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

Stefan Mühlbauer  
Leipzig University

Florian Sattler  
Saarland University  
Saarland Informatics Campus

Christian Kaltenecker  
Saarland University  
Saarland Informatics Campus

Johannes Dorn  
Leipzig University

Sven Apel  
Saarland University  
Saarland Informatics Campus

Norbert Siegmund  
Leipzig University  
ScaDS.AI Dresden/Leipzig

Abstract—Modern software systems often exhibit numerous configuration options to tailor them to user requirements, including the system’s performance behavior. Performance models derived via machine learning are an established approach for estimating and optimizing configuration-dependent software performance. Most existing approaches in this area rely on software performance measurements conducted with a single workload (i.e., input fed to a system). This single workload, however, is often not 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. Yet, so far, this aspect has not been systematically studied.

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

Our results demonstrate that workloads can induce substantial performance variation and interact with configuration options, often in non-monotonic ways. This limits not only the generalizability of single-workload models, but also challenges assumptions for existing transfer-learning techniques. As a result, 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 enabling desired functionality or tweaking non-functional aspects, such as performance or energy consumption. The relationship between 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 single-workload measurements that aim at reflecting performance behavior of 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 [11], as has been shown for the domains of SAT solving [12], [13], compilation [14], [15], video transcoding [16], [17], data compression [18], and code verification [19]. Beside apparent interactions, such as performance scaling with the size of a workload, qualitative aspects can result in intricate and inadvertent performance variations.

Take as an example two performance throughput distributions across the configuration space of the database system H2 in Figure 1. Here, the exact same configurations on two different parameterizations of the benchmark TPC-C have been measured. In this setting, the scale factor controls the modeled number of warehouses.

While for most configurations, throughput decreases when the scale factor is increased some configurations achieve even a higher throughput. This example illustrates that configuration-dependent performance can be highly sensitive to workload variation and that the performance behavior under different workloads can change in unexpected ways. In turn, this can render performance models based on a single workload useless, unless the configuration options’ sensitivity to workloads is accounted for.

Figure 1: Throughput distribution of 1954 configurations of the database system H2 for the TPC-C benchmark at two different scale factors.

A similar workload-specific performance distribution was described by Pereira et al. for the video encoder x264 [17].
To address this limitation, two different approaches have been pursued in the literature: performance modeling (1) based on existing knowledge [20]–[24], and (2) for a combined configuration-workload problem space [19], [25].

The first approach relies on transfer-learning techniques, in which, given an existing performance model, in a second step only the differences to a new environment or workload are learned. A transfer function encodes which configuration options’ influences on performance are sensitive to workload variation. While transfer learning is an effective strategy that is not limited to varying workloads [20]–[24], 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 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 [19] and high-performance computing [25]. However, the added complexity comes at substantial cost. Not only does it require substantially more measurements, one often lacks 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], [26]–[29] confirms the prevalence and importance of the influence of the workload on the performance of 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 two factors with software configuration options—influence performance.

To summarize, we make the following contributions:

- An empirical study of 25 258 configurations from nine configurable software systems on whether and how interactions of workload and configuration affect software performance;
- A detailed analysis illustrating that (1) system-level performance, and (2) the performance influence of individual configuration options can be sensitive to workload variation and often exhibit a non-monotonous relationship, caused by variation in the execution of option-specific code;
- A critical reflection of the suitability of single-workload models for predicting configuration-dependent performance and assumptions of state-of-the-art transfer-learning approaches in this area;
- An archived repository on zenodo.org with supplementary material, including performance and coverage measurements, configurations, and an interactive dashboard for data exploration to reproduce all analyses and additional visualizations left out due to space limitations.

II. PRELIMINARIES AND RELATED WORK

Software performance emerges from a variety of factors including configuration, workload, and hardware setup. In what follows, we revisit work that models the relationship between such factors (individually or in combinations) and software performance.

A. Performance Prediction Models

Configurable software systems is an umbrella term for any kind of software system that exhibits configuration options to customize its functionality [30]. While the primary purpose of configuration options is to select and tune functionality, each configuration choice may also have implications on non-functional properties (e.g., execution time or memory usage)—be it intentional or not. Performance prediction models approximate non-functional properties, such as execution time or memory usage, as a function of software configurations $c \in C$, formally, $\Pi : C \rightarrow \mathbb{R}$.

