ABSTRACT

Functional testing is widespread and supported by a multitude of tools, including tools to mine functional specifications. In contrast, non-functional attributes like performance are often less well understood and tested. While many profiling tools are available to gather raw performance data, interpreting this raw data requires expert knowledge and a thorough understanding of the underlying software and hardware infrastructure.

In this work we present an approach that mines performance specifications from running systems autonomously. The tool creates performance models during runtime. The mined models are analyzed further to create compact and comprehensive performance assertions. The resulting assertions can be used as an evidence-based performance specification for performance regression testing, performance monitoring, or as a foundation for more formal performance specifications.

Categories and Subject Descriptors
C.4 [Performance of Systems]: Measurement techniques, Modeling techniques; D.2.1 [Software Engineering]: Requirements/Specifications

General Terms
Measurement, Performance

Keywords
Specification mining, performance modeling, regression testing

1. INTRODUCTION

In many software projects, functionality is king. Functionality drives the project, and explicit performance specifications are missing: good performance is defined as the absence of performance regressions. Performance improvements are made on a need basis, that is after they manifested as performance bugs.

We believe the way forward are tools — tools that do not require human interventions and support software development by finding patterns and mining performance models and specifications that are easy to understand.

Previous work on performance focuses largely on high-performance computing [2, 6]. Typically high-performance computing incurs high execution costs. This justifies significant investments in performance optimizations and detailed performance modeling [12]. Other scenarios require up front performance specifications, too. For instance, if latency is a key consideration. In contrast, in many other applications performance is less mission critical. In these scenarios it is hard to justify the heavy investment required to explicitly specify and verify a performance model.

We target applications in which performance is not a primary concern but an important secondary objective. Many consumer-grade applications fall into this area (e.g. word processors, web browsers). We mine performance models during runtime automatically and autonomously. Our goal is to mine performance specifications of deployed applications in the field. The mined models specify the expected performance behavior and can be used to monitor deployed system and detect performance deviations. Insights gained by inspecting the gathered models might also be used to verify previously specified performance requirements and augment traditional approaches towards elicitation of performance specifications. In this paper we make the following contributions:

• We present a tool that extracts performance specifications during runtime. These specifications come in the form of performance models. The models are an evidence-based performance specification that describe the expected behavior of a system. We automatically detect interesting key features in the models and analyze their root causes. The root causes are captured in performance assertions. Assertions are small and easy to understand. We use dynamic binary instrumentation to enable adaptation of data collection during runtime. The adaptation operates autonomously and collects data incrementally. It does not require human guidance.

• We present a data analysis approach that extracts comprehensive performance models. We show that mined models are reliable and accurate. The models capture performance in relative terms, not absolute values. This increases platform independence and in crucial if we want
to mine performance specifications collaboratively across platforms.

• In our evaluation, we show that the tool mines performance models autonomously. Mined models are reproducible and consistent between multiple executions and across platform boundaries.

The paper is structured as follows. In Section 2 we give a brief, high-level overview of the system. Specific aspects are discussed in detail in Section 3. We evaluate our approach in Section 4. We review related work in Section 5. Finally, we examine possible future work and conclude the paper in Section 6.

2. OVERVIEW

Thorough performance testing helps to increase the quality of software and reduce the number of in-field defects. However, we observe that performance testing sometimes fail to get the required attention. We believe this is due to two factors: cost and time. In this work we introduce an automatic performance specification mining approach that can operate on deployed software and helps to reduce costs by extracting performance models from deployed systems autonomously without human guidance.

Figure 1 shows a performance model mined by our approach from an execution of Apache 2.0.64. Depending on the calling context the __semop functions exhibits 3 different performance behaviors. If the call path to __semop includes the edge e1, then we expect the __semop function to follow performance behavior #1. If the path includes edge e2, we expect behavior #2. Finally, if none of the 2 edges is included, we expect behavior #3. The key insight is that behavior #3 is very different from #1 and #2. Behavior #3 postulates that the execution of __semop will be faster than 70ms in 99% of the cases, compared to 41.9% for behavior #1 and 35.5% for #2. It is easy to understand why there is a performance difference: behavior #3 reflects the path that releases a mutex, whereas behavior #1 and #2 are on paths that acquire one. The model was collected autonomously and illustrates the capabilities of our approach. We discuss this model in greater detail in our evaluation in Section 4.3.

Mining such a performance specification is not easy. Because software is complex, it is possible to collect many different types of data at many different locations. Collecting all the data at all locations in parallel would result in a high performance overhead, probably rendering the system unusable. To tackle this challenge, our tool explores the monitored system incrementally. It starts with a minimal amount of data collection and uses the collected data to steer the mining process to promising targets, progressively accumulating an expressive body of knowledge.

Figure 2 provides a high level overview of our system. A software system is deployed across multiple sites. To support ongoing software engineering tasks (e.g., software evolution), we want to mine in-field performance specifications. The deployed and running systems are registered with a local stub controller that connects to a remote mining controller. The mining controller controls which information is collected in each deployed system. Once registered, the deployed systems start to generate and aggregate data, which is sent to a storage backend. One or more analysis processes retrieve the collected data, correlate it, and mine performance models and assertions. During the analysis the tool also judges about the stability and quality of the data and recommends next steps to the mining controller. The mining controller reviews the recommendations and selects a subset. Next, it sends instrumentation requests to the stub controller of the deployed systems leading to an adaptation of the information collected at each deployed system. This design is similar to the one presented in [22].

