Adressing the Impact of Workload Variation on the Performance of Configurable Software Systems

s ***Abstract*—The performance characteristics of a software system depends to a significant extent on its configuration and workload. State-of-the-art performance modeling approaches either address configuration-dependent or workload-dependent performance behavior. The interaction of both factors and how they influence performance have not been systematically studied so far. Understanding to what extent configuration and workload—individually and combined—cause a software system's performance to vary is key to understand whether performance models are generalizable, across different configurations and workloads. Assessing the impact and driving factors of such input sensitivity is key to develop strategies that obtain representative performance prediction models.**

To shed light on this issue, we have conducted a *systematic* empirical study, analyzing a multitude of configurations and workloads across ten software systems. We have obtained a substantial number of black-box performance measurements and enriched them with coverage data to assess whether and how configuration choices and workloads interact and shape software performance. We find that code coverage (i.e., *what* code is executed) and code utilization (i.e., *how* covered code is executed) are driving factors for workload-specific performance differences. Beyond code coverage testing, our findings motivate the use of dynamic code analyses to identify whether and in which way configuration options are sensitive to varying the workloads.

I. Introduction

Most modern software systems can be customized via configuration options to meet environmental demands. Configuration options can enable desired functionality or tweak non-functional aspects of a software system, such as improving performance or energy consumption. The relationship of configuration choices and their influence on performance has been extensively studied in the literature [1]–[9]. The backbone of performance estimation are prediction models that map a given configuration to the estimated performance value. Learning performance models relies on a training set of configuration-specific performance measurements. In state-of-the-art approaches observations usually employ only a single workload which aims at emulating a specific real-world application scenario.

The choice of the workload (i.e., the input fed to the software system) is known to influence the performance of configurable software systems in different ways as has been shown for the domains of SAT solvers [10], [11], compilation [12], [13], video transcoding [14], [15], data compression [16], and code verification [17]. Besides apparent interactions, such as performance scaling with the size of a workload, qualitative aspects can result in more complex and non-trivial performance interactions.

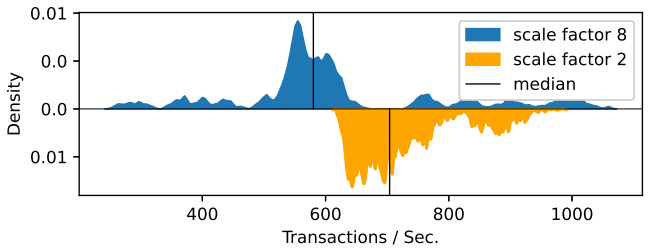

Figure 1: Performance distributions of the database system H2 run the TPC-C benchmark at different scale factors.

Take as an example the distributions of configuration-specific throughput of the database H2 in Figure 1. Here, we tested the exact same configurations on two different parameterizations of the benchmark TPC-C. In this scenario, the scale factor controls the modeled number of warehouses. While for most configurations, throughput decreases for a more complex benchmark, some configurations achieve higher throughput for a more complex benchmark. A similar example was outlined by Pereira et al. for the video encoder X264 [15]. That is, configuration-specific performance can be highly sensitive to workload variation and the behavior under different workloads can change in unforeseeable ways. In turn, this can render performance models based on a single workload useless, unless configuration options' sensitivity to workloads is accounted for.

To address this limitation, two different directions have been pursued in the literature. First, performance models trained using a specific workload can be adapted to another specific workload. Second, one can specify workload characteristics as further independent variables when modeling configuration-dependent performance [17]. The first strategy direction relies on transfer learning techniques, where, given an existing performance model, in a separate step only the differences to a new environment are learned. Such a transfer function encodes which configuration options' influence on performance is sensitive to workload variation. While transfer learning is an effective strategy that is not limited to varying workloads [18], but can also be applied to different versions [19]–[21], or hardware setups [22], its main limitation is that the transfer function is specific to the differences between two environments.

In contrast to transfer learning, a more generalist approach is to consider the input fed to a software system as a further dimension for modeling performance. Here, a workload can be characterized by properties that—individually or in conjunction

with software configuration options—influence performance. For such a strategy to work, one requires knowledge of the characteristics of a workload that influence performance. This strategy has been effectively tested for a variety of application-domains, such as program verification. However, the added complexity comes at significant cost. Not only does this require substantially more measurements, we often lack knowledge of which performance-relevant characteristics best describe workloads.

The existing body of research reflects the prevalence and importance of the workload influence on software systems. All these works are aware of the workload dimension as a factor of performance variation, yet little is known about the quality and driving factors of the *interplay* between configuration options and workloads. Our understanding of this cross-factor relationship lacks knowledge of the following aspects:

— How different is configuration-specific performance across different workloads?
— How many configuration options are responsible for differences in workload-specific performance behavior?
— What are the driving factors of the interplay between configuration options and workloads with regard to performance?

To answer these questions, we have conducted a systematic empirical study that sheds light on whether and how configuration options and workload choices interact with regard to performance. Specifically, we analyze 35 659 configurations and 103 workloads across ten configurable software systems to obtain a broad picture of the interaction of configuration and workload when learning performance models and estimating a configuration's performance (i.e., response time). Aside from studying the sole effects of workload variation on performance behavior, we explore possible driving factors. To this end, we enrich performance observations with corresponding statement coverage data to understand workload variation at finer granularity.

Our findings show that varying the workload can influence configuration-dependent software performance in different ways, including non-linear and non-monotonous effects. We identify different two factors that drive input sensitivity. First, the fact *whether* option-specific code is executed at all under a specific workload can determine whether an option has an impact on performance. Second, the frequency at which covered code is executed (loop executions, system calls etc.) can shape performance and is sensitive to the workload. In the following, we refer to these two different aspects as *code coverage* and *code utilization*.

A *key take-away* of our study is that, to maintain and improve performance model representativeness, an additional notion of input sensitivity has to be considered. We argue that the use of code analysis techniques to address input sensitivity when varying the workload and maintain and improve the representativeness of a performance-prediction model.

To summarize, we make the following contributions:

— An empirical study of 35 659 configurations and 103 workloads across ten configurable software systems on whether interactions of workloads with configuration options affect performance and what factors can drive such interactions;
— A detailed analysis that illustrates that variation in code coverage and code utilization due to varying workloads can affect the influence of configuration options on software performance;
— A companion Web site[1] with supplementary material including performance and coverage measurements, experiment workloads and configurations, and an interactive dashboard[2] to **reproduce** all analyses and additional visualizations left out due to space limitations.
— [23]

II. Background and Problem Statement

A. Performance Prediction Models

Configurable software systems are an umbrella term for any kind of software system that exhibits configuration options to customize functionality. While the primary purpose of configuration options is to select and tune functionality, each configuration choice may also have implications on non-functional properties—be it intentional or unintentional. All performance prediction models approximate non-functional properties, such as execution time or memory usage, as a function of software configurations $c \in C$, formally $\Pi : C \to \mathbb{R}$.

