of granularity has several advantages. A method is a very convenient reference for programmers and performance analysts, both when specifying

which parts of the program to analyze and when reading performance annotations. It is also a convenient granularity for the instrumentation purpose, since it clearly identifies the entry and exit points for the instrumentation, and also offers a clear indication of the parameters and features visible from the scope of the method. Also, contracts for functionalities most often refer to complete methods, so it is convenient to have uniformity in the granularity of the documentation for performance and functionality.

Thus the performance analysis is defined by (1) the set of names of target methods, and (2) the set of performance metrics to analyze. In this thesis we do not study the problem of choosing the target methods and instead leave this responsibility to the performance analyst in each case. When running the instrumented program, all the methods that are selected for the analysis are modified or augmented for data collection, at the same time. Ideally *all* the methods of a program could be instrumented at the same time. While there are no constraints on the number of software methods that can be analyzed at the same time, the main practical limitation is the total overhead introduced by the instrumentation, which could be a problem in a production environment. So, the performance analyst should select only the most relevant methods for the problem at hand. Similarly, any automation applying our performance analysis should limit the number of methods that are instrumented during a single execution of the target program.

From a high level point of view, the analysis consists of two distinct parts (Figure 4.1): the *instrumentation*, which extracts information from a running program, and the *statistical analysis*, which processes that information to create performance annotations. The statistical analysis is generic, and can be applied to data coming from any instrumented program. The instrumentation is to some extent generic, but parts of it are also necessarily specific of the implementation language and its runtime environment.

In this chapter we give a reference description of the two parts. In essence, we describe the information collected by the the instrumentation, and how the statistical analysis processes such information. Later, in Chapter 5, we detail our implementation of these ideas and algorithms in the Freud tool.

The rest of the chapter is organized as follows: in Section 4.1 we describe the instrumentation part to show the general requirements that any instrumentation should meet to support this performance analysis. In Section 4.2 we describe the statistical analysis including the detailed algorithms and statistical tools we use to model performance data. In Section 4.3 we discuss some ideas to create performance annotations that refer to other performance annotations.

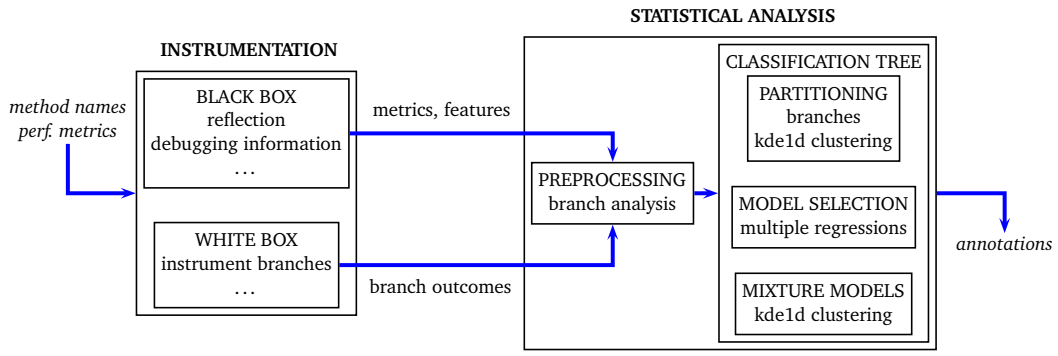


Figure 4.1. Overview of the derivation of performance annotations

4.1 Instrumentation

The first step involves analyzing and instrumenting the code of the target program to collect relevant data. For every execution of every target method, we collect one sample containing (1) a *unique id*, (2) the values of all the desired *performance metrics*, and (3) all the data that can be used to extract potentially relevant *features* of the input or of the state of the system. We also collect (4) the ids of all samples representing other target methods that have been executed on the same thread while the method was running. In other words, we log the *children* of the method.

All the information listed so far can be collected without ever looking at the code of the methods that we instrument. In other words, we can consider methods as black boxes to which we attach probes without any knowledge of their code. However, if the code is available, we can log additional information that helps in finding modalities in the performance behavior, and more generally that helps the statistical analysis produce good results. In particular, we can optionally log (5) the outcome of every executed conditional branch instructions (branch taken or not taken).

4.1.1 Data Collection

We now see more in details what the various pieces of information represent, and how they should be extracted by the instrumentation.

Ids

An id uniquely identifies a sample. The uniqueness of ids must be guaranteed across *all* the samples that are considered during a single statistical analysis, even if the samples come from different runs of the target program, possibly on different machines. In fact, ids are used for the composition analysis, which requires logging the descendants of a method, that is the methods actually called by the target method during its execution. For this purpose, the instrumentation must keep an internal stack to represent the execution status of the target program. When a new instrumented method is called, the instrumentation has a reference to the most recently called method that is also still active (that is, the top of the stack) so that it can add the new id to the list of children of the active method.

Metrics

Performance metrics are the dependent variable (Y) in performance annotations. Once we instrument a method, we always collect all the performance metrics chosen by the performance analyst. When measuring a performance metric for an instrumented method, we consider the cost incurred by the method itself, plus the cost incurred by all the descendants of the target method. In *gprof* terminology, we measure the *total* cost of the method, as opposed to the *self* cost. For example, when measuring the running time with start-stop timestamp pairs, we do not subtract the time taken by the children of the method.

Performance annotations account for metrics of different nature. Our analysis currently supports *running time*, *dynamic memory allocation*, *lock waiting time*, *lock holding time*, *number of minor page faults*, and *number of major page faults*.

In implementing the instrumentation for measurements, we consider two main objectives: minimum overhead and maximum accuracy, which crucially depends on the perturbation introduced by the instrumentation itself. We now describe the metrics and briefly discuss their implementation strategies and issues in different programming languages.

Running Time. Measuring running time is conceptually straightforward with start/stop timers or entry/exit timestamps, and many programming languages have timing features or at a minimum a clock features in their standard libraries. The most crucial parameter is the the clock resolution. Performance annotations use a *microseconds* timescale, which is well supported by modern architectures. A coarser scale for the measurement of time might be acceptable when measuring methods known to have orders of magnitude longer running times.

Memory Allocation. With memory allocation we measure the total amount of bytes of heap memory allocated by a method. We measure the total number of bytes allocated for all objects created during the execution of the method. Programming languages that use interpreters or virtual machines for the execution usually expose information about memory usage to the program through a dedicated API. Another approach is to instrument the low-level memory-management functions (e.g., `malloc` and `free`). The resolution that we use in performance annotations is 1byte.

Lock Holding/Waiting Time. Resources contention metrics are usually crucial in the performance behaviors of multi-threaded applications. Lock-holding time of a thread executing a method is the total duration of all the time intervals in which the thread holds a lock on a shared resource. Lock waiting time, instead, is the total time that the thread spends waiting for the acquisition of a shared resource. These metrics represent time intervals, and are measured in microseconds. This means that the same measurement techniques for running time (i.e. stop-watch timers or timestamps) apply. The difference is that the *start* and *stop* events for the time intervals are associated with the lock acquisition and release events, respectively, as opposed to entry and exit for a method.

Exactly which events one must instrument, depends on the programming language and the resources contention primitives. Typically, a lock hold is delimited by the successful completion of a lock acquisition request (e.g., *pthread_mutex_lock*) and the successful completion of a lock release request (e.g., *pthread_mutex_unlock*). Similarly, a lock wait is typically delimited by a call to a lock acquisition method (e.g., *pthread_mutex_lock*) and the successful termination of the same method.

In the end, locks-related metrics are time measurements, and therefore are subject to the same limitations regarding time resolution as running time.

Minor/Major Page Faults. These metrics are indicative of the interaction of the program with the memory subsystem of the operating system. Page faults are common to all operating systems that use virtual memory. We use the specific Linux definition to distinguish between minor and major page faults. This metric comes directly from the operating system, and so it is not language dependent. On Linux, thread-specific information about page faults can be found in the *procfs* filesystem, specifically in the file /proc/*pid*/task/*tid*/stat for process *pid* and thread *tid*. Page faults are measured with pure numbers, representing counters for such events.

Regardless of the metric, every performance measurement we used so far results in a scalar numeric value. While this is the list of performance metrics that we currently support in our Freud prototype, new performance metrics can be added with minimum effort. We will give a complete description of our approach to C/C++ instrumentation in Chapter 5.

Features

Features are the independent variables (X) in performance annotations. Our analysis selects the relevant features for each target method automatically. The selection is performed during the statistical analysis stage. Since the instrumentation alone does not know in advance which variables correlate with the performance behavior, the instrumentation collects *all* the information that is visible to the instrumented method within the program, as well as system features.

This process is therefore a bit more involved than collecting metrics. On the instrumentation side, we want to find all the values of local, global, or system variables that might have an effect on performance. We also want to use heuristics to extract quantities computed from program variables.

The first and most directly relevant variables that the instrumentation looks for are the parameters to the target method, as expressed in the program code. We collect both local and global variables that are visible from the scope of the method. We collect the values of scalar variables and explore structured (aggregate) variables to extract information about their fields. Ultimately, every structured type is decomposed into a set of scalar values, one for each field of the structured type. Since we are interested in *numeric* features, we represent Boolean values as 0 or 1, and strings with their length. We also follow references to objects and explore those objects to collect more features. We limit the depth of this exploration that might otherwise become arbitrarily complex. The depth is defined as the distance from the global variable or parameter that is directly accessed by the method.

In addition to the variables in the program, we also collect information that is present in the system but not necessarily in the instrumented program, such as the clock frequency of the processor, the number of active processes on the machine, and other system performance indicators. Since by definition this information resides outside of the instrumented program, typically it takes more time to access this information compared to the variables in the program.

A modern operating system provides a large amount of performance-related data. For example, *procfs* and *sysfs* on Linux expose hundreds of system features and many more for all the processes and devices. However, most of those fea-

tures are likely to have no relation with the performance of the system under analysis. Our approach is therefore to offer a restricted set of generally useful features, and then let the performance analyst select the most interesting features for the given application. For example, a generally useful feature is the clock frequency of the processor, which, unsurprisingly, we found to relate quite often with performance.

Finally, we heuristically discover *derived* features that are likely to affect performance. These are values computed from one or more values of state variables or parameters. For example, if we find that an object contains a pair of variables named *begin* and *end*, *first* and *last*, or *start* and *finish*, we log a new, derived feature computed as the difference between the two variables. The rationale, confirmed by experience, is that this feature might represent a size, and might therefore correlate well with performance.

The specific way to discover program variables automatically depends on the programming language of the target program. We will describe Freud's implementation that takes advantage of the debugging information that is shipped with C/C++ program binaries in Chapter 5. *Reflection* can also be used in other programming languages, such as Java, Python, or PHP. Reflection allows programs to access information about their own structure, and modify their own code. For example, it is possible to create proxies for the target methods. These proxies would examine and log the parameters they are given, prepare the collection of performance metrics, and execute the original method. Once the execution of the original method is finished, the control goes back to the proxy method that finalizes the collection of performance metrics, before returning.

Branch Outcomes

The data collection presented up to this point does not need access to the source code of the target methods. We can extract the list of parameters that are used by the methods from their interface definitions, such as their signatures. Therefore we can think of a target method as a black box to which we attach probes, without opening the box.

However, the code can of course provide useful information. In particular, software often exhibits different modalities, meaning different types of behaviors, and such modalities might be determined by the code selecting one of a number of distinct algorithms. We therefore collect additional information that relates to the code of the target method in the hope of finding high-level correlations that would better explain different performance modalities based on input features.

More precisely, we want to distinguish different execution paths in the target method. These execution paths are defined by the result of branch instructions. Each branch instruction in the code is a potential fork point where the algorithm might take a different path. And conversely, each execution path is uniquely identified by the sequence of the executed branch instructions and their outcomes.

In the most general case, the behavior might be determined by many branch instructions defining a complex class of execution paths, and each branch instruction might depend on previous computations. However, in some significant cases, the execution of even a single initial branch instruction executed only once might allow us to distinguish between, say, a slow path and a fast path. And furthermore that initial branch instruction might be directly tied to an input feature. In such cases, we could use the condition on the input feature expressed by that branch instruction as a scope condition to distinguish the slow modality from the fast modality.

In practice, there are different approaches to analyzing branch information. We could statically analyze the code of the program to parse the exact condition that is evaluated by these switches. Another approach, which only relies on dynamic analysis, is to log the outcome of such switches, to see whether they evaluated to true of false during a specific execution of the method. Again, the way one can concretely implement such instrumentation depends on the programming language. In the case of C/C++, we instrument every conditional branch instruction to log whether the branch is *taken* or *not taken*, as we will show in Chapter 5.

While not strictly necessary for our branch analysis, it might also be useful to record all executions and outcomes of the branch instructions, since they could be used as features during the performance analysis.

4.1.2 Producing Output

In summary, the instrumentation produces a log of records. Each record represents a single execution of a single method in the following format:

$$id, y_1, \ldots, y_m, x_1, \ldots, x_n, b_{1,1}, b_{1,2}, \ldots, \#, b_{2,1}, b_{2,2}, \ldots, \#, cid_1, cid_2, \ldots$$

where *id* is the unique id of the sample, 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, $b_{1,1}, b_{1,2}, \ldots$, followed by the outcomes of the second branch instruction, $b_{2,1}, b_{2,2}, \ldots$, and so on. Finally, $cid_1, cid_2, \ldots$ represent the ids of the children of the method.

The log entries are minimally structured and consist primarily of sequences of numbers, and therefore can be easily encoded in a compact binary format for efficiency.

Also, since we want to be able to instrument long running services, like a database server, a proxy, or an HTTP server, the instrumentation must be able to export the information while the target program is running, without blocking or perturbing its execution. For example, our Freud implementation uses a double buffering mechanism and a dedicated output thread to meet this requirement.

4.1.3 Perturbation and Overhead

For any performance monitoring tool, instrumentation overhead and perturbation can be a concern. First, it is important to distinguish between these two concepts.

Overhead is the increase in the cost of the execution of the program under analysis. Such increase might impact on the resources needed to execute the program, on the response time taken to serve a request, or on the concurrent number of requests that can be served. While the overhead might affect the ability to instrument software running on a production system with tight performance constraints, it does not affect the correctness of the measurements of performance metrics. For example this means that even if the instrumented program takes longer to execute, because it also needs to perform the instrumentation tasks, the measurements produced by the instrumentation report performance metrics as they would be in the original program, without any instrumentation or measurement device installed.

The overhead originates from the data collection activities that the instrumentation needs to perform. Such activities use CPU time and memory. Thus, any instrumentation incurs overhead, and should be designed to minimize it.

One common way of reducing overhead is to use sampling, which means to avoid logging all the data for *every* execution of the instrumented methods, and rather perform this expensive process only for a fraction of the executions. When using sampling, it is important to select the executions that are going to be logged so that they represent without bias the behavior of all the executions. For example, in Freud we use reservoir sampling to achieve this result.

Conversely, *perturbation* is the modification of the behavior of the method that is being observed as a consequence of the observation itself. Such modification can increase or reduce the cost of the execution of the instrumented method. Prior work has shown that even innocuous changes to code layout, similar to that caused by code instrumentation, can have considerable effects on

execution times [32]. It is extremely important to be aware of the risk of perturbation when designing the instrumentation, as it may threaten the validity of the measurements.

One of the first causes of perturbation might come from the instrumentation code itself. It is very important to avoid attributing the cost of the instrumentation to the instrumented method. This problem becomes more relevant when instrumenting recursive or nested methods. In these cases it is not possible to completely avoid the problem, because even the bookkeeping actions would need to be measured. Also, it is important to consider the effects of instrumentation in multi-threaded programs where there is a lot of synchronization and contention between threads. For example, instrumenting the lock-release operation on one thread would probably impact the running time of another thread that is waiting for the lock to be released.

In Section 5.4.2 we describe our efforts to mitigate the instrumentation overhead and minimize the perturbation for our tool Freud, and in Section 5.4 we experimentally validate the accuracy and robustness of our measurements.

4.2 Statistical Analysis

The second phase of our analysis takes the output logs produced by by the instrumentation, and outputs probabilistic performance annotations in the form of text and graphs. The statistical analysis is performed offline, so it does not need to meet strict requirements on running time and resources consumption.

The statistical analysis is generic, and we apply it to every target component for every target performance metric, with minor differences in the regression analysis for time based metrics. The output of the statistical analysis is one performance annotation for each method that is analyzed, for each performance metric.

As shown in Figure 4.1 (page 41), the statistical analysis consists of two phases. In the first phase we pre-process the data to extract branch information for the purpose of partitioning the records; in the second phase we build a *classification tree* to infer a set of models from the performance data with the help of such branch information. A performance annotation is the direct translation of the classification tree into our performance annotation language.

The classification tree defines a hierarchical partitioning of the performance records. Each node represents a part of the records of the parent node defined by a scope condition. The scope condition is a Boolean expression that contains at least one feature and that is true for all the records in that part. The records

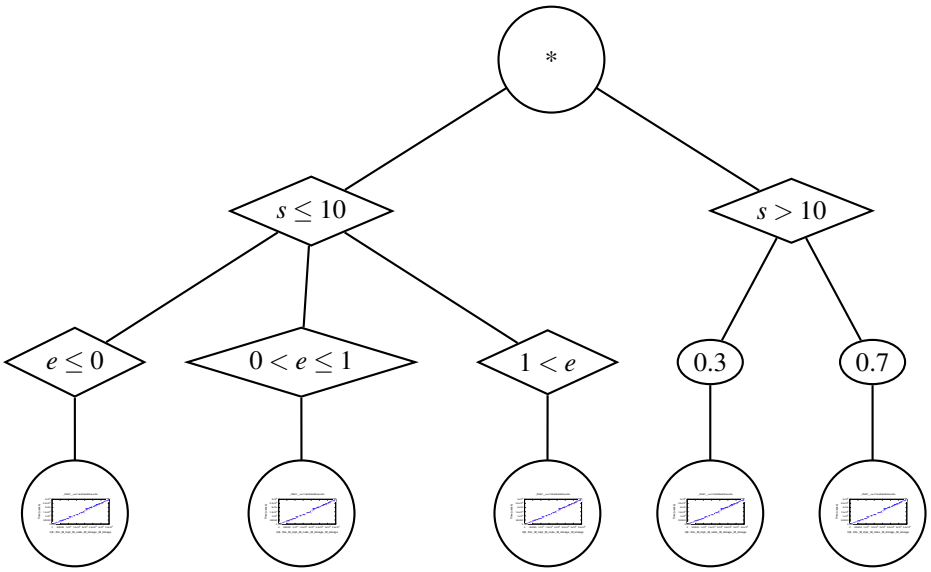


Figure 4.2. A classification tree with mixture models

in the leaves in the classification tree are modeled with distributions that depend on zero or more features. We use *regression analysis* to compute parametric distributions that correlate the relevant features to the performance metrics.

In the creation of the tree, we keep partitioning nodes until we either (1) find a distribution based on at least one feature that models the data (records) in the node well enough, or (2) we do not have any new partitioning conditions for that node. When can not formulate a valid regression model for the records in a node and we do not find a scope condition to further partition, we formulate *mixture models*. A mixture model is a combination of two or more sub-models each associated with an occurrence probability rather than a scope condition. Figure 4.2 shows an example of classification tree that uses mixture models.

The root node represents all the samples collected by the instrumentation. The complete set is split into two parts based on the value of the feature s: one part contains the records for which $s \leq 10$, the other contains the remaining records with $s > 10$. These two partitions are further split before valid models can be found. If $s \leq 10$ we further split based on the value of the enumeration variable e, while if $s > 10$ we rely on clustering to partition the data (mixture model). We find two clusters: one contains 30% of the samples, the other contains 70% of the samples. The leaves of the tree represent different models that we create to represent the data contained in those leaves. All these models contain distributions that correlate the performance metric under analysis with zero

or more features.

The partitioning conditions based on feature values correspond to scopes in our annotation language, while mixture models correspond to probability specifications. The models, i.e. the leaves of the tree, correspond to the distributions.

We now describe in greater detail the algorithms that we apply in each phase of the statistical analysis. In particular, we describe (a) how we pre-process the data to create partition candidates; (b) how we build the classification tree, defining scope conditions and producing regression models; and (c) how we use clustering to model data for which we could not find any good regression model.

4.2.1 Data Pre-processing

The goal of the pre-processing stage is to compute potential partitions of the data that can be used by the classification tree stage. To produce such information, we use information extracted or inferred from the code of the target methods.

This information allows our analysis to explain different performance modalities that stem from the algorithm implemented by the method, as opposed to the dynamic interaction of software with the rest of the system. While this is effective, our analysis also uses another strategy to partition the data, that is solely based on the performance metrics and feature values. We will describe this other strategy later in the next section.

There are two sources of information used by the pre-processing stage: *enumeration variables*, and *branch outcomes*.

Enumeration variables represent a promising source for finding different code paths taken by an algorithm. It is a common practice in writing computer programs to evaluate enumeration variables in switches that make the algorithm take different paths based on the value of the variable. So, we use every observed unique value of the enumeration variable as a potential partition. In other words, we create partition candidates in which each partition has all the samples collected by the instrumentation in which the enumeration variables assumes a specific value.

Partitioning based on branch outcomes is more involved, and is based on Algorithm 1. The algorithm is executed for each branch instruction of the method, and for each feature. We consider only branch instructions that are executed only *once* during the execution of the method, because those are the branches that likely represent a switch between two high-level distinct behavior modes such as fast and slow paths.

Algorithm 1 Branch analysis

Require: $O = \{t_1, f_1\}, \ldots, \{t_n, f_n\}$.
Require: f_i is the value of a specific feature in sample i.
Require: t_i is the outcome of a specific branch in sample i.
Ensure: Output: $P = \{p\}$.

