Identifying Software Performance Changes Across Variants and Versions

Stefan Mühlbauer
Leipzig University,
Germany

Sven Apel
Saarland University,
Germany

Norbert Siegmund
Leipzig University,
Germany

ABSTRACT

We address the problem of identifying performance changes in the evolution of configurable software systems. Finding optimal configurations and configuration options that influence performance is already difficult, but in the light of software evolution, configuration-dependent performance changes may lurk in a potentially large number of different versions of the system.

In this work, we combine two perspectives—variability and time—into a novel perspective. We propose an approach to identify configuration-dependent performance changes retrospectively across the software variants and versions of a software system. In a nutshell, we iteratively sample pairs of configurations and versions and measure the respective performance, which we use to update a model of likelihoods for performance changes. Pursuing a search strategy with the goal of measuring selectively and incrementally further pairs, we increase the accuracy of identified change points related to configuration options and interactions.

We have conducted a number of experiments both on controlled synthetic datasets as well as in real-world scenarios with different software systems. Our evaluation demonstrates that we can pinpoint performance shifts to individual configuration options and interactions as well as commits introducing change points with high accuracy and at scale. Experiments on three real-world systems explore the effectiveness and practicality of our approach.

CCS CONCEPTS

• **Software and its engineering** → **Software performance**; **Software evolution**; • **Computing methodologies** → **Active learning settings**.

KEYWORDS

software performance, software evolution, configurable software systems, machine learning, active learning

ACM Reference Format:
Stefan Mühlbauer, Sven Apel, and Norbert Siegmund. 2020. Identifying Software Performance Changes Across Variants and Versions. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3324884.3416573

ASE '20, September 21–25, 2020, Virtual Event, Australia

ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3416573

1 INTRODUCTION

Software performance plays a crucial role in users' perception of software quality. Excessive execution times, low throughput, or otherwise unexpected performance without added value can render software systems unusable. Poor performance is often a symptom of deficiencies in particular software components or the overall software architecture. Changes in the observed performance of a software system can be attributed to changes to the software at different levels of granularity (architecture, code, etc.).

Typically, modern software systems provide configuration options to enable users and admins customizing behavior to meet different user requirements. Configuration options usually correspond to pieces of selectable functionality (features), which contribute to overall performance with different proportions. That is, different configurations of a software system exhibit different performance characteristics depending on configuration decisions made by the user.

Performance changes during software evolution—intended or not—can affect all or only a subset of configurations since changes to a software system often relate to a particular configuration option or set of options. This is why performance bugs are rarely visible in default configurations, but revealed only in certain configurations [8]. If undetected, such perennial bugs can persist for the lifetime of a software system as software evolves. Adding this *technical debt* constantly can accumulate and entail trends of degrading performance quality [5].

Performance assessment and estimation is a resource-intense task with many possible pitfalls prevailing. Best practices for conducting performance measurements emphasize a dedicated and separated hardware setup to prevent measurement bias by side-processes and, subsequently, obtain reproducible results [18]. What is often overlooked is the fact that most software systems are configurable, which introduces another layer of complexity. Due to combinatorics, even for a small number of configuration options, the range of possible valid configurations renders exhaustive measurements infeasible.

Our goal is to enable the *retrospective* detection of changes in the performance of configurable software systems and pinpoint them to a specific option or interaction. For example, a patch of a certain feature is likely to affect only configurations where such feature is selected. We would like to know in which revision this patch was introduced and which features were affected. Many software systems are configurable, but have no performance regression testing routine set in place. To uncover performance deficiencies that emerged from revisions in the past development history, it is infeasible to test each and every commit and configuration. In essence, we face a combinatorial explosion along two dimensions: time (versions) and configuration space (variants). *First*, we aim

at finding changes in software performance from one version to another. The detection of such changes is referred to as *change point detection* [2, 3, 10, 19, 24, 25]. The main limiting factor is that exhaustive measurements across the configuration space are neither available nor feasible [31]. Change point detection techniques with both exhaustive measurement [2, 3] as well as limited data availability [10, 19, 24, 25] have been successfully applied to identify performance changes. *Second*, we aim at associating performance changes (from one version to another) with particular configuration options or interactions among them. There is substantial work regarding a related problem, which is estimating the influence of individual options on performance [7, 26, 28, 29] for a single version of the software system, ignoring the temporal dimension. Instead of estimating the influence of options and interactions on performance, we want to know: Which option or interaction is *responsible* for a particular change point in the version history of a software system?

