White-Box Performance-Influence Models: A Profiling and Learning Approach

Max Weber
Leipzig University
Germany

Sven Apel
Saarland University
Saarland Informatics Campus
Germany

Norbert Siegmund
Leipzig University
Germany

Abstract—Many modern software systems are highly configurable, allowing the user to tune them for performance and more. Current performance modeling approaches aim at finding performance-optimal configurations by building performance models in a black-box manner. While these models provide accurate estimates, they cannot pinpoint causes of observed performance behavior to specific code regions. This does not only hinder system understanding, but it also complicates tracing the influence of configuration options to individual methods.

We propose a *white-box approach* that models *configuration-dependent performance behavior* at the *method level*. This allows us to predict the influence of configuration decisions on individual methods, supporting system understanding and performance debugging. The approach consists of two steps: First, we use a coarse-grained profiler and learn performance-influence models for all methods, potentially identifying some methods that are highly configuration- and performance-sensitive, causing inaccurate predictions. Second, we re-measure these methods with a fine-grained profiler and learn more accurate models, at higher cost, though. By means of 9 real-world JAVA software systems, we demonstrate that our approach can efficiently identify configuration-relevant methods and learn accurate performance-influence models.

Index Terms—Configuration management, performance, software variability, software product lines

I. INTRODUCTION

Many software systems today are configurable, supporting multiple application scenarios, hardware platforms, and software stacks. Configuration options are used to tailor a system's behavior and its non-functional properties by (de-)activating or tuning corresponding code. Performance measures, such as response time and throughput, are among the most important non-functional properties of software systems [1], [2]. So, it is crucial to know how individual configuration decisions will influence a system's performance. Several approaches of accurately modeling and learning the performance behavior of configurable systems have been proposed in the literature [3]–[7]. The underlying idea is to sample a set of configurations from the configuration space and measure their performance. Machine-learning techniques, such as multi-variable regression [7], [8] or classification and regression trees [5], [6], [8] can be used to learn a performance-influence model from these measurements, to accurately predict the performance of unseen configurations or to find performance-optimal configurations with search-based techniques [9]–[12].
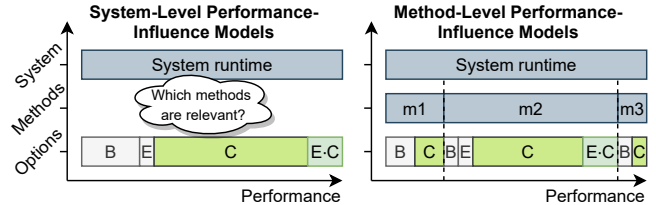

Fig. 1: Comparison of the black-box performance-influence modeling at system level (left) and white-box performance-influence modeling at method level (right), explaining how feature values at system level are composed of feature values at method level.

These and similar approaches based on parameter tuning [13] and algorithm selection [] have in common that they conceive the configurable software system as a *black box*. That is, they model the performance of a software system as a function of its configuration (i.e., a set of selected configuration options) without knowledge of the software's internals. For illustration, let us consider a database system with three features *Base* (B), *Encryption* (E), *Compression* (C), and E·C (interaction between E and C), as illustrated in Figure 1 on the left side. From such a system-level model, we can infer the most influential options (e.g., *Compression*) and possible interactions. However, we have no information about the root cause of interactions or influential options. We do not know where in the system's code base we spent execution time depending on the configuration. Essentially, developers want information at the level of individual methods, as illustrated in Figure 1 at right-hand side.

Knowing performance influences at the method level helps detecting performance bottlenecks [14], [15], pinpointing performance bugs [16], [17], or assigning performance tests in a CI pipeline to specific configurations. This is a developer's perspective, which is not supported by current black-box approaches.

From a performance-analysis perspective, there are monitoring and profiling techniques whose goal is to identify performance hot spots [18], [19]. Specialized performance engineers typically supervise performance of Web and cloud applications or identify bottlenecks using stress tests [20], [21]. Unfortunately, state-of-the-art approaches in this area

usually disregard the fact that today's software systems have huge configuration spaces. Typically, only a single configuration of a system is considered, which is insufficient since performance bugs and related issues are often configuration-dependent [17]. Few approaches aim at creating white-box performance models of configurable systems [3], [22], [23], but they either require explicit tracing information about which code regions are affected by which configuration options [3] or they rely on expensive and potentially imprecise static and dynamic program analysis [22], [23].

Our goal is to devise a white-box performance analysis technique for configurable software systems that, with high precision and at low cost, infers which methods are most affected by configuration decisions and which source-code regions exhibit the highest configuration-dependent performance variation. Our approach substantially widens the application scenarios of former black-box approaches [3]–[5] and introduces the concept of configurability and performance predictions to current white-box approaches [18], [19]. The right part of Figure 1 shows the essence of the approach: it creates a performance-influence model for each method of the system.

To realize our goal, we combine approaches from two fields: (1) profiling program behavior in a white-box manner and (2) predictive modeling of performance of configurable software systems. We use a two-step approach to direct the profiling activities to methods that are performance-relevant (i.e., contribute to a system's performance) and that are highly affected by configuration options. This way, we substantially reduce the influence of measurement overhead, thereby increasing prediction accuracy. In a nutshell, we draw samples from the configuration space based on well-established sampling strategies [24] and measure the selected system configuration with a low-overhead profiling tool. We use classification and regression trees (CART) to learn a performance-influence model per method based on the measured performance data. In this course, we identify methods that exhibit high performance variance, possibly caused by configuration options. In a second step, we re-measure all methods with high variance to improve the accuracy of the final performance models, this way, focusing on the difficult-to-learn methods. Here is the key: Methods are affected to different extents by configuration options, and only for a fraction of methods, a more fine-grained performance analysis is required. We use machine learning to find those methods.

Focusing on methods with high performance variation, we need to distinguish three causes: configuration variance, measurement variance, and a method's context variance. *Configuration variance* emerges from the (de-)selection of configuration options. This is what we aim to learn as influences of options and interactions on individual methods. *Measurement variance* corresponds to the measurement setup's inherent systematic bias. *Context variance* represents the performance variation of a method execution due to varying input parameters or program states. So, given these three types of variance, a method's performance can vary for each run of a software system, even with the same configuration. To learn accurate performance models for configurable software systems, we need to measure, distinguish, and control sources for all three types of variance.

To provide a robust empirical foundation as a base line for evaluation, we measured the performance of 9 software systems from various application domains, resulting in 19 years of CPU time of continuous measurements. We demonstrate that our approach can efficiently identify configuration-relevant methods and learn accurate performance models at the method level.

We contribute and evaluate not only an approach for learning white-box performance models, but also important empirical findings about the distribution and nature of configuration-dependent performance-relevant methods. We contribute an analysis that reveals the influence of the types of variance on the runtime of methods. These findings shall inform further work on tailoring and guiding sampling techniques, static code analyses, and performance-anomaly detection.

To summarize, we make the following contributions:

- An approach to learn white-box performance-influence models of configurable software systems at method level using profiling and prediction modeling;
- A performance analysis providing insights into statistical performance properties of methods related to configuration decisions;
- An evaluation of our approach with respect to prediction accuracy and scalability for 9 real-world software systems;
- A replication package including our implementation and measurements[1].

II. PRELIMINARIES AND RELATED WORK

A black-box performance-influence model does not contain information to map performance to source code. Yet, there exists substantial work conducting root-cause analysis, trying to locate, for example, performance bugs or memory leaks. However, prior work rarely takes configurability into account.

A. White-Box Performance Analysis

There are only few approaches that tackle performance analysis for configurable software systems. Reisner et al. [25] and Meinicke et al. [26] use symbolic and variational execution to analyze the behavior of interactions of configuration options at the level of control and data flow. They found that software systems in practice do have a much smaller relevant configuration space than theoretically possible, because only a few options interact at all. Many approaches in this area rely on this fact.