3. MINING AND ANALYSING DATA

Figure 3 presents a flowchart, illustrating how the tool decides which data to collect where. Assuming that the user did not provide hints on where to collect which kind of data via command line options, collection starts by detecting hot code functions. These are functions that are executed frequently. The tool simply samples the instruction pointer at a 100 ms interval. In the absence of user-supplied hints, measuring frequently executed functions is reasonable; stakeholders are typically more interested in the execution characteristics of frequently executed functions.

Once the system detects a hot function, it starts to monitor the function by accumulating timing data. The tool aggregates individual timings into histograms that are sampled as observations. After data is collected, it is analyzed. As a first step the system detects whether it collected a sufficient amount of data. To this end we partition the data into multiple clusters (Section 3.1) and compare the partitioning across time (cf. Section 3.2). If the partitioning does not change, we call it stable.

If the timing behavior is not stable, the system obtains more data until a threshold is met. The default threshold is set at 25,000 data points. It can be changed via a command
Figure 3: Incremental data collection flowchart.

line option. However, in our experience 25,000 data points are sufficient to detect stable key features if they exist.

If the system finds a stable partitioning, then this indicates that the function shows distinguishable timing behavior. As a next step the system continues to collect not only the time but also the call stack. The goal of collecting the call stack is to explain why the different clusters exist. The tool uses the gathered call stacks to extract call path patterns and stores them for later reference (cf. Section 3.3 and 3.4).

Finally, after the system collected timing behavior and call path patterns where appropriate, there is nothing else to do for this specific function at the moment. The tool continues by detecting call sites of the function. Detected callers are fed back to the system and used as new starting points. In this way the system propagates data collection up the call tree and incrementally covers the monitored application.

3.1 Data Partitioning

To partition the data, we tried Gaussian mixture models and peak detection, with limited success. Only a few of our collected distributions fit a Gaussian mixture model well. Peak detection works well with exponential distributions, but the resulting partitions tend to be fragile.

We settled on a different, third approach. The tool extracts the local minima of the gradient of the cumulative distribution function. The gradient is low at locations with a low density of observed values. The system partitions the data into multiple disjoint clusters along those areas of low density. Since each location partitions the data into two clusters, we call these locations cluster borders.

Sometimes our tool finds multiple cluster borders (local minima) in close proximity. These neighboring borders do not represent multiple useful partitions. Instead, they are artifacts caused by the limitations of the collected data. In particular the data is discrete, has limited precision and accuracy, and is finite.

To accommodate this, the tool aggregates detected borders that are close to each other. It does not match borders directly. Small fluctuations in the collected data might have a big impact on the precise location of the detected borders. Instead, the tool matches two borders if their mapped cumulative distribution function (CDF) values are close. For example, consider Figure 4. Let us assume the tool detected two local minima, \( x_1 \) and \( x_2 \). The distance between the locations of these two minima might be large. However, their CDF values are very close. Because the CDF values do not differ much, the number of elements between the two is small. These few elements are the only data points that will change cluster membership depending on whether the tool partitions at \( x_1 \) or \( x_2 \). Because the difference in the resulting labeling is small, both minima partition the data virtually in the same way. Our tool considers both borders as representatives of the same key feature if the difference between their CDF values is less than 2%.

**Definition 1.** Let \( F \) be a cumulative distribution function. Two positions \( x_1 \) and \( x_2 \) on the x-axis match with respect to \( F \) if \( F(x_1) \) and \( F(x_2) \) differ by less than 2% in absolute terms.

\[
\text{match}(x_1, x_2, F) \overset{\text{def}}{=} |F(x_1) - F(x_2)| < 0.02
\]

The threshold value of 2% can be set by the user. The value depends on the noise in the data as well as the preferred level of detail. We established 2% as a good threshold value empirically. Besides accommodating noise it prevents the detection of very small clusters by aggregating them into larger ones. This reduces the complexity of the resulting partitioning and led to good results in our evaluation studies.

After borders are matched, the tool picks one to represent the whole set. Most often the set only contains a single element and choosing one is trivial. In all other cases it chooses the leftmost one. Since all matched borders are close neighbors, the difference between them is small and the method of selection is non-critical.

3.2 Utility Score

Data collection is separated into what to collect and where to collect it. The decision is based on a utility score that quantifies how useful it would be to collect a specific type
of data at a specific location. We use the utility score to decide which type of data to collect next. We only collect data for the functions with the highest utility score. The utility score should have three properties:

1. If new data is different from previously collected data, collecting it did add information and potentially increased our understanding of the system. The utility score should be high.
2. If collected data stays the same over an extended period of time, we expect it to stay the same in the near future as well. Collecting this data will not add much value, and the tool could use the available resources more productively. For example, it could collect a different type of data instead. Thus, if the data stays the same for some time, then the utility of collecting this type of data at this specific location is low; the utility score should reflect this and should be low as well.
3. As the monitored system continues to be used, observed behavior might change due to shifting workload, evolving complexity of state, or a multitude of other reasons. Thus, collected data gets stale after some time; its value decreases over time. We have to re-collect it regularly to confirm that it is still accurate. Consequently, the expected utility of re-collecting the data increases over time. Thus, the utility score should increase over time, too.

From these three properties it is apparent that we have to be able to judge whether the data is the same over a period of time. To this end, we separate the stream of observed data into discrete observations.