Our main idea is as follows: We address the configuration complexity of this problem by selecting representative sample sets of configurations. Then, we uniformly sample a constant number of commits for each configuration and conduct respective performance measurements. Based on these measurements, we learn a prediction model. For each configuration, we estimate the likelihood of each commit being a change point (i.e., that performance of some configurations changes abruptly compared to the previous commit). Next, we leverage similarities in the performance histories of configurations that share common options. Since we sample the commits for each configuration independently, we obtain for each change point many estimations using measurements of different commits. Overall, this allows us to obtain more accurate estimations of performance-changing commits with tractable effort. Based on a mapping from configurations to predicted change point probabilities for each configuration and commit, we derive the options responsible for each particular change point, which we call *configuration-dependent change points*. In summary, we offer the following contributions:

- A novel technique to effectively identify shifts in performance of configurable software systems. It is able to pinpoint causative commits and affected configuration options with high accuracy and tractable effort.
- A feasibility demonstration of our approach by implementing an adaptive learning strategy to obtain accurate estimations with acceptable measurement cost.
- An evaluation using both synthetic and real-world performance data from three configurable software systems. Synthetic data let us assess our model and approach conceptually and at scale, whereas we are able to assess practicality with real-world data.
- A companion Web site[1] providing supplementary material including a reference implementation of our approach, performance measurement data, and additional visualizations.

[1]https://github.com/AI-4-SE/Changepoints-Across-Variants-And-Versions/ or an archived version at https://archive.softwareheritage.org/browse/origin/https://github.com/AI-4-SE/Changepoints-Across-Variants-And-Versions/

2 CONFIGURATION-DEPENDENT CHANGE POINTS

Performance as a property emerges from a variety of factors. Besides external factors, including hardware setup or execution environment [22], the configuration of a software system can influence performance to a large extent [29]. Most modern software systems exhibit configuration options that correspond to selectable pieces of functionality. With configuration options, we turn features on and off creating a variant of the software system. Depending on the configuration, different system variants with different behavior and performance can be derived.

2.1 Performance-Influence Models

Consider the following running example of a database management system (DBMS) with two selectable features: Encryption and compression. Either feature adds execution time to the overall performance, but, if both features are selected, the execution time is smaller than the sum of the individual features' contribution to performance: Less data is encrypted if it is compressed beforehand. An *interaction* in this setting is the combined effect of features Encryption and compression. We can assign each feature and interaction an *influence* that it contributes to the overall performance, if selected. In our example, the individual influences of the two features are positive (increasing execution time), whilst the interaction's influence is negative (decreasing execution time).

The influence of features on performance can be described using a *performance-influence model* [28]. A performance-influence model is a linear prediction model of the form $\Pi : C \rightarrow \mathbb{R}$, whereby C denotes a configuration vector (assignment of concrete values to configuration options) and the estimate is a system variant's real-valued performance:

$$\Pi(c) = \beta^T c = \begin{bmatrix} \beta_1 \\ \vdots \\ \beta_{2^{|F|}} \end{bmatrix}^T \cdot \begin{bmatrix} c_1 \\ \vdots \\ c_{2^{|F|}} \end{bmatrix} \qquad \begin{matrix} \beta_i \in \mathbb{R}^{2^{|F|}} \\ c_i \in \{0,1\}^{2^{|F|}} \end{matrix} \tag{1}$$

F denotes the set of all features. Our linear model has $|2^F|$ terms, whereby each term $t \subseteq F$ corresponds to a subset of F and is described by a coefficient β_f and a configuration parameter c_f. Coefficients encode the performance-influence of features and interactions. c is a vector and denotes the assignment of concrete values to configuration options or interactions. A configuration parameter c_t evaluates to 1, if all configuration options of the corresponding subset $t \subseteq F$ are selected, and 0 otherwise. The empty set corresponds to the invariable core functionality of the software system. Singleton subsets correspond to individual features, compound subsets to interactions of features.

For our DBMS example, we present a fully determined performance-influence model in Equation 2. The first two terms correspond to the features Encrpytion and Compression, respectively. The third term respresents the interaction of both features. The configuration parameter $c_{\text{Compr}\wedge\text{Encrypt}}$ evaluates to 1, if both c_{Encrypt} and c_{Compr} evaluate to 1. That is, $c_{\text{Compr}\wedge\text{Encrypt}} \equiv c_{\text{Encrypt}} \cdot c_{\text{Compr}}$. In the last term, we omit the configuration parameter $c_\emptyset$, as this term represents invariable functionality.

$$\Pi_{\text{DBMS}}(c) = \beta_{\text{Encrpyt}} \cdot c_{\text{Encrpyt}} + \beta_{\text{Compr}} \cdot c_{\text{Compr}} + \beta_{\text{Compr} \wedge \text{Encrpyt}} \cdot c_{\text{Compr} \wedge \text{Encrpyt}} + \beta_{\emptyset} \quad (2)$$

Prediction models of configuration-dependent performance of this form are not the only possible representation [4, 7], but linear models are easy to interpret [28]. We review extensions and alternatives to linear performance-influence models in Section 6.