Hoffmann et al. [27] use dynamic influence tracing to convert static parameters to dynamic control variables (global variables) with the goal of adapting properties of an application. They do not consider interactions among parameters and

---

cannot pinpoint code regions of interest. Their approach works only when static parameters are convertible, whereas our approach is agnostic to the type of configuration parameters.

Family-based performance measurement [3] aims at applying family-based analysis [28] to performance analysis. The idea is to create a variant simulator, which converts compile-time variability into run-time variability [29]. Then, the variant simulator is executed incorporating variability constraints. This way, multiple variants can be executed and measured in a single run by informing the analysis which method's performance is configuration-specific. On the downside, family-based performance measurement requires the construction of a variant simulator (or other variational representation), which is, in general, a non-trivial task [30].

Lillack et al. [31] use taint analysis to identify which code fragments are executed depending on which configuration options. This static code analysis technique is used in Config-Crusher for deriving performance models [22]. Specifically, ConfigCrusher employs static data-flow analysis to identify code regions whose performance is likely to be influenced by configuration decisions. It traces individual configuration options (represented by program variables)—following the call graph—and taints code regions influenced by configuration options or combinations thereof. Subsequently, ConfigCrusher merges regions to larger ones, such that it can efficiently measure performance of each individual region. Performance is measured by weaving instructions into the byte-code representation of the target program at the beginning and end of each region.

In contrast to our approach, ConfigCrusher requires modifying the source code of the target system to make the taint analysis run. Beside integrating an interface between analysis and target system, further substantial code refactorings are required to achieve scalability and precision [32]. The background is that configuration options are often stored in complex data structures (lists, maps, structs, etc.), causing the taint analysis to no longer differentiate among the options stored in the data structures. The result is that either all accesses to the data structure are tainted with all configuration options or the analysis stops tainting at this point. The first variant leads to memory explosion and timeouts, the second results in incomplete and possibly very short taints, rendering the resulting performance models inaccurate and effectively useless. To circumvent this problem, one can always refactor the entire system such that configuration options are stored in individual variables, which is usually infeasible in practice. Finally, in contrast to our approach, ConfigCrusher does not support numeric configuration options and is limited to single-threaded applications.

Velez et al. [23] proposed COMPREX, a tool that builds white-box performance models based on dynamic taint analysis and local performance measurement. COMPREX requires expensive dynamic analysis and focuses only on configuration specific code, whereas our approach covers the whole code base by building a model for each method of the system.

B. Black-Box Performance Analysis

Obtaining accurate performance models requires a series of measurements. Conducting measurements for each possible configuration, however, is infeasible due to the combinatorial complexity of the problem. Instead, sampling a representative subset of configurations can achieve high accuracy. For configurable software systems, there are various sampling strategies that are suitable for learning performance models [24], [33]: random sampling [5], [12], solver-based sampling [9], coverage-based sampling [34], [35], and distance-based sampling [36].

Courtois and Woodside [37] use regression splines to model the black-box performance behavior of a software system without taking configuration options into account. In the same vein, Israr et al. [38] and Mizan and Franks [39] obtain performance models at a coarser granularity using Layered Queuing Networks without considering variability, though. Instead, they produce models that provide event sequences for distributed systems. Westermann et al. [40] aim at finding optimal software configurations with performance modeling at the black-box level. However, they consider only little variability. Also, Krogmann et al. [41] build parameterized performance models at the component level. That is, they build black-box models for components and do not address fine-grained, possibly cross-cutting configuration options among several components. Ackermann et al. [42] propose an approach to automatically find a suitable machine-learning technique to learn black-box performance models using monitoring data. Grohmann et al. [43] use feature selection in the context of machine learning to obtain black-box performance dependencies. In contrast to these approaches, we use profiling information to build performance models at the method level, which is one of the main challenges and novelties of our approach.

Siegmund et al. [4], [44], [45] propose SPLConqueror, an approach to construct performance-influence models as linear functions over binary and numeric configuration options (or more complex combinations thereof). The key is to combine binary and numeric sampling and to symbolically learn the influence model in an iterative manner. Other approaches propose learning techniques based on classification and regression trees [5], [6] and spectral learning [11], or even learn when to stop the learning procedure [8]. Our approach takes advantage of these modeling capabilities to pinpoint performance properties, but at the method level.

Nair et al. [46] reduce the number of measurements by iteratively measuring and only adding configurations that improve accuracy the most. Another approach by Nair et al. [11] explores the configuration space by clustering, therefore, requiring measurements of only few representative configurations per cluster. This way, the sampling procedure can be directed to unveil performance influences and near-optimal configurations efficiently. However, all these approaches consider a software system as a black box, not allowing for pinpointing performance behavior and root causes in code.

C. Profiling

Profiling refers to the white-box analysis of the run-time behavior of a program execution with respect to memory consumption or execution time [47]. By contrast, measuring execution time of a system as a whole refers to black-box performance measurement.

Mytkowic et al. [48] analyzed the accuracy of Java profilers by comparing four commonly used profilers regarding their agreement on which methods are performance-critical. The profilers reported different sets of methods as hot-spots. Reasons for the disagreement are implementation details of the profilers (e.g., whether native methods are treated as part of the program) and the measurement overhead of the profiler (observer effect).

Some profiling approaches aim at automatically finding specific inefficient structures in the source code. Song and Lu [49] designed LDOCTOR, and Selakovic et al. [19] designed DECISIONPROF. Both tools search for redundant loops and optimization opportunities in the order of evaluating expressions. They focus only on specific code structures, but provide also suggestions for improvements.

There are many other approaches of how to profile different properties of software systems. However, these approaches do not consider configuration dependent performance variation. Still, profilers have been shown an important tool in industry to debug software systems with respect to resource usage. We do not want to replace but build on industry-strength profilers (i.e., JPROFILER) to reach our goal.

III. UNTANGLING PERFORMANCE VARIANCE

At the method level, learning performance models is a task that is highly sensitive to measurement bias. As execution times can be short, influences of concurrent processes can easily distort measurements. In the context of configurability and method context, many sources contribute to the overall variation in performance. To use machine learning effectively on measurement data, we need to quantify possible sources of performance variance and adjust our model accordingly. To this end, we conducted an analysis of possible causes of performance variance: measurement variance, configuration variance, and context variance. To devise an approach for learning method-level performance models, it is necessary to know how the variance in the execution time is composed and how to control the three contributing factors to pin down the influence of configuration options on performance.

For illustration, Figure 2 shows the execution time of a single method executed in three different configurations, repeated three times each. Each histogram shows the distribution of the execution time of all calls to that method. *Measurement variance* becomes visible when comparing rows: The performance distribution of a row's plots should not change since the measurement setup is the same. That is, any change here can only be caused by the measurement process (e.g., overhead) or measurement environment (e.g., context switches). *Context variance* is represented by the shape of the histogram. That is, for different contexts during a
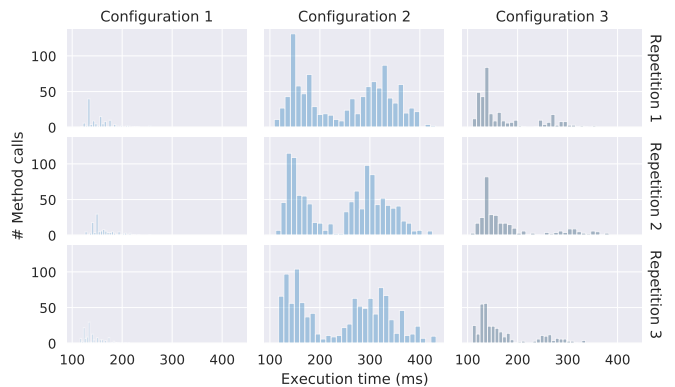

Fig. 2: Performance variances for different executions of method waitUntilSynced of PREVAYLER. Columns represent different configurations (configuration variance), rows represent different repetitions of a program execution (measurement variance), and each cell depicts the performance distribution of multiple executions of the same method in a single program run (context variance).

single program run, we might call the method with different parameters, different cache states, etc., leading to different execution times. So, a histogram shows the distribution of execution times for a single method in a single program run. Finally, *configuration variance* manifests as differences among plots of different columns. If the plots differ across the columns, the root cause of the performance changes are due to changes in the system's configuration. In what follows, we provide an in-depth analysis of three real-world subject systems (CATENA, H2, and PREVAYLER; see Section V-C for more details) regarding the three sources of variations. The goal is to obtain insights into which variance needs to be controlled when learning performance-influence models at method level. This helps us in devising sensible means for sampling, measuring, and learning instead of blindly applying an off-the-shelf machine-learning approach.