Such models do not rely on an understanding of the internals of a configurable software system, but are learned from a training set of configuration-specific performance observations. In this vein, finding configurations with optimal performance [24]–[26] and estimating the performance for arbitrary configurations of the configuration space is an established line of research [2]–[9]. Over the past years, several different machine-learning techniques, including probabilistic programming [1], multiple linear regression [2], classification and regression trees [5]–[7], Fourier learning [8], [9], and deep neural networks [3], [4] have shown effective to learn configuration-dependent software performance. The set of configurations for training can be sampled from the configuration space using a variety of different sampling techniques [27]. All sampling strategies aim at yielding a representative sample, either by covering the main effects of configuration options and interactions among them [28], or sampling uniformly from the configuration space [26], [29]. Most approaches share the perspective of treating a configurable software system as a black-box model at application-level granularity. Recent work has incorporated feature location techniques to guide sampling effort towards relevant configuration options [30], [31] or model non-functional properties at finer granularity [23], [32].

B. Varying Workloads

When assessing the performance of a software system, we ask how well a certain *operation* is executed, or, phrased differently, how well an *input fed to the software system*

[1] https://github.com/fse-submission-2022/workload-performance/
[2] https://workload-performance.herokuapp.com/

is processed. Such inputs, commonly called workloads, are essential to assessing performance, even detached from the specific context of configurable software systems. By nature, the workload of a software system is application-specific, such as a series of queries and transactions fed to a database system, a set of raw image files for video encoding, or an arbitrary file for data compression etc. Workloads can often be distinguished by characteristics (dimensions) they exhibit, such as their file size in general, or, the type of data to be compressed (text, binary data) for instance.

A useful workload for assessing performance or benchmarking should, in practice, closely represent the real-world scenario that the system under test is deployed in. To achieve this, a well-defined and widely employed technique in performance engineering is workload characterization [33], [34]. To select a representative workload, it is imperative to explore workload characteristics and validate a workload with real-world observations. This can be achieved by constructing workloads, among others, from usage patterns [35], or by increasing the workload coverage by using a mix of different workloads rather than a single one [36].

While workload characterization and benchmark construction is domain-specific, there are numerous examples of this task being driven by community efforts instead of individuals. For instance, the non-profit organizations Standard Performance Evaluation Corporation (SPEC) and Transaction Processing Performance Council (TPC) provide large bodies of benchmarks for data-centric applications or across different domains, respectively.

C. Representative Estimations?

While the notion of representative workloads is ubiquitous in an industry setting, we face a different situation in research on empirical performance models (cf. Section II-A): Most approaches provide accurate performance estimations, yet are based on observations gained by varying configurations while keeping the workload the same. The choice of workloads is an "orthogonal" problem [23] and can limit the model's generalizability to different workloads, especially if the training workload poorly represents real-world scenarios. While the configuration-specific behavior might be congruent across different workloads (i.e, a configuration scoring comparably for different workloads), we cannot rely on such assumptions in practice, where different workloads can indeed result in entirely different configuration-specific performance behavior.

Consider the illustratory example from Figure 2. Here, we measure performance under three different configurations and two different inputs, w_1 and w_2. The software system takes longer for w_2, regardless of the configuration. This could, for instance, be attributed to w_2 being more complex and performance scaling proportionally. However, if we ask which option is fastest in general there is no concrete answer since configuration 1 prevails for w_1 and configuration 2 for w_2. If we look at the corresponding performance prediction models below, we learn that the influence of o_1 is smaller than for o_2 in Π_{w_1} and vice versa in Π_{w_2}. Thus, there must be some hidden feature or variable, the "difference" between w_1 and w_2 that accounts for this performance variation beyond configuration options.

id	c_1	c_2	c_3	$\Pi_{w_1}(c)$	$\Pi_{w_2}(c)$
1	1	0	0	4 seconds	22 seconds
2	0	1	0	6 seconds	14 seconds
3	0	0	1	9 seconds	35 seconds

$$\Pi_{w_1}(c) = c_1 \times 3 + c_2 \times 5 + c_3 \times 8 + 1$$

$$\Pi_{w_2}(c) = c_1 \times 21 + c_2 \times 13 + c_3 \times 34 + 1$$

Figure 2: Input sensitivity in the case for a software system with three configuration options $C = (o_1, o_2, o_3)$.

Some aspects of this *input sensitivity* have been observed and documented before in the literature [15], [19], [37] and raise questions, such as: *Which options are input sensitive? What are the driving factors for input sensitivity? Can we estimate which options are input-sensitive?* We set out to answer these questions in this paper.

D. Workloads and Performance Prediction

To some aspects of the said questions, different approaches have been proposed to tackle the problem of input sensitivity.

a) Workload-aware Performance Modeling: Extending on workload characterization (cf. Section II-B), a strategy that embraces workload diversity is to incorporate workload characteristics into the problem space of a performance prediction model. Here, performance is modeled as a function of both the configuration options exhibited by the software system as well as the workload characteristics, formally $\Pi : C \times W \rightarrow \mathbb{R}$. The combined problem space enables learning performance models that generalize to workloads that exhibit characteristics denoted by W since we can screen for performance-relevant combinations of options and workload characteristics. Although this strategy is highly application-specific, it has been successfully applied to different domains, such as program verification [17]. However, its main disadvantages are twofold: The combined problem space (configuration and workload dimension) requires substantially more observations to screen for identifying performance-relevant options, characteristics, and combinations thereof. In addition, previous work found that only few configuration options are input sensitive [19] when varying the workload. That is, the problem of identifying meaningful, but sparse predictors is exacerbated since one must not only identify performance-relevant configuration options, but also input-sensitive ones.

b) Transfer Learning for Performance Models: Another strategy that builds on the fact that, across different workloads, only few configuration options are in fact input sensitive [19]. Here one first trains a model on a standard workload and, subsequently, adapts it to different workloads. Contrary to a generalizable workload-aware model, transfer learning strategies focus on approximating a transfer function that, without characterizing the workload, encodes the information of which

configuration options are sensitive to differences between a source and target pair of workloads. Training a workload-specific model and adapting it on occasion provides an effective means to reuse performance models, which is not limited to workloads [18], but has successfully been applied to different hardware setups [22], [38] and across versions [21]. The main shortcoming of transfer learning approaches is that they do not generalize to arbitrary workloads, since a transfer function is tailored to a specific target workload. Here, one trades off generalizability and measurement cost as learning a transfer function requires substantially fewer training samples.

While both directions are effective means to handle input sensitivity, to the best of our knowledge, there is no *systematic* assessment of the factors that drive the interaction between configuration options and workloads with regard to performance. Understanding scenarios that are associated with or even cause incongruent performance influences across workloads can help practitioners to employ established analysis techniques more effectively and can motivate researchers to devise analysis techniques dedicated to such scenarios.

III. Study Design

In what follows, we describe the general experiment setup and study design as well as research questions. We make all performance measurement data, configurations, workloads, and learned performance models available on the paper's companion Web site.

A. Research Questions

Our first two research questions shed light on the input sensitivity of the performance behavior of the studied software systems. We first take a look at systems as a whole (RQ_1) with regard to a large set of configurations and, subsequently, consider individual configuration options (RQ_2). Extending on the results of RQ_2, we explore possible driving factors and indicators for workload-specific performance variation (RQ_3).

