Analyzing the Impact of Workloads on Modeling the Performance of Configurable Software Systems

***Abstract*—**
Context: **Modern software systems often exhibit configuration options to meet customer requirements, including a system's performance behavior. Estimating and optimizing configuration-dependent performance with machine learning techniques is an established line of research.**

Problem: **Most existing approaches in this area rely on software performance observations under a single workload (input fed to a system), which is not necessarily representative of a software system's real-world application scenarios. 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**

Objective: **The interplay of workload variation and configurations on software performance has not been studied systematically, yet we require a better understanding to adequately assess the genralizability of singular-workload approaches.**

Method: **We conducted a systematic empirical study across 25 258 configurations across nine real-world configurable software systems to investigate the effects of workload variation at system-level performance and for individual configuration options. To explore driving causes for workload-configuration interactions, we enrich performance observations with option-specific code coverage information.**

Results: **Our results indicate that workloads can induce substantial performance variation and interact with configuration options, often in *non-monotonous* ways. This limits not only the generalizability of singular-workload approaches, but also questions assumptions for existing transfer learning techniques. We demonstrate that workloads should be considered when building performance prediction models to maintain and improve representativeness and reliability.**

I. Introduction

Most modern software systems can be customized by means of configuration options to enable desired functionality or tweak non-functional aspects, such as performance or energy consumption. The relationship of configuration choices and their influence on performance has been extensively studied in the literature [1]–[10]. 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 rely on only a single workload that aims to represent a typical real-world application scenario.

It is almost folklore that choice of the workload (i.e., the input fed to the software system) influences the performance of software systems in different ways as has been shown for the domains of SAT solving [11], [12], compilation [13], [14], video transcoding [15], [16], data compression [17], and code verification [18]. Besides apparent interactions, such as performance scaling with the size of a workload, qualitative aspects can result in intricate and inadvertent performance variations.

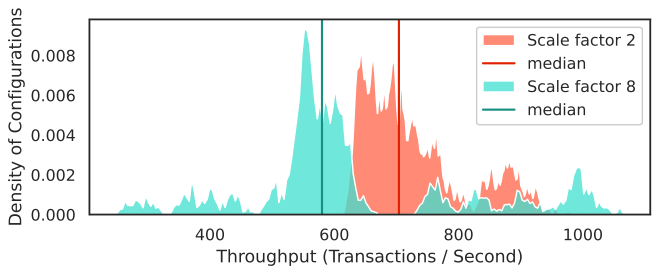


Figure 1: Throughput distribution of 1 954 configurations of the database system H2 run the TPC-C benchmark at different scale factors.

Take as an example two performance throughput distributions across the configuration space of the database H2 in Figure 1. Here, the exact same configurations on two different parameterizations of the benchmark TPC-C. have been measured In this scenario, the scale factor controls the modeled number of warehouses. While for most configurations, throughput decreases for a larger benchmark, some configurations achieve even a higher throughput[1]. 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 approaches have been pursued in the literature. The first approach relies on transfer learning techniques, where, given an existing performance model, in a separate step only the differences to a new environment are learned. 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 [19]–[23], its main limitation is that the transfer function is specific to the differences between two environments.

In contrast to transfer learning, a second and more generalist approach is to consider the input fed to a software system as a further dimension for modeling performance. A workload is

[1]A similar example was outlined by Pereira et al. for the video encoder x264 [16].

characterized by properties that—individually or in conjunction with software configuration options—influence performance. For such a strategy to work, one requires in-depth knowledge of the characteristics of a workload that influence performance. Let alone these characteristics can be mathematically modeled at all. This strategy has been effectively tested for a variety of application-domains, such as program verification and high-performance computing codes. However, the added complexity comes at significant cost. Not only does it require substantially more measurements, we often lack knowledge of which performance-relevant characteristics best describe a workload (e.g., what makes a program hard to verify or optimize).

The existing body of research [1]–[10], [24]–[27] confirms 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. Are workloads and configurations as factors influencing software performance orthogonal and can be treated independently or does their interplay give rise to intricate and inadvertent performance behavior?

