Analyzing System Performance with Probabilistic Performance Annotations

Daniele Rogora
Università della Svizzera italiana
daniele.rogora@usi.ch

Antonio Carzaniga
Università della Svizzera italiana
antonio.carzaniga@usi.ch

Amer Diwan
Google
diwan@google.com

Matthias Hauswirth
Università della Svizzera italiana
matthias.hauswirth@usi.ch

Robert Soulé
Yale University
robert.soule@yale.edu

Abstract

To understand, debug, and predict the performance of complex software systems, we develop the concept of *probabilistic performance annotations.* In essence, we annotate components (e.g., methods) with a relation between a measurable performance metric, such as running time, and one or more features of the input or the state of that component. We use two forms of regression analysis: regression trees and mixture models. Such relations can capture non-trivial behaviors beyond the more classic algorithmic complexity of a component. We present a method to derive such annotations automatically by generalizing observed measurements. We illustrate the use of our approach 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 the performance. Notably, we isolate a performance regression and identify the root cause of a second performance bug in MySQL.

Keywords: performance, performance analysis, instrumentation, dynamic analysis

ACM Reference Format:
Daniele Rogora, Antonio Carzaniga, Amer Diwan, Matthias Hauswirth, and Robert Soulé. 2020. Analyzing System Performance with Probabilistic Performance Annotations. In *Fifteenth European Conference on Computer Systems (EuroSys '20), April 27–30, 2020, Heraklion, Greece.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3342195.3387554

EuroSys '20, April 27–30, 2020, Heraklion, Greece

ACM ISBN 978-1-4503-6882-7/20/04…$15.00
https://doi.org/10.1145/3342195.3387554

1 Introduction

To manage complexity in programs, developers use a combination of tools [5, 6, 11, 12] and best practices (e.g., modularization and stylized documentation). These tools and practices help with *functionality* but not with run-time dynamical aspects, such as performance. Indeed, even for implementations of classic algorithms, such as sort algorithms, the dynamics are more complex than computational complexity alone. 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. For example, invoking an innocuous-sounding method, `getX`, 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; indeed we have personally experienced situations where an unintended change in performance significantly degraded user experience with the service (e.g., by resulting in more timeouts for the user). Thus, we need a system for determining and checking the performance characteristics of code.

This paper takes the first step towards such a system: it describes a formal notation for describing the performance characteristics of code and presents and evaluates an automated system for inferring the performance annotations that capture these characteristics.

To illustrate the challenges in understanding the performance of even a simple method, Figure 1 shows the running time of `list<int>::sort()` from the C++ standard library for a range of input sizes. To show the interactions with the memory subsystem we limit the amount of memory using Linux Control Groups (using much larger inputs would have the same effect). Despite the documented $O(n \log n)$ computational complexity of the function, Figure 1 shows that running time has three distinct modes: $n \log n$ for small inputs (although it appears nearly constant), linear increase with a steep slope for medium inputs, and a dramatic jump but lower slope for large inputs. How would a programmer wanting to use this code know about these modalities? The programmer could carefully study the code before using it. But apart from the fact that this would be very onerous and

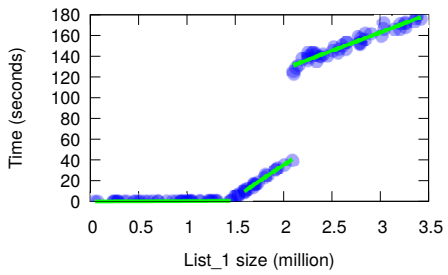


Figure 1. Running time for std::list::sort

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.

Today, developers rely on profiling [8, 14] to understand the performance of their code. Since profilers report aggregate resource usage (e.g., CPU cycles spent in a function), they do not relate the performance of a function to its input and offer no predictive capabilities. Recent work in algorithmic profilers [25] attempts to generalize profiles by relating the size of the input to amount of work performed by a function (e.g., number of loop iterations). Thus, algorithmic profilers attempt to discover the algorithmic complexity of code. 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.

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. We then use statistical analysis to build mathematical models relating input features to performance. Our models take modalities into consideration: in the `std::list::sort` example we produce three models, one for each modality. We translate these models into a semantically meaningful annotation language, which programmers can read and understand. This process is fully automated.

We have implemented our approach in a tool, named Freud. We evaluate Freud on three systems in real use: the MySQL database system; ownCloud, a distributed file hosting service offered by a Swiss academic ISP; and the x264 video encoder. Notably, Freud is able to isolate a performance regression in MySQL, and identify the root cause of a second performance bug that was supposedly fixed in prior releases, but in reality still exists.

Overall, Freud produces annotations and graphs that are easy to read and interpret for the performance analyst. The annotations relate a diverse set of metrics to input features, greatly increasing a developer's understanding of performance. These annotations can be used for documentation or run-time assertion checking. Moreover, the results generalize—even though the three case studies are very different in their functionalities, programming languages used, and deployment, we used the same analysis techniques on all of them.

In summary, this paper makes the following contributions:

- We develop a performance model in the form of probabilistic performance annotations (§2);
- 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 (§3);
- We describe a concrete implementation of the methodology in a tool named Freud, which automatically produces performance annotations for instrumented binaries (§4);
- We evaluate Freud through controlled experiments (§5), and analyze three complex systems: MySQL, the ownCloud storage service, and the x264 video encoder (§6).

2 Performance Annotations

We first describe how developers use our tool, *Freud*, to derive performance annotations for their programs (§2.1) and then present our full annotation language (§2.2).

2.1 Freud Analyzer

This section describes how Freud works for C++; our PHP implementation is similar, but uses a different tool set.

We assume the user has domain knowledge of the software being analyzed, and has selected the method they want to study.