2.2 Evolution of Performance Influences

Performance-influence models describe how features contribute to performance, but they do not allow for understanding configuration-dependent performance changes over time. We expand our DBMS example with a temporal dimension, considering a development history of hundred commits. In particular, we assume the following three events. (1) At commit 25, feature Compression has been modified, increasing the execution time; feature Encryption is introduced at commit 25 and can be combined with Compression. (2) At commit 50, the interaction of Compression and Encryption changes, resulting in an increased execution time. (3) At commit 80, the core functionality of the DBMS is refactored, which decreased execution time for all variants.

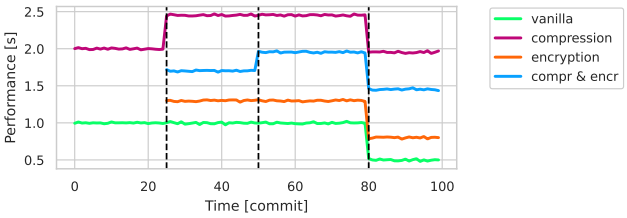


Figure 1: Performance of 4 variants with 3 change points

The performance histories of the four valid variants in Figure 1 exhibit three change points: commits 25, 50, and 80. These change points, however, do not affect all configurations. For instance, execution time decreases for all configurations at commit 80, but increases at commit 50 only for one configuration. Given this example, the target outcomes of our approach are (1) the locations of the three change points (commits 25, 50, and 80), (2) the association of such commits to the features and interactions (Compr, Compr ∧ Encrypt), and the invariable functionality, respectively.

2.3 Taming Complexity

A naive approach for identifying change points in the evolution of configurable software systems is to simply combine existing work on performance modeling of commit histories and performance prediction of software configurations.

For our example, we could measure all variants for each commit building a performance-influence model per commit. Having 4 configurations and 100 commits, this would result in 400 measurements. Since the number of configurations grows exponentially with the number of features, we end up with 2^n times T measurements where n is the number of features and T is the number of commits. Clearly, this does not scale along both dimensions: Already a few features would render even small commit histories intractable,

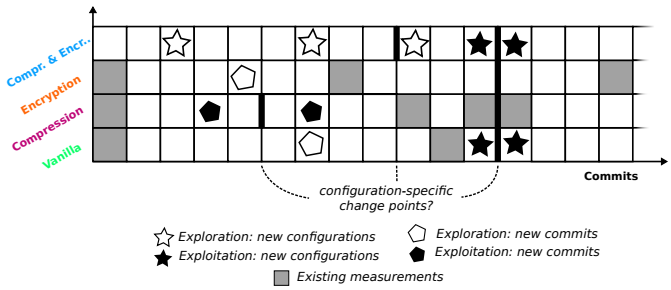


Figure 2: Active sampling strategies for a DBMS with 4 configurations

and even small configuration spaces make this approach infeasible given realistic commit histories of a few thousands of commits.

To obtain accurate estimations with a limited budget of measurements, we require a different approach. We propose an iterative and adaptive sampling approach. That is, we use an initial, but small sample set to explore the problem space and then increase the level of granularity at promising regions and dimensions (development history segments as well as individual features and interactions, in our case).

Based on the introductory DBMS example (cf. Figure 1), we illustrate our sampling strategy in Figure 2. Here, each box depicts a possible measurement (i.e., a pair of a configuration and a commit). Black bars represent change points hidden in the software system's performance histories (i.e., measurements immediately before and after a change point are dissimilar). The current state of the sample set comprises all measurements that are filled in grey.

To include new measurements into the sample set, we consider the following situations. On the one hand, our sample set might already hint to some possible change points and associated configuration options. On the other hand, our sample set might be too sparse, such that we cannot infer some change points yet. To address this trade-off, we devise two strategies, exploration and exploitation, both of which address both configurations and commits.

For *exploitation*, we sample new measurements to verify guesses based on the current state. That is, in Figure 2, we might include the measurements depicted by black-filled symbols (★/⬟). Black-filled stars (★) represent measurements that are of interest because we have already identified one change point for configuration Compression. We exploit this knowledge to test further configuration options involved in this change point. The black-filled pentagons (⬟) represent measurements that are interesting, because we have identified another early change point for Compression, but within a rather broad range of commits. Thus, we include further measurements from that range to narrow down the possible range for this particular change point.

For *exploration*, we might include the measurements denoted by white symbols (☆/⬠). White stars (☆) represent measurements that explore new configurations. In our example, the interaction between Compression and Encryption is the only configuration left to be measured. We include several commits of a particular new configuration to unveil possible performance variation that indicates possible change points. White pentagons (⬠) represent

measurements that are of interest because they increase the overall measurement coverage. We include measurements from large intervals of not yet measured commits, since possible change points can be hidden there.

These four strategies (exploration and exploitation of configurations and commits) prioritize measurements to be included next into the sample set. Adaptive sampling techniques have been successfully applied to obtain both performance-influence models [26, 29] and performance histories [19] before. However, it is unclear whether fewer measurements are sufficient to assess the performance-influence of configuration options and interactions with respect to version changes. This is what we address in this work.