We have conducted an empirical study that sheds light on whether and how choices of configuration and workload interact with regard to performance. Specifically, we analyzed 25 258 configurations across nine configurable real-world 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 what is driving the interaction between workload and configuration. To this end, we enrich performance observations with corresponding coverage data to understand workload variation with respect to executed code.

Our findings show that varying the workload can influence configuration-dependent software performance in different ways, including non-linear and non-monotonous effects. As a *key take-away* of our study, we found empirical evidence that single-workload approaches do not generalize across workload variations and that even existing transfer learning techniques are too limited to address non-monotonous performance variations induced by qualitative workload changes.

To summarize, we make the following contributions:

— An empirical study of 25 258 configurations across nine configurable software systems on whether and how interactions of workload and configuration affect software performance;

— A detailed analysis that illustrates that variation in code coverage due to varying workloads can affect the influence of individual configuration options on software performance;

— A companion Web site[2] with supplementary material including performance and coverage measurements, experiment workloads and configurations, and an interactive dashboard[3] for data exploration to *reproduce* all analyses and additional visualizations left out due to space limitations.

II. Background

A. *Performance Prediction Models*

Configurable software systems are an umbrella term for any kind of software system that exhibits configuration options to customize their 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 (execution time or memory usage)—be it intentional or not. Performance prediction models in this space approximate non-functional properties, such as execution time or memory usage, as a function of software configurations $c \in C$, formally $\Pi : C \rightarrow \mathbb{R}$.

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]–[27] and estimating the performance for arbitrary configurations of the configuration space is an established line of research [1]–[10]. Over the past years, several different learning and modeling techniques have shown to be effective to learn configuration-dependent software performance, including probabilistic programming [1], multiple linear regression [2], classification and regression trees [5]–[7], Fourier learning [8], [9], and deep neural networks [3], [4], [10]. The set of configurations for training can be sampled from the configuration space using a variety of different sampling techniques [28], [29]. All sampling strategies aim at yielding a representative sample, either by covering the main effects of configuration options and interactions among them [30], or sampling uniformly from the configuration space [27], [31]. 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 [32], [33] or model non-functional properties at finer granularity [34], [35].

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* is processed. Such inputs, commonly also 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 or a set of raw image files processed by a video encoder. Workloads can often be distinguished by the characteristics they exhibit, such as the amount and type of data to be processed (text, binary data).

[2]TODO: move supplementary web site to gitlab

[3]https://workload-performance.herokuapp.com

A useful workload for assessing performance should, in practice, closely resemble 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 [36], [37]. 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 from usage patterns [38] or by increasing the workload coverage by using a mix of different workloads rather than a single one [39].

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

C. Workloads and Performance Prediction

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 domains such as program verification [18]. 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 [20] 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.

not for all pos-si-ble

b) Transfer Learning for Performance Models: Another strategy builds on the fact that, across different workloads, only few configuration options are in fact input sensitive [20]. One first trains a model on a standard workload and, subsequently, adapts it to different a workload of choice. 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 demand provides an effective means to reuse performance models, which is not only limited to workloads [19], [22], [23], [40]. The main shortcoming of transfer learning is that it does not generalize to arbitrary workloads, since a transfer function is tailored to a specific target workload. Basically, one trades off generalizability for measurement cost because learning a transfer function requires substantially fewer training samples.

While both directions (workload-aware performance modeling and transfer learning) 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 (1) help practitioners to employ established analysis techniques more effectively and (2) motivate researchers to devise analysis techniques dedicated to such scenarios.

III. Study Design

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

A. Research Questions

The first two research questions investigate the workload-dependent input sensitivity of the studied software systems' performance behavior. We first take a look at the entire system (RQ$_1$) with regard to a large set of configurations and, subsequently, to individual configuration options (RQ$_2$). Furthermore, we explore possible driving factors and indicators for workload-specific performance variation of configuration options (RQ$_3$).