1) Performance Variation Across Workloads: Performance variation can arise from differences in the workload [39]. In a practical setting, the question arises whether, and if so, to what extent an existing workload-specific performance model is representative of the performance behavior of other workloads. That is, can a model estimating performance of different configurations be reused for the same software system but run with a different workload? Depending on the degree of similarity of the performance behavior across workloads, we obtain a clearer picture of the prevalence of input sensitivity and to what extent the strategies outlined in Section II-D might be applicable. To this end, we formulate the following research question:

RQ_1	*To what extent does performance behavior vary across workloads?*

2) Option Influence Across Workloads: At large, performance behavior is the resulting effect arising from multiple configuration options' and combinations' respective influences. To understand which configuration options are driving performance variation, in general, and which are input sensitive, in particular, we formulate the following research question:

RQ_2	*To what extent do influences of individual configuration options depend on the workload?*

3) Causes of Input Sensitivity: The first two research questions describe the performance behavior of our subject systems: Based on the results of related work, we expect configuration options to be, at least, to some extent input sensitive. To contextualize our findings, we switch our perspective to the code level. The goal is to understand the relationship between input sensitivity (i.e., variation in the performance influence of configuration options) and the execution of the subject system under varying workloads. We hypothesize that executions under different workloads also exhibit variation with respect to what code sections are executed and how this code is used. Using code coverage analysis—an easy to understand and widely employed technique—we are interested in how far one could infer or explain performance influence variation just based on code.

RQ_3	*Does the variation in configuration options' performance influence across workloads correlate with differences in the respective execution footprint?*

Table I: Subject System Characteristics

System	Lang.	Domain	Version	# O	# C	# W
JUMP3R	Java	Audio Encoder	1.0.4	19	4 196	6
KANZI	Java	File Compressor	1.9	24	4 112	9
DCONVERT	Java	Image Scaling	1.0.0-α7	17	6 764	12
H2	Java	Database	1.4.200	16	1 954	8
BATIK	Java	SVG Rasterizer	1.14	10	1 919	11
XZ	C/C++	File Compressor	5.2.0	33	1 999	13
LRZIP	C/C++	File Compressor	0.651	11	190	13
X264	C/C++	Video Encoder	baee400…	26	3 113	9
Z3	C/C++	SMT Solver	4.8.14	12	1 011	12

 # O: No. of options, # C: No. of configurations, # W: No of. workloads

B. Experiment Setup

1) Subject System Selection: We selected ten configurable software systems for our study. To ensure that our findings are not specific to one domain or ecosystem, the selection comprises an equal mix of Java and C/C++ systems from different application domains (cf. Table I). We include systems studied in previous and related work [15], [30], [32] and incorporate further ones with comparable size and configuration complexity. All systems operate by processing a domain-specific input fed to them (henceforth called *workload*). This study treats execution time as the key performance indicator with the exception of H2, where we report throughput.

2) Workload Selection: This study relies on a selection of workloads for each domain or software system. Ideally, each set of workloads is diverse enough to be representative of most possible use cases. We selected the workload sets in this spirit,

but cannot always guarantee a measurable degree of diversity and representativeness. This is due to the opacity of workloads: Beyond educated guesses, prior to conducting measurements, it is not possible to state which workload characteristics (size, scale, file type etc.) are performance-relevant. We discuss this aspect in the threats to validity. Below we outline the ten case studies along with the workloads tested.

For the *audio encoder* JUMP3R, the measured task was to encode raw WAVE audio signals to MP3 (JUMP3R). We selected a number of different audio files from the Wikimedia Commons collection[3] and aimed at varying the file size/signal length, sampling rate, and number of channels. Both applications share all workloads.

For the *video encoder* X264, the measured task was to encode raw video frames (y4m format). We selected a number of files from the "derf collection"[4], a set of test media for a variety of use cases. The frame files vary in resolution (low/SD up to 4K) and file size. For files with 4K resolution, we limited our measurements to encoding a subset of consecutive frames.

For the *file compression* tools KANZI, XZ, and LRZIP, we used a variety of community compression benchmarks that represent different goals, including mixes of files of different types (text, binary, structured data etc.) or single-type files. We augmented this set of workloads with custom data, such as the Hubble Deepfield image and a binary of the Linux kernel. Beyond this set of workloads, for XZ and LRZIP we added different parameterizations of the UIQ2 benchmark[5] to study the effect of varying file size.

For the *SMT solver* Z3, the measured task was to decide the satisfiability (find a solution or counter example) of a range of logical problems expressed in the SMT2 format. We selected the six longest-running problem instances from Z3's performance test suite and augmented it with additional instances from the SMT2-Lib repository[6] to cover different types of logic and increase diversity.

For the *SVG rasterizer* BATIK, the measured task was to transform a SVG vector graphic into a bitmap. We selected a number of resources from the Wikimedia Commons collection, primarily varying the file size.

For the embedded *database* H2, we used a selection of four benchmarks (SmallBank, TPC-H, YCSB, Voter) from OLTPBENCH [40], a load generator for databases that allows for using a variety of performance testing benchmarks. For each benchmark, we varied the scale factor, which controls the complexity (number of entities modeled) in each scenario.

For the *image scaler* DCONVERT, the measured task was to transform resources (image files, Photoshop sketches) at different scales (useful for Android development). We selected files that reflect DCONVERT's documented input formats (JPEG, PNG, PSD, and SVG) and vary in file size.

3) Configuration Sampling: For each subject system, we sampled a set of configurations. As exhaustive coverage of the configuration space is infeasible due to combinatorial explosion [41], for binary configuration options, we combine several coverage-based sampling strategies and uniform random sampling into an *ensemble* approach: We employ option-wise and negative option-wise sampling [2], where each option is enabled once (i.e., in, at least, one configuration), or all except one, respectively. In addition, we use pairwise sampling, where two-way combinations of configuration options are systematically selected. Interactions of higher degree could be found accordingly, however, it is computationally prohibitively expensive [41]. Last, we augment our sample set with a random sample that is, at least, the size of the coverage-based sample. To achieve a nearly uniform random sample, we used *distance-based sampling* [29]. If a software system exhibited numeric configuration options, we varied them across, at least, two levels to account for their effect.

4) Coverage Profiling: To assess what lines of code are executed for each combination of workload and software configuration, we used two separate approaches for Java and C/C++. For Java, we used the on-the-fly profiler JACOCO[7] that intercepts byte code running on the JVM at run-time. For C/C++, we added instrumentation code to the software systems using CLANG/LLVM[8] to collect coverage information. We split the performance measurement and coverage analysis runs to avoid distortion from the profiling overhead.

5) Measurement Setup: All experiments were conducted on three different compute clusters), where all machines within a compute cluster had the identical hardware setup: Cluster A with an Intel Xeon E5-2630v4 CPU (2.2 GHz) and 256 GB of RAM, cluster B with an Intel Core i7-8559U CPU (2.7 GHz) and 32 GB of RAM, and cluster C with an Intel Core i5-8259U (2.3 GHz) and 32 GB of RAM. All clusters ran a headless Debian 10 installation (kernel 4.19.0-17 for cluster A and 4.19.0-14 for clusters B and C). To minimize measurement noise, we used a controlled environment, where no additional user processes were running in the background, and no other than necessary packages were installed.