3 AN ALGORITHM FOR CHANGE POINT DETECTION

We propose an algorithm to detect substantial shifts in the performance of software configurations and associate them to individual commits and options or interactions. We lay out our approach as an iterative search across the commit history and configuration space. We provide an overview of our approach in Figure 3. It starts with a small initial sample set of measurements (i.e., performance observations of varying configurations and varying commits). Based on this sample set, it calculates for each configuration the likelihood of each commit being a change point.

Subsequently, our algorithm estimates a *candidate solution*, which is a set of pairs of a commit and a configuration option. Each of such pairs describes the estimated involvement of a configuration option in the shift of performance. Interactions are conceived as multiple tuples with identical commits. That is, if a commit occurs in multiple tuples, each with different configuration options, this indicates that the shift arises from an interaction among two or more options. Henceforth, we will refer to such pairs as *associations*.

After obtaining one candidate solution per search iteration, we augment the sample set of measurements with regard to two objectives: exploration and exploitation. *Exploration* aims at including previously unseen commits and configurations to improve coverage of the search space. *Exploitation* aims at including measurements in the sample set that, based on the previous candidate solution, may increase confidence in associations or rule out false positives (i.e., commits falsely identified as change points or options falsely associated with a commit). As for exploitation, we make an informed decision of which measurements are to be included, whereas exploration is agnostic of previous candidate solutions.

As each iteration yields a candidate solution, we keep track of associations in a *solution cache* and repeatedly update the confidence of associations. The rationale is not to lose previously identified change points, but at the same time, allow for removing identified changes points that are likely false positives due to new measurements. Some associations, especially in the beginning, might be influenced by sampling bias and can be removed if successive iterations do not repeatedly revisit these associations. The algorithm terminates if the solution cache does not change for a number of iterations or a maximum number of iterations/measurements has been reached. In what follo, we present the steps of our approach in detail.

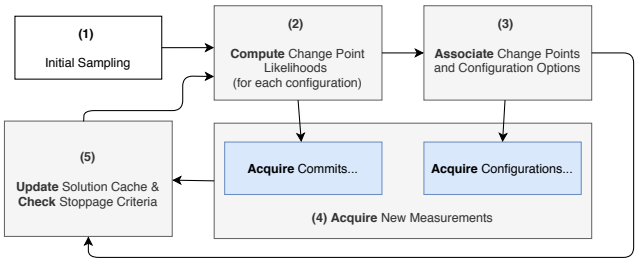


Figure 3: Overview of our approach: ① Selecting training data across configuration space and version history, ② estimating the change point probability distribution per configuration, ③ estimating change points as candidate solutions, ④ adaptively augmenting the training set, and ⑤ updating the solution cache and evaluating termination criteria.

3.1 Initialization and Sampling

The first step of our approach is to select a sample set of performance measurements. We select a relatively small, but fixed number of configurations, $n_{\text{configurations}}$. For each configuration, we select a fixed percentage of commits, r_{commits}, and assess their respective performance. The initial sample set in this setup is kept small and may not represent all relationships between configuration options and performance evolution. The rationale is that, to make our approach more scalable, we explore the combined search space and refine the temporal and spatial (i.e., configuration-related) resolution where necessary.

For the initial configuration sampling, we use distance-based sampling [14], which is a form of random sampling that strives for uniform coverage at low cost. In general, uniform random sampling of configurations is considered to yield the most representative coverage of a configuration space, but it is prohibitively expensive for real-world configurable software systems with constraints among options (i.e., not all combinations of configuration options are valid configurations). Distance-based sampling addresses this problem by demanding the number of selected options to be uniformly distributed to avoid local concentration.

The key idea of our algorithm to iteratively augment the training set addresses two issues: (1) The configuration space exhibits exponential complexity. (2) Interactions of higher degrees (i.e., interactions involving two or more configuration options) are possible, but relatively rare among configurable software systems [15, 16]. Therefore, instead of exhaustive sampling with respect interaction degrees, we iteratively add new configurations to our training sample to search previously undetected influences of options and interactions.

For each configuration in our sample set, the algorithm selects a small number of commits (e.g., one or two percent of all commits) for which it measures performance. The rationale of having only few commits is that, given a relatively large number of configurations, many similar configurations will exhibit change points of the same cause. We mitigate the poor temporal resolution by selecting the commits independently. Compared to a fixed sample of commits across all configurations, this way, each commit is more likely to be measured, at least, once. That is, we obtain change-point estimations for related configurations from independent training

samples. For instance, consider the third change point in the introductory DBMS example: At commit 80, all configurations exhibit a performance change. If we sampled two commits, 70 and 90, for all configurations, all that we would learn is that there is a change point somewhere between these two versions. Instead, our approach samples the commits 70 and 90 for the first two configurations, and commits 75 and 95, as well as 65 and 85 for the remaining two configurations respectively. Our best guess then is to assume a change point between commits 75 and 85 since all measurements agree with this conclusion. This way, we increase the temporal resolution while keeping the overall number of performance measurements manageable.