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

Freud uses debugging information for its code analysis of the binary and then uses Pin [18] to add instrumentation to collect features and metrics (§3.1,§4.1). For features, Freud collects the values of the arguments along with some relevant properties based on their type information.

The user then runs the program with multiple inputs, which may be standard workloads or special inputs picked to exercise the method that the user is interested in. The result of these runs are log files: each log entry gives the values for the metrics of interest along with all the features.

Freud then statistically analyzes the logs to produce annotations (§3). Freud can also produce graphical representations of the annotations. In fact, all the performance graphs shown in this paper are generated by Freud.

2.2 Annotation Language

Figure 1 shows that even our simple sort example has three modalities. Freud's output annotations for this method (Figure 2) reflect this.

```
1  std::list<int>::sort.time(this) {
2   uint s = *(this->_M_impl._M_node._M_storage._M_storage);
3
4   [s > 49584 && s < 1450341]
5   Norm(53350.31 - 2.10*s + 0.12*s*log(s), 12463.88);
6
7   [s > 1589482 && s < 2085480]
8   Norm(-90901042.29 + 63.11*s, 899547.29);
9
10  [s > 2098759 && s < 3415880]
11  Norm(56712024.50 + 35.38*s, 3379580.27); }
```

Figure 2. The Performance Annotation for Sort

Concretely, a performance annotation is an expression that characterizes a performance metric as a random variable (the dependent variable) whose distribution is a function of zero or more features of the input or state of the method (the independent variables). Thus, the basic form of a performance annotation is as follows: $Y \sim expr(x)$, which is read as: Y is a random variable distributed like $expr(x)$, where x is the relevant feature. Expressions with zero input features describe behaviors that are independent of the input or for which Freud found no relevant features.

Figure 2 gives the expression for the running time (dependent variable) of `sort`. Line 2 identifies the feature (named `s`) that Freud found relevant to the running time of this method. This feature corresponds to the variable that gives the length of the input list. Admittedly, this is not intuitively clear from the name of the variable that defines the feature (`_M_storage`). However, that is the exact name found in the implementation in the C++ standard library, which presumably would be meaningful to a developer of the library.

Freud's inferred annotations use *scope* conditions to separate modalities: $Y \sim [C_1]\ expr_1(x);\ [C_2]\ expr_2(x)$. This means that the run time follows the distribution $expr_1(x)$ if condition C_1 holds, and $expr_2(x)$ if condition C_2 holds, and so on. In other words, the annotation describes a type of regression tree where the scope conditions indicate partitions.

Figure 2 shows that Freud's annotation for the running time of `sort` has three scopes: (i) when the size is between 49584 and 1450341, (ii) when the size is between 1589482 and 2085480, and (iii) when the size is between 2098759 and 3415880. The interpretation of these constants requires further analysis. For example, as it turns out, the first threshold represents the number of elements of `list<int>` that exhaust the memory allocated to the program (each element uses 24 bytes, and the resident set size limit was 36MB, which means 1.5M elements).

In all three cases, Freud infers 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. Notice that the asymptotic complexity is still $O(n \log n)$ however, Freud could not distinguish that trend from the linear component in the given limited input range.

If Freud is unable to determine a relationship between features and scopes, it produces probabilistic annotations (i.e., a type of mixture model). For example, if Freud observes a performance behavior of $expr_1(x)$ 20% of the time and a performance behavior of $expr_2(x)$ 80% of the time then the annotation would take the form: $Y \sim \{0.2\}\ expr_1(x);\ \{0.8\}\ expr_2(x)$.

Freud can also infer models that combine probabilistic annotations within some given scoping conditions, as follows: $Y \sim [C_1]\ expr_1(x);\ [C_2]\{\ \{0.2\}\ expr_2(x);\ \{0.8\}\ expr_3(x)\ \}$.

Freud supports and infers three distributions: normal, binomial, and exponential. Furthermore its expressions may use all the standard relational and arithmetic operators and support standard mathematical functions (log, exp, etc.). For brevity, we omit the formal grammar of the annotation language, and rely on examples throughout this paper.

3 Derivation of Performance Annotations

We now describe a method to collect measurements and derive performance annotations. This method consists of two phases: instrumentation and then model selection.

3.1 Instrumentation

The first step involves analyzing and instrumenting the code to collect relevant data. For every execution of every target method, we collect (1) all the desired performance metrics, and (2) all the data that can be used to extract potentially relevant features of the input or the state of the system. Optionally, we log (3) the outcome of every executed conditional branch instructions (branch taken or not taken). These branch logs can improve the quality of the performance annotations (§3.2).

Metrics. Metrics are the dependent variables in performance annotations. In contrast to traditional profilers, Freud collects metrics besides running time, such as memory consumption and lock holding/blocking time. Moreover, the collection process is extensible, allowing us to easily add new metrics. When using Freud, the performance analyst explicitly selects which metrics are of interest to them. This is passed as a command line argument to the tool.

The exact procedures to collect performance metrics vary between metrics and programming languages. For example, to measure the running time for the executions of a method, we can wrap the method with code that logs timestamps at the entry and exit points of the method. To measure the amount of heap memory consumed, instead, we can either wrap a method with code that measures the total memory

allocated at a given instant, as we do with PHP, or we can track all the calls to the main memory allocation functions, as we do with C/C++ (§4.1).

Features. Features are the independent variable in performance annotations. Unlike with metrics, which are selected by the user, Freud selects features for each target method automatically. This process is therefore a bit more involved, as it amounts to statically finding the values of local or global variables that may impact the execution of the method. Freud selects three types of features: program variables, system variables, and quantities computed from program variables.