1) Performance Variation Across Workloads: Performance variation can arise from workload variation [41]. 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 also other workloads. That is, can a model estimating the performance of different configurations be reused for the same software system but run with a different workload? Clearly, it depends. But, analyzing systematically on how the degree of similarity of workloads and corresponding performance behaviors varies across the configuration space provide insights to what extent the strategies of transfer performance models (outlined in Section II-C) might be applicable. To this end, we formulate the following research question:

RQ$_1$	*To what extent does performance behavior vary across workloads and configurations?*

2) Option Influence Across Workloads: The global performance behavior emerges from the influences of several individual options and their interaction as well as the combined influence with the workload on performance. To understand which configuration options are driving performance variation,

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

in general, and which are workload sensitive, in particular, we state 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: From related work, we expect configuration options to be, at least, to some extent input sensitive. To contextualize our findings, we switch now our perspective to the code level. The goal is to understand the relationship between input sensitivity and the internal 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. Differences in performance influences of individual methods may stem from differences in program execution. Depending on whether an option is active or what value it has been set to, we may visit different code sections of the program or change their intensity. We are interested in how far one could infer or even explain performance variations just based on standard code-coverage analyses. The result would provide us either with a simple proxy for input sensitive configuration options or indicate that performance variations are more complex and non-trivial to detect a priori.

RQ$_3$ | *Does the variation of performance influence of configuration options across workloads correlate with differences in the respective execution footprint?*

B. Experiment Setup

1) Subject System Selection: We selected nine configurable software systems for our study. To ensure that our findings are not specific to one domain or ecosystem, we include a mix of Java and C/C++ systems from different application domains (cf. Table I). In particular, we include systems studied in previous and related work [16], [32], [34] and incorporate further ones with comparable size and configuration complexity (in terms of numbers of configurations and configuration options). All systems operate by processing a domain-specific input fed to them. 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 prior to conducting the actual measurements. Basically, this it what motivates this study in the first place. Nevertheless, we discuss this aspect as a threat to validity in Sections VI. Next, we outline the nine subject systems 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[4] 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"[5], 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[6] 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[7], to cover different types of logic and to 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 [42], a load generator for databases. 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.

[4]https://commons.wikimedia.org/wiki/Category:Images
[5]https://media.xiph.org/video/derf/
[6]http://mattmahoney.net/dc/uiq/
[7]https://smt-comp.github.io/2017/benchmarks.html

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 [43], 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, full interaction coverage is computationally prohibitively expensive [43]. 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* [31]. 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[8], which intercepts byte code running on the JVM at run-time. For C/C++, we added instrumentation code to the software systems using CLANG/LLVM[9] to collect coverage information. We split the performance measurement and coverage analysis runs to avoid distortion from the profiling overhead.

5) Measurements: 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 %.

add reference?

IV. Study Results

In this section, we present the results of our empirical study with regard to variation of system-level performance distributions (RQ$_1$) and the performance influence of individual configuration options (RQ$_2$) as well as the relationship of configuration-specific code-coverage and workload variation (RQ$_3$).

Table II: Three 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)

A. Comparing Performance Distributions (RQ$_1$)

1) Operationalization: We answer RQ$_1$ by pairwisely comparing the performance distributions from different workloads (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[10] [46]. 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. [47], for no combination, Cliff's δ [48] 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*: We test whether there exists a monotonous relationship between the two performance distributions. We use Kendall's rank correlation coefficient τ [49] 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, we have the pairs with a non-monotonous relationship (*NMT*). We summarize the category criteria as well as the category counts in Table II.

[8]https://www.jacoco.org/jacoco/trunk/doc/
[9]https://clang.llvm.org/docs/SourceBasedCodeCoverage.html

[10]We use non-parametric methods since performance-distributions are often long-tailed and multi-modal [44], [45] 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 %

2) Results: We depict the results of our classification in Table III. The observed range of relationships across the nine software systems exhibit no type that prevails across all software systems. All software systems, at least, in part, exhibit performance distributions that can be transformed into one another using a linear transformation (*LT*), such as shifting by a constant value or scaling by a constant factor. In particular, for JUMP3R and X264, we observe solely such behavior. The presence of linear transformations corresponds to experimental insights from Jamshidi et al., who encoded differences between performance distributions using linear functions [20].

Exclusively monotone transformations (*XMT*) are the exception and are exhibited only by five out of the nine 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 relationships (*NMT*) with different prevalence. For three systems (DCONVERT, H2, and Z3), the majority of transformations required is non-monotonous; for the remaining four systems (KANZI, BATIK, XZ, LRZIP), more than 10 % of workload pairs fall into this category.