**Definition 2.** An observation \( o \) is a tuple \((t_s, t_e, d, \tau)\) with \( t_s \) being the time collection of this observation started, \( t_e \) the time collection stopped, and \( d \) the raw data collected in the time frame \((t_s, t_e)\). \( \tau \) identifies the type of the collected data, for example whether it is timing data or call stack data. Then \( \text{start}(o) \) returns the start time \( t_s \) of \( o \), and \( \text{end}(o) \) returns the end time \( t_e \) of \( o \).

**Definition 3.** Let \( O_\tau \) be the set of all observations collected so far that are of type \( \tau \). We can define an order relation using the start time of the observations.

\[
o_1 < o_2 \quad \text{def} \quad \text{start}(o_1) < \text{start}(o_2)
\]

Since no two observations in \( O_\tau \) have the same start time, \( O_\tau \) is a totally ordered set.

**Definition 4.** Let \( O_\tau \) be a totally ordered set of observations. Then \( \text{last}(O_\tau) \) emits the newest observation in \( O_\tau \).

\[
o_1 = \text{last}(O_\tau) \quad \iff \quad o_1 \in O_\tau, \forall o_2 \in O_\tau \setminus \{o_1\}, \quad \text{start}(o_1) > \text{start}(o_2)
\]

\( \text{prev}(o) \) returns the observation immediately preceding \( o \).

\[
o_1 = \text{prev}(o_2) \quad \iff \quad o_1, o_2 \in O_\tau, \text{start}(o_1) < \text{start}(o_2), \quad \exists o_0 \in O_\tau, \text{start}(o_1) < \text{start}(o_0) < \text{start}(o_2)
\]

To determine whether to collect data of a certain type \( \tau \) at a specific location, the tool calculates the utility score. It compares newly gathered observations with all the other observations in \( O_\tau \). Our tool leverages the partitioning approach introduced in Section 3.1. It compares detected borders in new observations with borders extracted from previously collected observations. If the borders match, then all observations support the same partitioning.

**Algorithm 1 Calculate a set of common borders for the observations in \( O \).**

1: function \text{FINDSTABLEBORDERS}(O, \theta)  
2: \( F \leftarrow \text{CDF}(\text{MERGE}(O)) \)  
3: \( B \leftarrow \emptyset \)  
4: \( B_{\text{CDF}} \leftarrow \emptyset \)  
5: for all \( o \in O \) do  
6: \quad for all \( b \in \text{GETBORDERS}(o) \) do  
7: \quad \quad \( B_{\text{CDF}} \leftarrow B_{\text{CDF}} \cup \{F(b)\} \)  
8: \quad \text{BorderClusters} \leftarrow \text{MEANSHIFT}(B_{\text{CDF}})  
9: \quad for all \( C \in \text{BorderClusters} \) do  
10: \quad \quad if \( |C| \geq \theta * |\tau| \) then  
11: \quad \quad \quad \( B \leftarrow B \cup \{\text{centroid}(C)\} \)  
12: return \( B \)

**Algorithm 2 Find the first observations that matches \( B \) with all subsequent observations also matching \( B \).**

1: function \text{GETFIRSTSTABLE}(O_\tau, B, F)  
2: \( \text{last}_\text{match} \leftarrow \text{undefined} \)  
3: \( o \leftarrow \text{last}(O_\tau) \)  
4: while \( \text{matchBorderSets}(\text{GETBORDERS}(o), B, F) \) do  
5: \( \text{last}_\text{match} \leftarrow o \)  
6: \( o \leftarrow \text{prev}(o) \)  
7: return \( \text{last}_\text{match} \)

**Definition 5.** A set of borders \( B_1 \) matches another set of borders \( B_2 \) with respect to a cumulative distribution function \( F \) if we can match every border in \( B_1 \) with at least one border in \( B_2 \).

\( \text{matchBorderSets}(B_1, B_2, F) \iff \forall b_1 \in B_1 \exists b_2 \in B_2 \text{match}(b_1, b_2, F) \)

The tool does not directly compare new observations with old ones. Instead, it extracts a set of borders that can be found in the vast majority of previous observations. The reason for this extra step is simple: while most borders can be detected reliably in all observations, a few are more fragile. Sometimes a rather small difference in the collected data causes our data classification to detect a border in one observation but not in the other.

Algorithm 1 presents how our tool extracts stable borders. First, the algorithm merges all input observations \( O \) and obtains the corresponding cumulative distribution function (CDF) (line 2). Next, it iterates over all observations in \( O \). GETBORDERS implements the data partitioning approach described in Section 3.1. It maps the detected borders to their respective CDF value and accumulates them in \( B_{\text{CDF}} \) (lines 5 to 7). Next, it applies the common mean shift algorithm [8]. The mean shift algorithm returns a set of sets, each containing borders that belong to the same cluster. The algorithm filters clusters that have less than a threshold of \( \theta * |\tau| \) elements\(^7\) and aggregate the centroids of the remaining clusters in \( B \) (lines 9 to 11). Finally, it returns \( B \).

The tool compares new incoming observations to this set of stable borders \( B \). Algorithm 2 iterates over all observations starting with the newest (i.e. last) observation, \( \text{last}(O_\tau) \). It stops as soon as it finds a non-matching observation. It returns the last observation that matched.