The first and most directly relevant variables we use are the parameters to the method. We collect the values of scalar variables and we also transitively traverse pointers to extract field values of heap objects. We limit the depth of this exploration that might otherwise become arbitrarily complex. This analysis is language dependent, and in particular it is based on information about the types of the objects processed by the target methods. For example, in the case of C/C++, Freud uses debugging information to determine data types and to explore complex data structures (§4.1).

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.

We then 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.

Instrumentation Overhead and Perturbation. For any performance monitoring tool, instrumentation overhead and perturbation can be a concern, and Freud is no exception. 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 [16]. In §4.2 we describe our efforts to mitigate the instrumentation overhead, and in §5 we experimentally validate the accuracy and robustness of our measurements.

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

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

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

3.2 Model Selection

The second phase of our approach consists of an offline statistical analysis. Freud processes the logs to produce probabilistic performance annotations for the chosen components. Freud applies the same statistical analysis to every target component for every target performance metric.

The analysis is based on two statistical models, namely regression trees and mixture models. A regression tree defines a hierarchical partitioning of the performance records such that the records in each partition are defined by a scope condition and are modeled by a regression. A mixture model is a combination of two or more sub-models each associated with an occurrence probability rather than a scope condition. These two classes of models correspond to the scoped and probability specifications in our annotation language (§2.2).

At a high-level, the analysis proceeds in two phases. In the first phase, we attempt to formulate a regression tree, partitioning the records based on specific conditions on features ($x_1, x_2, \ldots$) and fitting a regression model within each partition. 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.

We now detail this high-level model selection process. We start with the model formulation that we invoke in every step of both high-level phases.

Model Formulation. Given a set of measurements of a metric y and corresponding features $x_1, x_2, \ldots$, we formulate multiple-regression models $h(x_1, x_2, \ldots)$ with multiple independent variables. We try four classes of models of increasing order: first constant, then x (linear), then $x \log x$, then x^2 (quadratic). 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 all the features of the regression obtain a good (low) p-score, or there are no features remaining in the regression. For each chosen model for each class, we compute the R^2 goodness-of-fit indicator. Since we want to avoid overfitting and choosing models with too many parameters, we use the Bayesian Information Criterion (BIC) to compare the relative quality of

the candidate models. We compute the BIC value for every model whose R^2 is above a fixed threshold, 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 is zero everywhere. Then, we group strongly correlated features into equivalence classes and use a single representative from each class in the regression analysis.

Regression Tree. We try to formulate a regression tree using Algorithm 1. We start by considering the whole set of logged records. If we find a good-enough regression model $h(x_1, x_2, \ldots)$ for a set of features $\{x_1, x_2, \ldots\}$, then that is the resulting model. Otherwise, we try to partition the set of measurements so as to find good regressions for the individual parts. Thus we proceed recursively with a hierarchical partitioning until we cover the entire set of records with either valid regressions or clustering information. We use input-dependent conditions to define the partitions.

Algorithm 1 Regression Tree Analysis

Input: $R = \{r_1, r_2, \ldots\}$.
Output: Output: $A = \{(C_1, h_1), \ldots\}$.
1: $A \leftarrow \emptyset$
2: $P \leftarrow \{(true, R)\}$
3: **while** P is not empty **do**
4: $(C, S) \leftarrow P.pop()$
5: formulate model $h(X)$ on S for some feature-set X
6: **if** $h(X)$ is a good model **then**
7: $A \leftarrow A \cup \{(C, h(X))\}$
8: **else if** S can be split into S_1 and S_2 on some cond. C' **then**
9: $P \leftarrow P \cup \{(C \wedge C', S_1), (C \wedge \neg C', S_2)\}$
10: **else**
11: *//formulate a mixture model h on S with clustering*
$h \leftarrow$ output of Algorithm 2 on $R = S$
12: $A \leftarrow A \cup \{(C, h)\}$
13: **end if**
14: **end while**

The input to Algorithm 1 is a set of log records where each record r_i represents a run of the target method and consists of a performance metric y_i, a set of features $x_{i,f}$, and a set of branch results $b_{i,j,k}$ for each branch j and each execution of k of branch j. The output is an annotation $A = [C_1]\, h_1; [C_2]\, h_2; \ldots$ consisting of a set of expressions h_i, each associated with a condition C_i.

Algorithm 1 iteratively processes various sets of input records until it covers the whole input set R. 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 S into two sets of measurements S_1 and S_2—such that C' is always true in S_1 and always false in S_2—and then proceeds recursively to look for good models for S_1 and S_2. If this hierarchical partitioning fails to yield a set of models for the whole input set—because a set S can not be modeled well by any regression $h(X)$ and it can not be further partitioned based on a known condition—then the algorithm fails to return a valid regression tree, and the analysis proceeds with clustering.

Clustering and Mixture Model. The clustering analysis (Algorithm 2) tries to partition the input set $R = \{\ldots r_i \ldots\}$ based on the values y_i of the performance metric rather than based on any condition on the input features. The output is an annotation $A = [C_1]\, h_1; [C_2]\, h_2; \ldots$ or $A = \{p_1\}\, h_1; \{p_2\}\, h_2; \ldots$ consisting of a set of expressions h_j, each associated with a condition C_j or a probability p_j.

For each cluster, Algorithm 2 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 the output annotation (lines 3–10). In this phase, unlike in Algorithm 1, we allow for some data points not to be covered by specific scoping conditions and distributions. 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).

Scope Conditions. Given a set S of measurements with features $x_1, x_2, \ldots$, Algorithm 1 looks 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. Algorithm 2 looks for conditions C that define S, such that C is true for all the measurements in S and no other measurements.