We ran each subject system *exclusively* on a single cluster: H2 on cluster A; DCONVERT, BATIK and JADX on cluster B; the remaining systems on cluster C.

For all data points, we report the median across five repetitions (except for H2), which has shown to be a good trade-off between variance and measurement effort. For H2, we omitted the repetitions as, in a pre-study, running on the identical cluster setup, we found that across all benchmarks the coefficient of variation of the throughput was consistently below 5 %.

IV. STUDY RESULTS

A. Comparing Performance Distributions (RQ_1)

1) Operationalization: We answer RQ_1 by pairwisely comparing the performance distributions from different workloads

[3] Linktowikimedia
[4] https://media.xiph.org/video/derf/
[5] linkzuuiq2benchmark
[6] LinktoSMT2-Lib
[7] https://www.jacoco.org/jacoco/trunk/doc/
[8] linktoLLVM

Table II: Four disjoint categories and criteria of relationships between pairs of workload-specific performance distributions.

	Category	Criteria
LT	Linear transformation	$r^* \geq 0.6$
XMT	Monotonous transformation	$r^* < 0.6$ and $\tau^* \geq 0.6$
NMT	Non-monotonous transformation	(otherwise)

(cf. the comparison in Figure 1) and by determining whether any two distributions are similar or, if not, can be transformed into each other: For the former case, we tested all combinbations of workload-specific performance observations with the Wilcoxon signed-rank test[9] [44]. For all combinations, we rejected the null hypothesis H_0 at $\alpha = 0.95$. To account for overpowering due to high and different sample sizes (cf. Table I), we further checked effect sizes to weed out negligible effects. Following the interpretation guidelines from Romano et al. [45], for no combination, Cliff's δ [46] exceeded a threshold effect size of $|\delta > 0.147|$. For the latter case, we are specifically interested in what *type* of transformation is necessary as this determines *how* complex a workload interacts with configuration options. Specifically, we categorize each pair of workloads with respect to the following aspects:

1) *Linear Correlation*: To test whether both performance distributions are shifted by a constant value or scaled by a constant factor, we compute for each pair of distributions Pearson's correlation coefficient r. To discard the sign of relationship, we use the absolute value and a threshold of $|r| > 0.6$ to indicate a linear relationship.
2) *Monotone Correlation*: Finally, we test whether there exists a monotonous relationship between the two performance distributions. We use Kendall's rank correlation coefficient τ [47] and a threshold of $|\tau| > 0.6$ for a monotonous relationship.

Based on these two correlation measures, we composed three categories that each pair of performance distributions can be categorized into. If both distributions exhibit a strong linear relationship, we classify them as linearly transformable (*LT*). If we observe a strong monotonous, but not a linear relationship, we classify such pairs as exclusively monotonously transformable into a separate category (*XMT*). Last, if the comparison yields no monotonous relationship, we can only transform them using non-monotonous methods (*NMT*). We summarize the category criteria as well as the category counts in Table II.

2) Results: We summarize the results of our classification in Table III. The observed range of relationships across the ten software systems has to type prevail across all software systems. All software systems, at least, in part, exhibit performance distributions which can be transformed into one another using an linear transformations (*LT*), such as shifting by a constant value or scaling by a constant factor. In particular, for JUMP3R and X264, we observe only such behavior. The presence of linear transformations corresponds to experimental insights from Jamshidi et al., who encoded differences between performance distributions using linear functions [19].

Exclusively monotone transformations (*XMT*) are the exception and are only exhibited by five out of the ten systems (KANZI, BATIK, XZ, LRZIP, Z3), twice with only one workload pair each (XZ and Z3). For all, except two systems (JUMP3R and X264) we observe non-monotonous transformations (*NMT*) with different prevalence. For three systems (DCONVERT, H2, and Z3) the majority of transformations is non-monotous, for the remaining four systems (KANZI, BATIK, XZ, LRZIP) more than 10 % of workload pairs fall into this category.

[9]We use non-parametric methods since performance-distributions are often long-tailed and multi-modal [42], [43] and, thus, fail to meet requirements for parametric methods.

Table III: Frequency of each category (cf. Table II) for each software system studied and pairs of workloads.

System	Σ_{pairs}	*LT*		*XMT*		*NMT*	
		abs	*rel*	*abs*	*rel*	*abs*	*rel*
JUMP3R	*15*	15	100.0 %	0	0 %	0	0 %
KANZI	*36*	28	77.8 %	4	11.1 %	4	11.1 %
DCONVERT	*66*	29	43.9 %	0	0 %	37	56.1 %
H2	*28*	13	46.4 %	0	0 %	15	53.6 %
BATIK	*55*	28	50.9 %	8	14.6 %	19	34.6 %
XZ	*78*	65	83.3 %	1	1.3 %	12	15.4 %
LRZIP	*78*	57	73.0 %	13	16.7 %	8	10.3 %
X264	*36*	36	100 %	0	0 %	0	0 %
Z3	*66*	10	15.2 %	1	1.5 %	55	83.3 %

Summary (RQ_1): Varying the workload causes a substantial amount of variation among performance distributions. Across workloads, we observed *mostly linear* (majority for for 70 % of the subject systems), but to a large extent, also *non-monotonous* relationships (majority for 30 % of the subject systems).

B. Input Sensitivity of Options (RQ_2)

1) Operationalization: To address RQ_2, we need to determine the configuration options' influence on performance and assess their variation across workloads.

Explanatory Model: To obtain accurate and interpretable performance influences per option, we learn an explanatory performance model using the entire sample set that is based on multiple linear regression [1], [2], [9]. Here, each variable in the linear model corresponds to an option and each coefficient represents the corresponding option's influence on performance. We limit the set of independent variables to individual options rather than including higher-order interactions to be consistent with the feature location used for RQ_3 where we determine option-specific, yet not interaction-specific code segments.

Standardization: To facilitate the comparison of regression coefficients across workloads, we follow common practice in machine learning and standardize our dependent variable by subtracting the population's mean performance and divide the result by the respective standard deviation. Henceforth, we refer to these standardized regression coefficients as *relative performance influences*. A beneficial side effect of

standardization is that the observed variation of regression coefficients for each configuration option cannot be attributed to shifting or scaling effects (affine transformation, category *LT* in Table II). This way, we can pin down the non-linear or explicitly non-monotonous effect that workloads may exercise on performance.

Handling Multicollinearity: Multicollinearity is a standard problem in statistics and emerges when features are correlated [48]. This can, for instance, arise from groups of mutually exclusive configuration options and result in distorted regression coefficients [1]. Although the model's prediction accuracy remains unaffected, we cannot trust and interpret the calculated coefficients. To mitigate this problem and, in particular, to ensure that the obtained performance influences remain interpretable, we follow best practices and remove specific configuration options from the sample that cause multicollinearity [1]. For the training step, we exclude all mandatory configuration options since these, by definition, cannot contribute to performance variation. In addition, for each group of mutually exclusive configuration options, we discard one group member. These measures reduced the variance inflation factor (indicating multicollinearity) to a negligble degree [49].

From the comparison of the relative performance influences, we can answer RQ_2 in detail and describe how options are sensitive to varying the workload, what characteristic traits describe the performance influences, and whether we can identify patterns.