A. Measurement Variance

Measurement variance affects the accuracy with which we get stable results while repeating experiments. High measurement variance adversely affects the accuracy of performance-influence models. Therefore, it is crucial to estimate measurement variance with a sufficient number of experiment repetitions. The aim of our analysis is to determine the number of repetitions needed to trust the estimated measurement variance.

Our analysis setup is as follows: We profile a given configurable software system with 50 repetitions and report the coefficient of variation ($c_v = \frac{\sigma}{\mu}$; σ represents the standard deviation of all method executions; μ the mean of all method executions in a single run) as a standardized measure of dispersion of a probability distribution to quantify the stability of measurement results [50]. To check whether measurement variance is independent of configuration variance, we ran-
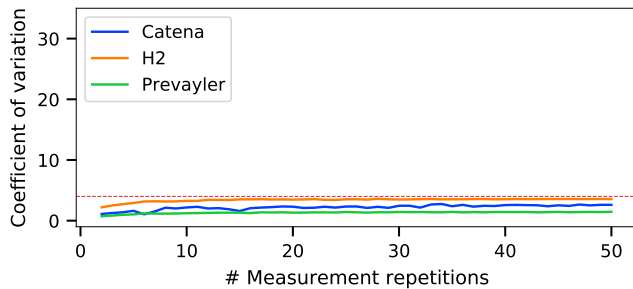
Fig. 3: Measurement variance with increasing number of repetitions. The dashed red line denotes the maximal measurement variance (4 %).
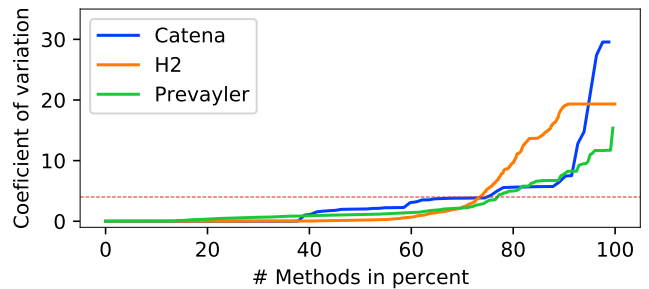

Fig. 4: Configuration variance of the methods per subject system sorted by their coefficient of variation. The dashed red line denotes the maximal measurement variance (4 %) such that all method execution times above the line change due to configuration decisions.

domly select 100 configurations for each of the three software systems.

To better visualize the effect of repeating experiments, we compute the coefficient of variation after each repetition and show it for our three configurable software systems in Figure 3. For all three systems, the variance is below 5 %. This indicates a reliable measurement setup and *the need for only a limited number of repetitions* (INSIGHT 1) as the coefficient remains stable already around three repetitions. Moreover, the coefficient is equal for all configurations, so configurations have no effect on measurement bias. Hence, we can neglect possible hidden dependencies here.

B. Configuration Variance

Configuration variance captures the variation in a method's execution time due to selecting different configuration options.

Our analysis setup is as follows: We profile a given configurable software system by measuring the execution time of each method. We aggregate the execution time per method and repeat this process five times to account for measurement bias. We repeat this process for different configurations. Again, we use the coefficient of variation per method as a measure to determine whether methods have a constant average execution time across different configurations. If the coefficient of variation is higher than the measurement variance (4 % for all of our subject systems; see Section III-A), the method's execution time is configuration-dependent.

Figure 4 depicts the coefficient of variation (y-axis) for each method (x-axis) ordered from low variation to high variation. We observe that *a large number of methods have only limited variance* (INSIGHT 2) and *only few methods exhibit high performance variance* (INSIGHT 3). Interestingly, the variance for these few methods is huge, and the percentage of methods affected by configuration decisions is also sensitive to the software system. From this analysis, we infer that method-level performance-influence models should concentrate on these highly varying methods; profiling all methods of the program would be wasteful. This is good news as profiling is usually expensive and affects measurement results. Hence, we conclude that an efficient and accurate learning approach would

first need to find the relevant methods and then concentrate learning these.

C. Context Variance

Context variance of a method's execution time originates from changes in the method's calling context (e.g., method parameters and cache state). In Figure 2, we visualize context variance as a histogram of performance values for each method execution in a single program run. We observe that the execution time of some methods remains constant during a program run, whereas other methods show high variance.

Overall, we observe *highly skewed execution times that heavily affect a method's average execution time* (INSIGHT 4). That is, we observe few but very large outliers, which are several orders of magnitude slower than about 99 % of the other method executions. This resembles a Cauchy distribution with no defined mean and standard deviation for these methods [51]. The problem for learning is that the Cauchy distribution is a well-known case where maximum-likelihood estimation fails and, subsequently, the likelihood principle in general [52], [53]. So, this can cause highly unreliable performance models.

Finally, we also see an interaction between context and configuration variance when analyzing Figure 2: *Not only the execution times vary, but also the number of method executions* (INSIGHT 5), so an algorithm that takes only the average execution time of a method in to account for learning the influence of options is doomed to fail. Instead, an accurate approach needs to account for the *number* of method calls in relation to their execution time.

D. Summary

From our analysis of variance, we can learn two important things: First, we see that the distribution of method executions changes for different configurations. That is, configuration influences a method's context causing a variation in the method's execution time. Second, based on our insight that some methods' performance values are Cauchy distributed, *we cannot resort to a sample-based profiling technique, but*
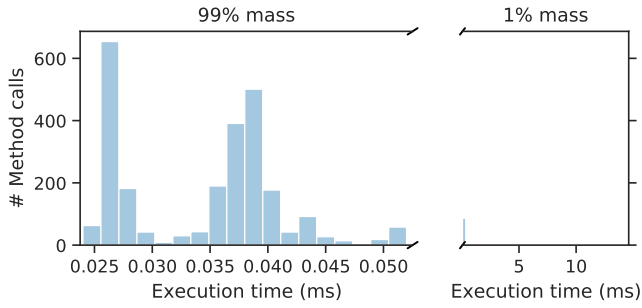
Fig. 5: Context variance of method rotr64; right: the 1 % longest running method executions; left: the remaining 99 % method executions.

rather need to tap the entire performance distribution using an instrumentation-based approach (INSIGHT 6). This is necessary as there might be large outliers that could skew the average method execution time substantially and would need to be filtered out. This is hardly possible with sample-based profiling.

IV. PERFORMANCE-INFLUENCE MODELING AT THE METHOD LEVEL

Our goal is to learn performance-influence models at the method level, so that we can pinpoint methods with high configuration-dependent performance variability and identify code regions that cause performance interactions. Our method-level performance-influence modeling approach builds on the insights that we gained from our variance analysis and is separated into two steps, as illustrated in Figure 6: coarse-grained analysis and fine-grained analysis.

We know that the execution time of only few methods varies for different configurations (INSIGHT 3), so we aim at identifying exactly these methods in the first step. For this purpose, we use a light-weight, coarse-grained profiler (JPROFILER) to obtain performance measures for all methods under different configurations using an established sampling approach. Then, we extract those methods that exhibit (i) a performance-relevant execution time (e.g., we filter out getter/setter methods) and (ii) a performance variation across different configurations. In the second step, we instrument the source code (INSIGHT 6) using the tool KIEKER to obtain an execution time distribution (an execution time for each method call). We filter from this distribution long running outliers (INSIGHT 4) and summarize the distribution as a histogram (INSIGHT 5). Finally, we learn one performance-influence model per method based on these fine-grained values.