The general idea is to find relatively simple and mutually exclusive conditions, $C_1, C_2, \ldots$ that define different behaviors. This can be done by associating classes of execution paths to behaviors, which in turn can be done statically, for example through symbolic execution, or dynamically, as we do with the instrumentation of branch instructions. For each measurement and therefore for each execution of the target method logged in S, the sequence of outcomes of branch instructions defines the exact path of that execution, which, together with the code of the branch instructions, defines a condition on the input features. It is among those conditions associated with each measurement that we look for scoping conditions. Note that these conditions are exact, in the sense that they are based on the code. Freud attempts to first use these conditions in Algorithm 1. If that fails, Algorithm 2 resorts to a black-box approach and formulates conditions by looking directly at the values of the features, as in the three ranges found for the sort example of Figure 1.

Further Refinements. Annotations derived from clustering, which we fall back to when we do not have enough

Algorithm 2 Clustering and Mixture Model

Input: $R = \{r_1, r_2, \dots\}$.
Output: $A = \{(p_1, h_1), \dots\}$ or $A = \{(C_1, h_1), \dots\}$.

$A \leftarrow \emptyset$
cluster R into $R_1, R_2, \dots$ 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 C_i that defines R_i **then**
      $A \leftarrow A \cup \{(C_i, h_i(X))\}$
    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

information to find good correlations and scoping conditions on input features, are valuable but limited. They are valuable because they still characterize the most common behaviors of the system. But they are also limited because they do not relate the behavior with the input or the state of the system, which means that the annotations can not be used to extrapolate and therefore to make predictions for different scenarios with different input or different state.

To overcome this limitation, 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.

4 Implementation

We describe the implementation of the ideas presented in the previous section within our open-source tool Freud.[1]

4.1 Instrumentation

We implemented two working instrumentation tools, one for PHP based on Runkit [19], and the other for C/C++ based on DWARF debugging symbols [21] and the Pin [18] instrumentation tool. These tools are designed to work with any program written in those programming languages, i.e., they do not target a specific software system. For brevity we only describe the C/C++ instrumentation.

We use debugging symbols to retrieve information about the input parameters passed to a target C/C++ method. For every method, we record the number of arguments, and for each argument, we collect the type and the position in memory that is valid just before the execution of the method.

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

The output of the analysis is a table of features. We then feed this table to the Pin tool that analyzes and instruments the binary file at the function level to record features.

Similarly, we use Pin instrumentation to collect performance metrics. For running time, we log the entry and exit timestamps for every target method. We instrument memory allocation primitives such as *malloc* or *new* to log dynamic memory allocation. We instrument the *pthread* mutex functions to collect lock holding/waiting time.

Finally, if we want to collect information about branches, we instrument the binary at the instruction level using Pin's

[1] https://github.com/usi-systems/freud

JIT mode. We instrument every conditional branch instruction to record the address of the branch, which we use as a unique id, and the branch outcome.

4.2 Overhead and Perturbation

Instrumentation necessarily introduces overhead. However, this is widely regarded as an acceptable cost, as the benefits of collecting the data outweigh the performance degradation. In order to minimize the run-time overhead of the instrumentation, we use sampling. We use a reservoir sampling algorithm to collect a fixed number of observations for every method or function for every run.

The instrumentation overhead depends on Freud's intrinsic features, such as the number and sizes of features recorded by Freud. It also depends on external factors, such as Pin's JIT compilation, which is variable and also difficult to analyze in part because Pin is proprietary software. Notice also that it is always possible to construct adversarial cases in which the overhead would dominate the execution time, for example by applying Freud to very simple functions executed in extremely hot loops. Still, in practice the overhead remains very limited, as we have verified experimentally in all the micro-benchmarks and concrete cases evaluated in Sections 5 and 6.

For Freud, the more pressing issue is that the instrumentation might perturb the measurements themselves. To reduce that risk, we subtract from the observations the execution time of the instrumentation code itself, and we invalidate measurements affected by Pin's JIT compilation. We set up a callback function, triggered when a new code trace is added to Pin's code cache, that invalidates the measurements that are in progress for all methods on the call stack.

4.3 Analysis

The regression analysis and all the other operations described in the remainder of the section are implemented as an off-line C++ program. The program uses the *R statistical package* library to compute regressions, mainly using the *lm* function (fitting linear models), and other basic functions to compute statistical properties of data sets.

Algorithm 2 clusters one-dimensional data (a single performance metric). Therefore, we use a variable-bandwidth kernel density estimator [20] to compute the one-dimensional density of the data. 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. To perform variable kernel density estimation, we use an own-made *R* port of the *akde1d* function [1], originally written for *Matlab*.

We find scoping conditions using the branch log, which contains the outcome of each executed conditional branch. If we find perfect correlations between some feature values and the outcome of specific branches, then we can infer that for some partitions of the values of the features, a specific method is taking a specific execution path. We use this information to partition the observations based on their execution paths. Then, we proceed with the same regression analysis that we described earlier.

5 Basic Validation

As a first evaluation, we conduct controlled experiments to (1) measure the running time for performing the analysis, (2) validate the accuracy of our measurements, and (3) test the robustness of the resulting performance annotations.

Running Time. The time for generating the static instrumentation code is negligible. For example, generating the instrumentation code for all of MySQL takes less than 10 seconds. The running time of the statistical analysis depends on a number of parameters, including number of features, the number of measurements, and the complexity of the code (e.g., number of branch conditions). For our unoptimized prototype, producing the annotation for a single function with 10 candidate features took, on average, less than 1 second.

Accuracy. We analyze sixteen functions, each exhibiting a well-defined behavior in terms of running times, which we control using *usleep*. 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. In all these cases, we first verify that Freud accurately derives a model of the expected class (quadratic, linear, etc.) for each function. We then measure the accuracy of the specific annotations through cross-validation for all sixteen functions with an overall minimum R^2 value of 0.9866.