$P \leftarrow \emptyset$
$max\_t \leftarrow INT\_MIN$
$max\_n \leftarrow INT\_MIN$
$min\_t \leftarrow INT\_MAX$
$min\_n \leftarrow INT\_MAX$
for $o \in O$ **do**
  if $t_i == true$ **then**
    $max\_t = max(max\_t, f_i)$
    $min\_t = min(min\_t, f_i)$
  else
    $max\_n = max(max\_n, f_i)$
    $min\_n = min(min\_n, f_i)$
  end if
end for

if $max\_n < min\_t$ **then**
  if $min\_n == max\_n$ **then**
    $P \leftarrow min\_n$
  else if $min\_t == max\_t$ **then**
    $P \leftarrow min\_t$
  else
    $P \leftarrow max\_n$
  end if
else if $max\_t < min\_n$ **then**
  if $min\_t == max\_t$ **then**
    $P \leftarrow max\_t$
  else if $min\_n == max\_n$ **then**
    $P \leftarrow max\_n$
  else
    $P \leftarrow max\_t$
  end if
else
  $P \leftarrow \emptyset$
end if

We look for conditions C that partition S into two parts S_1 and S_2, such that C is true for all measurements in S_1 and false for all those in S_2. The condition C checks whether the values of one specific feature in a given sample is bigger or smaller-or-equal than a reference pivot value.

The algorithm finds these pivot values, that represent different execution paths in the target method. To do so, we compute the minimum and maximum values of the selected feature for the executions in which the target branch evaluated to *true*, and for those in which the target branch evaluated to *false* (lines 6–14).

We then check whether the ranges of values of the feature for the two classes of samples do not overlap (lines 16–34). If that is the case, we conclude that the feature defines different execution paths, and therefore we use the boundary values of the feature as a pivotal point.

Notice that, since we are performing a statistical analysis on a finite number of samples, the boundary value that we find might not be exactly the value defined in the code. This happens when we have no samples in which the feature has the precise value specified in the condition in the code of the method. In other words, the boundary value that we use to produce partition candidates is the closest to the real one that we have observed through the instrumentation. This means that more samples are likely to reduce the difference between the computed pivotal point and the real one, and that eventually the difference will be zero.

4.2.2 Classification Tree

The goal of the second part of the statistical analysis is to build a classification tree that can be directly translated into a performance annotation. The inputs are all the samples collected by the instrumentation, with performance metrics and feature values, and the partition candidates produced by the pre-processing stage.

Our analysis keeps partitioning the data on a branch of the tree until either we find a good regression model that contains at least one feature, or we cannot split anymore the set of data points with scope conditions. If part of the records can not be modeled with a simple and accurate enough regression, and if no condition can be found to further split that part, then we proceed with the clustering of the records of that part based on the metric values (y, as opposed to $x_1, x_2, \ldots$).

After clustering, whether it is for a specific partition or for the entire data set, we still try to identify, for each cluster, a feature-dependent model as well as a defining condition based on the values of the features. If that also fails, we resort

to a simpler probabilistic characterization of each cluster, and we try to model the data of each cluster first with a regression and then with an input-independent model.

To recap, we have two types of scope conditions: derived from *branches*, and derived from *clusters*. The two types are identical in the performance annotations language, but they describe performance modalities that stem from different sources.

Partitioning based on *branches* are derived from information inferred from the code of the method. Therefore, these partitions help distinguishing different performance modalities as a consequence of the algorithm executed by the method. Also, assuming that the instrumentation collected enough samples, these partitions are correct, in the sense that they represent the exact behavior of the method. In other words, the condition on the features that define the scope are precise and correct.

On the other hand, partitions inferred after *clustering* are created through the performance measurements. This means that the precision, that is the difference between the values of the features specified in the scope conditions and the thresholds that distinguish different behaviors in the reality, is directly related to the precision of the clustering. At the same time, the big advantage of these scope conditions is that they can represent the performance modalities that are not given by the algorithm implemented in the target method, but rather by the dynamics of the interaction of the program with the rest of the system.

We now describe in detail the components of this classification tree analysis: data *partitioning*, computing *regression models*, and we *clustering* the data.

Building the Tree

We try to formulate a classification tree using Algorithm 2. We run this algorithm for every target method, for every performance metric.

The input to Algorithm 2 is a set of log records R where each record r_i represents a run of the target method and consists of a performance metric y_i, and a set of features values X. Additionally, the Algorithm takes as input a set of partition candidates C, produced in the pre-processing step. Each partition candidate c_i consists of a set of conditions $\{c_{i,1}, c_{i,2}, \ldots\}$, one for every partition candidate. Each condition $c_{i,j}$ specifies a feature name and two boundary values for such feature. The condition evaluates to $true$ for one sample only if the value of the feature for that sample falls within the range delimited by the two boundaries (strictly bigger for the lower boundary, smaller for the higher boundary).

The output is a classification tree $T = [S_1]h_1; [S_2]h_2; \ldots$ consisting of a set of

Algorithm 2 Classification tree analysis

Require: $R = \{r_1, r_2, \ldots\}$.
Require: $C = \{c_1, c_2, \ldots\}$.
Ensure: Output: $T = \{(S_1, h_1), \ldots\}$.

1: $T \leftarrow \emptyset$
2: $Q \leftarrow \{(true, R)\}$
3: **while** Q is not empty **do**
4: $(S, P) \leftarrow Q.pop()$
5: $h(X) \leftarrow regression(P, X)$ for some feature-set X
6: **if** $h(X)$ is a good model **then**
7: $T \leftarrow T \cup \{(S, h(X))\}$
8: **else if** P can be split into $\{P_1, \ldots, P_n\}$ on some condition c_i **then**
9: $Q \leftarrow Q \cup \{(S \wedge c_{i,1}, P_1), \ldots, (S \wedge c_{i,n}, P_n)\}$
10: **else**
11: {formulate a mixture model h on P with clustering}
12: $h \leftarrow$ output of Algorithm 3 on R
13: $T \leftarrow T \cup \{(S, h)\}$
14: **end if**
15: **end while**

expressions h_i, representing the models, each associated with a scope condition S_i.

Algorithm 2 iteratively processes various sets of input records until it covers the whole input set R with performance models. The first iteration starts by considering the whole set $S = R$. The algorithm looks for a good model $h(X)$ that predicts S (line 5). A model is considered good if the R^2 goodness-of-fit, which measures how well the model represents the data, is above a chosen threshold. If no good model is found (line 8) the algorithm looks for a condition c' that partitions P into two or more sets of measurements $P_1, \ldots, P_n$—such that $c'_{j,i}$ is always true in P_i and always false in all the other sets—and then proceeds recursively to look for good models for all the P_i.

If this hierarchical partitioning fails to yield any model for the whole input set—because no partition P can be modeled well by any regression $h(X)$ and no new partitions can be created based on known conditions—then the algorithm fails to return a valid regression tree, and the analysis proceeds with clustering the entire data set R.

Regression Model

Now we describe how we formulate regression models $h(X)$. Given a set of measurements of a metric y and corresponding features $x_1, x_2, \ldots$, we formulate multiple-regression models with interaction terms $h(x_1, x_2, \ldots, x_1x_2, \ldots)$ with multiple independent variables. We try to fit models of different classes to the data, and evaluate the quality of the resulting model. We try four classes of models of increasing order: first constant, then x (linear), then $x \log x$, then x^2 (quadratic). Conceptually, it is of course possible and indeed easy to consider other classes of models. However, ultimately we are interested in analyzing *systems,* and complexities above (quadratic) are rare in systems software.

For each class, we select *one* model. To choose the model for a class, we iteratively compute multiple regressions, progressively filtering the features that are less relevant for the regression. The process stops when (1) the R^2 value for the regression is too low (failure), (2) there are no features remaining in the regression (failure), or (3) all the remaining features of the regression obtain a good (low) p-score (success). We can stop the process and return a failure as soon as condition (1) is met because removing potential features from the regression can only decrease the R^2 value, since we are removing parameters—degrees of freedom—from the model. We stop on condition (2) because if we can only find a good correlation without any specific feature having a low p-score, then we can conclude that the regression is not statistically solid.

At the end of this process, we have at most one regression model for each class. Every regression has a R^2 value above a fixed threshold, and all the features having a p-score lower than a fixed threshold. Since we want to avoid overfitting and choosing models with too many parameters, we use the Bayesian Information Criterion [37] (BIC) to compare the relative quality of the candidate models. We compute the BIC value for the selected model of each class, and choose the model with the lowest BIC value.

To reduce the complexity of the regression analysis with numerous features, we first eliminate the features whose value does not have any variability in the samples. Then, we group strongly correlated features into equivalence classes and use a single representative from each class in the regression analysis.

For time-related metrics, and in general for performance metrics that are subject to noise in the measurements, we use additional heuristics to try to find regression models before resorting to clustering.

We explore data-assimilation heuristics to find meaningful regressions between metrics and features. Specifically, we *select* input measurements and find predictive models only on the selected data. We use two heuristics: we remove

additive and strictly positive noise, and we select the dominant cluster.

We apply noise removal when the performance metric we want to analyze is subject to only additive (positive) errors. One such metric is time (duration). The goal is to filter out additive noise from the measurements. Given a feature, we keep only the measurements that exhibit the lowest value for the performance metric, for a specific value of the feature that, in case of success, returns the lower boundary for the performance of the method, correlated to one feature.

Another heuristic that helps finding correlations when the performance metric is subject to considerable noise is the selection of the dominant cluster. Essentially, this is a special case of mixture modeling when the probabilities associated with all but a particular cluster of data points are very small. For these cases, we treat the outliers as noise, and do not use them in the regression. This is a simple form of robustness analysis. The goal is to keep only the samples that are most representative of the expected, average behavior of the method. Given a feature, we group the records by feature value. We then run a clustering algorithm within the groups of measurements. Then, for each feature value, we select the cluster with the most measurements as the representative for that feature value and drop all other measurements.

Since these heuristics filter the data being analyzed, and might produce biased results as a consequence, it is important to state that the heuristic has been applied in the performance annotation produced. To this extent, we have specific keywords in our language to indicate the use of heuristics in the analysis.

Clustering and Mixture Model

The clustering analysis (Algorithm 3) tries to partition the input set $R = \{\dots r_i \dots\}$ based on the values y_i of the performance metric rather than based on any condition on the input features. The output is either a classification tree $T = [S_1]h_1; [S_2]h_2; \dots$ where each model h_i is associated with a scope condition S_i, or a mixture model $T = \{p_1\}h_1; \{p_2\}h_2; \dots$ where each model h_i, is associated with a probability p_i.

The first step of the algorithm is to cluster the input set of records based on the values of the performance metrics y of the records. Since we are clustering 1-dimensional data (we consider only one performance metric at a time), we use Kernel Density Estimator clustering [6]. KDE clustering finds minima and maxima in the density distribution for the performance metric. We then use the minima as boundaries between clusters, and maxima as centroids.

For each cluster, Algorithm 3 first tries to find a good model and also a defining scope condition for the data points in the cluster, which then becomes part of

Algorithm 3 Clustering and mixture model

Require: $R = \{r_1, r_2, \dots\}$.
Ensure: $A = \{(p_1, h_1), \dots\}$ or $A = \{(S_1, h_1), \dots\}$.

$A \leftarrow \emptyset$
$\{R_1, R_2, \dots\} \leftarrow kde1d(R)$, based on the metric values y
for all clusters R_i **do**
  formulate model $h_i(X)$ on R_i for some feature set X
  if $h_i(X)$ is a good model **then**
    if there exists a condition S_i that defines R_i **then**
      $A \leftarrow A \cup \{(S_i, h_i(X))\}$
    end if
  end if
end for
if $A = \emptyset$ **then**
  for all clusters R_i **do**
    $p_i \leftarrow |R_i|/|R|$
    formulate model $h_i(X)$ on R_i for some feature-set X
    if $h_i(X)$ is a good model **then**
      $A \leftarrow A \cup \{(p_i, h_i(X))\}$
    else
      formulate input-independent model h_i on R_i
      $A \leftarrow A \cup \{(p_i, h_i)\}$
    end if
  end for
end if

the output annotation (lines 3–10). In order to find cluster-defining scope conditions, the algorithm searches for non overlapping ranges of values for some feature. For example, if we find two clusters R_1 and R_2, and we find that all the records in the cluster R_1 have values smaller or equal to 10 for the feature f, while all the records in the cluster R_2 have a value bigger than 10 for the same feature f, then use f as a cluster defining feature, with 10 as pivotal value. In this case, we would drop the probabilities associated with the clusters, and use $f <= 10$ and $f > 10$ as scope conditions.

With this type of partitioning, we can find partitions that describe modalities in the performance behavior that are not generated by specific execution paths in the code of target methods.

If this first phase does not yield a condition and a good model for even a single cluster, the algorithm continues with a search that includes input-independent models and that associates models with probabilities (lines 12–21). When we cannot formulate any regression model for the data points in a cluster, we formulate models using normal variables whose mean is the centroid of the cluster, and whose standard deviation is computed from the distances of all the data points in the cluster from the centroid.

4.3 Considerations on Composition

In this Section we discuss some ideas to reason about composition. With the word composition we refer to the use of performance annotations for different methods in the definition of another performance annotations, as we have shown in Chapter 1. In other words, composition means describing how the performance behavior of different methods which have some kind of inter-dependencies relate to each other, for one specific performance metric. We will not formally discuss the problem in this thesis, as we leave this topic for future work. Still, here are some considerations.

While in the end the resulting performance annotations would look identical, there are multiple ways to reason about composition. More precisely, there are multiple ways to infer the properties of composition.

On the one hand, composition might refer to the process of finding how the performance behavior of inter-dependent methods correlate to each other, when we have instrumented and observed all the methods involved. This means that we find correlations between the performance of the different methods that we observe.

On the other hand, with composition we might refer to the process of pre-

dicting the performance behavior of a method that we have never instrumented or observed, using the performance annotations of other methods. This kind of composition requires an execution model for the method for which we want to predict the performance. Either we ask the user to provide such model, or we run static analysis. It does not make sense to think of any dynamic analysis, since that would bring us back to the "easy thing".

4.4 Threats to Validity

While our analysis is effective in finding complex performance pattern in real world complex software, as we will show in Chapter 6, there are several risks to consider with the results and the information contained in our probabilistic performance annotations.

For example, instrumentation inevitably introduces overhead on the running time of the program being observed, and perturbation in the data collected. We will discuss and quantify these risks in Section 5.4.2.

Other types of risks are related to the *correctness* and the *completeness* of our results.

Correctness. The risk related to correctness is that of finding false correlations. Indeed with our approach there is the risk of finding false correlations between a performance metric and a specific feature. With false correlation we refer to casual correlations, that are present in the samples that we observe with the statistical analysis, but do not represent any real pattern caused by the program being analyzed. For example the running time of a `std::list<int>::sort()` function could be found to be correlated with the value of the first element of the list, because just by accident such correlation was indeed observed in a given set of samples.

While false correlations could admittedly deceive the performance analyst reading performance annotations, the risk becomes indefinitely limited with a growing number of samples that are collected. In other words, if the correlation is really accidental, then collecting more samples is going to uncover the error in the results. At the same time, if the correlation remains valid when more and more samples are collected, then it becomes more and more likely that the correlation is indeed existing, even if not immediately visible or intuitive from the code of the program.

Completeness. The risk related to completeness is that of producing performance annotations with not enough information to describe completely the performance behavior of software components. Performance annotations that do not account for the effect of some features, that in reality *do* have an effect on performance, cannot be used to make accurate predictions of the behavior in different environments and with different workloads.

As we said in Section 3.2.1, our performance annotations do not represent a ground truth but rather a model of what was observed. The information contained in performance annotations is what can be extrapolated from what has been observed in real world experiments. As a concrete example, if we never run experiments with different list sizes when analyzing `std::list<int>::sort()`, the resulting performance annotation will not have the size of the list as a relevant feature.

Also, our analysis cannot predict multi-modal performance behaviors without observing them. For example, Freud would not have been able to predict the big jump in the running time in the sort experiment with limited memory (Figure 1.1), without observing it.

In order to produce richer performance annotations we need to collect observations in which we have (1) a good variability for the values of the relevant features, and (2) information about all the potential relevant features.

As we said in Section 3.2.1, rather than documenting all the features, it makes more sense to document all the features that really matter. This means that we consider the variability of the features that can be observed in the typical workload for a system to be enough to represent the performance behaviors of interest.

While our approach for automatic feature exploration minimizes the risk of ignoring relevant features within the memory of the program, we rely on the performance analyst to identify relevant features in the system that is executing the program under analysis.

In our current implementation of Freud we consider the CPU clock speed as a system feature that is potentially relevant for the performance of software, but others exist. For example, workloads different from the program under analysis, running on the same machine, are likely to have an effect on the performance of the program under analysis. Other examples can be the architecture or the size of the cache memory of the processor executing the program, or the network bandwidth and latency.

While we do not consider all these feature candidates in the current implementation of the analysis, we argue that these features can be accounted for without any fundamental change in our analysis. For example, we can imagine

that Freud would compute a linear regression to correlate the running time of a method `download(std::string url)` with the feature *network_bandwidth*, if such feature was read from the system (in Linux, from *sysfs*).

Similarly, a different CPU architecture, or maybe the usage of a dedicated processor or of a GPU instead of the main CPU, could be encoded with an enumeration feature that would be used to partition the data points. Freud would be able to find different performance profiles, according to the hardware used to execute parts of the method.

Chapter 5

Freud

In this chapter we describe Freud, the tool that we implemented to analyze the performance of software systems to automatically derive performance annotations. Freud is an open source tool that is designed to operate with *any* C/C++ program, that is, it does not target any specific program. Freud uses Intel's *Pin* instrumentation tool to dynamically recompile binaries to be able to extract performance data at run time. Freud then performs a statistical analysis on the collected data using the R statistical package.

Freud was developed for Linux. In particular, Freud processes *64bit ELF* binaries containing *DWARF* debugging information, and uses the *procfs* to collect information about the state of the system. Freud also expects that the instrumented system uses the System V AMD64 application binary interface (ABI). These requirements correspond to the typical default configuration on practically all modern Linux 64-bit systems with the default GCC compiler. However, the choice of the Linux platform was not motivated by any unique feature of Linux systems. In fact, we also believe that Freud can be easily extended to handle more binary formats and operating systems thanks to its modular design.

Figure 5.1 shows a high level model of Freud, which is composed of three separate modules: *freud-dwarf*, *freud-pin*, and *freud-stats*. Dwarf and pin are the modules that constitute the instrumentation part of our analysis, while stats implements the statistical analysis. Each module results in a separate program, and the interaction between the different programs happens through physical files on the filesystem.

The typical usage pattern is as follows:

1. The performance analyst runs *freud-dwarf*, passing as input the names of the methods to instrument, and the path to the binary of the target program. This operation produces two files as output: *table.txt*, and *feature-*

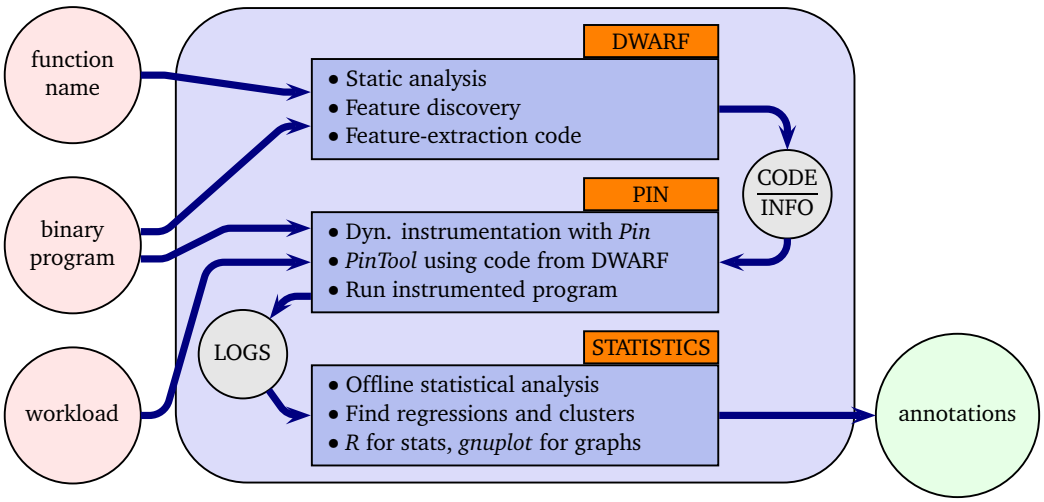


Figure 5.1. High-level architecture of Freud

processing.cc.

2. The performance analyst compiles *freud-pin*, which requires the *feature-processing.cc* file produced during the first step. This will result in a compiled PinTool. The performance analyst then runs the PinTool passing as input the binary of the program to analyze, the *table.txt* file, and the set of performance metrics to collect. The PinTool executes the target program and produces output logs while the program is running. If necessary, the performance analyst must supply an appropriate workload for the target program.

3. Finally, the performance analyst executes *freud-stats*, passing as input the logs produced at the previous step. Optional parameters are available to override the default threshold values used during the analysis, such as the minimum R^2 for the regressions. The tool produces performance annotations as output: *txt* files for the textual representation, and *eps* files for the graphs.

In the rest of this chapter we describe in details the three modules of Freud: the static analysis of the debugging information (*freud-dwarf*) in Section 5.1, the instrumentation (*freud-pin*) in Section 5.2, and the statistical analysis (*freud-stats*) in Section 5.3. In Section 5.4 we describe our validation of Freud, to assess that it produces corrects results with methods with known performance behaviors. Finally in Section 5.4.2 we discuss, with the help of some real world exam-

ples, the overhead and the perturbation introduced by Freud's instrumentation.