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

B. Workload Sensitivity of Individual 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 based on the entire sample set using 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.

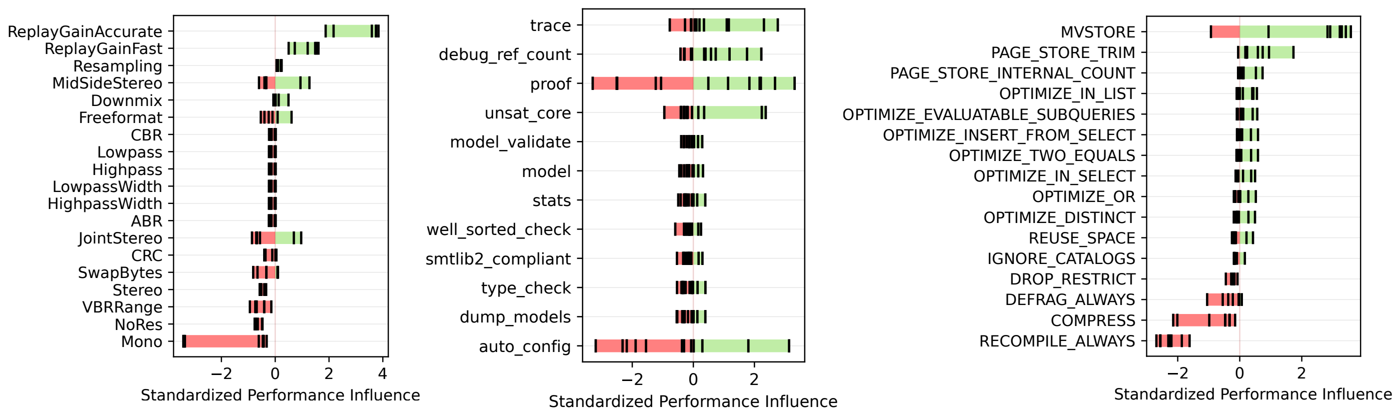
Handling Multicollinearity: Multicollinearity is a standard problem in statistics and emerges when features are correlated [50]. This can, for instance, arise from groups of mutually exclusive configuration options and result in distorted regression coefficients [1]. Although the model's prediction accuracy remains unaffected, we cannot trust and easily 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 negligible degree [51] .

VIF Anal-
y-
sis

2) Results: Our results show a wide variety of workload 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 in terms of an interactive dashboard provided as supplementary material. We strongly invite the interested reader to use this interactive dashboard to explore all distributions and result obtained in this study.

In Figure 2, we show the distribution of configuration options' performance influences for three of the nine software systems (JUMP3R, Z3, and H2). The following patterns refer to one row in Figures 2a, 2b, and 2c (configuration option) for one subject system each.

a) Conditional Influence: For some configuration options, we observe that they affect 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 six workloads presented as bar plots in Figure 3a. Selecting this option reduced the execution time substantially for two workloads, whereas for the other four workloads, the effect was substantially smaller.



(a) JUMP3R (b) Z3 (c) H2

Figure 2: Distribution of configuration options' performance influences for JUMP3R, Z3, and H2.

According to the documentation of tJUMP3R[11], 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: Another pattern we found is that the performance influence of most (relevant) configuration options exhibits a large spread. For example, option proof of the SMT solver Z3 can both increase or decrease the execution time as shown in Figure 3b. Compared to the example above, we cannot attribute this variation to an apparent workload characteristic. The global parameter proof enables tracking information, which is used for proof generation in case a problem instance is unsatisfiable. 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.

reference?

c) Scaling Anomaly: The anomaly pattern is shown for the configuration option MVSTORE of the database system H2 in Figure 3c (left). The option controls which storage engine, either the newer multi-version store or the legacy page store, is used. 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 (that control the workload complexity; expressed as number of rows), we found 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 effective only under higher load, this example demonstrates that performance influence is not guaranteed to scale with the workload as expected.

could be bug, could be interesting

[11]https://github.com/Sciss/jump3r/blob/master/README.md

Summary (RQ$_2$): Workloads can affect performance influences of configuration options in various ways (e.g., conditioning influence, introducing variance, having outliers). Yet identifying relevant workload characteristics is highly domain-specific and cannot be considered trivial.