A. Sampling Configurations

As a prerequisite to step 1, we sample a set $\hat{C}$ from the set C of all valid configurations. There is a substantial corpus of approaches that successfully applied different sampling strategies to obtain a representative set of configurations [8], [12], [24], [36]. Following previous work, we opt for feature-wise and pair-wise sampling for binary and Plackett-Burman

sampling for numeric options [4], but other sampling strategies might be appropriate, as well.

With feature-wise sampling, we obtain a set of configurations in which each option is enabled once. With pair-wise sampling, also called t-wise sampling with t = 2, this set is enriched by all pair-wise combinations of configuration options. We use the extended Plackett-Burman design, proposed by Wang and Wu [54], for sampling numeric configuration options. Compared to binary options, adding a numeric option with n different values increases the configuration space by factor n instead of factor 2. The Plackett-Burmann design selects a fixed set of configurations determined by a pre-chosen seed, which strongly reduces the effect of the combinatorial explosion.

B. Coarse-grained Profiling

Our approach automatically runs a given software system for each configuration of the sample set (denoted as *run*) with JPROFILER [2], a coarse-grained profiler that uses the JVMTI interface of the JVM. For each run, we obtain the absolute execution time and the number of calls for each method. We repeat each run five times and report the mean time to account for measurement bias (INSIGHT 1).

Next, we learn a performance-influence model per method $m \in M$ (where M is the set of all methods of a software system) from these measurements using classification and regression trees (CART) as the learning method [5]. In the case that all methods have been learned accurately, there is no need to continue with the second step. However, we have seen in our variance analysis that, typically, some methods are highly sensitive to context variance (INSIGHT 4 AND 5), which makes a second learning step necessary.

C. Filtering

To identify methods that are hard to learn and that contribute substantially to a system's performance (INSIGHT 5), we apply a filter to all methods M of a system obtaining a subset $M_{hard} \subseteq M$ for further measurement and learning (cf. Eq. 1). The filter relies on a predicate $\phi(m, \alpha, \beta, \gamma)$, with $\alpha, \beta, \gamma \in \mathbb{R}$ (cf. Eq. 2), which states whether a given method $m \in M$ belongs to the set of performance-relevant methods.

$$M_{hard} = \{ m \mid m \in M \land \phi(m, \alpha, \beta, \gamma) \} \qquad (1)$$

$$\frac{\Pi_{err}(m, \alpha) \qquad \Pi_{abs}(m, \beta) \lor \Pi_{rel}(m, \gamma)}{\phi(m, \alpha, \beta, \gamma)} \qquad (2)$$

$\Pi_{err}(m, \alpha)$ evaluates whether the error of the corresponding method's performance model exceeds the given threshold α (cf. Eq. 3). All methods that have been learned with a prediction error (mean absolute percentage error, MAPE, Eq. 4) of α or worse get selected. For this purpose, we compare the measured performance for method m of configuration c denoted with $\pi_c(m)$ with the performance $\pi'_c(m)$ predicted with the model of step 1. Following Siegmund et al. [4], we
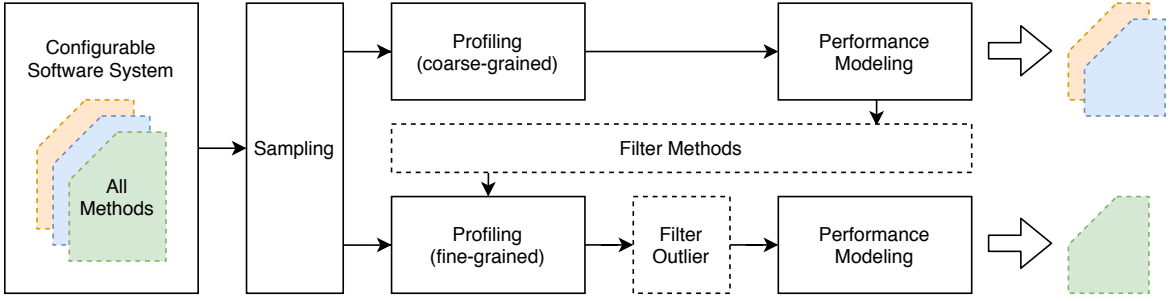
Fig. 6: Method-level white-box-modeling pipeline for configurable software systems

fix α to 5 % in our experiments, which represents an already strict filter criterion just above the measurement bias (cf. Section III-A). Increasing α decreases the number of methods that have to be analyzed further, but this way more inaccurate performance models are accepted.

$$\Pi_{err}(m, \alpha) = \text{MAPE}(m) \geq \alpha \qquad (3)$$

$$\text{MAPE}(m) = \frac{100}{|\hat{C}|} \cdot \sum_{c \in \hat{C}} \left| \frac{\pi_c(m) - \pi_c'(m)}{\pi_c(m)} \right| \qquad (4)$$

$\Pi_{abs}(m, \beta)$ (Eq. 5) evaluates whether a method's execution time is longer than β. A method's execution time here is defined as the accumulated execution times over a run (Eq. 6). By setting β, we control to which extent we want to invest measurement effort for short-running methods.

$$\Pi_{abs}(m, \beta) = \text{absPerf}(m) \geq \beta \qquad (5)$$

$$\text{absPerf}(m) = \frac{1}{|\hat{C}|} \cdot \sum_{c \in \hat{C}} \pi_c(m) \qquad (6)$$

$\Pi_{rel}(m, \gamma)$ (Eq. 7) evaluates whether a method has a relative run-time (Eq. 8) of more than γ in relation to the accumulated black-box time of the overall software system (sum of performance values of all methods, Eq. 9). Adjusting γ enables us to focus on methods that contribute the most to the overall performance of the system.

$$\Pi_{rel}(m, \gamma) = \text{relPerf}(m) \geq \gamma \qquad (7)$$

$$\text{relPerf}(m) = \frac{1}{|\hat{C}|} \cdot \sum_{c \in \hat{C}} \frac{\pi_c(m)}{\text{blackBoxPerf}(c)} \qquad (8)$$

$$\text{blackBoxPerf}(c) = \sum_{m \in M} \pi_c(m) \qquad (9)$$

To sum up, predicate ϕ selects methods that have been inaccurately learned (α), have a total run time of, at least, β, and contribute to the overall software's performance by, at least, γ percent.

D. Fine-Grained Profiling and Learning

We use KIEKER [18], an aspect-oriented JAVA performance profiling tool, to measure the methods M_{hard} obtained from

the filtering step. To analyze the variation across all calls of a method, we extended KIEKER by logging the assignment of values to method arguments of each method call together with the measured execution time.

Profiling with KIEKER involves three steps: First, we include an annotation (pointcut) into the source code of the subject system at the beginning of each selected method. Second, we compile the software system into an executable. Third, we execute the software with KIEKER as JVM argument (JAVA agent) to weave the monitoring code (advice) around method executions. We run our experiments with the same set of configurations and workloads as used in the coarse-grained profiling phase, obtaining performance data per method execution of the relevant methods. Based on our variance analysis, we filter outliers that make up 1 % of the longest execution times and we learn new models with the remaining aggregated data using CART.

V. EXPERIMENT SETUP

In this section, we present the measurement setup that we use for profiling as well as the software systems that we selected for evaluation.

A. Measurement Setup

All measurements ran on a cluster of 27 computers, each of which has an Intel Quad-Core processor, an SSD running a headless operating system (Ubuntu 18.04.3 LTS), an HDD to store experiment data, and 8 or 16GB of RAM.[3]

B. Measurement Procedure

For each subject system, we generate two sample sets for learning (according to the two sampling strategies of Section IV-A), as shown in Table I. For each learning set, we measure the runtime with our two-step approach, learning a model per method. We sample an additional test set of 100 fresh configurations at random. We use the test set to evaluate the prediction error of the models learned based on the learning sets. Furthermore, we repeat all measurements five times, which results in 94,000 measurement runs (RQ$_1$ and RQ$_2$). Additionally, we measure the black-box execution time of all configurations to determine the execution time without profiling (RQ$_3$).