5.1 freud-dwarf

As shown in Figure 5.1, *freud-dwarf* performs a *static analysis* on the binary of the target program. The binary must contain DWARF debugging symbols as generated, for example, by the *gcc* compiler with the *-g* flag. The goal of the static analysis is to discover all the potential features within the program that might affect the performance of all the methods that the performance analyst wants to analyze. As we said in the previous chapter, this means all the local and global variables that can be accessed by the selected methods. For each feature candidate, freud-dwarf produces C code to extract the values of such feature at runtime. This code is written in *feature-processing.cc*, which is to be later compiled and executed in *freud-pin*.

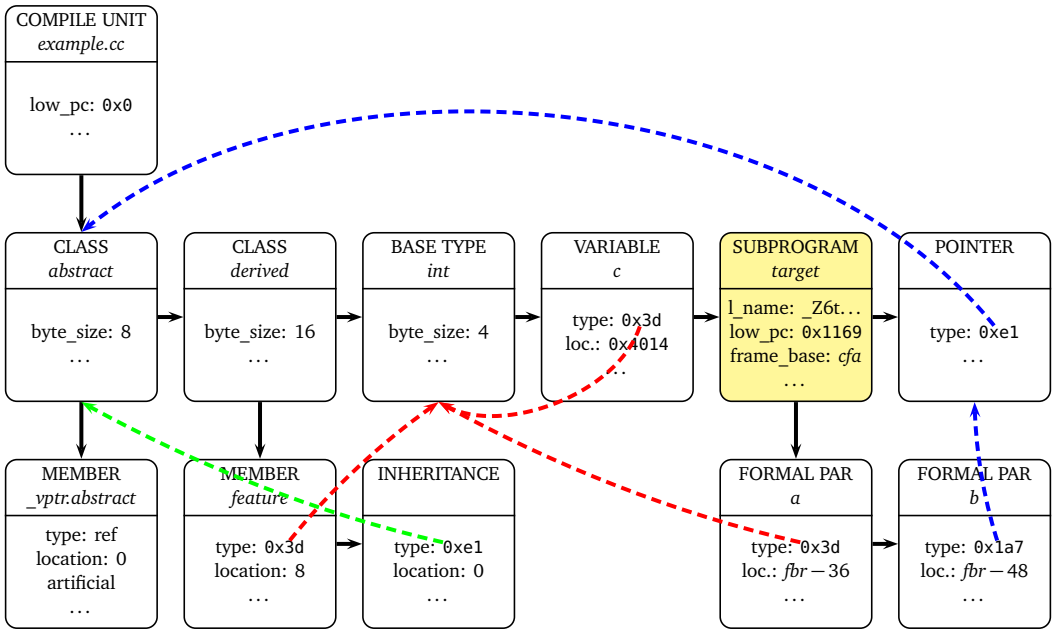
To perform the analysis, Freud takes advantage of the DWARF debugging information that is stored within the ELF binaries of the target programs. We now briefly introduce DWARF information and then describe the analysis performed by *freud-dwarf*. For illustration purposes, we use the C++ program shown in in Figure 5.2 as a target program. In this example, the performance analyst wants to create performance annotations for the method *target*, which takes two parameters *a* and *b*, and can also read the global variable *c*. The relevant feature in the example is the member *feature* of the class *derived*, that is read by the target method through the parameter *b*. Notice that *b* is statically defined to be of type *abstract* in the signature of `target` (line 13). *abstract* is a polymorphic class that can be downcast to *derived*, as in line 14.

5.1.1 DWARF

DWARF [40] is a common format for storing debugging information. This information relates data and locations in the compiled program to programming concepts and specifically to their human readable definitions in the source code. Such information is used by debuggers such as *gdb* to help the developers inspect the state of the program at runtime with reference to the source code.

DWARF uses a tree structure to represent the debugging information of an entire program. Each node of the tree is a Debugging Information Entry (DIE). DIEs represent different objects in the program, from compilation units to methods and variables. Each DIE has a set of DWARF-defined attributes, such as *name*, *type*, or *location*. The semantics of the attributes depends on the type of the DIE.

In this thesis we cover only a very limited set of attributes for presentation purposes. A complete documentation is available in the reference manual [10].




```
1 class abstract {
2       virtual void foo() = 0;
3 };
4
5 class derived: public abstract {
6 public:
7       int feature;
8       void foo() {};
9 };
10
11 int c;
12
13 void target(int a, abstract * b) {
14       derived * d = dynamic_cast<derived *>(b);
15 }
16
17 int main() {
18       target(0, new derived());
19 }
```

Figure 5.2. Example of DWARF tree for a C++ program

The upper part of Figure 5.2 shows the DWARF tree that corresponds to the

C++ program in the lower part of the picture. Each node of the tree is a DIE; the first line of each DIE represents its type, the second line is the name, while the rest is the set of relevant attributes used by Freud. For space efficiency reasons in the binary encoding, every DIE has an implicit reference to zero or one child, and a reference to zero or one sibling.

In order to reduce the size of the binaries, DWARF avoids duplicating information and instead uses cross-references to DIEs. For example, the DIE representing the `int` base type is referenced by three other DIEs in our program. We illustrate these references with dashed arrows in the Figure. Red arrows are for basic type specifications, blue arrows are for pointer types, and green arrows are for inheritance relations.

We now introduce the DIEs of interest for our analysis. Such DIEs represent information in the program such as types, methods, parameters to methods, and variables.

Subprograms

The methods in the program are represented by *SUBPROGRAM* DIEs in the tree. This kind of DIE has a few attributes that are used by Freud: (1) *name* or *linkage_name*, whose value represents the name (mangled name) of the method in C (C++) programs, (2) *low_pc* which is the address of the first instruction of the method, and (3) *frame_base*, which represents the address of the base of the activation frame of the method. The activation frame is the memory buffer that is allocated as stack memory to the method during its execution. In our example the `target` method has a linkage name that is created by the *g++* compiler. The value for the *low_pc* is a constant, while the value for the *frame_base*, *cfa*, stands for *Canonical Frame Address*, and is defined to be the value of the stack pointer at the call site in the previous frame. This value can typically be found in a specific register of the processor (*rbp*) at the beginning of methods.

Formal Parameters

The parameters taken by the methods are represented by *FORMAL PARAMETER* DIEs, which can be found among the children of the SUBPROGRAM DIE of the corresponding method. The most important attributes of these DIEs, besides the name, are (1) *type*, which is a reference to the DIEs representing the type of the parameter, and (2) *location*, which encodes the position in memory of the parameter.

The type is encoded as a chain of DIEs. The chain contains DIEs such as

POINTER or *CONST*, and ends in (a) a *BASE TYPE*, (b) a *STRUCTURE*, or (c) a *CLASS*. In our example the formal parameter `b` of `target` has a type attribute which points to a *POINTER DIE*, which in turns points to the *CLASS DIE* representing the `abstract` class.

Location attributes, instead, describe where the parameter is going to be located in the program memory. DWARF uses location expressions that encode instructions for a virtual stack machine. Such virtual machines instructions use absolute values, register values, or values present at given memory locations. The execution of these location-programs with the runtime registers and memory values as they are during the execution of the target program, produces the actual location.

This encoding with such stack machine programs is necessary because the location of variables and parameters changes during the execution of the target program, since CPU registers and the stack are used for various purposes and often interchangeably. So, the DWARF location expressions must be evaluated in a specific context, assuming a specific value for the program counter. Since we always know where our instrumentation is going to be executed within the program, most of the times we can evaluate the location expressions already during the static analysis of the debugging information, avoiding any overhead incurred for the evaluation at runtime.

Complex Types

Complex types, such as structures and classes in C/C++, are described with all their members and the corresponding memory layout in the DWARF debugging information. DWARF represents complex types such as classes and structures with *CLASS* and *STRUCTURE* DIEs. For our analysis, their most important children in the tree are *MEMBER* DIEs, which enumerate their members. Just like formal parameters, members have location and type attributes. While the type attribute is identical to those of the formal parameter DIEs, the location indicates the offset (in bytes) of specific members relative to the beginning of the memory buffer containing the complex type.

Let's go back to our example. The parameter `b` of the method `target` has a type field that points to a *POINTER* DIE, which in turns points to the *CLASS* DIE named `abstract`. `abstract` has only one member, `_vptr.abstract`, which we discuss later. At the same time, the `feature` member of the `derived` class is positioned 8 bytes after the beginning of the class. In more complex scenarios, the members of complex types can have complex types as well. These sub-trees representing complex types always end in base types.

Note from this example that, just by reading the debugging information referenced from the `target` DIE, we do not reach the information of the `derived` class and therefore we cannot see the `feature` variable. To handle these cases, we need our static analysis to be aware of inheritance and polymorphism in C++.

Inheritances

Several programming languages, such as C++, support class inheritance and polymorphic types. In C++, a class *B* inherits from a class *A* when *B* extends *A*. In other words, *B* has all the fields and methods declared by *A*, plus potentially more fields and methods as specified by the programmer. This relation between types is represented in DWARF with the *INHERITANCE* DIE. This DIE represents exactly the relation between a class and one of its base classes introduced in C++ by the colon (:) character, as in line 5 in our example program.

In our example, one of the children of the `derived` class in the DWARF tree is an *inheritance* DIE, indicating that `derived` is extending another class. The type of this DIE points to the type of the complex type that `derived` inherits from. Concretely, at the byte-level, these classes are as follows: first there are all the members of the inherited type, and then all the members of the derived type. The location attribute represents the offset, in bytes, at which the members of the derived class begin relative to the beginning of the inherited class.

In our example, we see that the class `derived` is extending the class `abstract`. Therefore, at byte-level an object of type `derived` will have all the members of `abstract` at the beginning, and then, with offset 0, the members of the type `derived`. Adding the offset of the inheritance with the offsets of each member, it is possible to reconstruct the complete memory layout of `derived` objects.

Polymorphic Types

With polymorphism, it is possible to use a single symbol to represent multiple different types. Such types might have different members, and therefore different features. While static, compile-time polymorphism allows our static analysis on debugging symbols to correctly access all the information that is stored in objects as they can be seen by the methods of the target program, dynamic (i.e. runtime) polymorphism can be a problem.

This can be seen in our example, in which our feature of interest is in the `derived` class. However, in the code and in the DWARF debugging symbols, our `target` method takes an object of type `abstract`, which does not carry any feature of interest. Since `abstract` has virtual methods, it uses dynamic poly-

morphism: objects that are declared to be of type `abstract` in the code, might actually be of different types at runtime.

The way this dynamic polymorphism is implemented in the binary program is compiler specific. *g++* uses pointers to virtual tables, such that polymorphic types contain, at the beginning of their memory space, `_vptr.` members that are added automatically by the compiler and are not visible from the source code. These are pointers to the *virtual tables* of the runtime objects. These tables contain pointers to methods, allowing the program to execute the methods specifically assigned to the objects, as appropriate for their runtime dynamic types.

Still, these virtual pointer tables do not uniquely identify a type, which is what we need for our purpose of accessing all the information carried by polymorphic objects at runtime. So, the compiler must add one more field to a polymorphic object to uniquely identify its dynamic type. In particular, the binary code generated by `g++` stores at address `_vptr.` -8 the address of the `__class_type_info` object that represents the actual type of the object. `__class_type_info` is itself a structure that contains a pointer to a char string representing the name of the dynamic type. In other words, we can programmatically retrieve the name of the runtime type of a polymorphic object, starting from the object's virtual table pointers and going through the object's `__class_type_info` descriptor.

Additionally, C++ allows multiple inheritances, which results in objects having more than one `_vptr` members at the beginning.

5.1.2 Extracting the Data

Figure 5.3 shows the two main phases of the computation performed by *freud-dwarf*; first the DWARF tree is explored to collect information about the selected methods, their parameters, and about the class hierarchies. In the second phase such information is used to produce the C code (*feature-processing.cc*) and other info (*table.txt*) used by *freud-pin*.

Methods and Parameters

First *freud-dwarf* finds the SUBPROGRAM DIEs of the methods selected by the users. To uniquely identify methods in C++ programs, the user of Freud must provide their *mangled* names, which is matched against the *linkage_name* attribute of every *SUBPROGRAM* DIE. In plain C, instead, the name of the methods must be unique, therefore the search is based on the value of the name attribute of the subprogram DIEs.

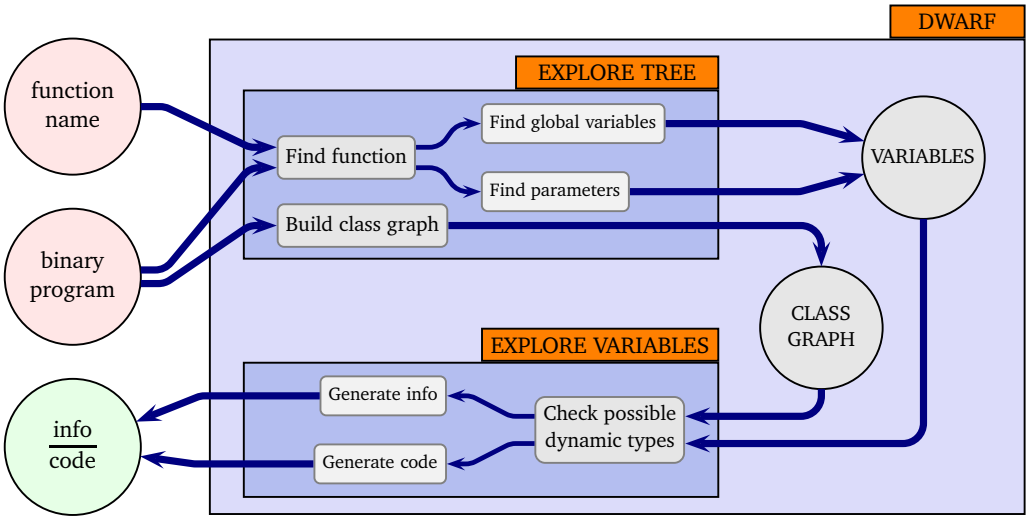


Figure 5.3. Architecture of *freud-dwarf*

Once the correct DIE has been identified, *freud-dwarf* collects information about the method: instrumentation entry point address, parameters, and global variables in the same scope.

Entry Point Address

freud-dwarf reads the *low_pc* attribute's value of the SUBPROGRAM DIE in order to find the address for the entry point into the methods.

In C/C++ programs, each method executes in a dedicated memory area in the program's memory. Such memory area is used to store parameters and local variables used by the method. The creation and setup of such memory area is performed before the actual code of the corresponding method is executed, in the so called *preamble* of the method.

The address defined by *low_pc* points to the method's code, at the beginning of the preamble. Since the goal of the instrumentation code is to read the value of the parameters, we want it to execute after the preamble is complete, when the memory area will be populated with the information we want to collect. So, we need to find the address of the first instruction of the method after the preamble.

To find it, *freud-dwarf* uses information from a different section of the program's binary, the *.debug_line* section of the ELF file. This section is effectively an index into the program information. Each line maps a memory address in the binary program to information in the source code of the program. *gcc* uses the convention of storing multiple lines with the same address pointing to the same

method, if there is a preamble for the method. The last address points to the first instruction of the method, after the preamble.

Parameters and Variables

Next, *freud-dwarf* looks for *FORMAL_PARAMETER* DIEs among the children of the method's SUBPROGRAM DIE. These DIEs represent the parameters that are passed to the method. The attributes that we use for our analysis, in addition to the name, are (1) *location* and (2) *type*.

Freud tries to evaluate DWARF location expressions during the static analysis. Since we know exactly the address in the program memory where our instrumentation code is going to execute, most of the times we can resolve the location expressions to a register, an absolute address in the program memory, or an offset from the value contained in a register at runtime. This allows for efficient evaluation of locations at runtime, minimizing the overhead of the instrumentation.

Freud also follows the chain of DIEs that describe the type of the parameters, including whether each parameter represents a pointer or not. We ignore const qualifiers, since Freud never writes or modifies any bit of information in the instrumented program. If one parameter is of a complex type that has a `_vptr.` member, then we know that the object, at runtime, might be of different types. We need to find all the possible runtime types to extract information that will be used at runtime to read all the information available.

In order to compute the number of potential runtime types for each complex type, Freud performs an exhaustive search in the DWARF tree. The goal of this search is to build a complete class graph, which represents all the hierarchies between classes in the program. We use this graph when writing the information about parameters: for each complex type, in addition to the type defined statically in the signature of the target method, we write all the possible derived classes (i.e. descendants, in the class graph). For each potential class, we write the hash of the name of the type, and the number and names of the possible features.

Additionally, while exploring the tree, *freud-dwarf* logs all the variables that are direct descendants of Compilation Unit DIEs. These are global variables which are visible to all the methods at the same level, or at lower levels in the same branch of the tree. We explore the location and type attributes of global variables as we do with formal parameters, but the location is typically an absolute address, which is resolved at runtime.

5.1.3 Generating Code and Info

Once we have all the information that is needed for our instrumentation, we have to create the actual instrumentation code and information that will be used by *freud-pin*.

The static information is written in a file called *table.txt*. This table contains one entry per target method, including the entry point address for the instrumentation, and information about parameters. For each parameter we have: name, location, whether the parameter represents a pointer, and one or more sets of basic features. Each set of features is associated with a different runtime type when we have polymorphic types. To identify the runtime type, we use a hash of the string representing the type.

The information described so far is used to set up the instrumentation: it finds the points in the program where to add jumps to the instrumentation code, and also finds the registers and memory addresses that contain information relevant to the parameters.

On the other hand, Freud must extract feature candidates from complex types, such as structures and classes. To do this, *freud-dwarf* creates specific C code in the *feature-processing.cc* file. This code is executed for every parameter and global variable that should be collected for a method. The code consists of a switch statement that evaluates the static type of the parameter that is going to be analyzed. Each parameter type requires different actions to extract features information. Some types also require different actions whether they are stored in memory or in registers. This is the case for smaller objects that can be passed completely through CPU registers. Pointers and references to complex types are always passed as addresses.

To speed up string comparisons, we use 64-bit hashes of the strings in all the instrumentation code.

The code to handle polymorphic types has one additional switch, to identify the actual type of objects at runtime. To perform this check, we hash the name of the dynamic type at runtime and compare the result with the hash value stored in *table.txt*. *vptr* points to the *__class_type_info* for the object, at runtime. *__class_type_info* + 8 points to a string representing the mangled name of the object. At runtime, in the PinTool, we hash that string and compare to the hash generated statically by *freud-dwarf*. The static hash is passed through *table.txt*.

Once the correct type at runtime is determined, we execute the actual feature extraction code. This code does not use any source-level declaration or definition of the objects it is inspecting. Instead, everything is performed at a low level, with only basic types and memory locations. The feature extraction code collects the

values of variables and complex types members 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. Whenever we need to dereference a pointer or read from unsafe memory locations, we use the `PIN_Safecopy` function, that performs checks on the validity of the memory address without causing SIGSEGV.

Also, while exploring the tree to document the structure of complex objects, *freud-dwarf* applies heuristics to create potential *derived* features. For example, we compute the difference between variables named start, begin, first and stop, end, last, respectively, and we log them as a derived feature representing a size or time span. For pointers and aggregate types (i.e., structs) we perform a traversal to reach nested or linked variables. Pointer traversal requires a runtime check to avoid de-referencing invalid pointers.

5.1.4 Parameters

Table 5.1 enumerates the parameters that the user can control in *freud-dwarf*.

We use `MAX_DEPTH` and `MAX_FEATURES` when looking for feature candidates, to limit the number of potential features that we consider.

With `MAX_DEPTH` we limit the distance, in terms of structures or classes, of the feature candidates from the direct variable that is accessed by a software component. For instance, `abstract.feature` is at depth 2 from the viewpoint of the method `target`, in the example of Figure 5.2.

With `MAX_FEATURES` we limit the number of features candidates that are collected for a single variable as seen by the target method.

We set a maximum to the depth of the exploration to limit the amount of data that we collect, in order to reduce both the overhead during the execution of the target program, and the running times for the statistical analysis.

It is not possible to characterize precisely the effect of these parameters on a generic program, since the number of additional features that are discovered at increasing depths depends on the specific program that is being analyzed.

In Section 5.4.3 we give concrete data to show the effect of the value of `MAX_DEPTH` on the number of features collected and on the running time for the statistical analysis in real world complex cases.

name	default	description
MAX_DEPTH	3	Maximum search distance from the direct parameters
MAX_FEATURES	512	Minimum number of features to collect before stopping the search even at depths lower than MAX_DEPTH

Table 5.1. Parameters and default values in *freud-dwarf*

5.2 Instrumentation

Once we have all the information about target methods and their potential features, we want to run the target program to extract information from it. This requires instrumenting the target program, which means modifying the program's binary code so that, in addition to the normal operations, it executes instructions to measure performance metrics, and extract feature values. For the instrumentation we use Intel Pin.

Pin allows for dynamic modification of the program binary. Every instruction of a compiled program can be analyzed and modified, and it is also possible to inject new instructions in the program. Pin allows for transparent instrumentation, which means that the original program is not aware of the instrumentation context, and all the relative addresses within the program remain unchanged. In other words, all the relative addresses provided by the debugging information remain valid, from the viewpoint of the instrumentation.

We will now describe in details our instrumentation tool, how it measures performance, extracts feature values, and create the logs for the statistical analysis. First, we discuss some of the constraints of Pin, and how they affect our choices for the instrumentation.

5.2.1 Intel Pin and Pin Tools

Pin is a proprietary tool. The distribution consists of a binary program, *pin*, and a set of libraries and headers, the *PinCRT* (Pin C Run Time).