2) Results: Our results from fitting performance prediction models show a wide variety of input sensitivity. Due to the size of our empirical study and space limitations we selected three configuration options that showcase different characteristic traits of observed input sensitivity. The exhaustive analysis for all configuration options is illustrated at the interactive dashboard provided as supplementary material.

a) Conditional Influence: For some configuration options, we observe that they become influential to performance only under specific workloads and remain non-influential otherwise. An example of such conditional influence is the configuration option "MONO" for the mp3 encoder jump3r. We illustrate the performance influence of this option across all workloads tested in the bar plot in Figure3. Selecting this option reduced the execution time greatly for two workloads, whereas for the other four workloads, the influence was substantially smaller.

According to the documentation of the software system, selecting this option for stereo files (i.e., audio files with two channels) results in averaging them into one channel. Indeed, the two workloads described above are the only ones that exhibit two audio channels in this selection. Hence, this example illustrates how a workload characteristic can condition the performance influence of a configuration option.

b) High Spread: In contrast to the binary example for conditional influence, the performance influence of most (relevant) configuration options exhibits spread of varying intensity. In the case of configuration option "proof" of the SMT solver z3, selecting this option could both increase or

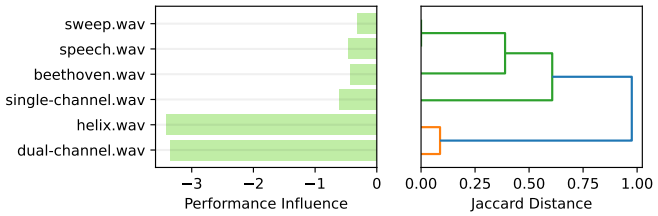

Figure 3: Option MONO (JUMP3R)

decrease the execution time as shown in Figure4. Compared to the example above, we cannot attribute this variation to an apparent workload characteristic accounts as in the case of conditional influence above. The global parameter "proof" enables tracking information in the SMT solver such that proof generation in the case of unsatisfiability is facilitated. Each workload in our selection contains multiple problems instances to decide satisfiability for. We conjecture that the ratio of satisfiable to unsatisfiable instances likely accounts for this high variation. From the user's perspective, any input to the solver is opaque in that satisfiability as a workload characteristic cannot be determined practically without a solver.

c) Scaling Anomaly: The last example we present illustrates that workload characteristics one would consider transparent at first glance do not necessarily have anticipated results. Consider the example of the configuration option MVSTORE of the database system h2. The option controls which storage engine, either the newer multi-version store or the legacy page store, is used. As shown in Figure5, we observe that selecting the newer multi-version store increases the measured throughput for all but one benchmark scenario. When tested with the Yahoo! Cloud Serving Benchmark (YCSB) with two different scale factors (which control the workload complexity; expressed as number of rows), we observe that the less complex parameterization (ycsb-600) results in a lower throughput. This is in stark contrast to the expectation that a more complex workload would show lower throughput. While it is possible that some optimizations of the multi-version store are only effective under higher load, this example demonstrates that performance influence is not guaranteed to scale with the workload as expected.

Summary (RQ_2): ...

Redo analysis from RQ2

C. Code Coverage and Performance (RQ_3)

1) Operationalization: To explore the possible driving factors of input sensitivity, we switch to the code level and analyze the relation of performance influences inferred for each configuration option (RQ_1) to code coverage information. Therefore, we augment our performance observations with code coverage information to assess differences in the execution under different workloads. Specifically, we are interested in code sections that implement option-specific functionality. From comparing the coverage information of option-specific code,

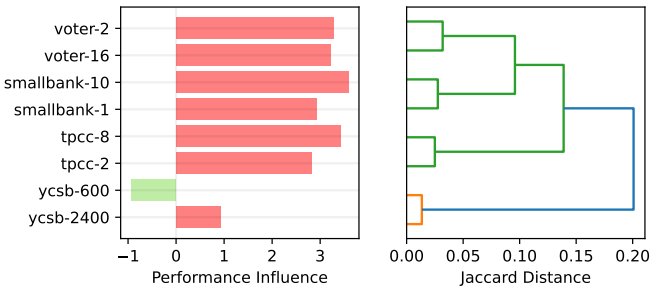

Figure 4: Option Mono (JUMP3R)

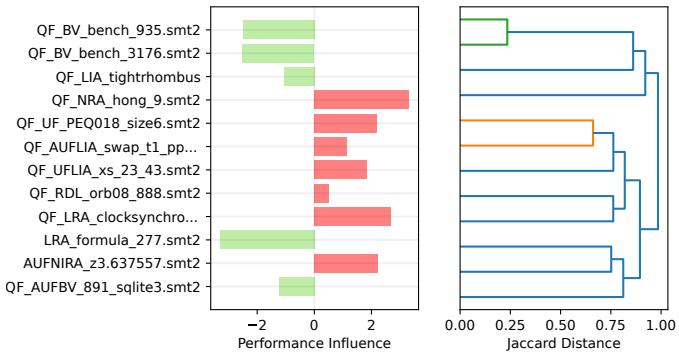

Figure 5: Option Mono (JUMP3R)

we can formulate different hypothetical scenarios explaining input sensitivity.

First, if we observe that the coverage of option-specific code is conditioned by the presence of some workload characteristic, we expect that such an option is only influential under respective workloads. This scenario enables us (to some extent) to use code coverage as a cheap-to-compute proxy for estimating the representativeness of a workload and, by extension, resulting performance models: For options that are known to condition code sections, we can maximize option-code coverage to elicit all option-specific behavior and, thus, performance influence. For instance, a database system could cache a specific view only if a minimum number of queries are executed. Here, the effect of any caching feature would be conditioned by the number of transactions resulting from the workload.

Second, if we observe performance variation across workloads in spite of similar or identical option-specific code coverage, we draw a different picture. Here, we cannot attribute performance variation to code coverage, yet have to consider differences in the workloads' characteristics as potential cause: The presence of a workload characteristic may influence not *what* code sections are executed, but *how* code sections are executed. For instance, in a simple case, a software system's performance may scale linearly with the input size. In a more complex case, the presence of a characteristic may determine how frequently an operation is repeated, as is the case for a database merge. Here, we would not elicit the worst-case performance if a previous transaction has sorted the data (e.g., by building an index).