\(^7\)In our evaluation we use a threshold value \( \theta \) of 0.9.
Using Algorithm 1 and Algorithm 2, the tool calculates the utility score in Algorithm 3. First, it determines stable borders for the last 10 observations, using Algorithm 1 (line 2 and 3). Next, it finds the first observation \( o_{first} \) that fits the extracted stable borders under the condition that all newer observations \( o_t > o_{first} \) also fit, using Algorithm 2 in line 5. Finally, with \( t_{start} = \text{start}(o_{first}) \) and \( t_{end} = \text{end}(\text{last}(O_s)) \), the utility score is calculated as

\[
    u = \min \left( 1, \frac{\text{current time} - t_{end}}{t_{end} - t_{start}} \right)
\]

Note that the utility score fulfills our requirements and is (1) low if observations are stable for a longer time, (2) high if new observations are significantly different from previous ones, and (3) increases over time so that data is periodically re-verified.

### 3.3 Capturing Performance Models

If the tool detects multiple performance classes for a specific function using our partitioning approach from Section 3.1, then it attempts to explain why there are different classes (compare the flowchart in Figure 3 introduced in Section 3).

To locate the cause for the existing clusters, the tool has to collect additional data. At the moment it collects the call paths that lead to a specific function. The assumption is that the calling context is an important factor that might contain crucial information to help discriminate between different performance behaviors.

Alternatively, one might also collect different data, for example, function argument values. Arguments are likely to influence the executing time of the called function. However, arguments of functions suffer from the deficit that they require interpretation. Except for trivial arguments, interpretation requires in-depth understanding of their structure and contained data. This calls for either manual annotation or elaborate static or dynamic analysis. In contrast, call paths are easy to capture and easy to interpret. Furthermore, in our experience they are sufficient to gain significant insights into system behavior.

If our approach finds interesting features in the timing behavior of a specific function (i.e., it finds at least one stable cluster border), then it automatically augments the data collection to extract not only the execution time but also call path information.

Using the call path information the tool examines whether the different classes can be explained by differences in the call paths traversed to reach the function. Is there a specific set of call paths that show similar performance behavior compared to all other call paths? If yes, we found a performance pattern that occurs only if the function is reached via a specific set of call paths.

Finding these sets of call paths is a two step process. First, our system analyzes the aggregated data of a specific function \( f \) and detects stable cluster borders. We described this step previously in Section 3.2. Algorithm 1 returns a set of stable borders. Second, it uses the detected cluster borders to classify observed execution data and group call paths that behave similarly. The borders partition the data into multiple stable clusters, \( C \).

**Definition 6.** Let \( T_p \) be the set of execution times observed while reaching the function via call path \( p \). Let \( C \) be a set of stable clusters. We assign each \( t \in T_p \) to a cluster that fits best, using \( f(it(t)) \). Next, we count for each cluster \( c \in C \) how many of the observed execution times \( t \in T_p \) are a member of this cluster.

\[
    \text{count}(c) \overset{\text{def}}{=} |\{t \in T_p, f(it(t)) = c\}|
\]

We normalize the result and obtain the normalized cluster distribution vector \( b_p \) for call path \( p \).

\[
    b_p = \left( \frac{\text{count}(c_1), \text{count}(c_2), \ldots, \text{count}(c_n)}{|T_p|} \right)
\]

The \( i \)-th element in \( b_p \) represents the percentage the executions times for path \( p \) fall into cluster \( c_i \in C \).

To decide whether two call paths \( p_1 \) and \( p_2 \) have similar behavior, the tool groups call paths using their normalized cluster distribution vector \( b_p \). It clusters these points using KMeans clustering. On important aspect of using KMeans is to choose \( k \), the number of clusters. We start with \( k = 1 \) and increase \( k \) as long as the adjusted mutual information score [26] increases. KMeans groups the call paths into disjoint sets \( G_1, G_2, \ldots, G_n \). Call paths that end up in the same group have similar \( b_p \). During runtime they are expected to behave similarly.

**Definition 7.** A performance model \( M \) is a tuple \((f, G, B, \lambda)\) with \( f \) being a function in the binary of an application. Let \( G \) be a grouping of call paths \( G = \{G_1, G_2, \ldots, G_n\} \), \( B \) a set of normalized cluster distribution vectors \( B = \{b_{c_1}, b_{c_2}, \ldots, b_{c_n}\} \), and \( \lambda \) a bijective mapping \( \lambda: G \rightarrow B \).

Thus, a performance model consists of a set of grouped call paths. The behavior of each call path group is described by one cluster distribution vector.

To denote the performance model for function \( f \), we write \( M_f \) for short. Whenever we need to distinguish models collected in different executions we write \( M_{f}^n \) to identify the model \( M_f \) collected in the \( n \)-th execution.

We compare multiple performance models in our evaluation to reason about reliability and platform independence of mined performance models. It is not difficult to compare two performance models. A difference occurs when two call path \( p_1 \) and \( p_2 \) are members of the same group in one model while they belong to two different groups in the other model. If we detect such a difference between the models, than one model postulates that both call paths should behave the same while the other model postulates that they should behave differently.

It is noteworthy that a performance model itself does not contain any direct notion of execution time. Execution times are expected to be very different across platforms. Instead, it groups call paths by behavior. Call paths in the same group are expected to behave similarly independent of concrete
values of execution times. Abstracting the execution times and only retaining their relationships is crucial to make a performance model portable across platforms.

3.4 Extracting Performance Assertions

Performance models tend to be large and contain a lot of information. Except for very small models the complexity of the models directly impacts readability. Complex models are difficult to understand.

To simplify a performance model $M_f$, our tool generalizes it. It searches for a minimal set of call edges that is sufficient to distinguish all call path groups from each other.