Such black-box 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 [26]–[29] and estimating the performance
for arbitrary configurations of the configuration space is an established line of research [1]–[10]. Over the decade, 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 [31], [32]. All sampling techniques aim at yielding a representative sample, either by covering the main effects of configuration options and interactions among them [33] or by sampling uniformly from the configuration space [29], [34]. Most sampling techniques 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 [35], [36] or to model non-functional properties at finer granularity [37], [38].

B. Varying Workloads

When assessing the performance of a software system, one asks how well a certain operation is executed, or, phrased differently, how well an input fed to the software system is processed. In the context of this study, we refer to such inputs as workloads. 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 sequence 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).

In practice, a useful workload for assessing performance should closely resemble the real-world scenario that the system under test will be deployed in. To achieve this, a well-defined and widely employed technique in performance engineering is workload characterization [39], [40]. 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 [41] or by increasing the workload coverage using a mix of different workloads rather than a single one [42].

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 and across different domains, respectively.

C. Workloads and Performance Prediction

Different approaches have been proposed to tackle the problem of workload sensitivity in performance prediction.

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 explicitly 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. This domain-specific strategy has been successfully applied to domains such as program verification [19], algorithm selection [43], or the parametrization of the Java microbenchmark harness [44]. In these instances, the characteristics (varying aspects of a workload) are explicitly specified and do not require further characterization.

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 interactions thereof. In addition, previous work found that only few configuration options are sensitive to workload variation [21]. That is, the problem of identifying meaningful, but sparse predictors is exacerbated since one must not only identify performance-relevant configuration options but also workload-sensitive ones. It is not possible to find such a characterization in every case. Even worse, a chosen characterization can be wrong and omit important factors or overestimate unimportant factors.

At large, the notion of the influence of workloads on configuration-dependent performance remains the exception in the literature: While a study related to ours explores and confirms the presence of interactions between the workload and configuration options [45], only few researchers even consider this dimension of the problem space.

b) Transfer Learning for Performance Models: Another strategy for workload-aware performance prediction builds on the fact that, across different workloads, only few configuration options are in fact workload sensitive [21]. One first trains a model on a standard workload and, subsequently, adapts it to a different 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 on 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 [20], [23], [24], [46]. 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 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 workload sensitivity, to the best of our knowledge, there is no systematic assessment of the factors that drive the interaction between configuration and workload 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 are concerned with the workload sensitivity of the studied software systems’ performance behavior. We first take a look at the entire system (RQ1) and its configurations and, subsequently, to individual configuration options (RQ2). In Sec. V, we explore possible driving factors and indicators for workload-specific performance variation of configuration options (RQ3).

1) Performance Variation Across Workloads: Performance variation may arise from workload variation [11]. 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 how the degree of similarity of workloads and corresponding performance behaviors varies across the configuration space provides insights into the extent the strategies of transferring performance models (outlined in Section II-C) might be applicable. To this end, we formulate the following research question:

RQ1: 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, in general, and which are workload sensitive, in particular, we state the following research question:

RQ2: To what extent do influences of individual configuration options depend on the workload?

B. Experiment Setup

1) Subject System Selection: We have 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 [17], [35], [37], and we 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 workload fed to them. Our study treats execution time as the key performance indicator with the exception of h2, for which we consider throughput.

2) Workload Selection: Our 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 Section VII.

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. We selected a number of different audio files from the Wikimedia Commons collection\(^3\) and varied 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 sizes.