The typical usage of Pin is as follows: users write their own Pin tools in C, and compile their tools against the PinCRT to create objects containing compiled code. Users then execute *pin* passing the compiled object as parameter, followed by a process to which Pin must attach or a binary program to run. After initialization, the Pin tool takes control of the execution of the process and of its memory

space, with the possibility of modifying the code.

Pin can be used in two mutually exclusive modes: *Just In Time (JIT)*, and *Probe*. With JIT, Pin re-compiles on-the-fly the binary code of the program to which it is attached. So, when using Pin's JIT mode, the binary code that is executed is effectively Pin's newly generated code, and not the original binary code. This is true for all the binary code of the target program, whether it is modified by the Pin tool or not. In JIT mode, it is possible to use all the features of Pin, including instruction-level instrumentation, which we use to log branch outcomes. The Probe mode instead only allows for a limited set of features. In particular, in Probe mode it is only possible to insert jump instructions in very constrained places in the program, or to replace entire methods with instrumentation methods.

Freud currently uses only the JIT mode of Pin. While using the Probe mode would reduce the overhead of our instrumentation, there are different problems with this latter approach: (1) in Pin's Probe mode there are only a few places where we can insert jumps, which is problematic for our instrumentation; and (2) replacing the entire method is not easy, because we would need to replace signatures that contain complex types from the instrumented software, and this must be done within a Pin tool that must be compiled and linked against the PinCRT.

In fact, the requirement that a Pin tool be compiled against the PinCRT poses various limitations to the external libraries and resources that can be used in the tool. In fact, PinCRT implements part of the standard C/C++ libraries, and replace the standard libraries provided by the system (e.g., *glibc* on Linux).

For example, this means that it is not possible to use tools and libraries such as the PAPI (Performance Application Programming Interface) library to collect performance measurements. Also, Pin tools cannot link against any object that is linked against the native system libraries. This makes it not viable to use declarations about complex types from the target program's source code, because (a) linking against the objects containing their definition is not possible, and (b) compiling their definitions against the PinCRT is unlikely to work without major modifications, which would alter the behavior, in addition to being quite a cumbersome task for users.

In order to use the definitions of complex types from the target program's source code from a Pin tool, one option would be to create some glue code that embeds all the instructions to extract features from complex types using the native system libraries and that can be executed from the Pin tool. This amounts to instrumentation code that extracts features from complex types using the source code with the definition of those types, possibly using the default getters provided

by the developers. This code would be compiled *statically* against the native C library of the system. From the Pin tool, it would then be possible to link to entry points in this compiled module. One advantage of this solution is that it would be possible to use "for free" the structure definitions and parsing code that comes with the instrumented program. Still, it would not be easy to compile such glue module pulling code and header files from a big, complex project. Indeed, we would have a chain of dependencies on more development headers and libraries that would complicate a lot the ideally simple glue module.

Given these constraints, we opted for a solution in which we create Pin tools that contain all the code that is needed to extract features from complex types, and can be compiled against the PinCRT without any external dependency.

5.2.2 freud-pin

Figure 5.4 shows the main operations of our Pin tool, *freud-pin*. In the first phase, during the startup of the target program, we setup the instrumentation context, and add jumps to our instrumentation code in specific points in the target program. We also create a new Pin thread that operates in the program's memory space, but is reserved to our instrumentation. Such thread is used to dump output logs while the target program is running, without blocking its execution, and with minimum overhead.

We will now describe more in details each phase and operation.

5.2.3 Adding Instrumentation

The first operation performed by *freud-pin* is to modify the target program's binary code to add jumps to the instrumentation code in different places. More precisely, we add the following jumps:

- At the entry points of target methods, we add a jump to the instrumentation entry code. We will describe in details this code later.
- At the exit points of target methods, we add jumps to the instrumentation exit code. We will describe in details this code later.
- For every branch instruction within the target methods, we add a jump to the branch logging code, which logs the outcome of the branch evaluations.
- At entry and exit points of specific standard library methods, such as `malloc` or `pthread_mutex_lock`, we add jumps to performance measurement code. We will give more details later.

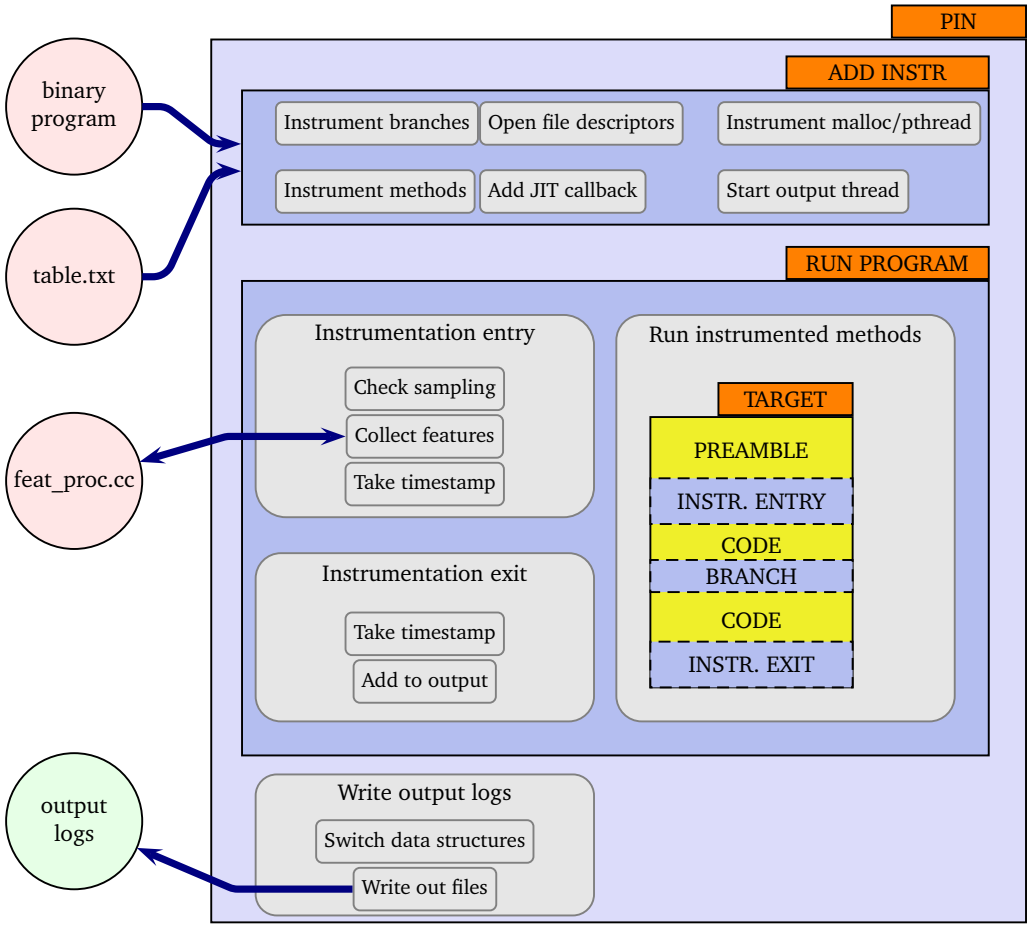


Figure 5.4. Architecture of *freud-pin*

As explained in Section 5.1, when we add jumps at the entry points of methods, we use the address of the first instruction of the method after the preamble. Such address is reported in *table.txt*.

All the addresses provided with the debugging information are provided as relative to the beginning of the target program's memory space. With the Pin API, on the other hand, we need to provide absolute memory addresses. Absolute addresses change at every execution of the program, as mandated by the Address Space Layout Randomization (ASLR) security feature, that is typically enabled on every modern Linux distribution.

To compute real absolute addresses, *freud-pin* finds the base address of the program's memory space at the beginning of each execution of the target program, reading the *procfs* filesystem. The complete addresses are computed as

the sum of the base address, and the relative addresses within the program's memory.

Conditional Branches

If we want to collect information about branches, we analyze the code of the target methods at the instruction level. To do so, we use Pin's JIT mode to find all the conditional jump instructions. We instrument those instruction to first jump to our instrumentation code, which records for every execution whether the jump has been taken, or not. Different branches are uniquely identified with their addresses.

This instruction level instrumentation can introduce a considerable overhead if the instrumented branches are executed many times. Branch instrumentation should therefore be avoided when exploring and when it is somehow known that the branch analysis is not needed.

5.2.4 Running the Program

When the setup of the instrumentation is complete, the target program is started, with the same command line options it would take during normal operations. The target program runs with our instrumentation code, which is executed when any of the jumps added during the setup phase is reached. Such instrumentation codes takes care of measuring performance metrics, and collect the values of potential features.

Metrics Collection

Given the limited number of possible approaches allowed in the PinCRT context, we will now describe our approach at performance measurements. In general, we access directly low-level information, minimizing the overhead introduced by the measurement.

Time. For time, our Pin tool reads directly the processor Time-Stamp Counter (TSC). The TSC on newer Intel and AMD processors is synchronized across all the CPU cores, and ticks at constant rate regardless of any power state and clock speed. Also, the TSC for all the CPU cores is reset at the same time by the Linux kernel during the boot of the system. Thus, the TSC can be used for wall clock timings ([23], Sec. 17.17.1).

We use the `rdtscp` instruction to read the TSC efficiently at the user level without context switch, minimizing the chances of instructions reordering by the processor. We take one timestamp just before the execution of the target method, and another one as soon as the control goes back to the instrumentation, when the execution of the target method ends.

The difference represents the number of TSC ticks that occurred during the execution of the method, even in case of context switches and threads being moved to other processors. To convert this metric to time, it is necessary to divide the number of ticks by the nominal speed of the processor.

Memory. We instrument `malloc` to add a jump to our instrumentation code at the entry point of the function. The instrumentation code reads the integer parameter passed to the function, and the thread id on which the function was called.

The integer parameter represents the size of the allocation requested, in bytes. This value is added to the counters associated with all the methods active on the same thread.

In order to increase the accuracy of the instrumentation, at the expense of additional overhead, it is possible to instrument also the exit point of `malloc`, in order to read the return value, and catch potential failures in the allocation.

Instrumenting `malloc` catches also the C++ `new` calls, which are implemented with `malloc` in *glibc*.

We measure the amount of memory bytes allocated during the execution of a method. In case the maximum amount of memory allocated during the execution is required, it is necessary to instrument also the `free` function.

Locks. We instrument `pthread` functions to log all lock operations, namely requests, completions, and releases. We produce two different metrics from these logs: waiting time, and holding time. We do not measure the details about multiple locks, and instead we just consider each thread as waiting for or holding a lock, or neither. It could be one or more locks. Still, there are only three basic cases we consider for methods run by threads that acquire/release locks: (1) a method can be called while a lock is held, (2) a method can be the caller of the lock acquisition functions, (3) a method can release a lock while it is running.

More in detail, we instrument:

- `pthread_mutex_lock`, at both the entry and exit points. When this function is called, we take a timestamp with `rdtscp` and store its value in a thread specific structure. When the function returns for the same thread,

we take another timestamp. The time difference represents time that all the active methods on the thread spent waiting to acquire a lock. If the return value is zero, then a new timestamp is taken to record the beginning of the lock acquisition by the thread.

- `pthread_mutex_trylock`, at the exit point. When this function returns zero, then a new timestamp is taken to record the beginning of the lock acquisition by the thread.

- `pthread_mutex_unlock`, at the exit point. When this function returns zero, then we compute the difference between the current timestamp and the timestamp representing the lock acquisition by the thread. The difference is the time during witch the thread had the lock.

- `pthread_cond_wait` and `pthread_cond_timedwait`, at the entry and exit points. When these functions are called, the thread implicitly releases the lock. When the functions return zero, the thread implicitly acquires the lock.

Page Faults. We use the *procfs* [27] to read the number of memory page faults. The thread specific information can be read in */proc/PID/task/TID/stat*, where PID is the Process ID, and TID is the Thread ID. We compute the difference between the values at the exit and entry points of the instrumented methods. We assume a specific interface to the information in the procfs, which might change in future versions of Linux. We do not use */proc/thread-self* because we want to open the file only once and keep the handle open. However, we rewind with fseek(0) on each read.

5.2.5 Collecting Features

All the code that extracts values from features within the program, such as parameters and variables, comes from *freud-dwarf*. Our Pin tool executes the switch in the *feature-processing.cc* file for every parameter of the target methods, and for every global variables they can access.

System features, instead, are collected by *freud-pin* autonomously. As an example, to read the clock speed of the processor executing a specific method we do as follows: we first read the thread id on which a thread is running (*/proc/PID/task/TID/stat*), then use the procfs to check on which CPU the thread is running, and finally read the CPU clock speed for that CPU (*/sys/devices/system/cpu/cpuX/cpufreq/scaling_cur_freq*). This might not be always correct, since

a context switch might change the CPU that is executing the thread, however it is still a reasonable solution. Indeed, power-related frequency scaling happens at a different time scale, and the CPU does not idle in between the measurement of the clock speed and the execution of the method. Also, it does make sense in comparing different CPU models or fixed power profiles.

In the current implementation, we do not explore other system features, but all the useful information about the system can usually be found in the procfs, and can be parsed as we do for the CPU clock speed.

5.2.6 Producing Output

We designed Freud to be usable also with long running processes, such as a web server, or a DBMS. This requirement implies that freud-pin must be able to produce the output logs even when the target program is running, without affecting its performance.

For this goal, we use Pin to spawn a dedicated thread that produces output files periodically. To minimize the impact on performance we do not block any thread that is executing the target program. To access safely the data structure, we use a double buffering mechanism. The data structure holding all the records collected during the execution of the target program is duplicated, so that at any time only one is active, and used to store new information. When the output thread wakes up, it switches the active data structure, so that the program threads start using the one that was previously inactive. Then, the output thread dumps the content of the data structure to output files and clears the data structure before going to sleep for another period of time.

One problem with this approach is that we might invalidate some measurements that are active when the buffer-switch happens. However, with an adequate sleeping period, the amount of missed measurements should be limited. When the target program ends, the output thread is activated to dump all the remaining log entries.

The default period for producing the output is 5 seconds, and can be set with a command-line parameter to *freud-pin*. Logically, the output follows the format described in Chapter 4, with one entry per observation. In order to reduce the size of the logs we apply two optimizations: (1) we use a binary encoding, and (2) we replace common information such as the strings that represent feature names with references to a single instance of such information.

5.2.7 Minimizing Perturbation and Overhead

We do several things to minimize the perturbation of the measurements. First, we do not introduce any lock whatsoever in our instrumentation. We still instrument multi-threaded programs, but every thread has its own data structures to write data.

One of the biggest sources of perturbation is the JIT recompilation. To avoid this, we discard every measurement during which the JIT compiler has been active.

We also measure the instrumentation time. This is necessary when there are more than 1 active methods, concurrently.

There might still be perturbations in multi-threaded applications whenever a method instrumented for time depends on another instrumented method on another thread. In that case, Freud effectively attributes the running time cost of the instrumentation of the second method to the running time total of the first method.

To minimize overhead, we use sampling, so as to collect a limited number of observations for each output dumping period. Since we are sampling from a potentially infinite stream of observations, we use reservoir sampling [46] to collect random samples for each target method. The default number of samples is 100 per dumping period, but this value is arbitrary, and should be adjusted by the performance analyst as needed for specific use cases. The number of samples per period can be specified with command-line parameters to *freud-pin*.

5.2.8 Parameters

Table 5.2 enumerates the command line parameters that are available in *freud-pin*. Freud users can use these parameters, in addition to the *freud-dwarf* parameters, to reduce the overhead of the instrumentation when some information is not needed for the performance analysis.

The parameters that contribute the most to the overhead are `branches`, which controls whether branch instructions of selected software methods are instrumented or not, and `logs_count`, which controls how many samples are collected, in conjunction with `dump_period`.

`dump_period` is used to decide how often the instrumentation writes the collected information to output files. A value of 0 can be used to disable the additional dumping thread, and write output files only once, at the end of the execution of the instrumented program.

Finally there are parameters to enable or disable the collection of all the per-

formance metrics, in addition to time, which does not introduce any observable overhead.

name	default	description
`branches`	true	Instrument branches
`logs_count`	100	Number of samples to take per dump_period for each symbol
`dump_period`	5000	Time (ms) between consecutive flush to the output
`locks`	false	Measure locks related metrics
`memory`	false	Measure memory allocations
`pfaults`	false	Measure page faults (needs procfs)
`procfs`	true	Read from procfs (needed for CPU speed)

Table 5.2. Parameters and default values in *freud-pin*

5.3 Statistical Analysis

The last step of our performance analysis is a statistical analysis, which is performed offline by our tool, *freud-stats*. *freud-stats* reads the logs produced by *freud-pin*, and produces performance annotations, in the form of text files with the textual representation, and EPS files with the plot representation.

5.3.1 freud-stats

freud-stats is a stand-alone program, that can be run offline on any system, that processes the binary logs produced with the interface implemented by *freud-pin*, and terminates the execution with the creation of performance annotations.

The parameters taken for the analysis are the names of the methods to analyze, and the performance metric. There are also some optional parameters, such as for setting a custom minimum R^2 for the regression analysis.

We implemented the algorithms of Section 4.2 in C++ in *freud-stats*. We use the *R statistical package* library for the basic blocks of the statistical analysis, and *gnuplot* to produce graphs.

We now detail some some parts of our specific implementation of some of the algorithms of Section 4.2. Our C++ code implements the algorithms to create the classification tree exactly as they are described there.

Pre-processing

We implement the pre-processing code directly in *freud-stats*. We find scoping conditions using the branch logs, which contain the outcome of each executed conditional branch. In our analysis, we consider only branches that are executed exactly one time for every execution of the target methods. These branches likely represent switches that distinguish slow and fast paths in the execution of the methods. We use Algorithm 1 to find perfect correlations between some feature values and the outcome of specific branches, and produce partitioning candidates based on the values of specific features.

Regressions

First we filter correlated features to keep only one representative for each equivalence class. To do so, we use the correlation matrix given by the *cor* function provided by *R*. We remove one of the two features, for any pair with a correlation coefficient bigger than 0.9, where 0 means no correlation, and 1 means perfect correlation. In the current implementation, we do not use any heuristic or algorithm to decide which of the two correlated features to drop. A future improvement might use some heuristics on the name of the features to keep the one that is more likely to be indicative of a size or a length, which would presumably be more meaningful for the performance analyst.

Once we have filtered the inter-correlated features, we proceed with formulating regressions, as described in Chapter 4. We use the *lm* function provided by *R* to compute linear models with interaction terms. To generate regressions we use a default R^2 value of 0.75, because in our experience it is a good trade-off to account for noise in the data, and still produce meaningful models. We still let the user override the default R^2 value. The minimum p-value that every feature must have in the regression is 2×10^{-11}.

In addition to what we said in Chapter 4, when we compare regression models from different classes (i.e. constant, linear, quadratic, logarithmic) with the BIC value, we additionally use a delta for the BIC to prefer simpler models over more complex ones. In other words, if we have two valid regressions r_1 and r_2, we prefer r_1 over r_2 if and only if $BIC(r_1) - BIC(r_2) - 10|deg(r_1) - deg(r_2)|$, where $deg(r)$ is the degree of the model r.

The idea of using a delta to reduce the probability of choosing more complex models with no big difference in the prediction quality of the model, comes from a statistical paper ([25], [36]) that discusses the use of BIC in the different scientific contexts. We choose 10 as the value for the delta, because it produces good

results in choosing the most appropriate models in our experiments.

Clustering

When we cluster the data points, we always consider only the performance metric, ignoring all the information about the features. This means that we always consider one-dimensional data. We choose to use a Kernel Density Estimator clustering, which is efficient with one-dimensional data, and does not require any prior assumption on the number of clusters.

The one parameter that is required by this KDE analysis, is the smoothing bandwidth (i.e. the window size) to use when computing the density of the distribution of values. For this purpose, we use a variable-bandwidth kernel density estimator [39] to compute the one-dimensional density of the data. We implement this technique in our own-made `R` port of the `akde1d` function [6], originally written for `Matlab`. `akde1d` automatically adjusts the smoothing bandwidth according to the local densities of the distribution under analysis.

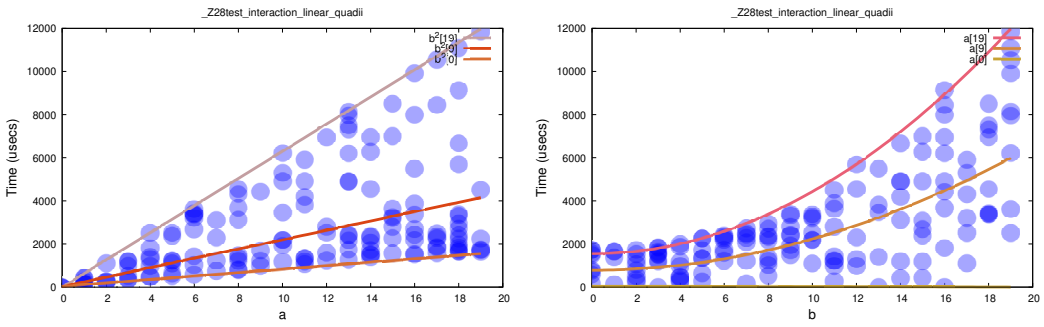
We then find the local minima and maxima in the density, and use those points to cluster the data. The minima represent the boundaries of the clusters, while the maxima represent the centroids. The number of clusters is thus defined by the number of local maxima in the density of the performance metric.