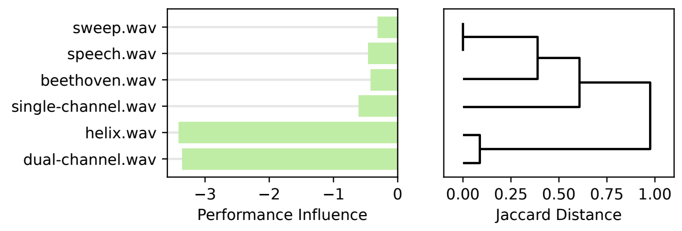
noch was substantielle

C. Relationship of Configuration-Specific Code Coverage and Workload Variation (RQ$_3$)

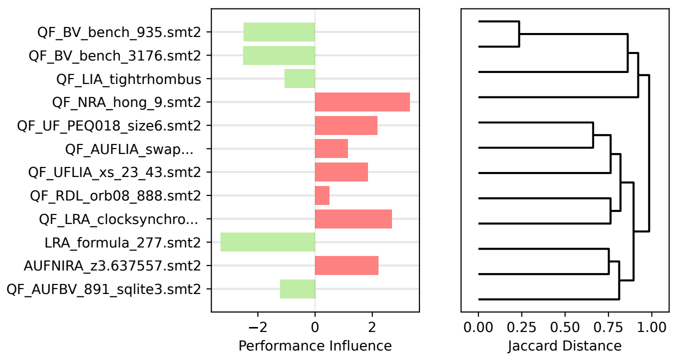
1) Operationalization: To explore whether and to what extent variations in the execution paths (that stem from the interplay of the given workload and configuration) correlate with performance variations, we employ an analysis based on code coverage information. That is, we augment our performance observations 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, we can formulate different hypothetical scenarios explaining performance-relevant 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 corresponding 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 option would be conditioned by the number of transactions resulting from the workload.

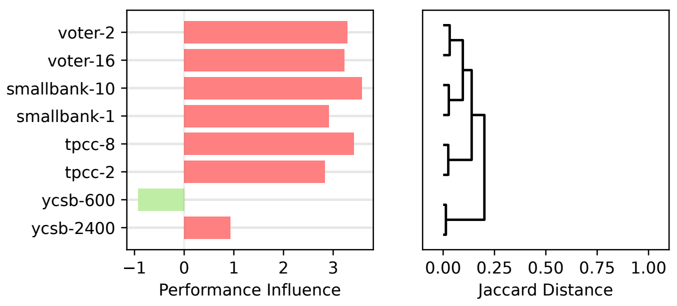
Second, if we observe performance variation across workloads in spite of similar or identical option-specific code coverage, we draw a different picture. In this case, we cannot



(a) Performance influence (left) and option-code coverage clustering for option Mono (JUMP3R).



(b) Performance influence (left) and option-code coverage clustering for option proof (Z3).



(c) Performance influence (left) and option-code coverage clustering for option MVSTORE (H2).

Figure 3: Workload-dependent performance influences of configuration options Mono (JUMP3R), proof (Z3), and MVSTORE (H2) (left sides of each subfigure) and option-code coverage clusterings for the the configuration options (right side of each subfigure).

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 table merge operation in a database system. 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* [52]. While there is a number of approaches based on static [32], [53], [54] and dynamic taint analysis [33], [55], [56], 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, by 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 [57]–[60], but root work on in program comprehension [61]–[65] and fault localization [66], [67]. Specifically, we follow a strategy akin to *spectrum-based feature location* [59]: 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 configuration sample (cf. Section III-B3) into two disjoint sets c_o and $c_{\neg o}$ such that option o is selected only in c_o and not in $c_{\neg_o}$. Next, we select from our code coverage logs the corresponding covered lines of code, S_o and $S_{\neg o}$. The rationale is that all shared lines between both sets are not affected by the selection of an option o. Thus, we compute the *symmetric set 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 consider only execution logs under workload w ($S_{o,w}$ and $S_{\neg o,w}$) and compute the symmetric set difference $\mathbb{S}_{o,w} = S_{o,w} \ \Delta \ S_{\neg o,w}$.