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

---

Table I: Subject System Characteristics

<table>
<thead>
<tr>
<th>System</th>
<th>Lang.</th>
<th>Domain</th>
<th>Version</th>
<th># O</th>
<th># C</th>
<th># W</th>
</tr>
</thead>
<tbody>
<tr>
<td>JUMP3R</td>
<td>Java</td>
<td>Audio Encoder</td>
<td>1.0.4</td>
<td>16</td>
<td>4196</td>
<td>6</td>
</tr>
<tr>
<td>KANZI</td>
<td>Java</td>
<td>File Compressor</td>
<td>1.9</td>
<td>24</td>
<td>4112</td>
<td>9</td>
</tr>
<tr>
<td>DCONVERT</td>
<td>Java</td>
<td>Image Scaling</td>
<td>1.0.0-a7</td>
<td>18</td>
<td>6764</td>
<td>12</td>
</tr>
<tr>
<td>h2</td>
<td>Java</td>
<td>Database</td>
<td>1.4.200</td>
<td>16</td>
<td>1954</td>
<td>8</td>
</tr>
<tr>
<td>BATIK</td>
<td>Java</td>
<td>SVG Rasterizer</td>
<td>1.14</td>
<td>10</td>
<td>1919</td>
<td>11</td>
</tr>
<tr>
<td>XZ</td>
<td>C/C++</td>
<td>File Compressor</td>
<td>5.2.0</td>
<td>33</td>
<td>1999</td>
<td>13</td>
</tr>
<tr>
<td>LRZIP</td>
<td>C/C++</td>
<td>File Compressor</td>
<td>0.651</td>
<td>11</td>
<td>190</td>
<td>13</td>
</tr>
<tr>
<td>x264</td>
<td>C/C++</td>
<td>Video Encoder</td>
<td>bace400...</td>
<td>25</td>
<td>3113</td>
<td>9</td>
</tr>
<tr>
<td>Z3</td>
<td>C/C++</td>
<td>SMT Solver</td>
<td>4.8.14</td>
<td>12</td>
<td>1011</td>
<td>12</td>
</tr>
</tbody>
</table>

# O: No. of options, # C: No. of configurations, # W: No. of workloads tested
We split the performance measurement and coverage analysis which intercepts byte code running on the JVM at run-time. For the embedded database h2, we used a selection of four benchmarks (SmallBank, TPC-H, YCSB, Voter) from OLTPBench [47], 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.

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 [48], 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 [48]. 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 distribution, we used distance-based sampling [34]. If a software system exhibited numeric configuration options, we varied them across, at least, two levels to measure 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], 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 [8] to collect coverage information. We split the performance measurement and coverage analysis runs to avoid distortion from the profiling overhead.

5) Measurements: We conducted all experiments on three different compute clusters, where all machines within a compute cluster had an 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 and BATIK on cluster B; the remaining systems on cluster C.

We collect performance data using the tools GNU TIME (execution time) and OLTPBENCH (throughput). For all data points, we report the median performance across five repetitions (except for H2), which has shown to be a good trade-off between variance and measurement effort [49]. Across these repetitions, most configurations exhibit only little variation (e.g., only a few seconds for whole-system benchmarks which run for several minutes): The ratio of configurations with a coefficient of variation (standard deviation divided by the mean) of less than 10% ranges from 91% (LRZIP) to 99% (X264). 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

In this section, we present the results of our empirical study with regard to variation of system-level performance distributions (RQ1) and the performance influence of individual configuration options (RQ2).

A. Comparing Performance Distributions (RQ1)

1) Operationalization: We answer RQ1 by pairwise 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 combinations of workload-specific performance observations with the Wilcoxon signed-rank test [9] [52]. We rejected the null hypothesis $H_0$ at $\alpha = 0.95$. To account for over powering 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. [53], for no combination, Cliff’s $d$ [54] exceeded a threshold effect size of $|d| > 0.147$. 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) Monotonous Correlation: We test whether there is a monotonous relationship between the two performance distributions. We use Kendall’s rank correlation coefficient $\tau$.
Table II: Three disjoint categories and criteria of relationships between pairs of workload-specific performance distributions.