Plots

We use *gnuplot* to produce graphs. To keep the graphs readable to the human performance analyst, we split multi-feature and therefore multi-dimensional graphs into various two-dimensional graphs. Each two-dimensional graph represents the correlation between one performance metric and one of the relevant features. The effect of the other inter-correlated features is represented taking some sample values for the other features. Figure 5.5 is one example, that shows the interaction between the features a, which contributes a linear complexity to the performance, and b, which contributes a quadratic complexity.

5.3.2 Checker

Freud also implements a checker, that can read performance annotations as input, and check them against the measurements coming from the instrumentation. The result of the analysis is a Boolean value stating whether or not the new behavior is compatible with the performance annotations. In other words,



(a) Main feature *a*

(b) Main feature *b*

```
void test_interaction_linear_quad(int a, int b).time {
features:
 int a = a;
 int b = b;

annotations:
 Norm(155.10*a + 15.52*b^2 - 7.62*b + 17.95*a*b^2 - 51.50*a*b, 21.87);
}
```

(c) Performance annotation

Figure 5.5. Example of plots for regressions with interaction terms

the checker implements a prototype for assertion checking using performance annotations.

5.3.3 Parameters

Table 5.3 enumerates all the parameters that are used in the current implementation of freud-statistics.

`CORRELATION_THRESHOLD` represents the minimum correlation value that two different feature candidates must have to be considered equivalent in the regression pre-processing phase. If two features are considered equivalent, then one of the two is dropped and not used to compute regressions. A higher value means less pre-filtering and therefore more pressure on R, that is going to compute regression models with more features. Highly correlated features are going to be filtered anyway by R during the iterative regression analysis phase. While a lower value for `CORRELATION_THRESHOLD` means less features used in the regression analysis, the risk of using a lower value is drop features that could be beneficial

name	default	description
CORRELATION_THRESHOLD	0.9	Maximum correlation between two features when filtering
MIN_DET	0.75	Minimum R^2 value for a correlation
GOOD_PVALUE	2×10^{-11}	Maximum p-value for which a feature is considered good in a regression
BIC_DIFF	10	Additional penalty for a higher-degree model for a regression
DROP_K	5	Number of bad features to remove at each iteration when computing regressions

Table 5.3. Parameters and default values in *freud-statistics*

in finding good regressions. The default value, 0.9, aims at filtering only highly correlated features, and is therefore a conservative choice. On the other hand, in complex cases with hundreds of feature candidates, it might be beneficial to lower the value in order to reduce the time for the statistical analysis.

MIN_DET is the minimum R^2 value that a regression must have in order to be considered valid. This is one of the most direct way of controlling the results of the statistical analysis. A higher value forces *freud-statistics* to try to compute regressions that fit better the observations. This is achieved with either (1) branch analysis, (2) noise removal, (3) main trend analysis, or (4) regression after clustering analysis. Setting an exceedingly high value for MIN_DET might hide some regression that would be informative for the performance analysts. Setting a low value, on the other hand, might make *freud-statistics* stop the exploration of the classification tree soon, and hide more precise regressions that could have been found with one of the approaches above.

GOOD_PVALUE is the maximum p-value that a feature must have in a regression to be considered relevant, and be used for more iterations in the iterative regression analysis. The default value is quite strict, meaning that only features that are considered strongly relevant are kept. A higher value makes freud-statistics keep features that are less relevant in the regressions, with the risk of using more features, and building more complex models. A lower value, instead, increases the risk of dropping features that are instead very relevant to determine the performance behavior in a performance annotation.

BIC_DIFF is the minimum absolute difference that there must be in order

to prefer a more complex model when computing regressions, as discussed in Section 5.3.1.

DROP_K is used in the iterative algorithm that computes the best regression model of a given complexity to represent the given set of samples (Section 4.2.2). When we cannot find any feature with a p-value lower than GOOD_PVALUE, we proceed removing the worst DROP_K features. A higher value for this parameter makes the model selection process quicker, but might remove from the regression model features that would be relevant to describe the performance.

5.4 Validation

In this Section we validate Freud, as a tool that automatically creates performance annotations for C/C++ methods. To such extent, we want to show that Freud (1) produces accurate performance annotations that correctly document the performance in non trivial scenarios, (2) does not perturb the performance behavior of observed programs, (3) completes the analysis in a "reasonable" time (i.e., minutes).

5.4.1 Accuracy

To validate Freud, we created a set of micro benchmarks consisting of simple methods written in C that have known performance behaviors. We analyze twenty-two methods, each exhibiting a well-defined behavior in terms of running times, which we control using `usleep`.

With this experiment we evaluate two fundamental functionalities that Freud must implement to produce performance annotations automatically: (1) the ability to find relevant features even when such features are found in complex structures and classes, and (2) the ability to compute the expected correlations and produce the expected clusters from the raw data.

The micro-benchmark source code and test programs are included with the Freud public distribution. We provide a complete description of all the functions of the micro-benchmark, the resulting performance annotations, and comment on the purpose and the results of each function in Appendix A.

We experiment with times linearly and quadratically correlated with a given value of an input feature, and also with running times chosen at random from a known distribution independent of any feature. We also have methods that exhibit different behaviors based on switches and enumeration variables, methods in which different features interact to define the performance behaviors, and

methods that use dynamic polymorphism to access features that would not be visible from a static analysis of the parameters. In all these cases, we first verify that Freud accurately derives a model of the expected class (quadratic, linear, etc.) for each method. We then measure the accuracy of the specific annotations through cross-validation for all twenty-two methods with an overall minimum R^2 value of 0.9866.

Also, the running time for *freud-dwarf* on the complete micro-benchmark program is of 0.019 seconds. The running time for freud-statistics varies from 0.3 seconds, for the analyses that do not need clustering, to 11 seconds when the statistical analysis needs to cluster the data points.

5.4.2 Overhead and Perturbation

Perturbation and overhead are two potential effects of the instrumentation applied by *freud-pin*. Ideally, we want to minimize both effects, so that the program under analysis runs identically in terms of performance (i.e., using the same resources) with and without the instrumentation.

Considering time as the performance metric, both perturbation and overhead might result in the same effect of increasing the running time for the program under analysis. However, perturbation and overhead are effectively very different phenomena.

With overhead we refer to a *degradation* of some performance metric of the programs under analysis, as a consequence of the instrumentation of the program. Examples are an increase of the running time, or an increase of the memory usage. Instrumentation necessarily introduces overhead. However, this is widely regarded as an acceptable cost, as the benefits of collecting the data outweigh the performance degradation. The important point, however, is that overhead does not necessarily threaten the validity of the data collected about the performance metrics.

On the other hand, with perturbation we refer to a *modification*, which might be positive or negative, of some performance metrics of the program under analysis, as a consequence of the instrumentation. Such modification is observed also by the instrumentation, which is therefore collecting perturbed information about performance metrics. In other words, when perturbation occurs, the instrumentation produces measurements of the performance that do not represent the real behavior of the program in the absence of the instrumentation.

In the design of Freud, our highest priority was to minimize the perturbation, as opposed to the overhead. For example, the addition of callback functions to check the activity of the JIT compiler of Pin is useful to limit the perturbation,

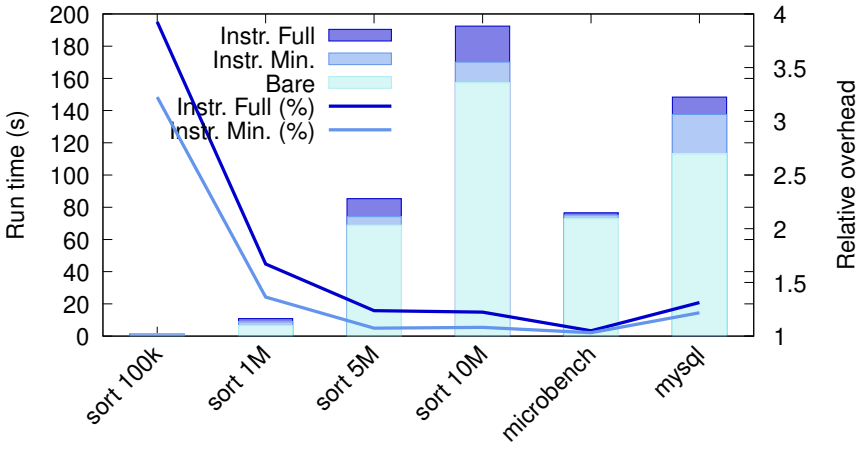


Figure 5.6. Runtime performance overhead of Freud

even though it increases the overall overhead on the target program.

One potential source of perturbation is the instrumentation itself. It is important that the instrumentation does not consider the cost incurred by the instrumentation itself, as costs generated by the instrumented methods. This could happen, for example, when instrumenting nested methods.

To test that this problem of measuring the instrumentation is not affecting Freud, we analyze the same twenty-two controlled functions under two configurations: normal instrumentation and double instrumentation (i.e., log everything twice). We would expect that if instrumentation perturbs data and produces wrong annotations, then doubling the instrumentation would double the perturbation and therefore lead to significantly different annotations. But that is not the case. The experiment confirms that the models for both configurations (for the same function) are of the same class (e.g., linear) and they are equivalent. We measure equivalence by checking that a model derived with double instrumentation predicts the data collected with normal instrumentation with the same high accuracy as the model derived with normal instrumentation, and vice-versa, with a minimum R^2 value of 0.9838.

Let's consider now the running time as a performance metric. The same ideas apply to others, such as memory usage, with no differences.

To evaluate the overhead introduced by *freud-pin*, we run some experiments on different programs.

Figure 5.6 shows the overhead introduced in some specific experiments: we test (a) the `sort` method of the `std::list<int>` class, executed in a simple program written for this specific test, (b) our validation micro-benchmark described

above, and (c) the MySQL DBMS version 5.7.24, executing a specific insertion benchmark.

To measure the overhead for the instrumentation of `sort`, we write a program that runs a loop a predefined number of times. In each iteration of the loop, the program creates a new list of random integers and sorts it. The lists have fixed size, that is passed as input to the program. In our experiments, we test with sizes of 100k, 1M, 5M, and 10M. In this program, we instrument only the `std::list<int>::sort` method. The instrumentation collects many features from complex data types.

Next, we test our micro-benchmark, in which we instrument *all* the methods at the same time. The instrumentation collects only few features for each methods, since there are no complex types that are big in terms of number of members.

Finally, we measure the overhead introduced when instrumenting MySQL. We let MySQL execute a well defined insertion benchmark, while we instrument the `mysql_execute_command` method. This is a high level method with access to a huge number of feature candidates.

For all the experiments, we measure the total time taken to initialize and run the target program. We do that using the *time* Unix utility program. For MySQL, we use a custom script to kill the server as soon as the insertion queries are executed. We run the programs (a) without any instrumentation (*bare*), (b) with the minimum instrumentation (*Instr. Min.*) needed to generate rich performance annotations for the methods, and (c) with the full instrumentation (*InstrḞull*), where every possible functionality of the instrumentation activated, even if not needed.

Figure 5.6 suggests that the overhead stems from three sources.

First, there are fixed initialization and termination costs. This is clear from the experiments with a shorter absolute running time, such as sort_100k and sort_1M, where the overhead is dominated by such costs, resulting in huge relative overheads. The fixed costs quickly become negligible, as it can be seen by the relative overheads for the sort experiments with more than 1M elements.

Second, there is overhead introduced by the instrumentation to read the feature values, read the branch outcomes, and measure the performance metrics. This overhead depends on the number and complexity of features collected, and on the level of instrumentation. The difference between the *full* and *minimum* levels of instrumentation shows that using all the instrumentation features, especially collecting the branch logs, has a considerable cost, and should be avoided when not necessary. The impact of this overhead on the overall performance of the instrumented program depends on the number of times an instrumented

method is executed, and on the duration of the method. For example, in the micro-benchmark test we only have methods running for at least milliseconds, which is long enough time to make the relative cost of the instrumentation to collect the few feature candidates almost negligible.

Finally, there is the overhead introduced by Pin, not directly caused by the operations of the instrumentation. In other words, this overhead would be visible even with a fake, completely empty instrumentation. This overhead is introduced by the Pin JIT recompilation of the code. On the one hand, it actually takes time to recompile the code. On the other hand, the recompiled code might not perform exactly as the original one. This might also introduce some perturbation in the performance behavior of the methods observed, but we have experimental evidence that the behavior of the recompiled code is similar to that of the original code. Bach et al. [1] observe the overhead introduce by the Pin JIT recompilation with different programs, and obtain overhead values that are comparable to what we also observe with our Freud instrumentation.

In all the experiments where the fixed initialization costs are not dominating the running time, we have a maximum relative overhead of 21% when using the needed instrumentation features, and 31% when instrumenting more than what is necessary to create informative performance annotations. Whether this is acceptable overhead for a production system depends on the specific domain of the application.

5.4.3 Running Time

The static analysis of the binary code (*freud-dwarf*) and the statistical analysis (*freud-statistics*) are performed offline with respect to the execution of the program under analysis. Therefore we did not invest time in optimizing the performance of those two components of Freud. Still, we consider the time for performing these two steps of the analysis to be reasonable. We give a precise characterization of those steps in the case of the micro-benchmark programs in Appendix A. Now we analyze the performance of freud-dwarf and *freud-statistics* on real-world complex programs.

Table 5.4 reports the time taken by freud-dwarf when analyzing three target methods. Two are the same `mysql_execute_command` method in two different versions of MySQL; one is `x264_8_encoder_encode` in x264. We measure the running time and the number of potential features found with different values for the `depth` parameter. It is interesting to notice that the running time does not really depend on the depth of the exploration, nor on the number of feature candidates that are discovered. The main parameters affecting the running time

are the size of the binary file containing the debugging information, and the complexity of the class hierarchy encoded in the debugging information.

target method	depth	features	time (s)
	1	106	9.055
`mysql_execute_command`	2	632	9.094
MySQL 5.7.24	3	655	9.105
(binary size 218MB)	4	734	9.125
	5	783	9.143
	1	103	32.808
`mysql_execute_command`	2	637	32.816
MySQL 8.0.15	3	675	32.787
(binary size 652MB)	4	727	32.843
	5	753	32.946
	1	99	1.13
`x264_8_encoder_encode`	2	992	1.17
x264	3	1023	1.18
(binary size 99MB)	4	1056	1.20
	5	1214	1.21

Table 5.4. Running time for *freud-dwarf* for three targets and different depths of feature exploration

The size of the binary is usually correlated with the amount of debugging information that is stored in the binary itself. Notice the difference between the sizes of the MySQL binaries for version 5.7.24 and version 8.0.15. A similar difference in the time required to parse these binaries can be noticed when using *gdb* to collect debugging information.

On the other hand, notice also that the `x264` binary is still of considerable size, and yet the time required for the static analysis is significantly lower. The main factor defining this difference is the absence of class objects among the potential features. Thus, freud-dwarf does not need to perform a complete exploration of the debugging information tree to build the complete class graph.

Table 5.5 shows the time taken by freud-statistics when analyzing the same two target methods `mysql_execute_command` and `x264_8_encoder_encode` of MySQL 5.7.24 and x264, respectively. A first, high-level consideration is that even with real-world complex software, with thousands of observations and several hundreds candidate features, freud-statistics completes the analysis in minutes, which we consider to be a reasonable time.

target	samples	candidate features	candidate branches	regression features	time (s)
MySQL$^{(a)}$	3038	655	34	8	124.00
x264$^{(b)}$	4251	992	13	66	232.47
x264$^{(c)}$	7649	992	13	72	807.14

Table 5.5. Statistics and performance (running time) of the statistical analysis for (a) the `mysql_execute_command` method of MySQL 5.7.24, (b) the `x264_8_encoder_encode` method of x264 with a sampling rate of 20 samples per second, and (c) the `x264_8_encoder_encode` method of x264 with a sampling rate of 50 samples per second.

Second, we observe that our pre-processing algorithm that filters out uninteresting feature candidates prior to the regression analysis is effective in reducing the number of features that are then considered in the computation of the regressions. For example, in the case of the `mysql_execute_command` method of MySQL, the algorithm selects 8 regression features out of 655 potential features identified through the static analysis.

Lastly, we look at the number of samples, which is the main size parameter for the analysis performed by *freud-statistics*. The second and third lines in Table 5.5 characterize the analysis on the same target `x264_8_encoder_encode` and with the same input but with different sampling rates. We first use the default sampling rate of 20 samples per second (see Table 5.3) and then about double that rate, resulting in a total of 4251 and 7649 samples, respectively. The results of Table 5.5 clearly show an increase in the analysis time. However, we also report that in terms of results for this case study, we do not need the higher sampling rate. A few hundreds of observations are often sufficient to produce informative performance annotations even in complex cases, with multi-modal performance behaviors affected by many different features and branches. In fact, all the performance annotations that Freud was able to produce for x264 that we show later in Chapter 6 were produced with the default sampling rate, and in all those cases we obtain the same result as with double that rate.

5.5 Other Contributions

Freud uses a C++ library to help parsing the debugging information. This library is open sourced under the MIT license. During the development of Freud, we expanded the library to handle the most common information that might be

found in the debugging information produced by gcc/g++. Some of the missing features that we added during the development of freud are eh_frames. In its current state, Freud can use the debugging information to analyze the vast majority of the debugging information for most if not all real-world C/C++ program. However, some rare constructs are still not handled. One such example is `expr_loc`.

Chapter 6

Evaluation

In this chapter we evaluate Freud and more generally the idea of performance annotations. To do so, we demonstrate the use of Freud on three real-world, complex software systems. The first system is MySQL, a well-known and widely used DBMS written in C/C++. The second system is the x264 library and application for encoding video streams into the H.264/MPEG-4 AVC compression format, which is written in C. We also report on some early experiments that show the use of performance annotations on a third complex software system written in a different programming language, namely ownCloud, a remote storage web application written in PHP.

We use Freud (an early prototype in the case of PHP) to derive performance annotations for all three systems for several metrics, including running time, dynamic memory allocation, and lock holding/waiting time.

For the case studies, the goal of the evaluation is demonstrative rather than quantitative. We show that Freud outputs performance annotations for many different software components of the three systems, that are easy to read and interpret for the performance analyst. Even though these three systems are very different in their functionalities, programming languages used, and deployment, we successfully applied the same analysis techniques to all of them. We intentionally choose functions to analyze and present that exhibit different features of performance annotations and/or interesting and sometimes counter-intuitive behaviors. We go from simple to progressively more articulate cases.

In summary, we demonstrate that Freud correctly identifies many interesting behaviors; in particular that Freud is effective even in cases where the performance depends on multiple features or on specific internal conditions (parameters or state) or external conditions (environment); that all such behaviors can be meaningfully described by relatively simple annotations that relate input features

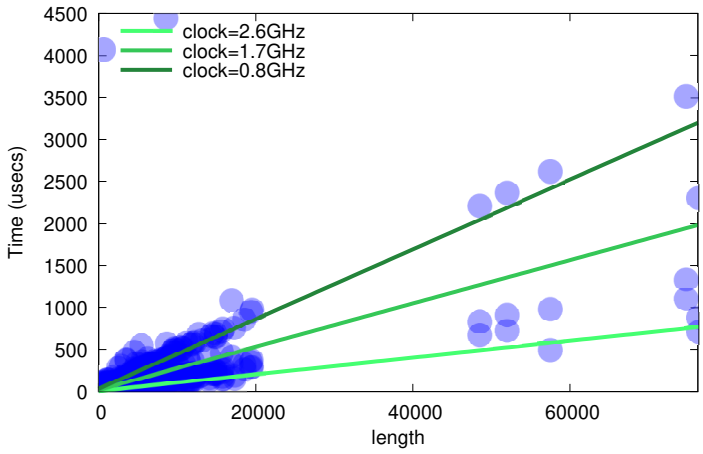
and measured metrics; that such annotations can be used not only as documentation but also to detect performance regressions and anomalies; that Freud and its annotations can be used to diagnose non-trivial performance bugs.

We do not evaluate Freud in terms of the *types* of performance annotations it finds, in particular we do not count or classify the methods for which Freud finds regressions as opposed to clustered annotations. This kind of evaluation is more pertinent to the workload than to Freud, since regressions by definition require a meaningful coverage of the relevant input features, which in turn depends on the workload. In our evaluation we analyze various methods, ranging from small utility methods to high level and complex methods, all taken from large software systems. Since most of these methods run deep within the execution of the system, we have limited control on the specific input taken by such methods, and in any case we do not artificially forge workloads to tweak specific input features. Instead, we intentionally use the systems with limited but realistic inputs and deployments.

The structure of the chapter is as follows: in Section 6.1 we create performance annotations for x264, which is a multi-threaded, heavily optimized program. In this Section we show how Freud can efficiently document the performance of complex methods, that are affected by different features. Performance annotations are of great benefit in understanding the behavior and the multi-threading model of complex, highly optimized and parallel, software. In Section 6.2 we use Freud to investigate the performance of MySQL. Notably, we use performance annotations to investigate bugs that affect different versions of MySQL, and to catch and quantify a performance regression that affects newer versions of MySQL. In Section 6.3 we show some preliminary results that demonstrate the use of the same performance analysis on a completely different system: we produce performance annotations for ownCloud, which is written PHP. We show that we can use the performance annotations to catch a performance anomaly that we deliberately introduce in a distributed system.