Robustness. To test for perturbations from the instrumentation, we analyze the same sixteen 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.

6 Case Studies

We now demonstrate the use of Freud on three real-world, complex software systems: ownCloud, a remote storage web application written in PHP; MySQL, a well-known and widely

used DBMS written in C/C++; and the x264 library and application for encoding video streams into the H.264/MPEG-4 AVC compression format, written in C. We use Freud to derive performance annotations for all three systems for several metrics, including running time, dynamic memory allocation, and lock holding/waiting time.

We note that for the case studies, the goal of the evaluation is demonstrative, as opposed to quantitative. We show that Freud outputs performance annotations and graphs 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 purposely chose 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 Freud automatically identifies such relevant input features, and is also able to reference them with names that are immediately meaningful to developers; that the resulting 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 nontrivial performance bugs.

6.1 ownCloud

For the first case study, we create 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 100,000 requests involving files of different sizes, name-lengths, 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. The textual representation of the performance annotation can also be easily parsed by other analysis tools, such as a performance checker. As a complement to the textual representation, the graphs illustrate the trends and the key performance parameters for the benefit of human analysts.

Figure 3 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.

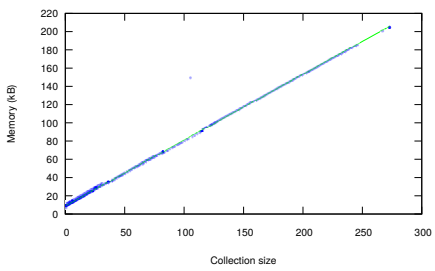
The first two graphs show two annotations for *generateMultiStatus*, a method that takes an array of file properties. The method iterates over the objects in the array, computing some data about each one. The graph on the left clearly shows that memory usage is linearly correlated to the size of the 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 the *emit_file_hooks_pre* method, 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.

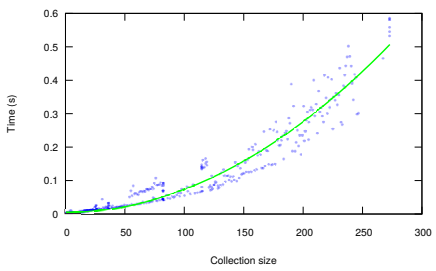
To perform the exploratory analysis described above, the user of Freud must (1) choose the methods to analyze and pass their names to the runkit tool that in turn creates the corresponding instrumentation code; (2) run the instrumented SWITCHdrive server to collect logs; and (3) run the statistical analysis tool to create performance annotations for some chosen performance metrics.

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.

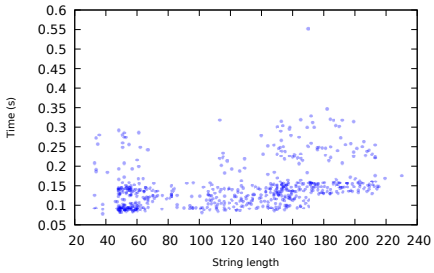
We proceed as follows: we first derive annotations using the two workloads described above. We then run the same workloads in a special setting in which we artificially introduce an anomaly in the system. In this setting, we use




```
generateMultiStatus.mem(collection) {
 uint s = collection.size;
 Norm(721.362*s + 8851.16, 2324.567); }
```




```
generateMultiStatus.time(collection) {
 uint s = collection.size;
 Norm(0.0066*s^2 + 0.000038*s + 0.0000036,
       0.022); }
```




```
emit_file_hooks_pre.time() {
 {0.77}Norm(0.13133923, 0.0007451);
 {0.23}Norm(0.23977998, 0.001); }
```

Figure 3. Some annotations for ownCloud: linear and quadratic regressions (left, center) and clusters (right).

annotations 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.

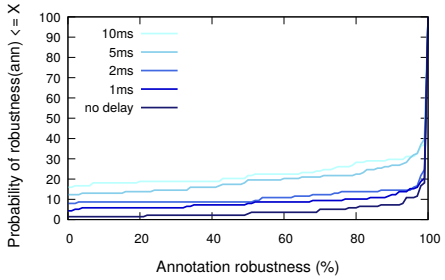


Figure 4. Robustness, use of annotations to detect anomalies.

Figure 4 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.

6.2 MySQL

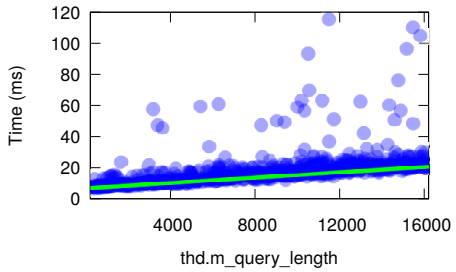
In the second case study, we use Freud to investigate two performance bugs reported on the MySQL bug tracker (n. *94296* and *92979*). To reproduce these bugs we use different versions of MySQL. We compile the unmodified code cloned from the MySQL github repository [15] with gcc 7.3 on a Ubuntu 18.04 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 use the instrumentation described in §4.1. We inject the workload locally using the MySQL client 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.