Our approach uses a common $A^*$ algorithm to find these edges. As input we supply the call path groups ($G_1, G_2, ..., G_n$). For each input path group $G_i$ the algorithm returns a boolean expression, $E_i$. Thus, as output we obtain a vector of boolean expressions $(E_1, E_2, ..., E_n)$. The atoms of the boolean expression represent call edges. The tool uses the total number of call edges as a cost function. Thus, the returned expressions are guaranteed to use a minimal number of call edges to distinguish between the call path groups.

A boolean expression can be instantiated with a call path. To instantiate an expression with a call path $p$, we substitute an atom with True if and only if the corresponding call edge is present in $p$. We write $E_i(p)$ to denote an expression $E_i$ that was instantiated with call path $p$.

The boolean expressions filter call paths. If and only if a previously observed call path is a member of the path group $G_i$, then the boolean expression evaluates to true. Thus, for all observed call paths $P$ with $P = \bigcup_{i=0}^{n} G_i$ the resulting expressions satisfy the two constraints:

$$E_i(p), \forall p \in G_i \quad E_i(p), \forall p \in P \setminus G_i$$

The expressions are a building block for compact performance assertions.

Definition 8. A performance assertion is a tuple $\langle f, ((E_1, b_{c1}), (E_2, b_{c2}),..., (E_n, b_{cn})) \rangle$ with $f$ being a function in the binary of an application. Let $E_i$ be a boolean expression that identifies all paths in cluster $G_i$, and $b_{ci}$ the aggregated normalized cluster distribution vector of $G_i$.

For example, assume the tool observed that function $z$ is reached via three different call paths $(p_1, p_2, p_3)$:

$$G_1: p_1 = (a \rightarrow b \rightarrow c \rightarrow z) \quad G_2: p_2 = (a \rightarrow b \rightarrow z) \quad G_2: p_3 = (a \rightarrow c \rightarrow z)$$

$a, b, c, z$ are functions in the monitored application. $a \rightarrow b$ means that function $a$ calls function $b$. Furthermore, assume function $z$ shows different performance behavior during runtime depending on the context of the call. For example, $z$ always executes very fast if called via $p_1$. However, if $z$ is called via $p_2$ or $p_3$ it takes a bit longer to complete in 20% of the cases.

Our tool will detect the difference in performance behavior and group the three call paths into two groups ($G_1$ and $G_2$), using the mechanisms described in Section 3.3. Next, the system will use the $A^*$ algorithm to extract the minimal set of edges that are sufficient to distinguish all call paths in $G_1$ from all call paths in $G_2$. The minimal solution is $E_1 = e$ and $E_2 = \neg e$ with $e$ representing the edge $b \rightarrow c$. The path groups $G_1$ and $G_2$ can be distinguished by the single edge $e$. All call paths in $G_1$ contain the edge $b \rightarrow c$ while all paths in $G_2$ do not contain it.

Using these two expressions the tool will create the following performance assertion $a_1$:

$$a_1 = (z, \{(b \rightarrow c), (1.0, 0.0)\}, (-b \rightarrow c), (0.8, 0.2))$$

This performance assertion $a_1$ captures the condensed essence of the collected data. If function $z$ is called via a call path that contains edge $b \rightarrow c$, then we expect all calls to behave the same and execute fast. However, if edge $b \rightarrow c$ is not present in the call path, only 80% of the calls should execute fast while the remaining 20% are expected to behave differently; they should execute slower.

4. EVALUATION

Our evaluation is designed to answer the following research questions:

RQ1: Can we mine performance models reliably?

RQ2: Are these performance models platform independent?

RQ3: How difficult is it to understand the resulting performance models and assertions?

RQ4: How high is the overhead incurred by data collection?

RQ5: Does data collection perturb the system behavior?

We evaluate our approach on three subjects, Apache 2.0.64, MySQL 5.0.88, and Okular 0.19.3. These subjects cover different aspects of the design space. MySQL 5.0.88 is multithreaded; Apache httpd 2.0.64 uses multi-processing; and Okular 0.19.3 is a GUI application.

To answer RQ1-RQ3, we run Apache and expose it to the banking workload of the SPEC-web2009 benchmark version 1.00.3 We run a single SPEC-web2009 client on the same machine as Apache to simplify the setup. We reduced the mean wait time between requests in SPEC-web2009 to speed-up execution of the benchmark. We run our experiments on two different machines. Machine A is equipped with an Intel Core i7-2600 processor with 3.4 GHz and 8 GB of RAM. Machine B is equipped with two Intel Xeon E5-2680 processor with 2.7 GHz and 32 GB of RAM.

4.1 RQ1: Reliability

To show that our approach mines performance patterns reliably, we run the experiment five times each on two different machines. Then we compare the models mined on the same machine with each other. In this way we show that we can mine coherent models in fixed environments.

Each execution of the experiments creates a set of performance models $M^f = \{M^f_1, M^f_2, ..., M^f_n\}$ for functions $f_1, f_2, ..., f_n$. Figure 5 summarizes our findings for all ten measurements using box plots. To create a graph, we compare collected performance models. We compare models using the mechanism introduced in Section 3.3, that is we check whether two models group the call paths in the same way. Each data point represents the statistics of a performance model $M^f$ of a specific function $f$.