3.2 Iteration: Change Point Likelihoods

For each configuration in our sample set, we estimate the probability of each commit being a change point for the corresponding configuration. To this end, we need to define what counts as a performance change. We use a user-defined threshold, which discriminates between measurement noise and performance changes such that different application scenarios as well as system-specific peculiarities can be accounted for. If the performance difference for a configuration between two commits exceeds this threshold, we count this difference as a performance change. Although manually defined, there are several possibilities to estimate this threshold automatically. Prior to learning, the measurement variation obtained by the repeating measurements for the same configuration multiple times can be estimated and employed as a minimum threshold. In addition, a relative or absolute threshold can be derived from the application context, such as a ten percent or ten seconds increase in execution time.

We encode the threshold in a step function θ_τ:

$$\theta_\tau(a,b) = \begin{cases} 0, & |\pi(a) - \pi(b)| < \tau \\ 1, & |\pi(a) - \pi(b)| \geq \tau \end{cases} \qquad a, b \in V, \tau \in \mathbb{R} \tag{3}$$

The function evaluates to 1 if the difference between performance π_a and π_b of commits a and b exceeds the threshold τ and 0 if not. Given a pair of commits, we can now decide whether performance has changed somewhere between the two commits. However, not each pair of commits is equally informative. The farther the distance between two commits, the lesser the information we can obtain, as there might be several change points in between. In addition, the effect of one change point between two commits can be shadowed by another change point in the opposite direction, such as that one change point increases the execution time and a second decreases the execution time again. We define the influence of each pair on our estimation by weighing each pair inversely proportionally to the distance between two commits.

$$p'(v) = \sum_{\{a \in V \mid a < v\}} \sum_{\{b \in V \mid b > v\}} \underbrace{\theta_\tau(a,b)}_{\text{step function}} \cdot \underbrace{(a-b)^{-2}}_{\text{weighting term}} \tag{4}$$

$$p(v) = \frac{p'(v)}{\sum_{i=1}^{n} p'(i)} \tag{5}$$

For a given commit $v \in V$, we can now estimate a change point probability by comparing each pair of commits before and after v. This is illustrated in Equation 4, where $p'(v)$ is the sum of the influence times the performance change indicator θ_τ for each pair of commits. In practice, however, measuring all commits V is undesirable. Therefore, we sample a small number of commits $T \subset V$ instead and compare each pair of commits before and after v to obtain an approximation of $p'(v)$. Last, to obtain a proper probability distribution, we need to normalize each value $p'(v)$, as illustrated in Equation 5. Consequently, we obtain an approximation $p(v)$ that represents a probability distribution with $\int_0^{|V|} p(v)\, dv = 1$. The resulting probability distribution as well as its approximation are illustrated in Figure 4, where each change point corresponds to a peak in the probability distribution.

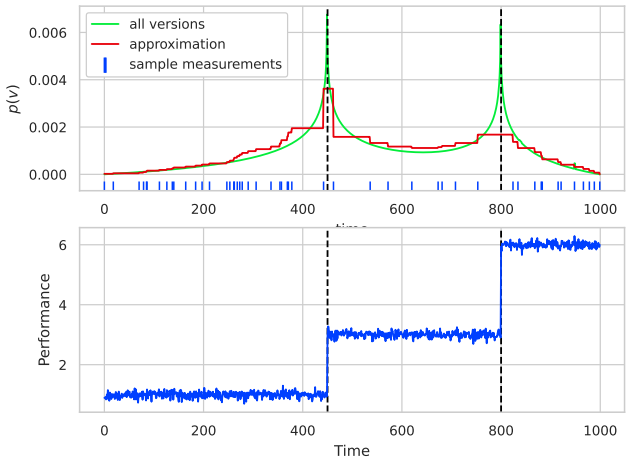


Figure 4: Performance history with 1,000 commits and two change points (bottom); Ground truth and approximated change point likelihood $p(v)$ (top)

3.3 Iteration: Assembling a Candidate Solution

We now have change point likelihood estimations for all configurations in our sample set. Different configuration options and interactions contribute to this change point likelihood as we have seen in the introductory DBMS example. In the following step, we estimate the coordinates (pair of commit and configuration option) of likely change points. So, we first estimate candidate commits based on the change point likelihood from the previous step. Then, we associate these candidate commits with configuration options.