6.1 x264

For the first case study, we use the most recent version of x264 at the time of writing (git commit 5493be8). x264 is an open-source library and utility for H.264/MPEG-4 video encoding [48]. We run all experiments on a 4-core, 8-thread Intel Core i7-6700HQ CPU at 2.60GHz, in a system equipped with 16GB of RAM and a NVMe SSD drive. We compile *x264* with gcc 8.3 on a Ubuntu 18.10 64-bit system.




```
ff_h2645_extract_rbsp(const uint8_t *src, int length, H2645RBSP *rbsp,
    H2645NAL *nal, int small_padding).time {
features:
 int l = length;
 int clock = system.cpu_clock;

annotations:
 Norm(43.32 + 0.055*l - 1.46e-05*clock - 1.75e-08*l*clock, 4.56);
}
```

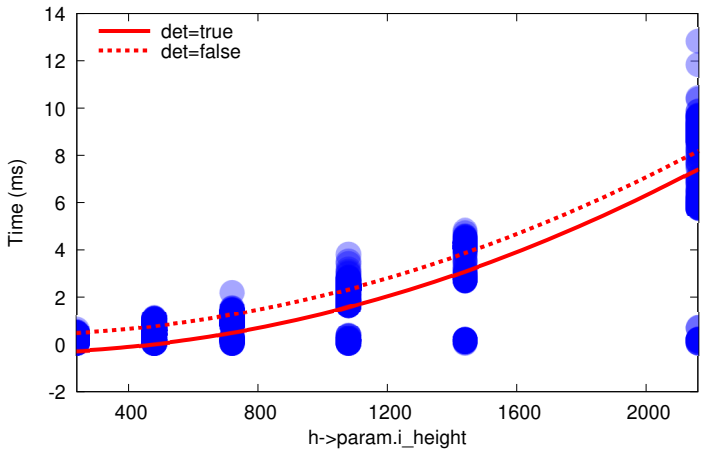
Figure 6.1. `ff_h2645_extract_rbsp`: running time.

As a workload, we use *x264* to convert or re-encode six different videos (different content) from either VP9 or H.264 to H.264. We re-encode each video at the original resolution. All videos have the same aspect ratio (16:9) but different vertical resolutions, ranging from 240p to 2160p, and different frame rates (25–30fps). We then run the experiments with 2, 4, 8, and 12 threads (default is 12). We also enable or disable the *sliced-threads* option to test both the *slice-based* and *frame-based* processing of *x264*. Finally, for some experiments we also use the Intel p-state driver to change the CPU clock speed between 800MHz, 1.6GHz, and 2.6GHz.

Selected Results. We start with `ff_h2645_extract_rbsp`, a utility method that extracts a raw bit stream from an h264 source buffer. Freud derives an annotation (Figure 6.1) that indicates that the running time increases linearly with the size of the input buffer. Moreover, Freud also finds an interaction be-

tween the *size* and *cpu_clock* features. Notice that *cpu_clock* is a system feature. Qualitatively, the annotation indicates that the running time is CPU-bound.

Proceeding now to the core of the encoding functionality, we analyze the `x264_8_encoder_encode` method. We choose not vary the CPU speed for the sake of clarity, since there are already many parameters that affect performance. `x264_8_encoder_encode` drives the encoding process by managing the pool of worker threads that perform the actual encoding. In *slice-based mode*, the method runs once for every output video frame; splits the frame in many slices; assigns those to the worker threads for processing; and waits for them to complete the frame. Every output frame is completed before the next one is processed. Conversely, in *frame-based mode*, `x264_8_encoder_encode` processes a set of frames at a time, assigning an entire frame to each worker thread.




```
x264_8_encoder_encode(x264_t *h, x264_nal_t **pp_nal, int *pi_nal,
    x264_picture_t *pic_in, x264_picture_t *pic_out).time {
features:
 int h = h->param.i_height;
 int d = pic_in->param.b_deterministic;

annotations:
 Norm(394.27 + 1.67e-03*d - 0.41*h + 2.88e-05*h^2, 14.96);
}
```

Figure 6.2. `encoder_encode`: running time without context switches.

We first run Freud so as to filter out the measurements affected by any context switch (voluntary or not). This is an intentional bias to remove some noise and

complexity, and to show that the actual work performed grows as expected with higher resolutions, as shown in Figure 6.2.

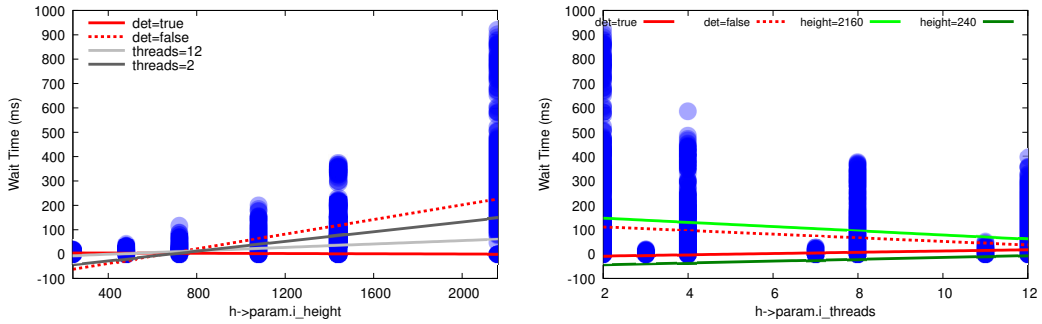
Interestingly, without this filter we see a different behavior. The running time of `x264_8_encoder_encode` is much higher, and the correlation with the video resolution is not as clear. We therefore analyze the time spent waiting on a condition variable, and find that it completely dominates the total running time. Freud, with branch analysis, finds that the waiting time in *sliced-mode* (when `h->param.b_sliced_threads=1`) correlates well with the number of threads and the video resolution (Figure 6.3). The waiting time grows with the resolution of the video, and is lower with more threads. We conclude that indeed `x264_8_encoder_encode()` always waits for the worker threads to finish processing their slices.

When the processing is *frame-based*, instead, there is little or no correlation between the collected features and the run time, and therefore Freud fails to find a good regression and instead performs a cluster analysis. The resulting model (condition `!sliced` in Figure 6.3) is still informative and shows that the majority of the *frame-based* executions of `x264_8_encoder_encode` wait for a very short time.

Moving to another analysis, `slice_write` is one of the most time consuming functions and is executed by the worker threads in all the processing modes (Figure 6.4). In this case Freud shows that having more threads has opposite effects on the running time depending on the processing mode. Also, Freud's annotations reveals that `x264_8_encoder_encode` is not synchronizing on each execution of `slice_write` in *frame-based* mode, since for some inputs `slice_write` has a much higher running time (Figure 6.4) than any waiting time observed for `x264_8_encoder_encode` (Figure 6.3).

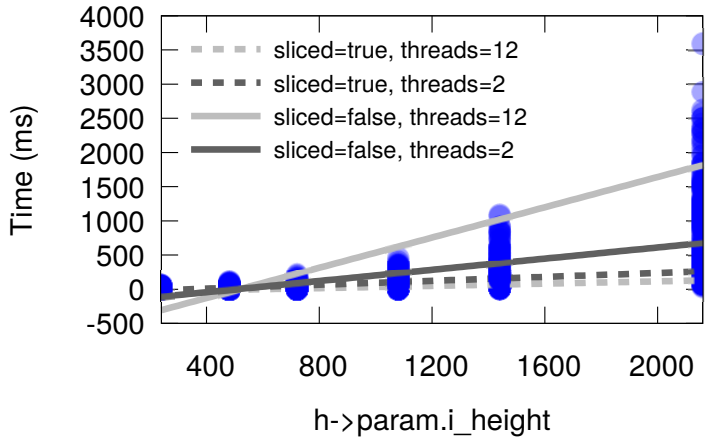
6.2 MySQL

In the second case study, we use Freud to investigate two performance bugs reported on the MySQL bug tracker (n. *94296* and *92979*). To reproduce these bugs we use different versions of MySQL. We compile the unmodified code cloned from the MySQL github repository [31] with gcc 7.3 on a Ubuntu 18.04 64-bit machine. We pass the `-g` and `-gstrict-dwarf` flags to the compiler to add standard DWARF debugging symbols in the ELF mysqld binary. We run the experiments on a Intel Xeon CPU E5-2670, with 64GB of ram, whose root partition is mounted on a Samsung SSD 850 PRO drive. We run the MySQL server with the default configuration. We inject the workload locally using the MySQL client




```
x264_8_encoder_encode(x264_t *h, x264_nal_t **pp_nal, int *pi_nal,
    x264_picture_t *pic_in, x264_picture_t *pic_out).wait_time {
features:
 bool sliced = h->param.b_sliced_threads;
 int height = h->param.i_height;
 int threads = h->param.i_threads;
 int dequant = h->thread.dequant4_mf;
 bool det = pic_in->param.b_deterministic;

annotations:
 [sliced]
 Norm(-56362 + 189.17*height - 3221.21*threads - 1378.66*dequant
       - 152.83*height*det - 6.48*height*threads + 10044*threads*det
     , 1.05e+05)
 [!sliced]
 {0.55}Norm(108.7, 188.65);
 {0.30}Norm(7282, 51465.24);
 ...
}
```

Figure 6.3. encoder_encode: wait time seen from two different features




```
slice_write(x264_t *h).time {
features:
 bool sliced = h->param.b_sliced_threads;
 int threads = h->param.i_threads;
 int height = h->param.i_height;

annotations:
 [sliced]
 Norm(-1.34e+05 - 3.65e+04*threads + 269.17*height + 69.76*threads*
     height, 2.51e+04);
 [!sliced]
 Norm(-4.94e+04 + 155.76*height - 6.17*height*threads, 4.87e+03);
}
```

Figure 6.4. `slice_write`, sliced vs. framed processing.

application on the same host that runs the server. We use modified versions of the workloads attached to the bug reports. We modify the workloads to obtain a greater variety for some feature values.

Bug 92979. The report describes a performance regression in MySQL 8 as compared to version 5.7 for a specific insertion workload. The regression has been verified but not fixed by the developers. The root cause is not known.

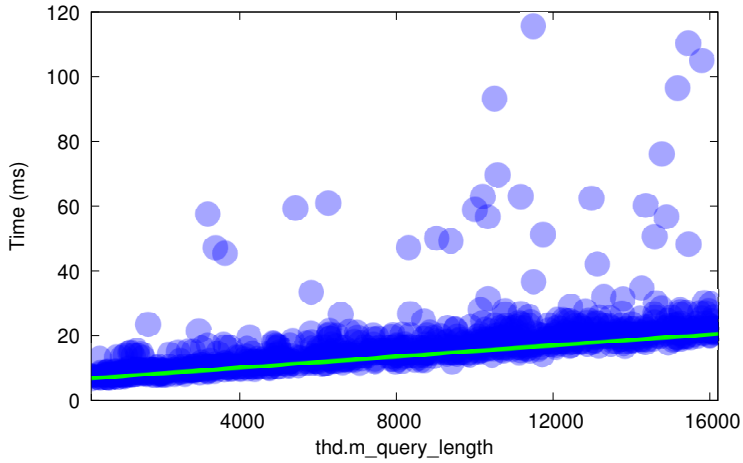
We replicate and analyze the bug using versions 5.7.24 and 8.0.11. As a workload, we use the MySQL dump attached to the bug report. The dump consists of a set of `INSERT` operations for a specific table. Using this set as a basis, we create a workload with a series of `INSERT` operations each inserting an increasing number of rows. As an entry point for our investigation, we instrument the high level function `mysql_execute_command`, which is present with the same signature in both versions.

Figure 6.5 shows the annotations and corresponding graphs generated by Freud for MySQL versions 5.7.24 (top) and 8.0.11 (bottom), respectively. The annotations and the graphs evidence the performance regression. Freud identifies the size of the input query as a relevant features, and formulates a linear performance model for both versions. However, Freud finds a significantly higher linear coefficient for version 8 than for version 5 (4.94 vs. 0.86).

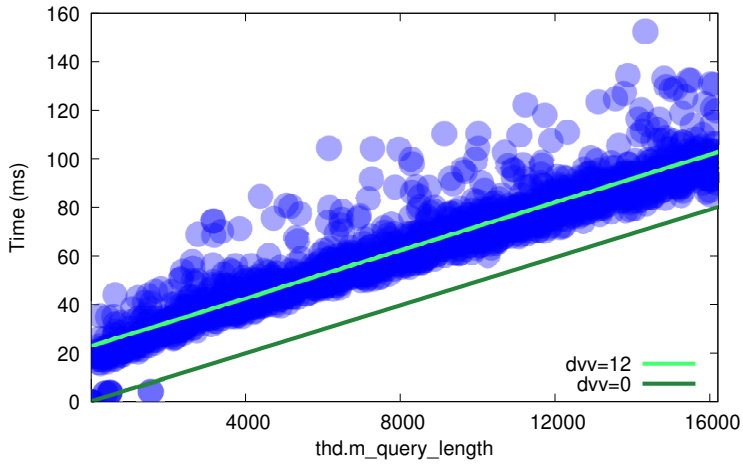
Notice that the running times for version 5 are lower, and therefore the measurements are affected in a greater proportion by other factors, such as the storage access times. Freud treats those factors as additive noise as discussed in Section 4.2.2. Also, unlike version 5, MySQL version 8 performs some additional startup operations during the execution of `mysql_execute_command` whose running time correlates with another input feature (`dynamic_variable_version`), as evidenced by the multiple regression found by Freud.

Bug 94296. Bug n. 94296 reports a difference in the execution time of functionally identical `SELECT` queries that use different operators. The performance is found to be worse when using a series of `IN` operators instead of a disjunction of conjunctions. The bug is marked as fixed in MySQL 8. However, our analysis with Freud demonstrates that the bug is still present in version 8.0.15.

The workload consists of two `SELECT` queries provided with the bug report. As with Bug n. 92979, we split the workload into multiple queries of increasing sizes. The queries are simple selections on a single table `t` (`SELECT * FROM t WHERE ...`) with the same logical condition on two columns $c1$ and $c2$ expressed with two different `WHERE` clauses: one with the `IN` operator, $(c1, c2)$ `IN`




```
mysql_execute_command(THD *thd, bool first_level).time {
features:
 int len = thd->m_query_string.len;

annotations:
 R Norm(6630.19 + 0.86*len, 15.78);
}
```

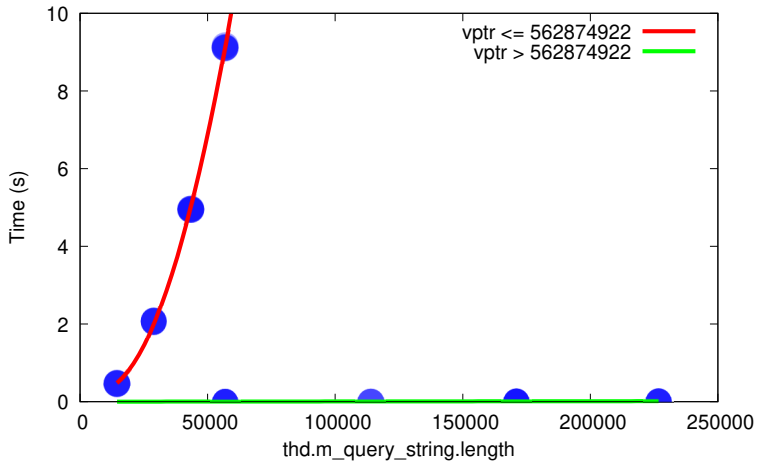



```
mysql_execute_command(THD *thd, bool first_level).time {
features:
 int len = thd->m_query_string.len;
 int dvv = thd->variables.dynamic_variable_version;

annotations:
 Norm(168.65 + 4.94*len + 1886.87*dvv, 2489.04);
}
```

Figure 6.5. `mysql_execute_command`, 5.7.24 (top) vs. 8.0.11.

$((v1_1, v2_1), \ldots, (v1_n, v2_n))$, the other with `AND/OR`, $(c1 = v1_1$ `AND` $c2 = v2_1)$ `OR` `...OR` $(c1 = v1_n$ `AND` $c2 = v2_n)$.

We start our analysis from `test_quick_select()`, which we find as the top-level function in MySQL that processes all the queries of the given workload. We use Freud to analyze the running time of `test_quick_select()`, obtaining the performance annotations shown in Figure 6.6.

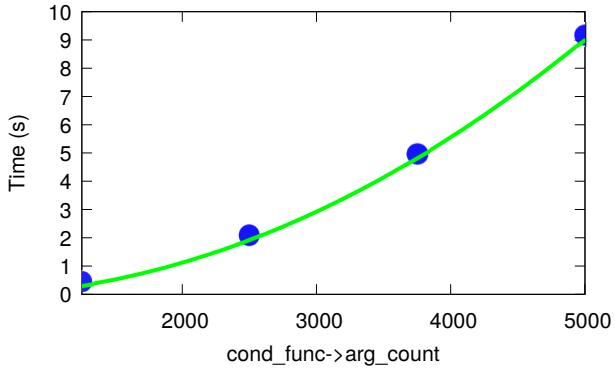



```
test_quick_select(THD * thd, Key_map keys_to_use, table_map prev_tables
    , ha_rows limit, bool force_quick_range, const enum_order
    interesting_order, const QEP_shared_owner * tab, Item * cond,
    Key_map * needed_reg, QUICK_SELECT_I ** quick).time {
features:
 int len = thd->m_query_string.len;
 int vptr = cond->_vptr.Parse_tree_node_tmpl;

annotations:
 [vptr <= 562874922]
 Norm(467533 - 50.21*len + 0.0036*len^2, 282711.59);
 [vptr > 562874922]
 Norm(-53.603 + 0.057*len, 157.57);
}
```

Figure 6.6. `test_quick_select()`: `IN` vs `AND/OR` query.

Freud automatically distinguishes two different behaviors that depend on the query type. `test_quick_select()` takes a `cond` parameter that is statically seen as a structure of the generic class `Item`, although its actual type is a different

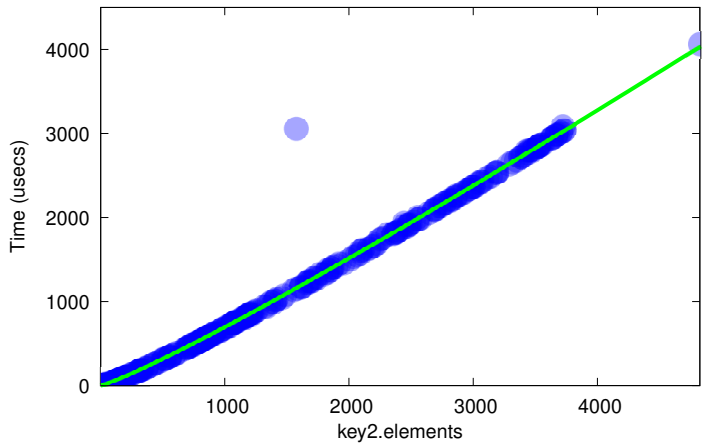
subclass when the query uses the IN operator and the AND/OR. The virtual table pointer, which Freud considers as a feature (logged as an integer), distinguishes the actual type and their different behaviors. The time grows linearly with the AND/OR operator (which produces longer query strings), but grows quadratically with the IN operator.




```
get_func_mm_tree(RANGE_OPT_PARAM * param, Item * predicand, Item_func *
     cond_func, Item * value, bool inv).time {
features:
 int ac = cond_func->arg_count;

annotations:
 Norm(156569 - 269.041*ac + 0.414447*ac^2, 15781.22);
}
```

Figure 6.7. `get_func_mm_tree`: arg_count feature

Through a simple manual inspection, we follow a chain of functions executed in the case of `IN` queries. We then analyze these functions with Freud to find the origin of the quadratic behavior. This analysis first points to `get_func_mm_tree`, that is executed only in the case of queries using the IN operator. In Figure 6.7 we see that the main feature affecting the quadratic behavior is *arg_count*, which is a field of the class *Item_func*. Going deeper with the analysis, we analyze `tree_or()` that in turn calls `key_or()`. `key_or()` computes the logical disjunction of two keys encoded with two RB-trees. Freud reveals that `key_or()` has a linear complexity but is called repeatedly, as many times as there are clauses, with a `key2` parameter that progressively grows in size (Figure 6.8) from zero to the number of clauses in the query. This arithmetic progression explains the overall quadratic behavior.



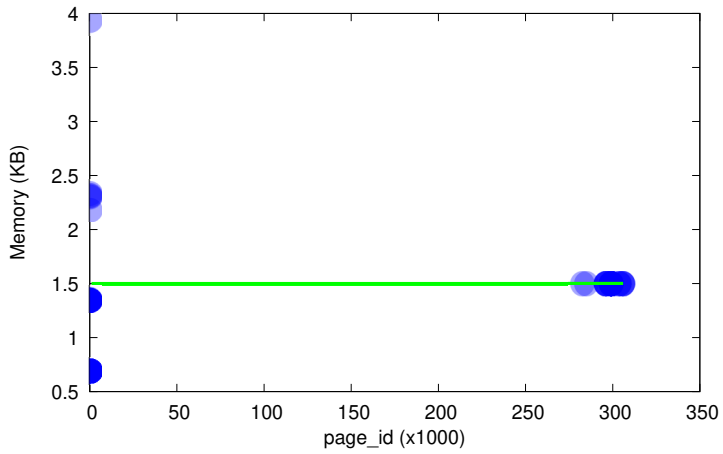

```
key_or(RANGE_OPT_PARAM * param, SEL_ROOT * key1, SEL_ROOT * key2).time
    {
features:
 int e = key2->elements;

annotations:
 Norm(-0.276 + 0.073*e + 0.062*e*log(e), 2.24);
}
```

Figure 6.8. Arithmetic progression of `key_or()`.

Reading the well documented code of `key_or()`, we find that `key_or()` is designed to compute a more general disjunction of *ranges,* as opposed to the specific values in the reported workload. This may suggest a specialized implementation and therefore a more radical bug fix.

Other metrics and other bi-modal behaviors. The annotation for method `fseg_create_general` (Figure 6.9) shows a case in which the behavior is bi-modal depending on a certain condition. If the `page_id` parameter is greater than 0, then the method allocates a fixed amount of memory. Conversely, if `page_id` is equal to 0, then the method allocates a variable amount of memory, which we could not predict successfully with the features collected for the experiment.