On the x-axis we show four entries #2 to #5. Each data point in entry #2 is created by comparing the model $M^f_1$ for a single function $f$ obtained during the first execution with the corresponding model $M^f_2$ collected during the second execution, i.e. $cmp(M^f_1, M^f_2)$. To create the data points for entry #3, we merge the two models and compare it with the model gathered during the third execution: $cmp(merge(M^f_1, M^f_2), M^f_3)$. Similarly, for data points

1https://www.spec.org/web2009/
we can still match more than 80% of the data points collected at machine A for most models (Figure 5a, right). In fact, the box plots are hardly noticeable because the tool can align 100% of the observations for many models. There are two outliers at about 40% for which our analysis fails to create reliable performance models. We tested whether problems during the merging of behavioral clusters could be caused by complex models. We were unable to find a correlation. We conclude that the performance behavior of these two functions is just too erratic to allow merging.

In Figure 5b we present equivalent measurements collected on machine B. The results are even slightly better than on machine A. In summary we succeed in collecting reliable performance models on both machines.

**Answer to RQ1:** Most performance models can be reliably aligned across multiple executions.

### 4.2 RQ2: Platform Independence

After showing that performance models can be reliably created and matched across multiple executions on the same machine, we proceed to show that models can also be merged across machines. This illustrates the ability to mine stable performance models across different hardware platforms.

We align the models collected on machine A with models created on machine B. Note that the two machines not only have different hardware configurations, but also run slightly different binaries; even though we run the same software, we compiled the binary on each machine; thus, the binaries are different. Especially some functions present in the version on machine B have been inlined in the version on machine A. While this creates some difficulties during merging, a reliable approach should handle these challenges.

To show whether performance models can be aligned across platforms, we merge the models collected on machine A with the corresponding models collected on machine B. We merge the models at position $n$ in Figure 5a with the corresponding models at the same position in Figure 5b. The resulting statistics for the merged models is depicted in Figure 5c. For most functions the performance models can be merged across different executions on different machines.

**Answer to RQ2:** Performance models collected on different platforms and with different binaries show high stability and can be matched successfully to a large extent.

### 4.3 RQ3: Complexity

Mined performance models tend to be complex. We already presented a small example of a performance model extracted from Apache 2.0.64 in Figure 1.\(^4\) Other models tend to be much larger. The size of the performance models motivated our approach to reduced them to their key components (cf. Section 3.4).

Our tool found the model depicted in Figure 1 in the following way. The approach detects \_\_semop as a frequently executed function; it starts to measure the execution time. After collecting some data the tool analyses it and finds one stable border at 70ms. This border separates the observed data into two distinct behavioral clusters. Instead of reporting the whole performance model and hoping for proper

---

\(^4\)We chose to show the model for the \_\_semop function mainly because the model is small enough to fit into the manuscript. A larger model for poll and Apache 2.0.64 is available at [https://drive.google.com/file/d/0B9FeGNLlmM1aTW1xdxhlREhiZDA](https://drive.google.com/file/d/0B9FeGNLlmM1aTW1xdxhlREhiZDA)
Using the number of unique edges in the performance assertions is just the number of atoms in the boolean expressions. In Figure 6 we compare the number of behavioral clusters to the number of atoms. Not surprisingly, the number of atoms increases with the number of clusters. Performance models that represent more complex performance behavior, that is models with many clusters, require expressions with more atoms to distinguish between these behaviors.

**Answer to RQ3**: Mined performance models tend to be complex. We succeed to reduce them to performance assertions that use small boolean expressions to distinguish performance behavior. With a maximum number of eight required atomic formulae we believe our performance assertions are compact enough to be read and understood by stakeholders.

### 4.4 RQ4: Performance Overhead

We collect performance data during execution. To collect this data, we add instructions into the running application. Executing these instructions has an effect on many measurable metrics; especially execution time is easily affected.

![Figure 7: Performance overhead of MySQL 5.0.88.](image)

We evaluated the performance impact of our mining approach using all three subjects, Apache, MySQL, and Okular. The experiments in this subsection are executed on machine A.

Okular is the default pdf viewer under KDE. To get a first impression, we mined performance specifications from Okular 0.19.3 while we were using it for our daily work. Not surprisingly, we did not perceive any performance impact. Okular is a GUI application and thus waits for user input most of the time. To assess the actual overhead incurred, we measured the time it took to parse and render a 19 MB large pdf file containing 3883 pages. Once the file is loaded we load it again. In this way we eliminate idle time and keep the application busy at 100% CPU load. In total we load the file 10 times. Mean as well as median time to load the file is 12.74 seconds with $\sigma = 0.03$. Next, we employ our performance mining approach. Mean time increases by 16.0% to 14.8 seconds and median time increases by 13.9% to 14.5 seconds ($\sigma = 0.94$).

The impact of data collection on execution time is hard to predict and often does not correlate linearly with the amount of instrumentation inserted. Figure 7 presents an extreme example. We started a single instance of the MySQL server. Next, we ran the `test-select` benchmark. Once the benchmark terminates, we start it again leading to another iteration (x-axis). We leave the same MySQL server running across benchmark invocations.

Mining performance specifications for MySQL 5.0.88 leads to a speedup for most iterations. We attribute this speedup to cache and pipeline effects. Even minor and seemingly harmless changes to the execution environment can have a dramatic impact by the stakeholders, the system extracts distinguishing features that discriminate different performance behavior (Section 3.4). After finding the stable border at 70ms our tool continues and collects the execution times as well as the call stack.

Analyzing the gathered call stacks, the tool detects three distinct performance behaviors. Using the $A^n$ algorithm it finds two edges that are sufficient to distinguish between the three performance behaviors. Our algorithm chooses edges $e_1$ and $e_2$ in Figure 1. The resulting performance assertion annotated with concrete time values is shown in Table 1.