---

[3]For a single software system, we conducted all measurements either on the systems with 8 or 16GB memory.

TABLE I: Overview of subject systems. $|F|$ denotes the number of configuration options (binary and numeric); $|C|$ denotes the number of the valid configurations C; $|\hat{C}_{FW}|$ denotes the number of configurations sampled feature-wise; $|\hat{C}_{PW}|$ denotes the number of configurations sampled in a pair-wise manner.

System	Domain	$\|F\|$	$\|C\|$	$\|\hat{C}_{FW}\|$	$\|\hat{C}_{PW}\|$
BATIK	SVG rasterizer	31	$9.6 \cdot 10^4$	28	337
CATENA	Password hashing	12	$1.0 \cdot 10^9$	875	2625
CPD	Copy-paste detector	7	$1.1 \cdot 10^4$	40	115
DC	Image density converter	24	$3.4 \cdot 10^6$	1600	9700
H2	Database	16	$6.5 \cdot 10^{11}$	375	2275
KANZI	Data compression	40	$4.3 \cdot 10^3$	34	458
PMD	Source-code analyzer	11	$5.1 \cdot 10^2$	36	104
PREVAYLER	Database	12	$1.3 \cdot 10^5$	250	400
SUNFLOW	Rendering engine	6	$5.4 \cdot 10^6$	125	n/a

C. Subject Systems

We evaluate our approach with 9 real-world software systems. Our selection includes configurable JAVA applications, covering different domains, including databases, rendering engines, and static code analyzers. Our selection was driven by covering a diverse set of domains, having memory and CPU-intensive tasks, and providing configuration options that affect performance of the system. We provide an overview of the software systems in Table I. Next, we present the systems and benchmark workloads. When possible, we reused existing workloads provided by the respective software systems. [4]

The BATIK rasterizer converts SVG files to a raster format. As workload, we used the DACAPO benchmark suite [55], which contains a set of SVG images of different sizes that can be used for performance tests.

CATENA is a secure password scrambling framework that implements a corresponding hashing function. As workload, we used its password hashing benchmark. With the provided configuration options, it is possible to select one out of four graphs as well as different seeds and security values that influence how much main memory has to be used to encrypt or decrypt a password and how long this process takes.

CPD is a code duplication detector. It detects duplicate source code sections to support developers with code refactoring. As a benchmark workload, we detect code duplicates in CATENA's code base.

DENSITY-CONVERTER (DC) is an image density converter that, given an image or folder, converts these inputs into image formats with different resolutions. As workload, we used a set of high resolution images provided by the developers.

H2 is an open-source relational database system that can operate both in an embedded and a client-server setting. As workload, we use the subset of tests of the POLEPOSITION benchmark, with which developers compare H2 to other database applications.

KANZI is a lossless data compressor. It provides various configuration options for composing and tuning the compression process. As workload, we used the SILESIA corpus benchmark [5].

PMD is an extensible cross-language static code analyzer that checks source code against a set of rules. As a workload, we selected all rules that try to identify performance violations in the system to analyze. The system that we analyzed is PREVAYLER.

PREVAYLER is an open-source object persistence library for JAVA supporting in-memory storage. It provides a scalability and performance benchmark consisting of transaction-processing and query scalability tests that are applied to a JDBC-compatible database.

SUNFLOW is an open-source global illumination rendering system. It provides a selection of example scenes (objects to illuminate and render), of which we selected the *golden* scene (a teapot in a colored room with one light source and 128×128 pixels).

VI. EVALUATION

The goal of our approach is to pin down the influence of configuration options on individual methods. In our evaluation, we address three research questions:

RQ1: Can we learn accurate performance-influence models at the method level?

Previous work has shown that performance can be accurately modeled for a system as a whole. However, it is unclear whether this level of accuracy can be achieved when modeling performance at the method level, due to measurement overhead and the various sources of variance that we have described in Section III.

RQ2: How do system-level and method-level models compare in terms of information they provide?

Knowing the influence of a configuration option at the system level is helpful for tuning a system's performance. However, identifying the root cause of the influence of features helps developers, for example, to spot performance bugs and to focus on specific performance tests in a CI pipeline.

RQ3: What is the relation of the runtimes of profiled and unprofiled methods?

An important measure for validity is whether the actual (unprofiled) method execution time relates to the measured (profiled) method execution time. Profiling introduces overhead, and therefore the models we learn may be biased. With this question, we aim at quantifying the extent of the profiling overhead.

As ConfigCrusher is closest to our approach (cf. Section II), it would be a natural candidate for comparison. In Section VI-D, we report on why a comparison is not feasible, though.

---

[4]We provide all benchmarks on the supplementary Website: https://git.io/JtnTa or an archived version at https://archive.softwareheritage.org/browse/revision/2e61f8ce57498194c2af0cd76e87498a174f07fa/

[5]Silesia Corpus: http://sun.aei.polsl.pl/~sdeor/index.php?page=silesia
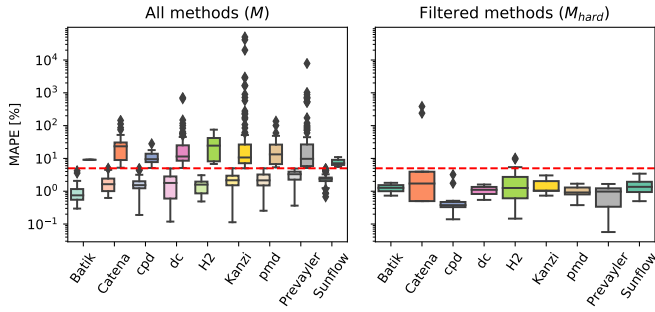
Fig. 7: Error (MAPE) of method-level performance-influence models of all methods (M) and of the filtered methods (M_{hard}). Dashed red line denotes 5 % MAPE.

TABLE II: Method and correlation analysis. $|M|$ denotes the total number of methods and $|M_{hard}|$ the number of filtered methods. Configuration-wise linear correlation (LC) and rank correlation (RC) of black-box measurements (BB) using a coarse-grained profiler (CG) and a fine-grained profiler (FG).

System	$|M|$	$|M_{hard}|$	BB vs. CG		BB vs. FG	
			LC	RC	LC	RC
BATIK	122	9	0.82	0.89	0.76	0.87
CATENA	128	7	0.94	0.98	0.81	0.95
CPD	125	11	0.25	0.55	0.57	0.76
DC	435	3	0.51	0.91	0.53	0.96
H2	238	43	0.32	0.42	0.12	0.34
KANZI	386	27	0.65	0.87	0.38	0.77
PMD	218	5	0.32	0.72	0.18	0.38
PREVAYLER	166	7	0.94	0.97	0.84	0.88
SUNFLOW	128	6	0.62	0.62	0.26	0.38

A. Method-Level Performance Models (RQ_1)

a) Operationalization: To answer RQ_1, we calculate the MAPE of the performance-influence models of each method separately. We follow the measurement procedure described in Section V-B. We assume an influence model to be sufficiently accurate if it predicts the execution time of the method in the test set with an average error below 5 %, which is stricter than 10 % as used in previous work [5].

b) Results: Figure 7 summarizes the results regarding RQ_1. The plot on the left shows the prediction-error distribution of all method-level models based on data measured with the coarse-grained profiler. All methods that are below the dashed red marker (5 % prediction error) can be accurately learned based on a single profiling run per configuration, which makes up 84.8 % of all methods. Hence, the Π_{err} condition of our filter selects about 15 % of the methods for step 2. When using the additional conditions with respect to performance relevance (Π_{abs} and Π_{rel}), we obtain a set comprising of about 6 % of all methods. Table II depicts for each system the set of filtered methods, denoted as M_{hard}.