A code inspection confirms that `fseg_create_general()` contains a switch that follows different execution paths depending on the value of `page_id`. If `page_id` is equal to 0, the method allocates new memory for a new `segment`.




```
fseg_create_general(space_id_t space_id, page_no_t page, ulint
    byte_offset, ibool has_done_reservation, mtr_t * mtr).mem {
features:
 int page_id = page;

annotations:
 [page_id > 0]
 Norm(1496, 0);
}
```

Figure 6.9. `fseg_create_general`: branch analysis.

Otherwise the method returns an already allocated buffer, and performs other operations, which cause some memory consumption.

6.3 ownCloud

Finally, we run some experiments in a completely different scenario: we created a scaled-down replica of a real-world data center that runs a cloud application called SWITCHdrive. SWITCHdrive is a file-hosting cloud service similar to Dropbox operated by SWITCH, the national ISP for academic institutions in Switzerland. Our replica runs on 12 instead of 41 servers, but it is otherwise identical in its structure and configuration, including applications, web servers, storage layers, virtualization stack, hosts, and network. We focus our analysis on the ownCloud application, which is written in PHP.

We use the WebDAV filesystem interface (*davfs*) provided by ownCloud to

mount a user folder on a spare client machine. We then use this folder to apply two workloads. The first runs an rsync operation that copies a large tree of directories and files (the complete Linux kernel source) to the mounted directory. The second workload consists of more than 100 thousand requests involving files of different sizes, name-lenghts, and relative path lengths. To improve the clarity of the visualization without losing generality, we limit our presentation to about two thousand requests.

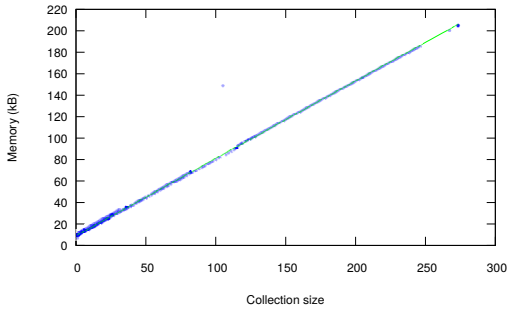
Selected Results. At a basic level, performance annotations provide human readable documentation of the actual performance of methods. Figure 6.10 shows some interesting behaviors that exemplify common correlations (or lack thereof) found in ownCloud. The x-axis indicates the relevant feature; the y-axis indicates the measured metric.

The first two graphs show annotations for `generateMultiStatus`, a method that takes an array of file properties. The method iterates over the objects in the array, computing some data about each one. The graph on the left clearly shows that memory usage is linearly correlated to the size of the input array. As it turns out, this linear behavior can be clearly deduced from the code. The data is also perfectly linear because the chosen metric (memory usage) is not affected by noise. The graph in the center shows the run time, which is more noisy. But again, a regression produces a fitting performance model, which in this case is quadratic. Although not immediately obvious, this quadratic behavior indeed corresponds to the algorithmic complexity of the code.

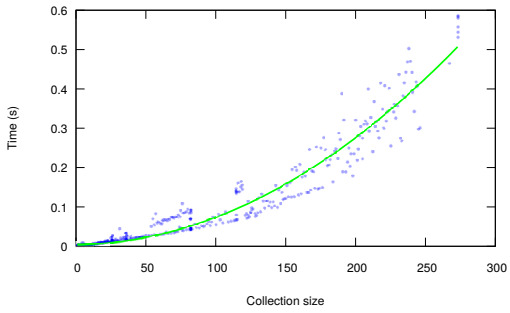
Finally, the graph on the right shows the run time for `emit_file_hooks_pre`, which is used to invoke callback functions associated with file events. The graph shows the running time over string-length of the `path` parameter. However, in this case Freud did not find good regressions with this or any other feature, and therefore formulated an annotation based on clusters of the running time metric.

Use of Annotations in Anomaly Detection. Having derived several performance annotations, we ask whether these annotations would serve developers and system operators beyond their value as documentation. In particular, we try using annotations as assertions and therefore as failure or anomaly detectors. We then verify that such detectors are sensitive to real anomalies at the same time as they are robust with respect to different workloads.

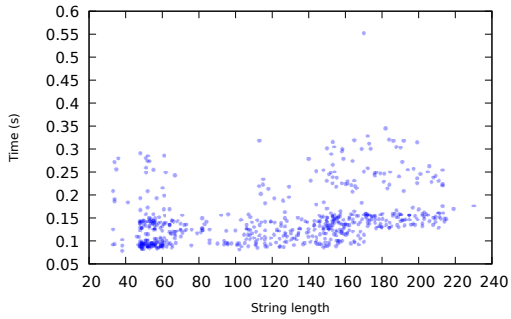
We proceed as follows: we first derive annotations using the two workloads described above. We then run the same workloads in a special setting in which we artificially introduce an anomaly in the system. In this setting, we use anno-




```
generateMultiStatus(
   array $fileProperties,
   $strip404s = false
).mem {
features:
 int s = count(fileProperties);

annotations:
 Norm(721.362*s + 8851.16, 2324.567)
     ;
}
```




```
generateMultiStatus(
   array $fileProperties,
   $strip404s = false
).time {
features:
 int s = count(fileProperties);

annotations:
 Norm(0.0066*s^2 + 0.000038*s +
     0.0000036, 0.022);
}
```




```
emit_file_hooks_pre($exists, $path,
     &$run).time {
features:

annotations:
 {0.77}Norm(0.13, 0.00075);
 {0.23}Norm(0.24, 0.001);
}
```

Figure 6.10. Some annotations for ownCloud: linear and quadratic regressions (top, middle) and clusters (bottom)

tations as standard one-sample statistical tests to compare measured metrics to the idealized model given by the annotation. We record an assertion violation whenever the test indicates that the measurements do not conform to the annotation. For each run, we then count the number of assertions passed (or failed) for all the instrumented methods.

As an anomaly, we introduce an artificial network latency between the virtual machine hosting the database server and the rest of the cluster, varying the delay from 0 to 10 milliseconds. The case of 0ms serves as a robustness check of the assertions generated during training.

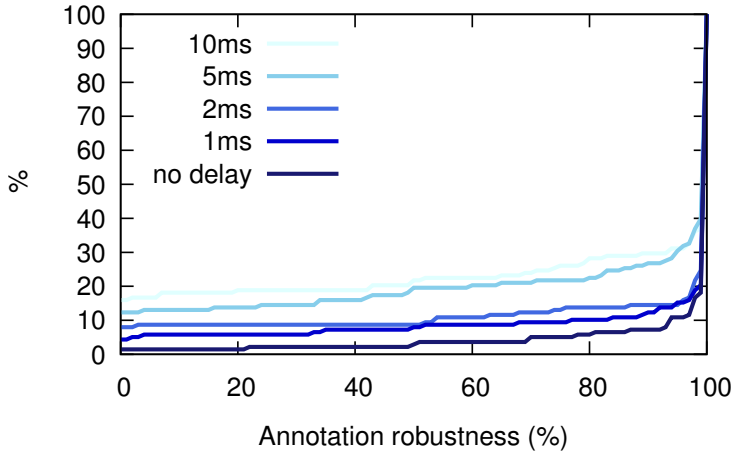


Figure 6.11. Robustness, use of annotations to detect anomalies

Figure 6.11 shows the cumulative distribution function of the passed assertions as a percentage of all assertions. The graph shows that assertion failures clearly expose a change in the performance behavior of the system, thereby signaling a performance problem. The overall difference may appear small. Indeed, we only tweak the behavior of the database component, leaving all the other components in their original configurations. This means that only a fraction of the 138 methods analyzed are affected by the slow database. In fact, we observe that the methods whose annotations are violated more often in the different experiments are those that directly or indirectly involve database operations.

Chapter 7

Conclusion and Future Work

This thesis introduces probabilistic performance annotations as a new tool for software performance analysis and specification.

Performance annotations describe the observed performance behavior of software components with concrete performance metrics, such as the running time, expressed in seconds, or the memory usage, expressed in bytes.

Differently from traditional profilers, which report only aggregate information, performance annotations describe performance with cost functions that correlate a performance metric with one or more features. Features represent either information internal to the program under analysis, such as values of parameters or variables, or external information coming from the system where the target program is running, such as the CPU clock speed or the number of virtual cores.

Such correlations allow performance annotations to extrapolate and therefore predict the performance of software components beyond what has actually been observed. Also, performance annotations can be used to make assertions about the observed behavior of software that can then be used as oracles in performance testing and analysis. For example, it is possible to compare the expected performance with the observed behavior to find performance regressions between different versions of the software.

In addition to the notion of performance annotations, we have also introduced Freud. Freud is a tool that automatically creates meaningful performance annotations for `C/C++` software. Freud is quite simple to use: the performance analyst selects some methods to be analyzed along with one or more performance metrics. If necessary, the analyst also provides a specific workload, or otherwise runs the program normally in a laboratory environment but also possibly in the production environment. Beyond that, the performance analysis is completely automated, and produces performance annotations describing the expected be-

havior of software methods, in the form of text files and plots.

We have shown experimentally that Freud produces meaningful and useful performance annotations automatically, even for very large and complex software such as MySQL and the x264 codec. Still, the ideas introduces in this thesis are intended to establish a new approach to performance analysis, and we see Freud as a first step in that direction that also leaves many new lines of research open.

One of the problems that we did not discuss in this thesis is *workload generation*. The richness of our performance annotations is directly correlated to the richness of the performance behaviors that the instrumentation could observe on the real software. For example, if we identify a potential feature that seems to be correlated with performance, but that feature does not have enough variability to give a strong p-value in our regression analysis, then we cannot confidently use such feature in the corresponding performance annotation. If on the other hand we could generate a workload to expose observations with more variability for such feature, then we would be able to prove (or disprove) the hypothesized correlations. The generation of such workloads is an interesting research direction.

Another interesting research direction that we only briefly touched upon is that of *compositional* annotations. With this term, we refer to the problem of automatically creating performance annotations in which we use references to performance annotations of other methods. Such annotations could describe not only methods that we could observe through the instrumentation, but also methods that were never observed or executed. In the case in which we want to predict the performance of a software method without ever observing it, we are not applying a dynamic analysis as we do with Freud, but instead we need to perform a *static* analysis. In other words, we need to analyze the code of the method to infer the way in which the method uses and interacts with other methods with known performance behaviors and annotations. Some techniques that might be helpful in researching such static analysis approach include probabilistic symbolic execution, probabilistic programming, and Monte Carlo analysis.

With the ownCloud example we have shown that performance annotations can be used even in complex distributed software that runs at many different layers and on different physical machines. This example suggests a research direction that would expand on specific instrumentation for distributed systems. The base observation is that most of the operations happening in distributed systems running in data centers require the interaction of different physical or virtual machines. Such interaction is usually concretely implemented through RPC calls. From the viewpoint of the performance analyst, the performance of

a single method running on a specific system depends also on the performance behavior of other methods possibly running on different systems. In addition, the performance analyst might want to assign the entire performance costs generated in the data center to the execution of one specific method in one single node. To be able to perform this kind of analysis, we need to be able to correlate events happening in different parts of a data center. The correlation should be defined on the basis of a causality-relation, in which the execution of some methods on one node in the data center is the effect caused by the execution of some other method on another node. Finding this type of correlations requires a *distributed instrumentation*. Such instrumentation might inject unique traces ids in the RPC requests of the data center. Examples of this distributed instrumentation are Google Dapper [38], or Zipkin [43].

Finally, there are some more concrete directions to explore: on the one hand, it would be interesting to development more instrumentation modules for Freud, to instrument programs written in other programming languages, such as Java or Python. The development should benefit from the well defined requirements and interface for the format of the logs, described in this thesis.

Another concrete direction for future work would be optimization work on the current instrumentation of Freud. As we have shown in Figure 5.6, there is a considerable difference in the overhead when the instrumentation is logging all the potential features, all the branches, and all the metrics, as opposed to the case in which the instrumentation is reducing the collection and measurements to what is strictly necessary to produce specific performance annotations. The overhead might be a problem in production environments, in which we instrument software that is running in a system (e.g. a data center) that is serving real user requests. On the other hand, in the current implementation there is no way to know which features and branches will be useful to the statistical analysis in the creation of the performance annotation, and therefore the instrumentation collects every possible source of information. One optimization that we could apply is to feed the result of the statistical analysis back to the instrumentation. With this information, the instrumentation could enable or disable the collection of specific features and branches and also adjust the sampling rate *on-the-fly*. In other words the idea is to dynamically control 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 is to reduce the overhead of the instrumentation to the point that Freud can be safely used with systems running in production environments.

Appendix A

Microbenchmark

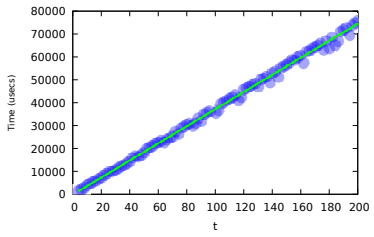
In this Appendix we describe in detail the micro-benchmark introduced in Section 5.4.1. We use this micro-benchmark to validate Freud and the results of the performance analysis. For each function of the micro-benchmark we report (1) the complete code of the function, (2) the performance annotations that Freud produced, and (3) the running time for the statistical analysis. We also discuss briefly why each specific function is part of the micro-benchmark, and what functionalities of Freud are being tested.

The instrumentation of the micro-benchmark binary program including all the functions below took *0.019 seconds* on a Intel Xeon CPU E5-2670, using a single CPU core. For each function of the micro-benchmark, we find three to five feature candidates.

We collect 149 observations for each function. Each function is executed 150 times, but the first one is automatically discarded because it is affected by JIT recompilation.

A.1 Function `test_linear_int`

```
void test_linear_int(int t) {
   for (int i = 0; i < t; i++) {
      usleep(250);
   }
}
```



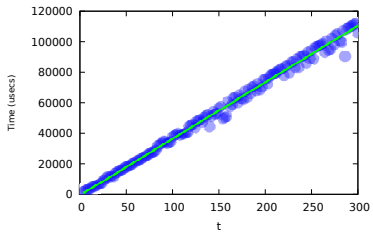

```
void test_linear_int(int t).time {
features:
 int i = t;

annotations:
 Norm(-754.85 + 373.86 * t, 1306916);
}
```

(Time for statistical analysis: 0.379s)

In this function we have a linear correlation between the int parameter, passed by value, and the running time. We test both the correctness of the statistical part in finding the linear correlation, and of the instrumentation in reading feature values from the CPU registers. With the default `gcc` optimization level (i.e., `-O2`) the integer parameter is stored in a CPU register.

A.2 Function `test_linear_int_pointer`

```
void test_linear_int_pointer(int *t) {
   for (int i = 0; i < *t; i++) {
      usleep(250);
   }
}
```




```
void test_linear_int_pointer(int *t).time {
features:
```

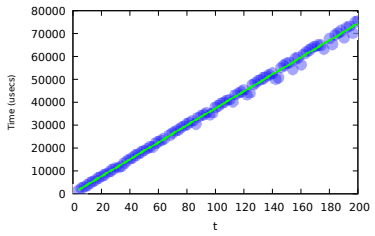
```
 int t = *t;

annotations:
 Norm(-1454.12 + 370.27*t, 8560724);
}
```

(Time for statistical analysis: 0.370s)

This function is really similar to `test_linear_int`, with the only difference being the location of the relevant feature. The parameter `t` is now passed as a pointer, which means that Freud will find the address of the feature in a CPU register.

A.3 Function `test_linear_float`

```
void test_linear_float(float t) {
   for (int i = 0; i < (int)t; i++) {
      usleep(250);
   }
}
```




```
void test_linear_float(float t).time {
features:
 int t = t;

annotations:
 Norm(371.30 * t, 1141729);
}
```

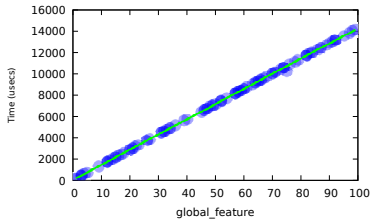
(Time for statistical analysis: 0.388s)

A linear correlation between the float parameter, passed by value, and the running time. With the default gcc optimization level, float parameters are stored

in the CPU XMM registers, when processor have such registers. This requires the instrumentation to be able to read packed values from such 128bit registers.

A.4 Function test_linear_globalfeature

```
void test_global_feature() {
   for (int i = 0; i < global_feature; i++) {
      usleep(70);
   }
}
```




```
void test_global_feature().time {
features:
 int g = global_feature;

annotations:
 Norm(33.15 + 142.48 * g, 6796);
}
```

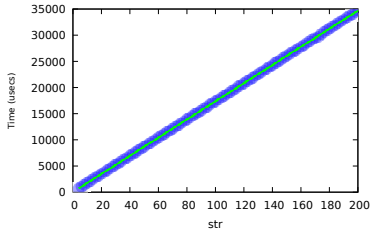
(Time for statistical analysis: 0.369s)

A linear correlation between an integer global variable, and the running time. Global variables are found at static addresses in the program memory space. This test requires *freud-dwarf* to be able to correctly identify global variables in the scope of selected methods, and *freud-pin* to be able to read values from absolute addresses.

A.5 Function test_linear_charptr

```
void test_linear_charptr(char * str) {
   for (int i = 0; i < strlen(str); i++) {
      usleep(100);
```

```
   }
}
```



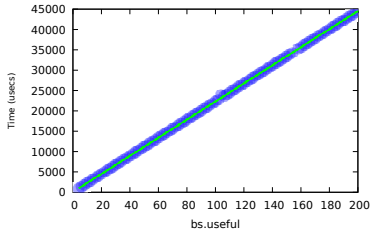

```
void test_linear_charptr(char * str).time {
features:
 int s = strlen(str);

annotations:
 Norm(-59.69 + 173.45*s, 785);
}
```

(Time for statistical analysis: 0.383s)

A linear correlation between the length of a zero-terminated C string and the running time. The memory reads in the native expression are wrapped with Pin_SafeCopy in the feature extraction code. `strlen` is a special feature that Freud implements through heuristics (the behavior of which can be controlled with compile time switches). When Freud finds a char pointer, it tries to compute the length of the C string pointed to by the pointer.

A.6 Function `test_linear_structs`

```
void test_linear_structs(struct basic_structure * bs) {
   for (int i = 0; i < bs->useful; i++) {
      usleep(150);
   }
}
```



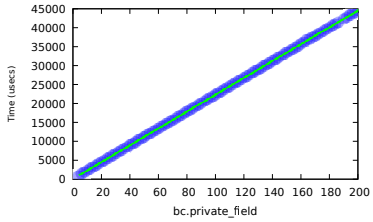

```
void test_linear_structs(struct basic_structure * bs).time {
features:
 int u = bs->useful;

annotations:
 Norm(-8.45 + 222.43 * u, 21147);
}
```

(Time for statistical analysis: 0.386s)

A linear correlation between the length of a field of a structure that is passed by reference through a pointer and the running time. With this method, we test the exploration of C structures performed by *freud-dwarf*.

A.7 Function `test_linear_classes`

```
void test_linear_classes(basic_class * bc) {
   int pf = bc->get_private_field();
   for (int i = 0; i < pf; i++) {
      usleep(150);
   }
}
```




```
void test_linear_classes(basic_class * bc).time {
features:
 int p = bc->private_field;
```

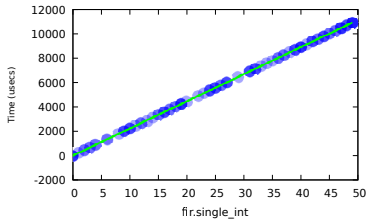
```
annotations:
 Norm(0.33 + 222.45 * p, 5753);
}
```

(Time for statistical analysis: 0.401s)

A linear correlation between the length of a private field of a class that is passed by reference through a pointer and the running time. With this method, we test the exploration of C++ classes performed by *freud-dwarf*.

A.8 Function `test_linear_fitinregister`

```
void test_linear_fitinregister(fit_in_register fir) {
   int pf = fir.get_int();
   for (int i = 0; i < pf; i++) {
      usleep(150);
   }
}
```




```
void test_linear_fitinregister(fit_in_register fir).time {
features:
 int s = fir.single_int;

annotations:
 Norm(9.40 + 222.60 * s, 850);
}
```

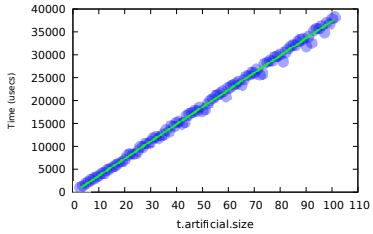
(Time for statistical analysis: 0.370s)

A linear correlation between the length of a private field of a class that is passed by value, and the running time. Since the class is small, the actual content

is packed in CPU registers directly. This tests both the code generated by *freud-dwarf* to handle structures that can fit in CPU registers, and the ability of *freud-pin* to unpack values in CPU registers.

A.9 Function `test_linear_vector`

```
void test_linear_vector(std::vector<int> * t) {
   for (int i = 0; i < t->size(); i++) {
      usleep(250);
   }
}
```



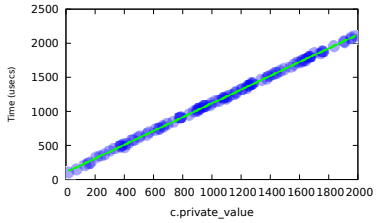

```
void test_linear_vector(std::vector<int> * t).time {
features:
 int t.artificial.size = t->_M_impl._M_finish - t->_M_impl._M_start;

annotations:
 Norm(-287.62 + 375.38 * t.artificial.size, 293989);
}
```

(Time for statistical analysis: 0.379s)

A linear correlation between the size of a std::vector and the running time. Our heuristics create an artificial feature, called size, as the difference between the finish and start fields of the vector class.