2) Locating Configuration-Dependent Code: To reason about option-specific code, we require a mapping of configuration options to code. The problem of determining which code section implements which functionality in a software system is known as *feature location* [50]. While there are a number of approaches based on static [30], [51], [52] and dynamic taint analysis [31], [53], [54], we employ a more light-weight, but also less precise approach that uses code coverage information, such as execution traces. The rationale is that, by exercising feature code, for instance via enabling configuration options or running corresponding tests, its location can be inferred from differences in code coverage. Applications of such an approach have been studied not only for feature location [55]–[58], but root work on in program comprehension [59]–[63] and fault localization [64], [65]. Specifically, we follow a strategy akin to *spectrum-based feature location* [57]:

We commence with obtaining a baseline of all code that can be associated with a configuration option in the scope of our workload selection. Since we are looking for workload-specific differences in option-code coverage, the expressiveness of such a baseline depends on the diversity of the workloads in question. To infer option-specific code, we split our sample (cf. Section ?) into two disjoint sets c_o and $c_{\neg o}$ such that option o is selected only in c_o and vice versa. Next, we select from our coverage logs the corresponding covered lines of code, S_o and $S_{\neg o}$. The rationale is that all lines common between both sets are not affected by the selection of an option o. Thus, we compute the *symmetric difference* $\mathbb{S}_o = S_o \ \Delta \ S_{\neg o}$ to approximate option-specific or, at least, option-related code sections. To finally obtain code sections that are option-specific under a specific workload w, we repeat the steps above. Here, we only consider execution logs under workload w ($S_{o,w}$ and $S_{\neg o,w}$) and compute the symmetric difference $\mathbb{S}_{o,w} = S_{o,w} \ \Delta \ S_{\neg o,w}$.

3) Comparing Execution Footprints: From (a) the information about which code sections are specific to a configuration option and (b) how much of these sections is actually covered under different workloads, we can compare the workload-specific execution footprint for each option. By comparing the sets $\mathbb{S}_{o,v}$ and $\mathbb{S}_{o,w}$ for any two workloads v and w, we can estimate similarity between the option-code coverage via the Jaccard set similarity index. A Jaccard similariy of zero implies that there is no overlap in the lines covered under each workload, whereas a Jaccard similarity of 1 implies that the exact same code was covered. Based on this pairwise similarity metric $sim_o(v, w)$, we can compute a distance $d_o(v, w) = 1 - sim(v, w)$ and cluster all workload-specific execution profiles. We use agglomerative hierarchical clustering with full linkage to construct dendrograms. In this bottom-up approach, we iteratively add execution footprints to clusters and merge sub clusters into larger ones depending on their Jaccard similarity to each other.

4) Results: We report our findings for the relationship between execution footprints and performance influences for the same configuration options presented for RQ_2 since these illustrate likely causes of input sensitivity and the limitations of solely relying on code coverage. Nonetheless, the dashboard

on the supplementary web site provides diagrams and inferred feature code for all configuration options.

We report our findings for the relationship between execution footprints and performance influences for the same configuration options presented for RQ2 since these illustrate likely causes of input sensitivity and the limitations of solely relying on code coverage. Nonetheless, the dashboard on the supplementary web site provides diagrams and inferred option-specific code for all configuration options.

The dendrograms beside the visualizations of performance influences in Figures 3, 4, and 5, respectively, illustrate how similar the covered lines of option-specific code under each workload are.

In many cases, where a configuration option is "conditionally influential", the respective option-specific code under the interacting workloads fall into a cluster, as with the option-specific code for Mono in Figure3. In this particular example, the dendrogram can be somewhat misleading as the number of common lines of code between workloads helix and dual-channel is far greater than between the other four workloads. Hence, differences in the coverage of option-specific code can account for, at least, some input sensitivity.

The other two examples, configuration options "proof" (z3) and "MVSTORE" (h2) provide a different picture. Akin to the variation of performance influence of option "proof", the clustering for this configuration option (cf. Figure4) shows that some clusters are disjoint, and, thus, the option-specific code is highly fragmented depending on the workload.

Similarly to option "proof", for "MVSTORE" we see that the option-specific code is highly fragmented, yet all four benchmarks constitute clusters of high internal similarity. In the context of the observed variation of performance influences for the Yahoo! Cloud Serving Benchmark (YCSB), we see that even very high similarity in the covered code can virtually either improve or deteriorate performance.

Summary (RQ_3): ...

V. Discussion

A. Option-level Input Sensitivity

The observed input sensitivity for configuration options exhibits a wide range of characteristics. While a large portion of options scales proportionally with workload complexity or remains unaffected by workload variation, the performance influences of several configuration options appear sensitive to the workload. So far, the existing body of work on modeling and optimizing configuration-specific performance largely neglects the impact of workload variation at the cost of generalizability. Option-specific input sensitivity is a strong motivation for assessing influential factors beyond configuration options more thoroughly and rigorously. Performance variation that is unaccounted for poses the risk of distorting performance estimations and, thus, rendering performance models useless in practice. Although transfer learning is used in literature to reuse existing performance models across different workloads, our findings let us raise concerns about its effectiveness and generalizability.

In their exploratory analysis, Jamshidi et al. reuse existing performance models by learning a linear transfer function across workloads [19]. Our results from RQ_1 have shown non-monotonous performance relationships across workloads, which, in practice, can be challenging to capture with such transfer functions. The presence of *non-monotonous interactions* between workloads and configuration options provides a strong argument for employing more apt machine learning techniques. Ensemble-based approaches, such as gradient boosting regression trees have shown promising results for transfer across versions and are an avenue for future work [21].

Lesson 1: Option-level input sensitivity can challenge robust and practical configuration-specific performance models, yet is neglected in state-of-the-art approaches.

B. Driving Factors for Input Sensitivity

Extending on the findings from RQ_2 and RQ_3, to treat option-specific input sensitivity, a key challenge is to infer which options are susceptible to workload variation. Despite the limitations faced with the coverage-based approach from RQ_3 (missing baseline), we have correlated missed option code with variation in the observed performance. Our findings point to more detailed code analysis, such as line hit counts or execution logs, and more profound domain knowledge as key ingredients for mastering the input sensitivity assessment. In literature, the heuristic to consider only influential configuration options for transfer learning has shown effective results [18]. However, we question whether this approach would generalize in practice in the light of the conditional performance influences observed in Setions IV-B and IV-C. Clearly, more research is needed to provide an empirical base and framework for sensitivity assessment.

Lesson 2: Understanding the driving factors for input sensitivity is necessary for providing representative workloads and identifying sensitive configuration options.

VI. Threats to Validity

Threats to *internal validity* include measurement noise which may distort our classification into categories (Section IV-A) and model construction (Section IV-B). We mitigate these threats by repeating each experiment five times (except for H2; cf. Section III-B5) and reporting the median as a robust measure in a controlled environment. Moreover, the coverage analysis (cf. Section III-B4) entails a noticeable instrumentation overhead, which may distort performance observations. We mitigate this threat by separating the experiment runs for coverage assessment and performance measurement. In the case of H2, the load generator of the OLTPBench framework [40] ran on the same machine as the database since we were testing an embedded scenario with only negligible overhead.

Threats to *external validity* include the selection of subject systems and workloads. To ensure generalizability, we select software systems from various various application domains as well as two different programming language ecosystems (cf. Table I). In lieu of domain knowledge, we cannot select workloads systematically with respect to workload characteristics due to workloads inherent opacity. We address this issue by varying likely relevant characteristics and, where possible, reusing workloads across subject systems of the same domain. Achieving true representativeness is desirable, yet intractable. The goal of this selection is to study the *presence* and quality of workload-option interactions, but not their *prevalence*. Hence, we are confident that this selection does not invalidate our findings.