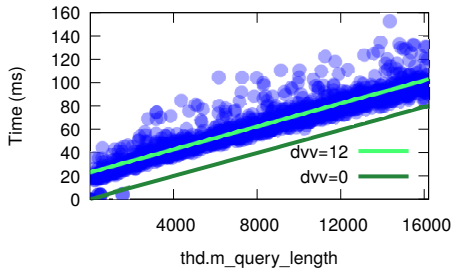
We replicate and analyze the bug using versions 5.7.24 and 8.0.11. As a workload, we use the MySQL dump attached to the bug report. The dump consists of a set of *INSERT* operations for a specific table. Using this set as a basis, we create a workload with a series of *INSERT* operations each inserting 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 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).




```
mysql_execute_command.time(thd) {
 uint len = thd->m_query_string.len;
 Norm(6630.19 + 0.86*len, 15.78); }
```




```
mysql_execute_command.time(thd) {
 uint len = thd->m_query_string.len;
 uint dvv = thd->variables.dynamic_variable_version;
 Norm(168.65 + 4.94*len + 1886.87*dvv, 2489.04); }
```

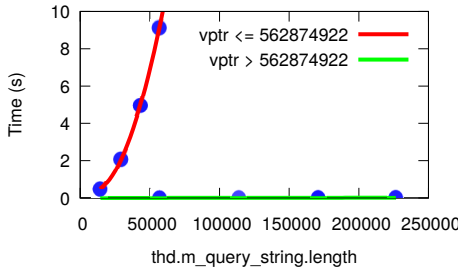
Figure 5. *mysql_execute_command*, 5.7.24 (top) vs. 8.0.11.

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 §3.2. Also notice that, differently from version 5, MySQL version 8 performs some additional startup operations during 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* $((v1_1, v2_1), \ldots, (v1_n, v2_n))$, 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 highest-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.



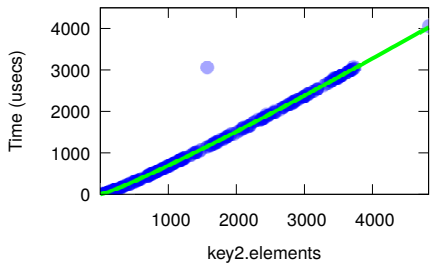

```
test_quick_select.time(thd, cond) {
 uint len = thd->m_query_string.len;
 uint vptr = cond->_vptr.Parse_tree_node_tmpl;
 [vptr <= 562874922] Norm(467533 - 50.21*len + 0.0036*len^2,
       282711.59);
 [vptr > 562874922] Norm(-53.603 + 0.057*len, 157.57); }
```

Figure 6. *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.

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 points to *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 7) from zero to the number of clauses in the query. This arithmetic progression explains the overall quadratic behavior.

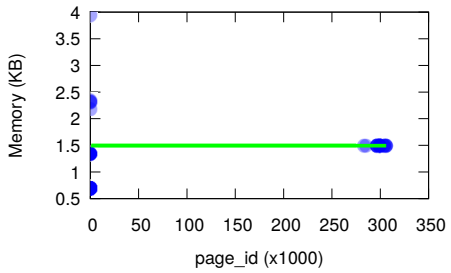
Reading the well documented code of *key_or()*, we find that *key_or()* is designed to compute a more general disjunction of *ranges*, as opposed to the specific values in the reported workload. This may suggest a specialized implementation and therefore a more radical bug fix.




```
key_or.time(key2) {
 uint e = key2->elements;
 Norm(-0.276 + 0.073*e + 0.062*e*log(e), 2.24); }
```

Figure 7. Arithmetic progression of *key_or()*.

Other metrics and other bi-modal behaviors. The annotation for the *fseg_create_general()* method (Figure 8) 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.




```
fseg_create_general.mem(page_id) {
 [page_id > 0] Norm(1496, 0); }
```

Figure 8. *fseg_create_general*: branch analysis.

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

Use of Freud in Identifying Performance Regressions. For bug 92979, Freud could be used to react to a performance regression. Freud would have first been used to create performance annotations for some symbols, either running the instrumented version of MySQL directly in the runtime deployment, or executing some performance tests before deploying the new version. The result of this analysis consists of a set of textual performance annotations, in addition to the graphs. When a newer version of MySQL become available, the performance analyst uses Freud to produce the new logs for the same symbols that had been already analyzed. This time, the statistical analysis tool takes additionally as input the textual performance annotations produced with the previous version of MySQL. The analysis finds a significant difference between the two performance profiles, and signals the anomaly, producing the new performance annotation that the performance analyst can use to see the differences and start the investigation.

Use of Freud in Performance Debugging. With bug 94296 we show a different use of Freud. This time the anomalous performance behavior is noticed by MySQL users, and reported on the bug tracker. The performance analyst, reading the bug description, decides to produce performance annotations for some specific symbols. The analyst passes the list of names of symbols to Freud, and executes the instrumented version of MySQL with the workload attached to the bug report. In this (proactive) scenario, the performance analyst reads the graphs produced by Freud to quickly see and characterize the anomalous behavior. The knowledge acquired by the analyst drives the investigation toward other symbols of MySQL, which finally lead to the root cause of the resulting performance behavior.

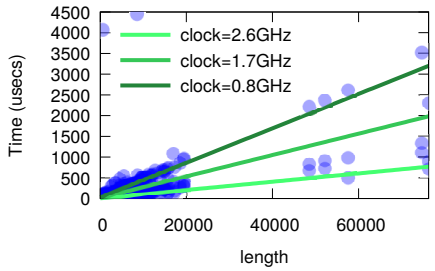
6.3 x264

For the third case study, we use the most recent version of x264, an open-source library and utility for H.264/MPEG-4 video encoding [24]. We run all experiments on a 4-core, 8-thread Intel Core i7-6700HQ CPU at 2.60GHz, in a system equipped with 8GB of RAM and a NVMe SSD drive. We compile *x264* with gcc 8.3 on a Ubuntu 18.10 64-bit machine.

As a workload, we use *x264* to convert or reencode six different videos (different content) from either VP9 or H.264 to H.264. We reencode 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.