3.3.1 Candidate Commits. For each configuration, we compute an approximation of the change point likelihood over commits (cf. Figure 4) to identify local maxima (peaks). We select such peaks under the condition that the peak change point likelihood is greater than a threshold:

$$t_{\text{CPL}} = \frac{1}{|V|} + N_{\text{CPL}} \cdot \underbrace{\sqrt{\frac{1}{|V|} \cdot \sum_{v \in V} \left(\frac{1}{|V|} - p'(v)\right)^2}}_{\text{standard deviation of } p'(v)} \tag{6}$$

The threshold t_{CPL} is the average change point likelihood over all commits plus a factor N_{CPL} times its corresponding standard deviation over all commits. The factor N_{CPL} (default value: 3) allows

us to filter peaks that do not stand out enough and might be false positives. The greater N_{CPL} is, the stricter the filtering of peaks. That is, for each configuration in our learning set, we obtain a set of commits (configuration-specific candidate commits) that represent possible change points. To reduce variation among the obtained commits, we cluster the commits using kernel density estimation (KDE). The purpose of KDE in our setting is to estimate the probability mass function of whether a commit is a change point. The local maxima of this KDE, subsequently, represent our set of candidate commits. Note that, if two distinct change points are almost coincident commit-wise, they can be mistakenly identified by our approach as one change point that, subsequently, can be associated with configuration options belonging to the two individual change points.

3.3.2 Associating Commits and Options. For each candidate commit that we obtain, we want to know which configuration options are most likely responsible for the respective peak. Hence, we estimate the influence of each configuration option on the change point likelihood. The core idea is to train a linear model $M : [0,1]^{n_{\text{options}}} \rightarrow [0,1]$, in which each configuration option's coefficient corresponds to its influence.
Instead of an ordinary linear regression model, we use a linear model that implements the L_1 norm for regularization (LASSO). In addition to the least squares penalty, this technique favors solutions with more parameters coefficients set to zero. This technique is commonly used to help prevent over-fitting, decrease model complexity, and eliminate non-influential model parameters (configuration options, in our case) [30]. The effect of L_1 regularization in a model is specified by an additional hyper-parameter λ. We tune this hyper-parameter using threefold cross-validation.
For each model (for a candidate commit), we consider a configuration option as associated with a commit, if its influence c_i (i.e., the absolute value of its coefficient) is greater than a threshold:

$$t_{\text{influence}} = \frac{1}{n_{\text{options}}} + N_{\text{influence}} \cdot \underbrace{\sqrt{\frac{1}{n_{\text{options}}} \cdot \sum \left(\frac{1}{n_{\text{options}}} - c_i\right)^2}}_{\text{standard deviation of } c_i} \tag{7}$$

The parameter $N_{\text{influence}}$ (default value: 3) allows us to filter only estimations with low variation. If the intercept of a model exceeds this threshold, we consider this candidate commit a change point that is not configuration-specific (i.e., it affects all configurations). The outcome is a list of *associations*: pairs of candidate commits and (likely) influential configuration options.

3.4 Iteration: Acquiring New Measurements

The last step in each iteration is the acquisition of new measurements. We extend the existing sample with both new commits for configurations already sampled as well for an additional set of new configurations. The role of including new data is twofold. First, previously assessed configurations or commits might not have captured unseen performance shifts. Therefore, a portion of new data is acquired without further knowledge (exploration). Second, a candidate solution might over- or under-approximate associations and can contribute to the algorithm's overall estimation. Therefore, sampling the second portion of data is guided by exploiting each iteration's candidate solution (exploitation).

3.4.1 Acquiring Commits. The exploration of new commits follows a simple rule: For each configuration, we sample a number commits that exhibit the maximum distance to already sampled commits.
The exploitation of an iteration's candidate solution employs the estimation of a configuration's change point likelihood (cf. Section 3.2). We randomly sample among those commits for which the change point likelihood indicates a possible change point, but is not confident. In detail, we select a number of commits for which the change point likelihood is greater than the average (1 divided by the number of commits), but smaller than the average plus the standard deviation of change point likelihood over all commits. By sampling in this range of commits, we incorporate existing knowledge (above average likelihood), but control over-fitting by only sampling commits with maximum likelihood.

3.4.2 Acquiring Configurations. The exploration of new configurations is similar to the initial sample selection strategy (distance-based sampling) described in Section 3.1. We select a constant number of configurations (default value: 75) for exploration.
The exploitation part of acquiring new configurations is guided by the current set of candidate solutions. Each candidate solution describes a change point and associated configuration options. These associated options can be a correct assignment or be an over- or under-approximation. In the latter cases, too many or too few options are associated with a change point. We exploit the existing candidate solutions in a way that addresses both under- as well as over-approximation. We select new configurations using a constraint solver and therefore can specify additional constraints. In the case of under-approximation, too few relevant configuration options are associated with a change point. Given a candidate change point, we require that all associated options keep enabled and that 50 % of all not-associated options can be selected. This way, we keep already associated options, but allow new configuration options to be included. Likewise, in the case of over-approximation, too many irrelevant configuration options are associated with a change point. Given a candidate change point, we require that variation only occurs among 50 % of the options associated with the change point. That is, we can remove up to 50 % of configuration options and narrow down the selection of relevant configuration options.
We limit the total number of new measurements per iteration with a budget $n_{\text{measurements}}$. This budget is split between acquisition for commits and configurations with a factor $n_{\text{commits\_to\_configs}} = 0.5$. The measurement budget for commit acquisition is further split between exploration and exploitation with a factor $n_{\text{commits\_explore}} = 0.5$. That is, initially, 50 % of the budget are used for configuration acquisition and 25 % for commit exploration and exploitation, respectively. As the algorithm proceeds, we multiply the factor $n_{\text{commits\_to\_configs}}$ by 0.95 and the factor $n_{\text{commits\_explore}}$ by 0.9. That is, the algorithm shifts (1) towards sampling configurations in depth and (2) focuses on exploitation in later iterations. The rationale is that we consider pinpointing commit to configuration options a more difficult task than finding performance-changing commits.