VII. Conclusion

Most modern software systems exhibit configuration options to customize behaviors and meet user demands. Configuration choices, however, can also affect the performance of a software system. State-of-the-art approaches model configuration-dependent software performance, yet overlook variation due to the workload. Until now, there exists no *systematic* assessment of what is driving the effect that input sensitivity of individual configuration options' influence on performance. We have conducted an empirical study of 35 659 configurations and 103 workloads across ten configurable software systems to characterize the effects that varying the workload can have on configuration-specific performance. We compare performance measurements with code coverage data to identify possible factors that drive input sensitivity. We found that the interactions between options and workloads are driven by workload characteristics conditioning the execution of option-specific code sections as well as determining how option-specific code sections are executed. Our findings highlight the necessity to consider input sensitivity when modeling configuration-dependent performance as varying the workload resulted in a substantial number of non-monotonous relationships, which limits a performance model's representativeness. Code analysis can provide an effective strategy to differentiate between sources of input sensitivity to obtain more representative performance models.

References

[1] J. Dorn, S. Apel, and N. Siegmund, "Mastering Uncertainty in Performance Estimations of Configurable Software Systems," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*. ACM, 2020, pp. 684–696.

[2] N. Siegmund, A. Grebhahn, S. Apel, and C. Kästner, "Performance-Influence Models for Highly Configurable Systems," in *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2015, pp. 284–294.

[3] H. Ha and H. Zhang, "DeepPerf: Performance Prediction for Configurable Software with Deep Sparse Neural Network," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1095–1106.

[4] Y. Shu, Y. Sui, H. Zhang, and G. Xu, "Perf-AL: Performance Prediction for Configurable Software through Adversarial Learning," in *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ACM, 2020.

[5] J. Guo, K. Czarnecki, S. Apel, N. Siegmund, and A. Wasowski, "Variability-aware Performance Prediction: A Statistical Learning Approach," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 301–311.

[6] A. Sarkar, J. Guo, N. Siegmund, S. Apel, and K. Czarnecki, "Cost-Efficient Sampling for Performance Prediction of Configurable Systems," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 342–352.

[7] 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*, vol. 23, no. 3, pp. 1826–1867, 2018.

[8] Y. Zhang, J. Guo, E. Blais, and K. Czarnecki, "Performance Prediction of Configurable Software Systems by Fourier Learning," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2015, pp. 365–373.

[9] H. Ha and H. Zhang, "Performance-Influence Model for Highly Configurable Software with Fourier Learning and Lasso Regression," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, 2019, pp. 470–480.

[10] S. Falkner, M. Lindauer, and F. Hutter, "Spysmac: Automated configuration and performance analysis of sat solvers," in *Theory and Applications of Satisfiability Testing – SAT 2015*, M. Heule and S. Weaver, Eds. Springer International Publishing, 2015, pp. 215–222.

[11] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "SATzilla: Portfolio-Based Algorithm Selection for SAT," *Journal of Artificial Intelligence Research*, vol. 32, no. 1, p. 565–606, jun 2008.

[12] Y. Ding, J. Ansel, K. Veeramachaneni, X. Shen, U.-M. O'Reilly, and S. Amarasinghe, "Autotuning Algorithmic Choice for Input Sensitivity," *SIGPLAN Not.*, vol. 50, no. 6, p. 379–390, jun 2015.

[13] D. Plotnikov, D. Melnik, M. Vardanyan, R. Buchatskiy, R. Zhuykov, and J.-H. Lee, "Automatic Tuning of Compiler Optimizations and Analysis of their Impact," *Procedia Computer Science*, vol. 18, pp. 1312–1321, 2013, 2013 International Conference on Computational Science.

[14] A. Maxiaguine, Y. Liu, S. Chakraborty, and W. T. Ooi, "Identifying "Representative" Workloads in Designing MpSoC Platforms for Media Processing," in *2nd Workshop on Embedded Systems for Real-Time Multimedia, 2004. ESTImedia 2004.*, 2004, pp. 41–46.

[15] J. A. Pereira, M. Acher, H. Martin, and J.-M. Jézéquel, "Sampling Effect on Performance Prediction of Configurable Systems: A Case Study," in *Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE)*. ACM, 2020, p. 277–288.

[16] M. Khavari Tavana, Y. Sun, N. Bohm Agostini, and D. Kaeli, "Exploiting Adaptive Data Compression to Improve Performance and Energy-Efficiency of Compute Workloads in Multi-GPU Systems," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019, pp. 664–674.

[17] U. Koc, A. Mordahl, S. Wei, J. S. Foster, and A. A. Porter, "SATune: A Study-Driven Auto-Tuning Approach for Configurable Software Verification Tools," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 330–342.

[18] P. Jamshidi, M. Velez, C. Kästner, and N. Siegmund, "Learning to Sample: Exploiting Similarities across Environments to Learn Performance Models for Configurable Systems," in *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2018, p. 71–82.

[19] P. Jamshidi, N. Siegmund, M. Velez, C. Kästner, A. Patel, and Y. Agarwal, "Transfer Learning for Performance Modeling of Configurable Systems: An Exploratory Analysis," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, p. 497–508.

[20] P. Jamshidi, M. Velez, C. Kästner, N. Siegmund, and P. Kawthekar, "Transfer Learning for Improving Model Predictions in Highly Configurable Software," in *Proceedings of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 2017, p. 31–41.

[21] H. Martin, M. Acher, L. Lesoil, J. M. Jezequel, D. E. Khelladi, and J. A. Pereira, "Transfer Learning Across Variants and Versions: The Case of Linux Kernel Size," *IEEE Transactions on Software Engineering (TSE)*, pp. 1–1, 2021.

[22] Y. Ding, A. Pervaiz, S. Krishnan, and H. Hoffmann, "Bayesian Learning for Hardware and Software Configuration Co-Optimization," Tech. Rep.

[23] X. Han, T. Yu, and M. Pradel, "ConfProf: White-Box Performance Profiling of Configuration Options," in *Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE)*. ACM, 2021, p. 1–8.

[24] V. Nair, T. Menzies, N. Siegmund, and S. Apel, "Using Bad Learners to Find Good Configurations," in *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2017, pp. 257–267.

[25] V. Nair, Z. Yu, T. Menzies, N. Siegmund, and S. Apel, "Finding Faster Configurations Using FLASH," *IEEE Transactions on Software Engineering (TSE)*, vol. 46, no. 7, pp. 794–811, 2020.

[26] J. Oh, D. Batory, M. Myers, and N. Siegmund, "Finding Near-Optimal Configurations in Product Lines by Random Sampling," in *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2017, pp. 61–71.

[27] C. Kaltenecker, A. Grebhahn, N. Siegmund, and S. Apel, "The Interplay of Sampling and Machine Learning for Software Performance Prediction," *IEEE Software*, vol. PP, 04 2020.

[28] N. Siegmund, S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, and G. Saake, "Predicting Performance via Automated Feature-Interaction Detection," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 167–177.

[29] C. Kaltenecker, A. Grebhahn, N. Siegmund, J. Guo, and S. Apel, "Distance-based Sampling of Software Configuration Spaces," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1084–1094.

[30] 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 (ASE)*, pp. 1–36, 2020.

[31] M. Velez, P. Jamshidi, N. Siegmund, S. Apel, and C. Kästner, "White-Box Analysis over Machine Learning: Modeling Performance of Configurable Systems," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1072–1084.