Applying the second step, we are able to model nearly all of the remaining methods accurately as shown in Figure 7 on the right-hand side. From a total number of $|M_{hard}| = 118$ methods across all systems, only 5 methods cannot be learned

accurately in step 2. A closer manual analysis of these methods revealed that they depend either heavily on thread-dependent file IO operations in the case of H2 or nested loops (number of loops depends exponentially on an option) that copy an internal state array (width of array depends exponentially on a configuration option) in the case of CATENA.

> Answering RQ_1, the coarse-grained profiling step is able to learn models with a MAPE below 5 % for 84.8 % of all methods. Applying the fine-grained profiler in a second step, the MAPE is below 5 % for 95.8 % of the performance-relevant methods.

B. Tracing Option Influences (RQ_2)

a) Operationalization: To answer RQ_2, we focus on the importance of configuration options and interactions in white-box models. Specifically, we learn one random forest [56], consisting of 100 classification and regression trees, at system-wide level as well as one random forest for each method and extract all options and interactions. For this analysis and without loss of generality, we concentrate on the two options (or interactions) per system that have the largest performance influence on the system determined by the black-box model (similar to the scenario sketched in Figure 1). We aim at identifying the root cause for high influences by analyzing all performance-influence models at method level. Since there might be hundreds of methods per system, we analyze only performance-relevant methods, whose total sum can explain 80 % of the system performance. Having determined influential options and methods, we count those methods as root cause for which the configuration options (or interactions) we are interested in, have an influence that exceeds the measurement error. Furthermore, we sort the methods by the options' influences revealing, which method contributes most to an option's influence.

b) Results: Tracing the influence of an option from the system level to the method level uncovers the cause of its influence. We present the results for CATENA in Figure 8. There are some configuration options and interactions that have an high influence on performance compared to the others; 12 options have no relevant influence. Focusing on an option of interest—the most important option in our example—reveals that only a small portion (12 out of 128) of the methods contribute to the option's system-wide performance influence. These 12 methods are responsible for more than 2/3 of the system's performance. This kind of information is not available in black-box models. It does not only help selecting important configuration options for guiding sampling and performance tuning, but also identifying the small set of methods that causes possible performance bugs, this way, facilitating performance bug detection of configurable systems.

> Answering RQ_2, white-box performance-influence models can successfully guide us to performance-relevant methods that are dependent on influential configuration options and interactions.
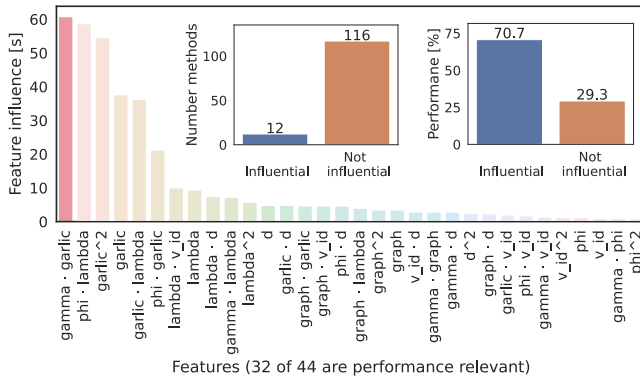
Fig. 8: Overview of the influence of options and interactions on CATENA's performance. The background plot shows the distribution of all influential options and interactions (influence greater then the measurement error). The small inner plots focus on the interaction *gamma · garlic*, which is the most influential option/interaction of the model. The left plot shows the number of methods that contribute and do not contribute to the interaction's performance. The right plot shows the performance portion of these methods.

C. Profiled vs. Unprofiled Methods (RQ₃)

a) Operationalization: To answer RQ₃, ideally we would need to compare the execution time between a profiled method and an unprofiled method. Since we cannot know an unprofiled method's execution time, we use a proxy to infer the actual unprofiled method execution times. For this purpose, we consider the black-box execution time of a system as the aggregation of all true method's execution times. We compare this time against the aggregated (i.e., system-wide) predictions of white-box performance models for all methods. By repeating this process for all measured configurations, we approximate the relation of our estimates to the actual method execution times.

We use two different indicators to quantify the relation: Pearson's correlation coefficient and Spearman's rank correlation coefficient. The former tests whether there is a linear dependency between the aggregated, system-wide execution time with and without profiling. A high linear correlation would indicate that it is possible to infer the unprofiled execution time from the profiled execution time with a constant factor. This way, white-box performance influence models could even provide a precise prediction of method performance running in operation (without profiler). A high linear correlation is unlikely, though, because the overhead during profiling increases while the number of profiled methods grows. Spearman's rank correlation coefficient tests whether the order between profiled and unprofiled execution times is preserved. A high rank correlation indicates that fast configurations measured without profiling stay fast, even if profiling is enabled. This would mean that our approach has accurately learned the relative influences of configuration options per method.

b) Results: In Table II, we show the correlation between system-wide unprofiled execution time compared to using the
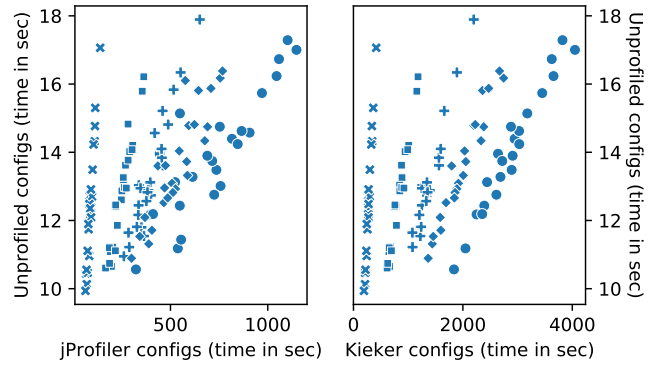

Fig. 9: Configuration-wise execution time of SUNFLOW: black-box measurements vs. profiling. Left: using JPROFILER, right: using KIEKER. Different symbols visualized different numeric values of configuration option SAMPLES.

coarse-grained profiler (BB vs. CG) and unprofiled execution time compared to using the fine-grained profiler (BB vs. FG). As expected, rank correlation is higher than linear correlation across all subject systems and for both profilers. That is, the fastest configurations remain the fastest independently of whether we use our learned models for predicting execution times or measuring execution time. This is good news as this property has been shown the main tuning objective for configurable systems [57]. Furthermore, we can see that different subject systems exhibit different correlations. Some systems, such as BATIK, PREVAYLER, and CATENA, exhibit even nearly perfect linear correlation. For them, the execution time depends strongly on the configuration for both types of experiments: measuring with a profiler and measuring the overall execution time of a program. There are also subject systems for which rank correlation is much higher than linear correlation: DENSITY CONVERTER, CPD, and PMD. There are two systems stand out with a generally low correlation: SUNFLOW and H2, which we analyze next.

Figure 9 shows the dependency between unprofiled execution time and aggregated execution time measured with a profiler per configuration for SUNFLOW. We concentrate on the measurement overhead as the main cause for a low correlation. Interestingly, when highlighting configuration options, we observe a strong pattern for the overhead. There are multiple groups that follow a linear trend, but with different slopes. The determining factor for this slope is the configured value of the numeric configuration option SAMPLES. This option is used as a seed for method calculatePhotons that compute the global illumination of the scene as part of the rendering process. By increasing the numeric value for this option, the overhead increases disproportionally. For H2, there is also a strong pattern, again, caused by a numeric configuration option (ANALYZEAUTO) [6]. This is important for other studies in this area: Profiling overhead of configurable system depends on configuration options.

---

[6]More details on the supplementary Website.

Answering RQ_3, we observed a generally high rank correlation between system-wide profiled and unprofiled performance, demonstrating that the use of a profiler does not change the relative importance of configuration options and that white-box models are able to reveal performance-relevant methods. Some show even a linear correlation. A notable exception to this rule is a numeric option in SUNFLOW, which directly impacts the overhead introduced by the profiler.