3) Comparing Execution Traces: 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 traces 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 the similarity between the option-code coverage via the Jaccard set similarity index. A Jaccard similarity of zero implies that there is no overlap in the code 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 corresponding distance metric $d_o(v, w) = 1 - sim(v, w)$ 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. The dashboard on the supplementary Web site provides diagrams and inferred feature code for all configuration options. The dendrograms next to the visualizations of performance influences in Figures 3a, 3c, and 3b, respectively, illustrate how similar the covered lines of option-specific code under each workload are. The dendrograms depict the Jaccard similarity clustering, where (from left to right) the split points indicate what Jaccard distance individual sets of lines or subclusters exhibit. The further to the left the point is, the more similar are the constituent parts.

We observe that, in many cases, where a configuration option is ”conditionally influential” (cf. Section IV-B2a), the respective option-specific code under the interacting workloads fall into a cluster, as with the option-specific code for Mono in Figure 3a. 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, the configuration options proof (Z3) and MVSTORE (H2), provide a different picture. Akin to the variation of performance influence of proof, the clustering for this configuration option (cf. Figure 3c) shows that some clusters are disjoint, and, thus, the option-specific code is highly fragmented depending on the workload.

In the same vein, 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.

For cases where we did not detect any differences in code coverage despite substantial differences in an option's performance influence across workloads, our results suggest that the way *how* code was executed (i.e., how frequently methods or loops are executed) is shaping performance.

Summary (RQ_3): Varying the workload can condition the execution of option-specific code coverage and cause performance differences. However, there is no single driving cause of variation: Code utilization depending on workload characteristics is a likely further cause accounting for the majority of variation in the performance influence of an option.

V. Discussion

Our results paint a clear picture of workload-induced performance variations of individual options. This sheds light on the extent of representativeness of singular-workload performance models. But, this is not the end of the story: We saw distinct patterns of complex variations that challenge transfer-learning approaches, which set out to overcome the workload specificity of models. Next, we discuss these points in more detail.

A. *Workload Sensitivity and Singular-Workload Approaches*

The observed workload sensitivity of 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 are sensitive to the workload. So far, the existing body of work on modeling [1]–[9] and optimizing [24]–[27] configuration-specific performance largely neglects the impact of workload variation at the cost of generalizability. Our findings from RQ_2 demonstrate that individual configuration options becoming influential under specific workloads or shifting their influence from beneficial to detrimental (or vice versa) are not uncommon. Revisiting the introductory example from Figure 1, our study provides evidence for performance variation that is unaccounted for. This poses the risk of distorting performance estimations.

unklar, kann wohl weg

Beyond performance estimation, using performance models as surrogates for finding configurations with optimal performance properties is not without risk. For instance, there are several approaches utilizing the rank or importance of options [25], [27]. Given the observed input sensitivity, such rankings remain susceptible to the choice of workload.

Insight: Workload sensitivity challenges the robustness and generalizability of singular-workload performance models, yet is neglected in state-of-the-art approaches. Even more, robust techniques using only rankings or relative importance of options are inapplicable for certain workload variations.

B. *Adressing Workload Variations*

In Section II-C0a, we have outlined the existing body of work that aims at incorporating workload variations into performance modeling [18]–[21]. In spite of the effectiveness of individual approaches, our results raise questions about assumptions and methods used for transfer learning [19], [20] in this setting.

1) Transfer Learning: In their exploratory analysis, Jamshidi et al. reuse existing performance models by learning a linear transfer function across workloads [20]. Our results from RQ_1 have shown non-monotonous performance relationships across workloads, which is 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 advanced machine learning techniques.

The more recent transfer learning approach *Learning to Sample* [19] improves over the prior exploratory work by Jamshidi et al. [20]. It operates under the assumption that sampling for a new context, such as workloads, should focus on the influential options and interactions from a previously trained performance model. While this approach has shown to be effective, our results from RQ_2 contradict the basic assumption of stable influential options. To illustrate this in the context of our study, we select a pair of workloads for each of the nine subject systems studied and compare the ranking of configuration options with regard to their absolute

Table IV: Common top five influential configuration options among pairs of workloads.