[32] M. Weber, S. Apel, and N. Siegmund, "White-Box Performance-Influence Models: A Profiling and Learning Approach." IEEE, 2021, pp. 1059–1071.

[33] S. Ceesay, Y. Lin, and A. Barker, "A Survey: Benchmarking and Performance Modelling of Data Intensive Applications," in *IEEE/ACM International Conference on Big Data Computing, Applications and Technologies (BDCAT)*, 2020, pp. 67–76.

[34] A. V. Papadopoulos, L. Versluis, A. Bauer, N. Herbst, J. v. Kistowski, A. Ali-Eldin, C. L. Abad, J. N. Amaral, P. Tůma, and A. Iosup, "Methodological Principles for Reproducible Performance Evaluation in Cloud Computing," *IEEE Transactions on Software Engineering (TSE)*, vol. 47, no. 8, pp. 1528–1543, 2021.

[35] M. C. Calzarossa, L. Massari, and D. Tessera, "Workload Characterization: A Survey Revisited," *ACM Computer Survey*, vol. 48, no. 3, Feb. 2016.

[36] Z. M. Jiang and A. E. Hassan, "A Survey on Load Testing of Large-Scale Software Systems," *IEEE Transactions on Software Engineering (TSE)*, vol. 41, no. 11, pp. 1091–1118, 2015.

[37] L. Liao, J. Chen, H. Li, Y. Zeng, W. Shang, J. Guo, C. Sporea, A. Toma, and S. Sajedi, "Using black-box performance models to detect performance regressions under varying workloads: an empirical study," *Empirical Software Engineering (ESE)*, pp. 1–31, 2020.

[38] P. Valov, J.-C. Petkovich, J. Guo, S. Fischmeister, and K. Czarnecki, "Transferring performance prediction models across different hardware platforms," in *ICPE*. ACM, 2017, p. 39–50.

[39] S. Kounev, K.-D. Lange, and J. von Kistowski, *Systems Benchmarking*, 1st ed. Springer International Publishing, 2020.

[40] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux, "OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases," *Proceedings of the VLDB Endowment*, vol. 7, no. 4, pp. 277–288, Dec. 2013.

[41] C. Henard, M. Papadakis, M. Harman, and Y. Le Traon, "Combining Multi-Objective Search and Constraint Solving for Configuring Large Software Product Lines," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 2015, pp. 517–528.

[42] C. Curtsinger and E. D. Berger, "STABILIZER: Statistically Sound Performance Evaluation," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. ACM, 2013, p. 219–228.

[43] A. Maricq, D. Duplyakin, I. Jimenez, C. Maltzahn, R. Stutsman, and R. Ricci, "Taming performance variability," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 409–425.

[44] M. Lovric, *International Encyclopedia of Statistical Science*, 1st ed. Springer, Dec. 2010.

[45] J. Romano, J. D. Kromrey, J. Coraggio, J. Skowronek, and L. Devine, "Exploring methods for evaluating group differences on the NSSE and other surveys: Are the t-test and Cohen's d indices the most appropriate choices?" in *Annual Meeting of the Southern Association for Institutional Research*, 2006, pp. 1–51.

[46] N. Cliff, "Dominance statistics: Ordinal analyses to answer ordinal questions." *Psychological Bulletin*, vol. 114, pp. 494–509, 1993.

[47] M. G. Kendall, "A new measure of rank correlation," *Biometrika*, vol. 30, no. 1/2, pp. 81–93, 1938.

[48] J. I. Daoud, "Multicollinearity and Regression Analysis," *Journal of Physics: Conference Series*, vol. 949, p. 012009, dec 2017.

[49] R. M. O'Brien, "A caution regarding rules of thumb for variance inflation factors," *Quality & Quantity*, vol. 41, no. 5, pp. 673–690, 2007.

[50] J. Rubin and M. Chechik, "A Survey of Feature Location Techniques," in *Domain Engineering: Product Lines, Languages, and Conceptual Models*, I. Reinhartz-Berger, A. Sturm, T. Clark, S. Cohen, and J. Bettin, Eds. Springer, 2013, pp. 29–58.

[51] M. Lillack, C. Kästner, and E. Bodden, "Tracking Load-Time Configuration Options," *IEEE Transactions on Software Engineering (TSE)*, vol. 44, no. 12, pp. 1269–1291, 2018.

[52] L. Luo, E. Bodden, and J. Späth, "A Qualitative Analysis of Android Taint-Analysis Results," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 102–114.

[53] J. Bell and G. Kaiser, "Phosphor: Illuminating Dynamic Data Flow in Commodity JVMs," *ACM SIGPLAN Notices*, vol. 49, no. 10, p. 83–101, Oct. 2014.

[54] C. H. P. Kim, D. Marinov, S. Khurshid, D. Batory, S. Souto, P. Barros, and M. D'Amorim, "SPLat: Lightweight Dynamic Analysis for Reducing Combinatorics in Testing Configurable Systems," in *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2013, pp. 257–267.

[55] W. E. Wong and J. Li, "An Integrated Solution for Ttesting and Analyzing Java Applications in an Industrial Setting," in *Proceedings of the Asia-Pacific Conference on Software Engineering (ASPEC)*, 2005, p. 8.

[56] M. Sulír and J. Porubän, "Semi-automatic Concern Annotation using Differential Code Coverage," in *Proceedings of the IEEE International Scientific Conference on Informatics (ISCI)*, 2015, pp. 258–262.

[57] G. K. Michelon, B. Sotto-Mayor, J. Martinez, A. Arrieta, R. Abreu, and W. K. Assunção, "Spectrum-based Feature Localization: A Case Study using ArgoUML," in *Proceedings of the International Conference on Software Product Lines (SPLC)*. ACM, 2021, pp. 126–130.

[58] A. Perez and R. Abreu, "Framing Program Comprehension as Fault Localization," *Journal of Software: Evolution and Process*, vol. 28, pp. 840–862, 2016.

[59] N. Wilde and C. Casey, "Early Field Experience with the Software Reconnaissance Technique for Program Comprehension," in *Proceedings of the International Conference on Software Maintenance (ICSM)*, 1996, pp. 312–318.

[60] N. Wilde and M. C. Scully, "Software Reconnaissance: Mapping Program Features to Code," *Journal of Software Maintenance: Research and Practice*, vol. 7, no. 1, pp. 49–62, 1995.

[61] K. D. Sherwood and G. C. Murphy, "Reducing Code Navigation Effort with Differential Code Coverage," Department of Computer Science, University of British Columbia, Tech. Rep. 14, 2008.

[62] A. Perez and R. Abreu, "A Diagnosis-Based Approach to Software Comprehension," in *Proceedings of the International Conference on Program Comprehension (ICPC)*. ACM, 2014, pp. 37–47.

[63] B. Castro, A. Perez, and R. Abreu, "Pangolin: An SFL-Based Toolset for Feature Localization," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1130–1133.

[64] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong, "Fault Localization using Execution Slices and Dataflow Tests," in *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, 1995, pp. 143–151.

[65] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A Survey on Software Fault Localization," *IEEE Transactions on Software Engineering (TSE)*, vol. 42, no. 8, pp. 707–740, 2016.