D. Comparison to CONFIGCRUSHER

Closest to our approach is CONFIGCRUSHER [22]. The main difference to our approach is that CONFIGCRUSHER relies on static taint analysis with the goal of determining which code regions are affected by which configuration options to weave measurement code at according statements. Due to this conceptual difference, we face both qualitative as well as technical challenges that render a comparison infeasible. As explained in Section II-A, there are three ways of propagating taints through a program: (A) taint every access to the data structure(s) that hold(s) the configuration options; (B) stop tainting at this point; and (C) rewrite the program, such that all options are stored in individual variables and are accordingly accessed across the code-base.

Variant C is infeasible in practice and also not in our case, as a substantial rewrite is not only impractical for larger systems such as H2, but this would also change the program structure such that a performance comparison has no longer a common ground. For the purpose of comparison, we tried variant A first, but quickly run into timeouts and memory limitations, due to the inherent limitations of static code analysis. We communicated with CONFIGCRUSHER's main author, who confirmed our findings.

In a second attempt, we followed variant B. We again consulted the main author of CONFIGCRUSHER for guidance to avoid introducing bias and setting up the subject system consistently with the original approach. As a result, we obtained reasonable taints for DENSITY-CONVERTER. For H2, the largest of our subject systems, we ran into memory overflows for all analyzed configuration options. The analysis of SUNFLOW, BATIK, and PREVAYLER produced tainted code regions of size 1, which are basically useless. The reason was that the configuration options are immediately stored in a data structure, leading to a termination of the taint analysis. We provide all analysis log files at our supplementary Website.

In addition to the limitations of the taint analysis, CONFIG-CRUSHER can handle only binary configuration options, whereas our approach can handle numeric options (as we do in our evaluation). Furthermore, CONFIGCRUSHER is capable of tainting only single-threaded applications due to the underlying taint analysis. Our approach can produce performance models also for multi-threaded applications, such as H2.

E. Threats to Validity

The selection of the profiler represents a threat to construct validity. We mitigated this threat by a pre-study (not shown in the paper) where we evaluated several profilers. JPROFILER is an industrial-strength profiler with low overhead, which turned out to be the best choice for the first phase. However, to obtain fine-grained performance data, we required more flexibility and opted for KIEKER. A threat to internal validity arises from the measurement overhead introduced by the profiler. We reduce this threat by selecting a low-overhead profiler for measurement in the coarse-grained step and devoted a whole research question to analyze its influence.

The selection of subject systems threatens external validity. Although we cannot claim that we can learn white-box models accurately for all JAVA systems with proper profiling capabilities. Our results show that this is in principle possible for a large, industry-relevant branch of configurable software systems.

VII. CONCLUSION

We have proposed an approach to learn white-box performance-influence models at the method level, enabling tracing configuration effects from the system level to individual methods. Based on a pre-study on 3 software systems, we analyzed possible causes of performance variance of method execution times to design an integrated profiling and learning approach. We found that the majority of methods can be easily learned, as they either do not contribute much to the system's overall performance or do not contribute to configuration variance. Based on these insights, we have devised a two step approach in which we learn performance-influence models for all methods of a software system using a cheap, coarse-grained profiler in a first step, and filter inaccurate and relevant methods to be measured and learned again with an expensive, fine-grained profiler in a second step. We found that, despite the overhead introduced by profiling, the correlation between profiled and unprofiled method execution time is high and that white-box models can accurately predict the profiled execution time. More importantly, we were able to show that performance models at the method level can be used to pinpoint the contribution of individual configuration options and interactions to individual methods, helping developers to chase configuration-related performance bugs or to focus performance testing on specific configurations.

REFERENCES

[1] C. U. Smith, "Software performance engineering," in *Performance Evaluation of Computer and Communication Systems*. Springer, 1993, pp. 509–536.

[2] M. Woodside, G. Franks, and D. C. Petriu, "The future of software performance engineering," in *Future of Software Engineering (FOSE)*. IEEE, 2007, pp. 171–187.

[3] N. Siegmund, A. von Rhein, and S. Apel, "Family-based performance measurement," in *Proc. Int. Conf. Generative Programming and Component Engineering (GPCE)*. ACM, 2013, pp. 95–104.

[4] N. Siegmund, A. Grebhahn, S. Apel, and C. Kästner, "Performance-influence models for highly configurable systems," in *Proc. Europ. Software Engineering Conference and ACM SIGSOFT Symp. Foundations of Software Engineering (ESEC/FSE)*. ACM, 2015, pp. 284–294.

[5] J. Guo, K. Czarnecki, S. Apel, N. Siegmund, and A. Wąsowski, "Variability-aware performance prediction: A statistical learning approach," in *Proc. Int. Conf. Automated Software Engineering (ASE)*. IEEE, 2013, pp. 301–311.

[6] J. Guo, D. Yang, N. Siegmund, S. Apel, A. Sarkar, P. Valov, K. Czarnecki, A. Wasowski, and H. Yu, "Data-efficient performance learning for configurable systems," *Empirical Software Engineering (EMSE)*, vol. 23, no. 3, pp. 1826–1867, 2018.

[7] N. Siegmund, M. Rosenmüller, C. Kästner, P. Giarrusso, S. Apel, and S. Kolesnikov, "Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption," *Information and Software Technology (IST)*, vol. 55, no. 3, pp. 491–507, 2013.

[8] A. Sarkar, J. Guo, N. Siegmund, S. Apel, and K. Czarnecki, "Cost-efficient sampling for performance prediction of configurable systems (t)," in *Proc. Int. Conf. Automated Software Engineering (ASE)*. IEEE, 2015, pp. 342–352.

[9] C. Henard, M. Papadakis, M. Harman, and Y. Le Traon, "Combining multi-objective search and constraint solving for configuring large software product lines," in *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE, 2015, pp. 517–528.

[10] A. S. Sayyad, J. Ingram, T. Menzies, and H. Ammar, "Scalable product line configuration: A straw to break the camel's back," in *Proc. Int. Conf. Automated Software Engineering (ASE)*. IEEE, 2013, pp. 465–474.

[11] V. Nair, T. Menzies, N. Siegmund, and S. Apel, "Faster discovery of faster system configurations with spectral learning," *Automated Software Engineering*, vol. 25, no. 2, pp. 247–277, 2018.

[12] J. Oh, D. Batory, M. Myers, and N. Siegmund, "Finding near-optimal configurations in product lines by random sampling," in *Proc. Europ. Software Engineering Conference and ACM SIGSOFT Symp. Foundations of Software Engineering (ESEC/FSE)*. ACM, 2017, pp. 61–71.

[13] S. Wang, C. Li, H. Hoffmann, S. Lu, W. Sentosa, and A. I. Kistijantoro, "Understanding and auto-adjusting performance-sensitive configurations," in *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2018, pp. 154–168.

[14] D. Shen, Q. Luo, D. Poshyvanyk, and M. Grechanik, "Automating performance bottleneck detection using search-based application profiling," in *Proc. Int. Symp. Software Testing and Analysis (ISSTA)*. ACM, 2015, pp. 270–281.

[15] O. Ibidunmoye, F. Hernández-Rodriguez, and E. Elmroth, "Performance anomaly detection and bottleneck identification," *ACM Computing Surveys*, pp. 1–35, 2015.

[16] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," *Proc. Int. Conf. Programming Language Design and Implementation (PLDI)*, vol. 47, pp. 77–88, 2012.

[17] X. Han and T. Yu, "An empirical study on performance bugs for highly configurable software systems," in *Proc. Int. Symp. Empirical Software Engineering and Measurement (ESEM)*. ACM, 2016, pp. 1–10.

[18] A. Van Hoorn, J. Waller, and W. Hasselbring, "Kieker: A framework for application performance monitoring and dynamic software analysis," in *Proc. Int. Conf. Performance Engineering (ICPE)*. ACM, 2012, pp. 247–248.

[19] M. Selakovic, T. Glaser, and M. Pradel, "An actionable performance profiler for optimizing the order of evaluations," in *Proc. Int. Symp. Software Testing and Analysis (ISSTA)*. ACM, 2017, pp. 170–180.