3.5 Solution Cache and Stoppage Criteria

After each iteration, we insert the candidate solution in a solution cache. This *solution cache* is a mapping of associations to weights indicating a degree of confidence. The rationale is that, if an association is included repeatedly in an iteration's candidate solution, it is likely a true positive (i.e., a true change point). By contrast, an association that is included only a few times is likely a false positive and can be discarded. We update the solution cache after each iteration in three steps. First, all associations that are either newly included or have been seen before have their weight increased by a constant factor $w_{\text{increase}} = 1$. Our default value for k is set to 3. Second, the weights of all associations in the solution cache are multiplied by a constant decrease factor $e_{\text{decrease}} \in]0, 1[$. We set the default value for e_{decrease} to 0.3. Last, we remove all associations from the solution cache if their weight is smaller than a threshold t_{drop}. We define t_{drop} as the weight an association exhibits if it is included once in a candidate solution but not in $k \in \mathbf{N}$ successive iterations. Effectively, value for t_{drop} is $(e_{\text{decrease}})^k$ since the increment w_{increase} is 1.

Similarly to the conditions for dropping an association from the solution cache, the algorithm terminates if no association is dropped from the solution cache for k iterations in a row and all association's weights are greater than 1. As a fallback termination criterion, the algorithm also terminates if a user-specified maximum number of measurements m_{max} or number of iterations i_{max} is reached.

4 EVALUATION

When evaluating our approach with a single experiment, we face a conflict between internal and external validity, a problem that is prevalent in software engineering research [27]. To assure internal validity, the assessment of our approach with respect to accuracy requires prior knowledge of change points as ground truth, which is hardly obtainable as this would require exhaustive performance measurements across commits and configurations. Moreover, to assure external validity, we require a great degree of variation among subject systems (e.g., number of commits and options, domains of subject systems, etc.) to learn about scalability and sensitivity. So, for a fair assessment, we require not only a large set of systems, but also the respective ground truth performance measurements. The caveat is that it is practically impossible to conduct such exhaustive performance measurements in the large, which was the main reason for proposing our approach in the first place.

To address this dilemma in our evaluation, we conduct two separate experiments, based on synthetic and real-world performance measurements. The first set of experiments uses synthesized performance data providing a controlled experimental setup to assess scalability, accuracy, and efficiency at low cost while simultaneously being able to simulate different scenarios by varying the number of change points and affected configurations in the synthesized data. The second sett of experiments uses a batch of real-world performance measurements of three software systems as a necessarily incomplete ground truth to explore whether our algorithm can be practically applied to real-world systems.

With this split experiment setup, we aim at answering the following two research questions:

RQ$_1$: Can we *accurately* and *efficiently* identify configuration-specific performance change points?

RQ$_1$: Can we *practically* identify configuration-specific change points *in a real-world setting*?

4.1 Controlled Experiment Setup

We break down the research question RQ$_1$ into three objectives to study the influence of the size of configurable software systems, change point properties, and measurement effort.

4.1.1 Influence of System Size. We are interested in how the size of a configurable software system influences the accuracy and efficiency of our approach. To answer this question, we synthesize performance data for systems of varying size in terms of number of configuration options and the number of commits. For the number of configurations n_{options}, we selected a range that resembles configurable software systems studied in previous work [19]; likewise, we selected a range for the number of commits n_{commits} to cover young as well as mature software systems. We present the ranges of the two size parameters in Table 1.

4.1.2 Influence of Change Point Properties. A change point may correspond to a single option or an interaction among multiple options. Furthermore, two change points might be only a few or many commits apart. Therefore, for the synthesized software systems, we vary both the total number of change points as well as the degree of interactions that a software system contains. The number of change points ranges from one to ten, reflecting findings of a recent study about performance change points [19]. We sample the degree of interactions from a geometric distribution (the discrete form of an exponential distribution), which is specified by a single parameter p between 0 and 1, henceforth called $p_{\text{interaction}}$. The greater the value of $p_{\text{interaction}}$, the less likely we generate higher-order interactions. The rationale of this setting stems from previous empirical work [16, 17] that has shown that, by far, most performance issues are related to only single options, or interactions of low degree. We present the specific ranges of the two parameters in Table 1.