<table>
<thead>
<tr>
<th>Category</th>
<th>Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>LT Linear</td>
<td>$r^* \geq 0.6$</td>
</tr>
<tr>
<td>XMT Monotonous</td>
<td>$r^* &lt; 0.6$ and $r^* \geq 0.6$</td>
</tr>
<tr>
<td>NMT Non-monotonous</td>
<td>(otherwise)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>System</th>
<th>$\Sigma_{pairs}$</th>
<th>LT abs rel</th>
<th>XMT abs rel</th>
<th>NMT abs rel</th>
</tr>
</thead>
<tbody>
<tr>
<td>JUMP3R</td>
<td>15</td>
<td>15 100.0%</td>
<td>0 0%</td>
<td>0 0%</td>
</tr>
<tr>
<td>KANZI</td>
<td>36</td>
<td>28 77.8%</td>
<td>4 11.1%</td>
<td>4 11.1%</td>
</tr>
<tr>
<td>DCONVERT</td>
<td>66</td>
<td>29 43.9%</td>
<td>0 0%</td>
<td>37 56.1%</td>
</tr>
<tr>
<td>H2</td>
<td>28</td>
<td>13 46.4%</td>
<td>0 0%</td>
<td>15 53.6%</td>
</tr>
<tr>
<td>BATIK</td>
<td>55</td>
<td>28 50.9%</td>
<td>8 14.6%</td>
<td>19 34.6%</td>
</tr>
<tr>
<td>XZ</td>
<td>78</td>
<td>65 83.3%</td>
<td>1 1.3%</td>
<td>12 15.4%</td>
</tr>
<tr>
<td>LRZIP</td>
<td>78</td>
<td>57 73.0%</td>
<td>13 16.7%</td>
<td>8 10.3%</td>
</tr>
<tr>
<td>x264</td>
<td>36</td>
<td>36 100.0%</td>
<td>0 0%</td>
<td>0 0%</td>
</tr>
<tr>
<td>Z3</td>
<td>66</td>
<td>10 15.2%</td>
<td>1 1.5%</td>
<td>55 83.3%</td>
</tr>
</tbody>
</table>

$\tau$ [55] 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 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 III.

2) Results: We list 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). In particular, for JUMP3R and x264, we observe solely such behavior. The presence of linear transformations corroborates experimental insights from Jamshidi et al., who encoded differences between performance distributions using linear functions [21].

Exclusively monotonous transformations (XMT) are the exception and are found only in 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 differing prevalence. For three systems (DCONVERT, H2, and Z3), the majority of transformations required is non-monotonous; for the other four systems (KANZI, BATIK, XZ, LRZIP), more than 10% of workload pairs fall into this category.

Summary (RQ1): 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 (RQ2)

1) Operationalization: To address RQ2, 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]. 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 RQ1, 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 (LT). 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 arises when features are correlated [56]. It can, for instance, be caused 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 when learning a model. These measures reduced multicollinearity to a negligible degree [57]. After these corrections, we observed no configuration options exceeding a variance inflation factor (indicating multicollinearity) of 5.

2) Results: Our results show a wide variety of degrees 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 workload sensitivity that we observed. An 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 results 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). Each vertical bar depicts the relative performance influence under a specific workload, the colored ranges depict positive (green) and negative (red) performance influences. 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 of the MP3 encoder JUMP3R. We illustrate the performance influence of this option across six workloads presented as bars in the last row of bars in Figure 3a. Selecting this option reduced the execution time substantially for two workloads, whereas, for the other workloads, the effect was far smaller.

According to the documentation of JUMP3R, 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 with reduced execution time 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 (row 3). 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 the case a problem instance is unsatisfiable. Each workload in our selection contains multiple problem 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 scaling anomaly pattern is shown for the configuration option MVSTORE of the database system H2 in Figure 3c (left). This 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. Using the Yahoo! Cloud Serving Benchmark (YCSB) with two different scale factors (which control the workload complexity, expressed as number of rows), we found that the lower complexity parameterization (YCSB-600) resulted in lower throughput. This is in stark contrast to

Figure 2: Distribution of configuration options’ performance influences under different workloads for JUMP3R.

Figure 2: Distribution of configuration options’ performance influences under different workloads for Z3.

Figure 2: Distribution of configuration options’ performance influences under different workloads for H2.

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.

\[ \text{JUMP3R: https://github.com/sciss/jump3r/blob/master/README.md} \]
Summary (RQ$_3$): Workloads can affect performance influences of configuration options in various ways (e.g., conditioning influence, introducing variance, having outliers). We can correlate some variation of performance influences with workload characteristics, yet identifying relevant workload characteristics is highly domain-specific and cannot be considered trivial.

V. EXPLORING CAUSES OF WORKLOAD SENSITIVITY

The results for the first two research questions demonstrate sensitivity of the influence of configuration options to the workload due to non-monotonous interactions, which is consistent with the findings of a related study [45]. Before we discuss implications for performance modeling in the Section VI, we investigate the underlying factors that drive workload sensitivity.