In this case study we show another proactive usage scenario, in which an engineer could use Freud to explore and get a better understanding of a complex application like *x264*. The complexity of the performance of this program stems on the one hand from the tightly optimized and highly parallel processing model, and on the other hand from the large number of available options that select different computation modes and optimization levels. In this scenario, the engineer creates performance annotations for symbols whose name, or code, seem interesting from a performance viewpoint.

Selected Results. We start with *ff_h2645_extract_rbsp()*, a utility function that extracts a raw bit stream from an h264 source buffer. Freud derives an annotation (Figure 9) that indicates that the running time increases linearly with the size of the input buffer. Moreover, Freud also finds an interaction between the *size* and *cpu_clock* features. Qualitatively, the annotation indicates that the running time is cpu-bound.




```
ff_h2645_extract_rbsp.time(length, cpu_clock) {
 uint l = length;
 uint clock = cpu_clock;
 Norm(43.32 + 0.055*l - 1.46e-05*clock - 1.75e-08*l*clock, 4.56);}
```

Figure 9. *ff_h2645_extract_rbsp*: running time.

Moving to the core of the encoding functionality, we analyze the *x264_8_encoder_encode* function. For clarity, we do not vary the CPU speed, since there are already many parameters that affect performance. *x264_8_encoder_encode* drives the encoding process, managing the pool of worker threads that perform the actual encoding operations. In *slice-based mode*, *x264_8_encoder_encode* 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.

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 the left part of Figure 10.

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 in *sliced-mode* (*h->param.b_sliced_threads=1*) the waiting time correlates well with the number of threads and the video resolution (Figure 10, right). 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 10) 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 11). 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 11) than any waiting time observed for *x264_8_encoder_encode* (Figure 10, right).

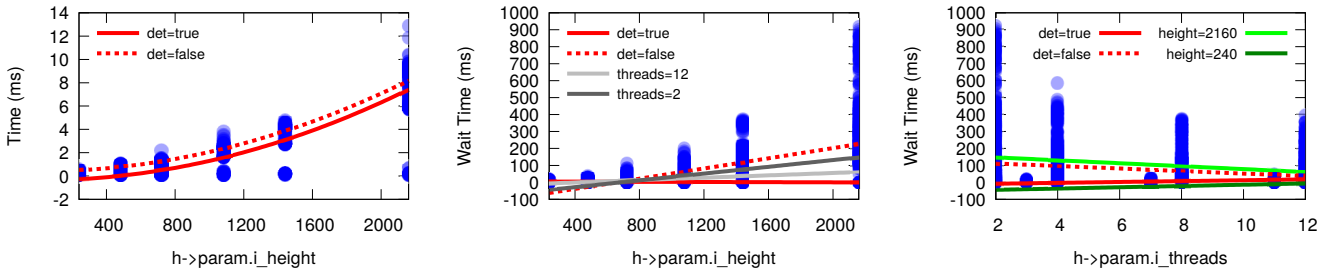
7 Related Work

Performance Assertion Specification. The idea of *assertions* that specify performance properties is not new. PSpec [17] 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 manually specify performance assertions, and the tool checks whether the assertions hold for a given trace. PSpec also includes a solver that uses linear regression to automatically infer the coefficients to be used in the assertions.

Vetter and Worley [23] use assertions on code segments for three scenarios: throwing performance exceptions, validating performance models, and adapting an algorithm dynamically at runtime. Their assertions are expressions involving hardware performance counters (e.g., cycles, instructions, or cache misses), application variables, and constants. Both of the above approaches allow developers to specify fixed bounds (e.g., on runtime), but they do not provide a probabilistic approach based on distributions. Moreover, they do not infer performance assertions automatically from scratch.

Deriving Models of Code Performance. Traditional profilers measure the execution *cost* (e.g., running time, executed instructions, cache misses) of a piece of code. They produce profiles such as: function f was called a total of 1000 times, 10% of program execution time was spent in function f, invocations of function f took 4 ms on average. Trend Profiling [7], Algorithmic Profiling [25] and Input-Sensitive Profiling [3, 4] represent a new form of profiling: instead of only measuring the execution *cost*, these profilers characterize a *cost function*: the 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.

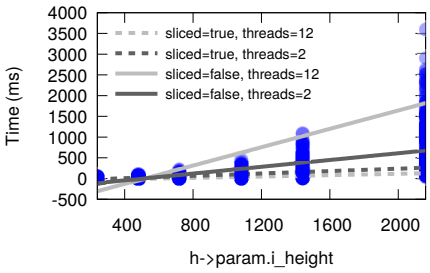
Freud expands on this idea in four ways: (1) Unlike algorithmic profiling, which measures cost in terms of platform-independent iteration counts, we measure running time. This




```
x264_8_encoder_encode.time(h, pic_in) {
 uint h = h->param.i_height;
 uint d = pic_in->param.b_deterministic;
 Norm(394.27 + 1.67e-03*d - 0.41*h + 2.88e
       -05*h^2, 14.96);
 ... }
```

```
x264_8_encoder_encode.wait_time(h, pic_in) {
 bool sliced = h->param.b_sliced_threads;
 uint height = h->param.i_height;
 uint threads = h->param.i_threads;
 uint dequant = h->thread.dequant4_mf;
 bool det = pic_in->param.b_deterministic;
 [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 10. *encoder_encode*: running time without context switches (left); wait time (center, right).




```
slice_write.time(h) {
 bool sliced = h->param.b_sliced_threads;
 uint threads = h->param.i_threads;
 uint height = h->param.i_height;
 [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 11. *slice_write*, sliced vs framed processing.

is a real metric that includes the effects of lower layers of the system, such as virtual memory or caching. This kind of integration of information from different system layers is similar to Vertical Profiling [10], but with a focus on cost functions instead of cost. (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. (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.