Subject System	**Workload 1**	**Workload 2**	**# Common Top 5 Options**
JUMP3R	helix.wav	sweep.wav	3
KANZI	vmlinux	fannie_mae_500k	1
DCONVERT	jpeg-small	svg-large	1
H2	tpcc-2	tpcc-8	3
BATIK	village	cubus	4
XZ	deepfield	silesia	4
LRZIP	artificl	uiq-32-bin	3
X264	sd_crew_cif_short	sd_city_4cif_short	5
Z3	QF_NRA_hong_9	QF_BV_bench_935	3

performance influence (cf. RQ_2). In Table IV, we show for each pair, how many configuration options are ranked in the top five (most influential) and shared across both workloads. Only for X264, the top five ranking is identical. For the other workload pairs, we see that the rankings are inconsistent and, thus, not a reliable heuristic for transfer learning.

As the performance influence of configuration options can be conditioned by workload characteristics, a more appropriate metric to guide sampling would be to assess, which configuration are input-sensitive rather than focusing on influential ones. This reiterates the problems described for most kinds of performance prediction approaches above.

2) Workload-aware and Configuration-aware Performance Modeling: While there is little body of work that *explicitly* considers the impact of factors beside configuration options on performance [18], our results from RQ_2 support the domain- or application-specific performance modeling. For instance, for several configuration options of JUMP3R, we can confidently associate input sensitivity with a workload characteristic. To abstract more from application-specific approaches, a notion of input sensitivity as a form of uncertainty is a promising avenue for further work. Recent work on using probabilisitc programming to learn performance prediction models [1] could be adapted to encode input sensitivity in a similar way.

Insight: Applying transfer learning to adapt performance models to new workloads must lift the assumption of stability of influential options. Domain-specific and workload-aware approaches are promising and should be extended on.

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 address 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. Our 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 [42] ran on the same machine as the database since we were testing an embedded scenario.

Threats to *external validity* include the selection of subject systems and workloads. To ensure generalizability, we select software systems from various application domains as well as two different programming language ecosystems (cf. Table I). To increase the representativeness of our workloads, we vary relevant characteristics (e.g., input types, input sizes, and problem classes) and, where possible, reuse workloads across subject systems of the same domain. Although there might be additional workload characteristics, our results demonstrate already for this 0selection severe consequences for existing performance modeling approaches. So, further variations could only enhance our message.

Threats to *statistical conclusion validity* include our choice of correlation metrics in Section IV-A. The correlation metrics are widely used to describe statistical relationships provide an overview of the effect of workload on the population of tested configurations.

warum?

VII. Conclusion

Configuration options are a key mechanism for optimizing the performance of modern software systems. State-of-the-art approaches of modeling configuration-dependent software performance yet often ignore performance variation caused by changes in the workload. So far, there is no *systematic* assessment of whether, and if so, to what extent can workload variations render singular-workload approaches inaccurate. We have conducted an empirical study of 25 258 configurations across nine 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 similarities of executed code of different workloads compared against performance variations.

more findings in die conclusion

We find that workload variations affect software performance not only at the system-level, but also the influence of individual configuration options, often in a non-monotonous way. We critically reflect on prevalent patterns we found in our subject systems and aim at raising awareness to the missing notion of input sensitivity in existing approaches in this area. Our study provides an empirical basis that questions the practicality and generalizability of state-of-the-art approaches as well as the validity of assumptions under which existing transfer learning approaches operate.

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] J. Cheng, C. Gao, and Z. Zheng, "HINNPerf: Hierarchical Interaction Neural Network for Performance Prediction of Configurable Systems," *ACM Transactions on Software Engineering and Methodology*, 2022, to appear.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

[24] Chen, Tao and Li, Miqing, "Multi-Objectivizing Software Configuration Tuning," 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, 2021, pp. 453–465.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel, "A comparison of 10 sampling algorithms for configurable systems," in *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 2016, p. 643–654.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

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

[35] 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.

[36] 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.

[37] 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.

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

[39] 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.

[40] 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.

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

[42] 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.

[43] 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.

[44] 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.

[45] 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.

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

[47] 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.

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

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

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

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

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

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

[61] 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.

[62] 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.

[63] 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.

[64] 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.

[65] 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.

[66] 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.

[67] 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.