We hypothesize that executions under different workloads also exhibit variation with respect to what code sections are executed (and which are not) and how this code is used (e.g., number of method invocations or loop iterations). 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 do so with varying frequency. To investigate to what extent one could infer or even explain performance variations based on different code execution profiles, we apply standard code-coverage analyses under different configurations and workloads. Our research question is as follows:

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

Exploiting a potential relationship between workload sensitivity of configuration options and differences in software execution could be beneficial. Instead of testing various combinations of configurations and workloads, code analysis can serve as a cost-effective way to detect workload sensitivity of individual options by identifying workload-specific differences in program execution.

A. Operationalization

To explore whether and to what extent performance variations correlate with variations in the execution paths (that stem from the interplay of the given workload and configuration), we employ an analysis based on code coverage information (cf. Sec. III-B4).

We combine performance observations with code coverage data to evaluate the execution under different workloads, specifically focusing on code sections implementing option-specific functionality. By comparing the coverage of this code, we can develop hypothetical scenarios explaining workload 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 influential only under the 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 workload-specific number of transactions.

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 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. In a simple case, a software system’s performance may scale linearly with the workload 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).

1) 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 [58]. While there is a number of approaches based on static [35], [59], [60] and dynamic analysis [36], [61], [62], we employ a more light-weight, but also less precise approach, that uses code coverage information. 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 [63]–[66], but root in work on program comprehension [67]–[71] and fault localization [72], [73].

Specifically, we follow a strategy akin to spectrum-based feature location [65]: 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 $c_0$ and $c_{\sim o}$ such that option $o$ is selected only in $c_0$ and not in $c_{\sim o}$. Next, we select from our code coverage logs the corresponding covered lines of code, $S_o$ and $S_{\sim o}$. The rationale is that all shared lines between both sets are not affected by the selection of option $o$. Thus, we compute the symmetric set difference $S_o = S_o \Delta S_{\sim o}$ 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 \( S_{o,o,w} \)) and compute the symmetric set difference \( S_{o,w} = S_{o,w} \Delta S_{o,o,w} \).

2) 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 \( S_{o,v} \) and \( S_{o,w} \) for any two workloads \( v \) and \( w \), we can estimate the similarity between the option-code coverage with 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 \( S_{o,v}(v, w) \), we can compute a corresponding distance metric \( d_{o,v}(v, w) = 1 - S_{o,v}(v, w) \) 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 subclusters into larger ones depending on their Jaccard similarity to each other.

B. Results

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 workload 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 are under each workload. The dendrograms depict the Jaccard similarity clustering, where the split points indicate what Jaccard distance individual sets of lines or subclusters exhibit. The farther to the left the point is, the more similar are the components.

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 workload sensitivity.

The other two examples, the configuration options proof (3.3) 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.
VI. DISCUSSION

Our results draw a clear picture of workload-induced performance variations of individual options. This sheds light on the extent of representativeness of single-workload performance models. But, this is not the end of the story: We saw complex variations that challenge transfer-learning approaches, which aim to overcome the workload specificity of models.

A. Workload Sensitivity and Single-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 [26]–[29] configuration-specific performance largely neglects the impact of workload variation at the cost of generalizability. Our findings from RQ2 demonstrate that unexpected interactions of configuration options with the workload are not uncommon, which can distort performance estimations.

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 [27], [29]. Given the observed workload sensitivity, such rankings remain susceptible to the choice of workload.

Insight: Workload sensitivity challenges the robustness and generalizability of single-workload performance models, yet it is neglected in state-of-the-art approaches. Worse, 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 [19]–[22]. Despite the effectiveness of individual approaches, our results raise questions about assumptions used for transfer learning [20], [21] 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 [21]. Our results from RQ1 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 motivates employing more advanced machine learning techniques.

The more recent transfer-learning approach Learning to Sample [20] improves over the prior exploratory work by Jamshidi et al. [21]. 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 RQ2 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 performance influence (cf. RQ2). 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. For these 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 workload 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 work that explicitly considers the impact of factors beside configuration options on performance [19], our results from RQ2 support idea of domain- or application-specific performance modeling. For instance, for several configuration options of JUMP3R, we can confidently associate workload sensitivity with a workload characteristic. To abstract more from application-specific approaches, a notion of workload sensitivity as a form of uncertainty is a promising avenue for further work. Work on using probabilistic programming to learn performance models [1] could be adapted to encode workload sensitivity.