**Performance behavior #3** is the most distinguished one. Whenever we enter the `__semop` function and we neither passed edge $e_1$ nor edge $e_2$, that is we reached `__semop` from `proc_mutex_sys_release`, `__semop` returns after less than 70ms in 99% of the cases. The other two performance behaviors, #2 and #3, are very different from #1. For behavior #2 64.5% of the executions take longer than 70ms. Behavior #3 is similar: 58.1% of the executions take longer than 70ms. Both pass through `proc_mutex_acquire`. Behaviors #2 and #3 are similar, but the difference is significant across multiple executions. Using only two unique atoms, edges $e_1$ and $e_2$, we can reduce the performance model of `__semop` with three different behavioral clusters to three simple boolean expressions. These expressions are shown in the second column of Table 1.

We believe that these expressions can help stakeholders to understand performance differences. Investigating performance behavior by scrutinizing the complete call tree can be a daunting task, especially for larger trees. Instead, a stakeholder can inspect the boolean expressions. These expressions quickly point to the important differences between call paths that result in different performance behavior.

How difficult it is to understand the mined performance models and assertions? To answer this question (RQ3), the best approach would be to measure the perceived complexity using a large population of engineers that leads to statistically sound results. Unfortunately, we do not have such a large population at our disposal.\(^5\) Thus, we use a proxy metric and quantify the complexity of a performance model using the number of unique edges in the performance assertion. The number of unique edges in the performance assertion is just the number of atoms in the boolean expressions.

**Table 1**: The performance assertion extracted from the model of `__semop` (cf. Figure 1).

<table>
<thead>
<tr>
<th>Behavior</th>
<th>Expr.</th>
<th>$t &lt; 70ms$</th>
<th>$t \geq 70ms$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$e_1 \land \neg e_2$</td>
<td>41.9%</td>
<td>58.1%</td>
</tr>
<tr>
<td>2</td>
<td>$\neg e_1 \land e_2$</td>
<td>35.5%</td>
<td>64.5%</td>
</tr>
<tr>
<td>3</td>
<td>$\neg e_1 \land \neg e_2$</td>
<td>99%</td>
<td>1%</td>
</tr>
</tbody>
</table>

\(^5\)A comprehensive field study is left as future work.
severe impact on execution times [19]; our approach modifies and rearranges the execution image, a much larger change.

Figure 7 also confirms an observation we made while evaluating Okular; the variability in the runtime measurements for the instrumented case is much larger. This is caused by our automatic adaptation approach; new data is constantly pouring in, leading to adaptations of the instrumentation and changing overhead levels.

As a third subject we ran Apache http 2.0.64. This version uses multiprocessing instead of multithreading in the default configuration. We use the same setup as in RQ1 to RQ3 except that we increased the execution time to 7.5 hours to observe long term effects. Average response time increases from 390ms (σ = 0.5) to 435ms (σ = 61.2), an increase of 11.1%.

All numbers in this subsection should be taken with a grain of salt; the incurred overhead can be easily manipulated and reduced using sampling. Our approach does not aim at minimal overhead; instead, we try to gather data as fast as possible, without incurring a too high overhead. What defines a too high overhead is subject to debate; therefore, a user can specify the target amount of overhead. In this section we used a target overhead of 10%. We could have used a smaller target overhead which would have decreased the reported performance overheads as well.

Answer to RQ4: Collecting data affects execution time. Incurred overhead is modest and stays reasonably close to our configured target overhead of 10%.

4.5 RQ5: Perturbation

The effect that data collection has on an application is interesting from two different perspectives. First, the performance overhead has to be reasonably small to ensure that the impact on total performance and throughput is manageable and does not render the system unusable. Second, measuring should not perturb the behavior of an application beyond reasonable bounds.

Collecting data at runtime not only delays execution, it is likely to perturb the collected data in unpredictable ways. To minimize perturbation of collected data, we carefully designed the collection process to inject an almost constant number of instructions into the application. We strive for a constant overhead. A constant overhead has the main benefit that it only affects the nominal execution time but not the shape of the distribution itself. To illustrate trends and get a grip on worst-case overhead and perturbation, we looked into how data collection affects the runtime of a fast executing function. We executed MySQL 5.0.88 and ran the test-select benchmark. We sampled the instruction pointer at random intervals. We collected 20,727 samples that map to 264 different unique functions. From these 264 functions we randomly selected a function that (1) executes very fast, (2) is called frequently, and (3) has a non-trivial distribution of execution times.

By chance we selected the _mi_bin_search function. To measure the effects of runtime instrumentation, we manually augmented the source code of _mi_bin_search with two calls to clock_gettime to measure the execution time. Note that manually adding instructions to the source code will itself perturb the time measurements. There is an inherent limitation towards achievable accuracy and precision regarding time measurements without relying on extra hardware support or full scale simulation. However, we expect static instrumentation to affect time measurements less than dynamic instrumentation. Thus, we feel comfortable to use the executions times collected using static instrumentation as a reference point to evaluate the relative overhead and perturbation of our dynamic instrumentation.

Figure 8 shows three graphs generated from 20 runs each. The graph at the top plots the execution times measured by static instrumentation without any instrumentation. The graph in the middle plots the execution times measured by static instrumentation while we also injected instructions to collect execution time dynamically. Thus, it represents the overhead added to a function by measuring execution times dynamically. Finally, the graph at the bottom plots the execution times measured by dynamic instrumentation. We can draw several conclusions.

First, using dynamic runtime instrumentation to measure execution time adds an average overhead of 424ns and a median overhead of 420ns.