[20] N. Snellman, A. Ashraf, and I. Porres, "Towards automatic performance and scalability testing of rich Internet applications in the cloud," in *Proc. Europ. Conf. Software Engineering and Advanced Applications (SEAA)*. IEEE, 2011, pp. 161–169.

[21] M. B. Chhetri, S. Chichin, Q. B. Vo, and R. Kowalczyk, "Smart cloudbench–automated performance benchmarking of the cloud," in *Proc. Int. Conf. Cloud Computing (CLOUD)*. IEEE, 2013, pp. 414–421.

[22] M. Velez, P. Jamshidi, F. Sattler, N. Siegmund, S. Apel, and C. Kästner, "ConfigCrusher: towards white-box performance analysis for configurable systems," *Automated Software Engineering*, vol. 27, no. 3, pp. 265–300, 2020.

[23] M. Velez, P. Jamshidi, N. Siegmund, S. Apel, and C. Kästner, "White-box analysis over machine learning: Modeling performance of configurable systems," in *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE, 2021.

[24] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel, "A comparison of 10 sampling algorithms for configurable systems," in *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE, 2016, pp. 643–654.

[25] E. Reisner, C. Song, K.-K. Ma, J. S. Foster, and A. Porter, "Using symbolic evaluation to understand behavior in configurable software systems," in *Proc. Int. Conf. Software Engineering (ICSE)*. ACM, 2010, pp. 445–454.

[26] J. Meinicke, C.-P. Wong, C. Kästner, T. Thüm, and G. Saake, "On essential configuration complexity: Measuring interactions in highly-configurable systems," in *Proc. Int. Conf. Automated Software Engineering (ASE)*. ACM, 2016, pp. 483–494.

[27] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard, "Dynamic knobs for responsive power-aware computing," in *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2011, pp. 199–212.

[28] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake, "A classification and survey of analysis strategies for software product lines," *ACM Computing Surveys*, vol. 47, no. 1, pp. 1–45, 2014.

[29] A. von Rhein, T. Thüm, I. Schaefer, J. Liebig, and S. Apel, "Variability encoding: From compile-time to load-time variability," *Journal of Logical and Algebraic Methods in Programming*, vol. 85, no. 1, pp. 125–145, 2016.

[30] A. von Rhein, J. Liebig, A. Janker, C. Kästner, and S. Apel, "Variability-aware static analysis at scale: An empirical study," *ACM Trans. Software Engineering and Methodology (TOSEM)*, vol. 27, no. 4, pp. 1–33, 2018.

[31] M. Lillack, C. Kästner, and E. Bodden, "Tracking load-time configuration options," *IEEE Trans. Software Engineering (TSE)*, vol. 44, no. 12, pp. 1269–1291, 2018.

[32] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden, "Mining apps for abnormal usage of sensitive data," in *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE, 2015, pp. 426–436.

[33] C. Kaltenecker, A. Grebhahn, N. Siegmund, and S. Apel, "The interplay of sampling and machine learning for software performance prediction," *IEEE Software*, vol. 37, no. 4, pp. 58–66, 2020.

[34] M. F. Johansen, Ø. Haugen, and F. Fleurey, "An algorithm for generating t-wise covering arrays from large feature models," in *Proc. Int. Software Product Line Conference (SPLC)*. ACM, 2012, pp. 46–55.

[35] D. Marijan, A. Gotlieb, S. Sen, and A. Hervieu, "Practical pairwise testing for software product lines," in *Proc. Int. Software Product Line Conference (SPLC)*. ACM, 2013, pp. 227–235.

[36] C. Kaltenecker, A. Grebhahn, N. Siegmund, J. Guo, and S. Apel, "Distance-based sampling of software configuration spaces," in *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE, 2019, pp. 1084–1094.

[37] M. Courtois and M. Woodside, "Using regression splines for software performance analysis," in *Proc. Int. Workshop Software and Performance (WOSP)*. ACM, 2000, pp. 105–114.

[38] T. A. Israr, D. H. Lau, G. Franks, and M. Woodside, "Automatic generation of layered queuing software performance models from commonly available traces," in *Proc. Int. Workshop Software and Performance (WOSP)*. ACM, 2005, pp. 147–158.

[39] A. Mizan and G. Franks, "An automatic trace based performance evaluation model building for parallel distributed systems," in *Proc. Int. Conf. Performance Engineering (ICPE)*. ACM, 2011, pp. 61–72.

[40] D. Westermann, J. Happe, R. Krebs, and R. Farahbod, "Automated inference of goal-oriented performance prediction functions," in *Proc. Int. Conf. Automated Software Engineering (ASE)*. ACM, 2012, pp. 190–199.

[41] K. Krogmann, M. Kuperberg, and R. Reussner, "Using genetic search for reverse engineering of parametric behavior models for performance prediction," *IEEE Transactions on Software Engineering*, vol. 36, no. 6, pp. 865–877, 2010.

[42] V. Ackermann, J. Grohmann, S. Eismann, and S. Kounev, "Blackbox learning of parametric dependencies for performance models," in *MODELS Workshops*, 2018.

[43] J. Grohmann, S. Eismann, S. Elflein, J. V. Kistowski, S. Kounev, and M. Mazkatli, "Detecting parametric dependencies for performance models using feature selection techniques," in *Proc. Int. Symp. Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2019, pp. 309–322.

[44] N. Siegmund, M. Rosenmuller, C. Kastner, P. G. Giarrusso, S. Apel, and S. S. Kolesnikov, "Scalable prediction of non-functional properties in software product lines," in *Proc. Int. Software Product Line Conference (SPLC)*. IEEE, 2011, pp. 160–169.

[45] N. Siegmund, S. S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, and G. Saake, "Predicting performance via automated feature-interaction detection," in *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE, 2012, pp. 167–177.

[46] V. Nair, Z. Yu, T. Menzies, N. Siegmund, and S. Apel, "Finding faster configurations using FLASH," *IEEE Trans. Software Engineering (TSE)*, vol. 46, no. 7, pp. 794–811, 2018.

[47] J. Du, N. Sehrawat, and W. Zwaenepoel, "Performance profiling of virtual machines," in *Proc. Int. Conf. Virtual Execution Environments (VEE)*. ACM, 2011, pp. 3–14.

[48] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Evaluating the accuracy of Java profilers," in *Proc. Int. Conf. Programming Language Design and Implementation (PLDI)*. ACM, 2010, pp. 187–197.

[49] L. Song and S. Lu, "Performance diagnosis for inefficient loops," in *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE, 2017, pp. 370–380.

[50] B. Everitt and A. Skrondal, *The Cambridge Dictionary of Statistics*. Cambridge University Press, 2002, vol. 106.

[51] N. S. Pillai and X.-L. Meng, "An unexpected encounter with Cauchy and Lévy," *The Annals of Statistics*, pp. 2089–2097, 2016.

[52] T. S. Ferguson, "Maximum likelihood estimates of the parameters of the Cauchy distribution for samples of size 3 and 4," *Journal of the American Statistical Association*, vol. 73, no. 361, pp. 211–213, 1978.

[53] J. A. Reeds, "Asymptotic number of roots of Cauchy location likelihood equations," *The Annals of Statistics*, pp. 775–784, 1985.

[54] J. Wang and C. J. Wu, "A hidden projection property of Plackett-Burman and related designs," *Statistica Sinica*, pp. 235–250, 1995.

[55] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. L. Hosking, M. Jump, H. B. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," in *Proc. Int. Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 2006, pp. 169–190.

[56] P. Geurts, D. Ernst, and L. Wehenkel, "Extremely randomized trees," *Machine learning*, vol. 63, no. 1, pp. 3–42, 2006.

[57] V. Nair, T. Menzies, N. Siegmund, and S. Apel, "Using bad learners to find good configurations," in *Proc. Europ. Software Engineering Conference and ACM SIGSOFT Symp. Foundations of Software Engineering (ESEC/FSE)*. ACM, 2017, pp. 257–267.