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

3) Identifying Workload Sensitivity via Code Analysis: Our findings from RQ3 show that it is possible to identify workload

<table>
<thead>
<tr>
<th>System</th>
<th>Workload 1</th>
<th>Workload 2</th>
<th># Common</th>
</tr>
</thead>
<tbody>
<tr>
<td>JUMP3R</td>
<td>helix.wav</td>
<td>sweep.wav</td>
<td>2</td>
</tr>
<tr>
<td>KANZI</td>
<td>vmlinux</td>
<td>fannie_mae_500k</td>
<td>1</td>
</tr>
<tr>
<td>DCONVERT</td>
<td>jpeg-small</td>
<td>svg-large</td>
<td>2</td>
</tr>
<tr>
<td>H2</td>
<td>tpcc-2</td>
<td>tpcc-8</td>
<td>3</td>
</tr>
<tr>
<td>BATIK</td>
<td>village</td>
<td>cubus</td>
<td>4</td>
</tr>
<tr>
<td>XZ</td>
<td>deepfield</td>
<td>silesia</td>
<td>4</td>
</tr>
<tr>
<td>LRZIP</td>
<td>artifcil</td>
<td>uig-32-bin</td>
<td>3</td>
</tr>
<tr>
<td>X264</td>
<td>sd_crew_cif_short</td>
<td>sd_city_4cif_short</td>
<td>4</td>
</tr>
<tr>
<td>Z3</td>
<td>QF_NRA_hong_9</td>
<td>QF_BY_bench_935</td>
<td>3</td>
</tr>
</tbody>
</table>

Table IV: Common top five influential configuration options among pairs of workloads.
sensitivity through code analysis. This can be done using systematic coverage testing, which can be easily incorporated into CI/CD pipelines along with other code analyses, such as hit counts. While this low-cost metric can enhance existing approaches and help to interpret and contextualize performance estimations, it is important to note that more detailed analyses may be required to fully explain all performance variation. These findings can be applied in practice, for example, by using code coverage data to estimate up-front whether an option is input-sensitive and annotating existing performance models with a usage score per option. These results are important for understanding the performance of configurable software systems and for designing effective benchmarks.

**Insight:** Code analysis can be used to identify workload sensitivity and inform benchmark design in configurable software systems, but it is important to consider the limitations of this approach.

**VII. Threats to Validity**

Threats to internal validity include the presence of 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 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 increase 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 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 selection severe consequences for existing performance modeling approaches. So, further variations could only strengthen our message.

**VIII. Conclusion**

Configuration options are a key mechanism for optimizing the performance of modern software systems. Yet, state-of-the-art approaches of modeling configuration-dependent software performance largely ignore performance variation caused by changes in the workload. So far, there is no systematic assessment of whether, and if so, to what extent workload variations can render single-workload approaches inaccurate. We have conducted an empirical study of 25,258 configurations from nine configurable software systems to characterize the effects that varying workloads can have on configuration-specific performance. We compare performance measurements with coverage data to identify possible similarities of executed code of different workloads compared to performance variations.

We find that workload variations affect software performance not only at the system-level, but also affect the influence of individual configuration options on performance, often in a non-monotonic way. While in some cases, we can correlate performance variations with the workload-conditioned execution of option-specific code, workload characteristics influence the utilization of option-specific code in further non-trivial ways (e.g., number of method calls).

We critically reflect on prevalent patterns, that we found in our subject systems and aim at raising awareness to the missing notion of workload sensitivity in existing approaches in this area. Our study provides an empirical basis that questions the practicality and generalizability of existing single-workload approaches as well as the validity of assumptions under which existing transfer-learning approaches in this area operate.

**IX. Acknowledgements**

We thank our reviewers for their thoughtful and constructive comments. Apel’s work has been funded by the German Research Foundation (DFG) under contract AP 206/11-2 and grant 389792660 as part of TRR 248 – CPEC. Siegmund’s work has been supported by the German Research Foundation (DFG) under the contract SI 2171/2-2 and by the German Ministry of Education and Research (BMBF) and State Ministry for Science and Cultural Affairs of Saxony (SMWK) in the program Center of Excellence in AI research “Center for Scalable Data Analytics and Artificial Intelligence Dresden/Leipzig”, project identification ScaDS.AI.

**REFERENCES**