Second, comparing the top graph with the graph in the center we notice that dynamic instrumentation not only adds overhead, but also perturbs the execution time. It follows that the overhead is not constant. We attribute the perturbation largely to cache effects. Comparing the statically instrumented version with the dynamically instrumented one, we noticed an increase in the number of instruction cache misses by 66.3% from 0.96 billion to 1.59 billion.

Third, the execution times measured by an external observer (graph in the middle) are more perturbed that the execution times collected by our approach (bottom graph). The data collected by our approach is less perturbed, because after time is measured, it is aggregated and communicated; this adds another layer of overhead and perturbation.

This observation led to another improvement in our approach. Perturbation is especially problematic if the collection of executions times is nested. We detect nested collections and annotate gathered data with statistics about the level of nesting. The analysis module takes these statistics into consideration. If the analysis module decides the
perturbation is too large to draw reliable conclusions, it ignores the data and re-collects it in isolation, that is it only collects data for one function. In this way we prevent the accumulation of perturbation and maximize data validity.\textsuperscript{6}

**Answer to RQ5:** Collected data is perturbed by the observer effect. However, even for a function we chose deliberately to illustrate the expected worst-case scenario, key features of the collected data are kept intact. This supports our perception that mined performance models reflect actual behavior.

### 4.6 Threats to Validity

We limited our evaluation to three subjects and two machines. This threatens the external validity of our results. Furthermore, we used the SPEC-web2009 banking benchmark for our evaluation. While the benchmark randomizes the workload, requests are still relatively structured with limited diversity. This might decrease the complexity of our models and might increase the correlation between mined performance models. An extensive field study would provide more support and could eradicate these threats but is beyond the scope of this paper.

### 5. RELATED WORK

Mining functional specifications focuses on extracting interaction patterns [1, 15]. System logs can also be used to mine behavioral performance models by leveraging existing timestamps [21]. It is hard to reason about non-functional properties and measure them precisely [16]. This work contributes to the existing approaches. Our approach is scalable, collects data collection in-field without human intervention, and creates meaningful performance models that aid in comprehension and can be further refined into performance assertions to enforce the models with minimal instrumentation.

Our performance assertions can be interpreted as a specification of an acceptable performance spectrum and thus might be considered an instantiation of a program spectrum [11].

In an attempt to determine the complexity of a piece of code, some related work correlates different run time parameters, most often input size, with execution cost [2, 27]. Even though it might be possible to calculate the input size dynamically at runtime for standard structures [27, 9], measuring input size for more complex structures (e.g. graphs) is harder and might require consultation of an expert. Due to this potential pitfall we do not take input sizes into account.

The notion of a performance assertion is not new [25, 24]. However, our performance assertions differ significantly in that they incorporate information about the call stack and are collected autonomously.

Automatic behavioral regression testing is an in-house functional testing tool that runs different versions of a software against generated test inputs, records changes in the behavior, and reports it to the developer [14]. Another approach uses probabilistic symbolic execution during in-house testing to approximate the distribution of execution times [7]. One major challenge of in-house performance testing is the dependency on a representative workload profile that reflects real world usage. Creating a representative test workload that reflects real-world usage is challenging [10]. As a consequence wrong workload assumptions account for more than 35% of all performance bugs [13]. Our work differs. We target in-field performance regression testing. We believe it can be used to leverage the user base and extract patterns in deployed systems to augment in-house testing [22]. In-field testing is not new. For example, it has been used to monitor different coverage criteria [23, 3], as well as exploring the configuration space of software programs [18].

To gain platform independence, Stochastic Performance Logic compares relative executions times of functions [5]. In our work we compare the grouping of call paths leading to a single functions based on the execution times instead.

We use Dyninst as an instrumentation backend [4]. We believe it is possible to use existing profilers to replace or augment our data collection. One prerequisite is to have fine grained control about what data is collected when. For example, we believe we could leverage the probes mechanism in the YourKit\textsuperscript{7} profiler to extend our data analysis to Java processes. We could also use OProfile\textsuperscript{8} to include hardware performance counters. However, the sampling nature of OProfile might proof problematic especially for functions with low total runtime. Our approach tries to be as accurate as possible; we do not sample unless the overhead actually observed during runtime is too high. Other profilers, e.g. gprof\textsuperscript{9}, require static instrumentation during compilation. We believe dynamic instrumentation is crucial to enable the monitoring of deployed applications.

Our tool is lightweight as compared to more heavyweight binary instrumentation approaches like e.g. Valgrind [20] or Pin [17].

### 6. CONCLUSION

Defining non-functional specifications like performance is costly. Regularly, especially in agile environments, precisely defining performance specifications up-front is counterproductive. We tackle these challenges and mine performance specifications automatically, without human intervention. We demonstrated that we can match models across machine boundaries. This is a crucial step to achieve our overall goal, a system that uses volunteer computing to collaboratively mine performance specifications.

There are multiple interesting avenues that should be explored in the future. It would be worthwhile to study the stability of performance models across an exhaustive set of subjects. The goal of performance specification mining is to help stakeholders accomplish their goals. A comprehensive field study that evaluates the usability and applicability of mined performance specifications would be valuable.

### References


\textsuperscript{6}http://sourceware.org/binutils/docs/gprof/

\textsuperscript{7}http://www.yourkit.com

\textsuperscript{8}http://oprofile.sourceforge.net

\textsuperscript{9}http://sourceware.org/binutils/docs/gprof/