Many performance models use static features, such as calling context [2], application configuration [9], OS version [22], or hardware platform [13]. 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.

8 Conclusion

This paper presents performance annotations, which formulate probabilistic models of the performance of software systems. These models correlate measured metrics with semantically meaningful input or system state, giving developers and performance analysts a greater understanding of system behavior. We demonstrate the automatic derivation and the use of performance annotations with three case studies: the ownCloud storage service; the MySQL database; and the x264 video encoder. Overall, performance annotations are a first step towards a more general goal of providing developers and designers with useful tools for analyzing and understanding the behavior of applications.

References

[1] Zdravko Botev, Joseph Grotowski, and Dirk Kroese. 2010. Kernel Density Estimation via Diffusion. *The Annals of Statistics* 38 (11 2010). https://doi.org/10.1214/10-AOS799

[2] Marc Brünink and David S. Rosenblum. 2016. Mining Performance Specifications. In *Proc. 24th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE '16)*. ACM, New York, NY, USA, 39–49. https://doi.org/10.1145/2950290.2950314

[3] Emilio Coppa, Camil Demetrescu, and Irene Finocchi. 2012. Input-Sensitive Profiling. In *Proc. 2012 PLDI (PLDI '12)*. ACM, New York, NY, USA, 89–98. https://doi.org/10.1145/2254064.2254076

[4] Emilio Coppa, Camil Demetrescu, Irene Finocchi, and Romolo Marotta. 2014. Estimating the Empirical Cost Function of Routines with Dynamic Workloads. In *Proc. 2014 IEEE/ACM International Symposium on Code Generation and Optimization (CGO '14)*. ACM, New York, NY, USA, 230:230–230:239. https://doi.org/10.1145/2581122.2544143

[5] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon System for Dynamic Detection of Likely Invariants. *Science of Computer Programming* 69, 1-3 (Dec. 2007), 35–45. https://doi.org/10.1016/j.scico.2007.01.015

[6] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Statã. 2002. Extended Static Checking for Java. In *Proc. 24th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE '16)*. ACM, 234–245. http://doi.acm.org/10.1145/512529.512558

[7] Simon F. Goldsmith, Alex S. Aiken, and Daniel S. Wilkerson. 2007. 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)*. ACM, New York, NY, USA, 395–404. https://doi.org/10.1145/1287624.1287681

[8] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. 1982. Gprof: A Call Graph Execution Profiler. In *Proc. 1982 ACM SIGPLAN Symposium on Compiler Construction (SIGPLAN '82)*. ACM, New York, NY, USA, 120–126. https://doi.org/10.1145/800230.806987

[9] Jianmei Guo, Krzysztof Czarnecki, Sven Apely, Norbert Siegmundy, and Andrzej Wasowski. 2013. Variability-Aware Performance Prediction: A Statistical Learning Approach. In *Proc. 28th IEEE/ACM International Conference on Automated Software Engineering (ASE'13)*. IEEE Press, Piscataway, NJ, USA, 301–311. https://doi.org/10.1109/ASE.2013.6693089

[10] Matthias Hauswirth, Peter F. Sweeney, Amer Diwan, and Michael Hind. 2004. Vertical Profiling: Understanding the Behavior of Object-Priented Applications. In *Proc. 2004 OOPSLA (OOPSLA '04)*. ACM, New York, NY, USA, 251–269. https://doi.org/10.1145/1028976.1028998

[11] Johannes Henkel and Amer Diwan. 2003. Discovering Algebraic Specifications from Java Classes. In *Proc. 17th ECCOP (ECOOP '03)*. 431–456.

[12] Johannes Henkel and Amer Diwan. 2004. A Tool for Writing and Debugging Algebraic Specifications. In *Proc. 26th ICSE (ICSE '04)*. IEEE Computer Society, Washington, DC, USA, 449–458. http://dl.acm.org/citation.cfm?id=998675.999449

[13] Kenneth Hoste, Aashish Phansalkar, Lieven Eeckhout, Andy Georges, Lizy K. John, and Koen De Bosschere. 2006. Performance Prediction Based on Inherent Program Similarity. In *Proc. 15th International Conference on Parallel Architectures and Compilation Techniques (PACT '06)*. ACM, New York, NY, USA, 114–122. https://doi.org/10.1145/1152154.1152174

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

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

[16] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2009. Producing Wrong Data Without Doing Anything Obviously Wrong!. In *Proc. 14th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*. ACM, New York, NY, USA, 265–276. https://doi.org/10.1145/1508244.1508275

[17] Sharon E. Perl and William E. Weihl. 1993. Performance Assertion Checking. In *Proc. 14th ACM Symposium on Operating Systems Principles (SOSP '93)*. ACM, New York, NY, USA, 134–145. https://doi.org/10.1145/168619.168630

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

[19] runkit 2019. runkit. http://php.net/manual/en/book.runkit.php.

[20] George R. Terrell and David W. Scott. 1992. Variable Kernel Density Estimation. *Annals of Statistics* 20, 3 (09 1992), 1236–1265. https://doi.org/10.1214/aos/1176348768

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

[22] Eno Thereska, Bjoern Doebel, Alice X. Zheng, and Peter Nobel. 2010. Practical Performance Models for Complex, Popular Applications. In *Proc. 2010 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '10)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/1811039.1811041

[23] Jeffrey S. Vetter and Patrick H. Worley. 2002. Asserting Performance Expectations. In *Proc. 2002 ACM/IEEE Conference on Supercomputing (SC '02)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1–13.

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

[25] Dmitrijs Zaparanuks and Matthias Hauswirth. 2012. Algorithmic Profiling. In *Proc. 2012 PLDI*. 67–76. http://doi.acm.org/10.1145/2254064.2254074