A.10 Function `test_derived_class`

```
void test_derived_class(const abstract_class_1 &c) {
   const derived_class &dc = dynamic_cast<const derived_class &>(c);
   usleep(dc.get_pv());
}
```



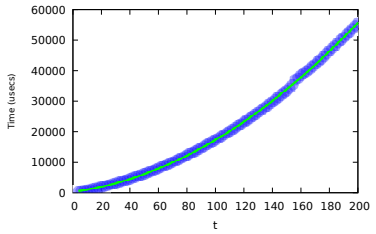

```
void test_derived_class(const abstract_class_1 &c).time {
features:
 int c = c.private_value;

annotations:
 Norm(100.93 + 1.01 * c, 258);
}
```

(Time for statistical analysis: 0.374s)

A linear correlation between the value of a private field of a class and the running time. The class that contains the relevant feature is of type derived_class, which inherits from abstract_class_1. On the other hand, no useful feature can be found in abstract_class_1. This tests (1) the ability of *freud-dwarf* to produce correct information about the hierarchy of classes in C++ programs, and (2) the ability of *freud-pin* to use such information at run time to extract features from objects that have a different type at runtime, compared to the type at compile time.

A.11 Function `test_quad_int`

```
void test_quad_int(int t) {
   for (int i = 0; i < t; i++) {
      usleep(t);
   }
}
```



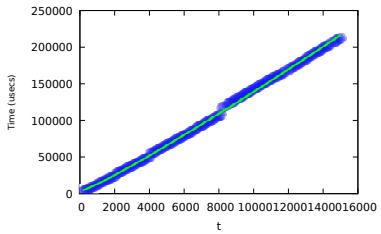
```
void test_quad_int(int t).time() {
features:
 int t = t;

annotations:
 Norm(63.51 * t + 1.07 * t ^ 2, 45612);
}
```

(Time for statistical analysis: 0.373s)

A quadratic correlation between the int parameter, passed by value, and the running time. Here we test the ability of *freud-statistics* to produce a model with the expected complexity.

A.12 Function `test_nlogn_int`

```
void test_nlogn_int(int t) {
   for (int i = 0; i < t; i++) {
      usleep(log2(t));
   }
}
```



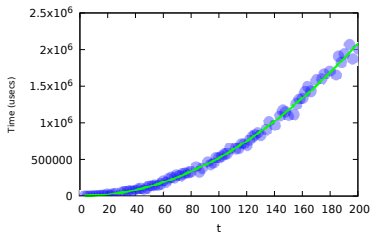

```
void test_nlogn_int(int t).time {
features:
 int t = t;

annotations:
 Norm(2585.09 + 1.03 * t * log(t), 477983);
}
```

(Time for statistical analysis: 0.372s)

An $n \log n$ correlation between the int parameter, passed by value, and the running time. Here we test the ability of *freud-statistics* to produce a model with the expected complexity. Notice how we must test with bigger absolute values for the features (controlled by the input parameters) in order to distinguish the $n \log n$ case from the linear and quadratic cases.

A.13 Function `test_quad_int_wn`

```
void test_quad_int_wn(int t, int nlevel) {
   for (int i = 0; i < t; i++) {
      unsigned int noise = rand() % t * 100;
      usleep(t + noise);
   }
}
```



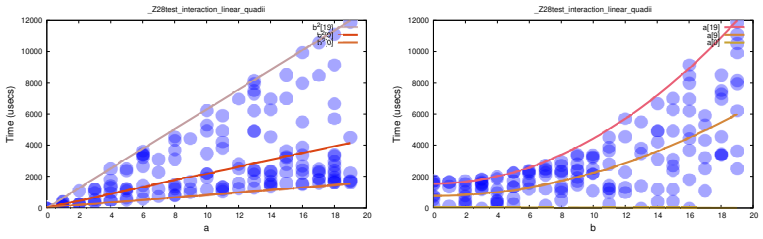

```
void test_quad_int_wn(int t, int nlevel).time {
features:
 int t = t;

annotations:
 Norm(6269.55 + 52.18 * t^2, 991720824);
}
```

(Time for statistical analysis: 0.358s)

A quadratic correlation between the int parameter with added random noise, passed by value, and the running time. We add noise around the expected performance to test the ability of *freud-statistics* to identify the correct complexity class even when there is residual noise in the measurements, when such noise cannot be correlated to any feature.

A.14 Function `test_interaction_linear_quad`

```
void test_interaction_linear_quad(int a, int b) {
   for (int i = 0; i < a; i++)
      usleep(b*b);
}
```




```
void test_interaction_linear_quad(int a, int b).time {
features:
 int a = a;
 int b = b;

annotations:
 Norm(155.10*a + 15.52*b^2 - 7.62*b + 17.95*a*b^2 - 51.50*a*b, 21.87);
}
```

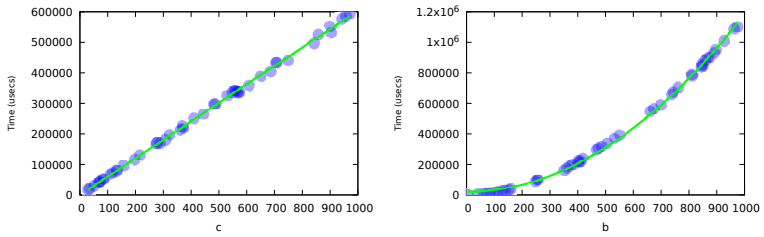
(Time for statistical analysis: 0.523s)

In this method we let two features interact to define the resulting performance behavior. We test both the ability of *freud-statistics* to identify such correlation, and to produce human readable 2-d plots for such cases.

A.15 Function `test_linear_branches`

```
void test_linear_branches(int a, int b, int c) {
   if (a > 10) {
      for (int i = 0; i < b; i++) {
         usleep(b);
      }
   }
   else {
      for (int i = 0; i < c; i++) {
         usleep(450);
```

```
      }
   }
}
```




```
void test_linear_branches(int a, int b, int c).time {
features:
 int a = a;
 int b = b;
 int c = c;

annotations:
 [ a <= 10 ]
 Norm(-392.24 + 587.35 * c, 424838118);

 [ a > 10 ]
 Norm(17743.87 + 1.14 * b ^ 2, 581010480);
}
```

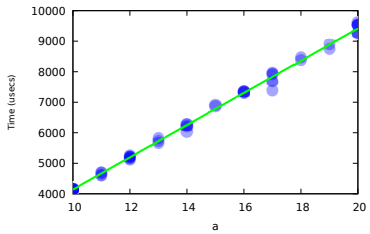
(Time for statistical analysis: 0.514s)

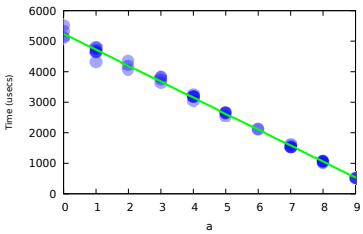
The branch defined by feature a triggers either a linear or quadratic behavior. We test the ability of our analysis to create and explore a complex classification tree.

A.16 Function `test_linear_branches_one_f`

```
void test_linear_branches_one_f(int a, int b, int c) {
   if (a < 10) {
      for (int i = 0; i < 10 - a; i++) {
         usleep(400);
      }
   }
   else {
```

```
      usleep(4000);
      for (int i = 0; i < a - 10; i++) {
         usleep(400);
      }
   }
}
```






```
void test_linear_branches_one_f(int a, int b, int c).time {
features:

annotations:
 [ a <= 9 ]
 Norm(5122.16 - 514.55 * a, 41817);

 [ a > 9 ]
 Norm(-1189.46 + 532.05 * a, 33452);
}
```

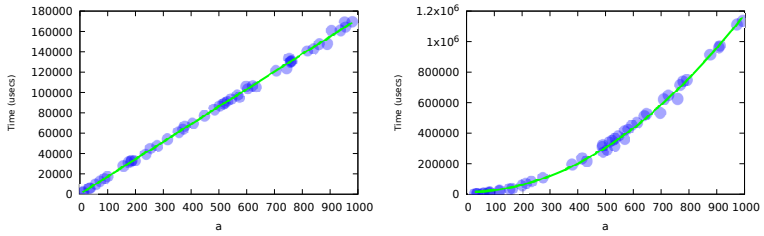
(Time for statistical analysis: 0.486s)

The case is very similar to `test_linear_branches`, but this time the branch is defined by the same feature that is also used in the regression analyses. Here we are testing the classification tree exploration performed by *freud-statistics*.

A.17 Function `test_multi_enum`

```
void test_multi_enum(enum command c, int a) {
   switch (c) {
      case CMD_CONSTANT:
         usleep(100000);
         break;
      case CMD_LINEAR:
         for (int i = 0; i < a; i++) {
```

```
            usleep(100);
         }
         break;
      case CMD_QUAD:
         for (int i = 0; i < a; i++) {
            usleep(a);
         }
         break;
   }
}
```



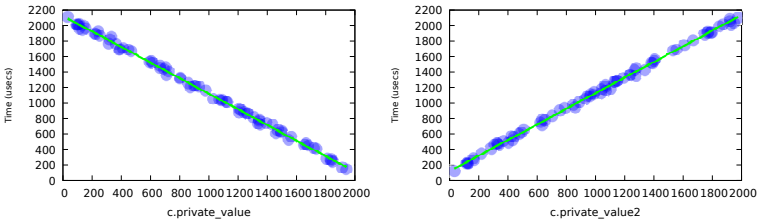

```
void test_multi_enum(enum command c, int a).time {
features:
 int c = c;
 int a = a;

annotations:
 [ 0 <= c ]
 Norm(10435, 124398);

 [ 0 < c <= 1 ]
 Norm(354.35 + 171.54 * a, 5249978);

 [ 1 < c ]
 Norm(19357.97 + 1.14 * a^2, 647021694);
}
```

(Time for statistical analysis: 0.447s)

With this experiment, we test the identification and usage of partitions generated by enum variables in the code of target programs.

A.18 Function `test_grand_derived_class2`

```
void test_grand_derived_class2(const abstract_class_2 *c) {
   const grand_derived_class2 *gdc =
        dynamic_cast<const grand_derived_class2 *>(c);
   if (gdc)
      usleep(gdc->get_pv());
   const derived_class *dc = dynamic_cast<const derived_class *>(c);
   if (dc)
      usleep(2000 - dc->get_pv());
}
```



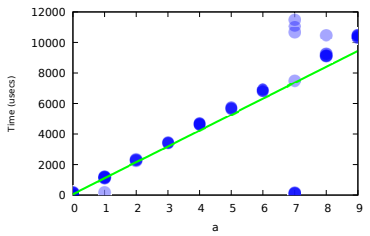

```
void test_grand_derived_class2(const abstract_class_2 *c).time {
features:
 int p = c->private_value;
 int p2 = c->private_value2;

annotations:
 [ c_vptr.abstract_class_2 <= 1844335335 ]
 Norm(2123.23 - 1.01 * p, 627);

 [ c_vptr.abstract_class_2 > 1844335335 ]
 Norm(116.40 + 1.01 * p2, 527);
}
```

(Time for statistical analysis: 0.455s)

Here we have two different possible dynamic types for the input parameter. One inheritance is two steps away from the compile time type. Also, the actual type of the input parameter at run time is used for the branch analysis, as a partitioning feature.

A.19 Function `test_main_component`

```
void test_linear_main_component(int a) {
   int p = rand() % 100;
   if (p < 25)
      usleep(rand() % 1500 + 10000);
   else if (p >= 25 && p < 50)
      usleep(rand() % 100);
   else {
      for (int i = 0; i < a; i++) {
         usleep(1000);
      }
   }
}
```




```
void test_linear_main_component(int a).time {
features:
 int a = a;

annotations:
 M Norm(64.88 + 1042.19 * a, 4697777);
}
```

(Time for statistical analysis: 4.582s)

Main component analysis. The method has a linear time complexity with respect to a only in half of its executions. In the other half, it exhibits difference performance behaviors, that do not depend on any feature. Our statistical analysis correctly identifies the clusters representing the most observed behavior, and produces a linear regression for those observations. Notice how the time required by the statistical analysis is bigger than all the cases observed so far, since the main trend analysis requires clustering, which is the slowest part of the analysis in the current implementation. Also, notice how the textual performance annotation contains an **M** keyword before the **Norm** expression, to indicate that the expression represents only a subset of the data points collected, filtered with the

main trend analysis.

A.20 Function `test_random_clustering`

```
void test_random_clustering(int a) {
   int rnd = rand() % 100;
   if (rnd < 30)
      usleep(900);
   else if (rnd < 70)
      usleep(600);
   else
      usleep(300);
}
```

```
void test_random_clustering(int a).time() {
features:

annotations:
 {0.32} Norm(422, 528);
 {0.32} Norm(740, 888);
 {0.34}Norm(1034, 1138);
}
```

(Time for statistical analysis: 11.285s)

This function takes one input parameter, in addition to the `global_variable`, but does not use any feature to select a specific behavior. Clustering always takes more time than finding regressions for a specific data set. On the one hand clustering is always performed if the regression analysis failed, on the other hand our implementation of the variable KDE clustering is the slowest part of the statistical analysis. We do not produce any plot automatically, since we have no cost function to draw.

Bibliography

[1] Moshe Bach, M. Charney, R. Cohn, Elena Demikhovsky, Tevi Devor, K. Hazelwood, A. Jaleel, Chi-Keung Luk, Gail Lyons, Harish Patil, and A. Tal. Analyzing parallel programs with pin. *Computer*, 43:34–41, 2010.

[2] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. Model-based performance prediction in software development: A survey. *IEEE Trans. Softw. Eng.*, 30(5):295–310, May 2004.

[3] Paul Barham, Rebecca Isaacs, Richard Mortier, and Dushyanth Narayanan. Magpie: Online modelling and performance-aware systems. In Michael B. Jones, editor, *HotOS*, pages 85–90. USENIX, 2003.

[4] Steffen Becker, Heiko Koziolek, and Ralf Reussner. Model-based performance prediction with the palladio component model. In *Proceedings of the 6th International Workshop on Software and Performance*, WOSP '07, page 54–65, New York, NY, USA, 2007. Association for Computing Machinery.

[5] Antonia Bertolino and Raffaela Mirandola. CB-SPE Tool: Putting component-based performance engineering into practice. In Ivica Crnkovic, Judith A. Stafford, Heinz W. Schmidt, and Kurt Wallnau, editors, *Component Based Software Engineering: 7th International Symposium, CBSE 2004*, volume 3054 of *LNCS*, pages 233–248. Springer, May 2004.

[6] Z. I. Botev, J. F. Grotowski, and D. P. Kroese. Kernel density estimation via diffusion. *Annals of Statistics*, 38(5):2916–2957, 10 2010.

[7] Marc Brünink and David S. Rosenblum. Mining performance specifications. FSE '16, pages 39–49, New York, NY, USA, 2016. ACM.

[8] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '04, page 2, USA, 2004. USENIX Association.

[9] Bihuan Chen, Yang Liu, and Wei Le. Generating performance distributions via probabilistic symbolic execution. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 49–60, New York, NY, USA, 2016. Association for Computing Machinery.

[10] DWARF Debugging Information Format Committee. Dwarf debugging information format. http://dwarfstd.org/doc/DWARF4.pdf, 2010.

[11] Emilio Coppa, Camil Demetrescu, and Irene Finocchi. Input-sensitive Profiling. PLDI '12, pages 89–98, New York, NY, USA, 2012. ACM.

[12] Emilio Coppa, Camil Demetrescu, Irene Finocchi, and Romolo Marotta. Estimating the Empirical Cost Function of Routines with Dynamic Workloads. CGO '14, pages 230:230–230:239, New York, NY, USA, 2014. ACM.

[13] Miguel de Miguel, Thomas Lambolais, Mehdi Hannouz, Stéphane Betgé-Brezetz, and Sophie Piekarec. Uml extensions for the specification and evaluation of latency constraints in architectural models. In *Proceedings of the 2Nd International Workshop on Software and Performance*, WOSP '00, pages 83–88, New York, NY, USA, 2000. ACM.

[14] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tsc̃hantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45, December 2007.

[15] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Statã. Extended static checking for java. FSE '16, pages 234–245. ACM, 2002.

[16] Simon F. Goldsmith, Alex S. Aiken, and Daniel S. Wilkerson. Measuring Empirical Computational Complexity. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 395–404, New York, NY, USA, 2007. ACM.

[17] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A Call Graph Execution Profiler. SIGPLAN '82, pages 120–126, New York, NY, USA, 1982. ACM.

[18] Vincenzo Grassi, Raffaela Mirandola, and Antonino Sabetta. From design to analysis models: A kernel language for performance and reliability analysis of component-based systems. In *Proceedings of the 5th International*

Workshop on Software and Performance, WOSP '05, page 25–36, New York, NY, USA, 2005. Association for Computing Machinery.

[19] Jianmei Guo, Krzysztof Czarnecki, Sven Apely, Norbert Siegmundy, and Andrzej Wasowski. Variability-aware Performance Prediction: A Statistical Learning Approach. ASE'13, pages 301–311, Piscataway, NJ, USA, 2013. IEEE Press.

[20] Johannes Henkel and Amer Diwan. Discovering algebraic specifications from java classes. ECOOP '03, pages 431–456, 2003.

[21] Johannes Henkel and Amer Diwan. A tool for writing and debugging algebraic specifications. ICSE '04, pages 449–458, Washington, DC, USA, 2004. IEEE Computer Society.

[22] Kenneth Hoste, Aashish Phansalkar, Lieven Eeckhout, Andy Georges, Lizy K. John, and Koen De Bosschere. Performance Prediction Based on Inherent Program Similarity. PACT '06, pages 114–122, New York, NY, USA, 2006. ACM.

[23] Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes 3A, 3B, 3C, and 3D: System Programming Guide. https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-3a-3b-3c-and-3d-system-programming-guide.html, 2020.

[24] JProfiler. https://www.ej-technologies.com/products/jprofiler/overview.html, 2019.

[25] Robert E. Kass and Adrian E. Raftery. Bayes factors. *Journal of the American Statistical Association*, 90(430):773–795, 1995.

[26] Heiko Koziolek. Performance evaluation of component-based software systems: A survey. *Perform. Eval.*, 67(8):634–658, August 2010.

[27] THE /proc FILESYSTEM. https://www.kernel.org/doc/Documentation/filesystems/proc.txt, 2020.

[28] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. *ACM Trans. Comput. Syst.*, 35(4), December 2018.

[29] Adrian Mos. Compas: Adaptive performance monitoring of component-based systems. In *In Workshop on Remote Analysis and Measurement of Software Systems (RAMSS) at 26th International Conference on Software Engineering (ICSE)*, 2004.

[30] Adrian Mos and John Murphy. A framework for performance monitoring, modelling and prediction of component oriented distributed systems. In *Proceedings of the 3rd International Workshop on Software and Performance*, WOSP '02, page 235–236, New York, NY, USA, 2002. Association for Computing Machinery.

[31] MySQL Server. https://github.com/mysql/mysql-server, 2019.

[32] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing Wrong Data Without Doing Anything Obviously Wrong! ASPLOS XIV, pages 265–276, New York, NY, USA, 2009. ACM.

[33] Trevor Parsons, John Murphy, et al. Detecting performance antipatterns in component based enterprise systems. *J. Object Technol.*, 7(3):55–91, 2008.

[34] Sharon E. Perl and William E. Weihl. Performance Assertion Checking. SOSP '93, pages 134–145, New York, NY, USA, 1993. ACM.

[35] Pin - A Dynamic Binary Instrumentation Tool. https://software.intel.com/en-us/articles/pintool, 2019.

[36] Adrian E. Raftery. Bayesian model selection in social research. *Sociological Methodology*, 25:111–163, 1995.

[37] Gideon Schwarz et al. Estimating the dimension of a model. *The annals of statistics*, 6(2):461–464, 1978.

[38] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[39] George R. Terrell and David W. Scott. Variable kernel density estimation. 20(3):1236–1265, 09 1992.

[40] The DWARF Debugging Standard. http://dwarfstd.org, 2019.

[41] Eno Thereska, Bjoern Doebel, Alice X. Zheng, and Peter Nobel. Practical performance models for complex, popular applications. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '10, pages 1–12, New York, NY, USA, 2010. ACM.

[42] Eno Thereska, Bjoern Doebel, Alice X. Zheng, and Peter Nobel. Practical Performance Models for Complex, Popular Applications. SIGMETRICS '10, pages 1–12, New York, NY, USA, 2010. ACM.

[43] Twitter, Inc. *Distributed Systems Tracing with Zipkin*, June 2012.

[44] Pavel Valov, Jean-Christophe Petkovich, Jianmei Guo, Sebastian Fischmeister, and Krzysztof Czarnecki. Transferring performance prediction models across different hardware platforms. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ICPE '17, pages 39–50, New York, NY, USA, 2017. ACM.

[45] Jeffrey S. Vetter and Patrick H. Worley. Asserting Performance Expectations. SC '02, pages 1–13, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[46] Jeffrey Scott Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software*, 11:37–57, 1985.

[47] Xiuping Wu and Murray Woodside. Performance modeling from software components. In *Proceedings of the 4th International Workshop on Software and Performance*, WOSP '04, page 290–301, New York, NY, USA, 2004. Association for Computing Machinery.

[48] x264. https://www.videolan.org/developers/x264.html, 2019.

[49] Dmitrijs Zaparanuks and Matthias Hauswirth. Algorithmic profiling. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 67–76. ACM, 2012.