4.1.3 Influence of Measurement Effort. We want to understand how the invested measurement effort affects accuracy and efficiency. An initial sample set chosen too small or large might fail to cover change points to exploit or waste measurement effort. In addition, the number of measurements per iteration can be selected irrespective of the software system size, resulting in too few measurements per configuration, and, thus failing to identify change points. That is, for the initialization of our algorithm, we vary both the initial number of configurations and the number of measurements per iteration. We select the number of configurations in the initial sample set as $n_{\text{changepoints}}$ times the number of configuration options, and the number of measurements per iteration from a range of three values. We fix the percentage of commits per configuration at 3 percent, which has been a promising sampling rate in previous work [19]. We present the ranges of the two parameters in Table 1.

4.1.4 Operationalization. We synthesize performance data by initializing each option with a randomly selected influence (cf. coefficients from Equation 1) from the range $[-1, 1]$. In addition, we randomly select a number of interactions among options to introduce

Table 1: Parameter ranges for the synthetic experiment.

	Parameter	Range
Synthesized Systems	n_{options}	8, 16, 32, 64
	n_{commits}	1000, 2500
	$n_{\text{changepoints}}$	1, 2, 5, 10
	$p_{\text{interaction}}$	0.5, 0.7, 0.9
Initialization	N_{initial}	2, 5, 10
	$n_{\text{measurements}}$	100, 200, 500

interactions of varying degrees. The interaction degree (i.e., number of selected configuration options) follows a geometric distribution. We assign to each interaction a real-valued influence uniformly from the range $[-1, 1]$. We define six parameter ranges (cf. Table 1) for the number of configuration options n_{options}, the number of commits n_{commits}, the number of change points $n_{\text{changepoints}}$, the parameter for the geometric distribution of interaction degrees $p_{\text{interaction}}$, a factor N_{initial}, and the number of measurements per iteration $n_{\text{measurements}}$. The initial number of configurations is the product of n_{options} and N_{initial}. We construct the Cartesian product of all parameter ranges and specify a maximum number of 30 iterations. In addition, we synthesize for each parameter combination in the parameter grid five different software systems by employing different seeds such that no seed is used twice. The total number of experiments with seeds is 10,800.

For each parameter combination, we record the number of measurements at each iteration, the required number of iterations for termination, as well as each iteration's candidate solution. To assess the accuracy of a parameter combination's outcome (as well as of intermediate iterations), we use the F_1 score, a combination of precision and recall. *Precision* refers to the fraction of correctly identified associations (true positives) among the associations of the retrieved (candidate) solution. *Recall* is the fraction of total number of the relevant associations (i.e., those we intend to find). The F_1 score is the harmonic mean of precision (P) and recall (R), defined as $F_1 = 2 \cdot \frac{P \cdot R}{P+R}$. In our context, we defined a correctly identified change point when a commits falls in a narrow 5 commit interval from the ground truth. To assess efficiency, we employ the required number of iterations for termination and required measurements in relation to the F_1 score as proxy metrics.

4.2 Real-World Experiment Setup

In the second experiment, we evaluate our approach from a practitioner's perspective, where the properties and whereabouts of change points are unknown. By means of three real-world configurable software systems, we investigate in particular practical challenges and check whether our algorithm is able to detect performance-relevant commits with respect to configuration options.

4.2.1 Exploratory Pre-Study. Of course, we cannot obtain complete ground truth data of our selected subject systems since the search space is exponential in the number of configuration options. However, to provide some context for interpretation of our results, we measured performance for a representative subset of configurations

Table 2: Project characteristics for our three subject systems

Name	#Options	#Commits
XZ	16	1193
LRZIP	9	743
OGGENC	12	935

across all commits. To be precise, we sampled configurations using feature-wise, negative feature-wise, pair-wise, and also uniform random sampling strategies, which have been successfully applied to learn performance influences before [1]. We sampled as many random configurations as there are valid configurations sampled with pairwise sampling, which amounts to 79 configurations for LRZIP, 152 for OGGENC, and 161 for XZ. An overview of the three software systems is given in Table 2.

As a workload for the file compression tools LRZIP and XZ, we used the Silesia corpus, which contains over 200 MB of files of different types. It was designed to compare different file compression algorithms and tools and has been used in previous studies [19]. For the audio transcoder OGGENC, we encoded a raw WAVE audio file of over 60 MB from the Wikimedia Commons collection. For all three subject systems, we assess performance by reporting the execution time.

All measurements were conducted on clusters of Ubuntu machines with Intel Core 2 Quad CPUs (2.83 GHz) and 8 GB of RAM (XZ and LRZIP) and 16 GB of RAM (OGGENC). To mitgate measurement bias, we repeated each measurement five times. The coefficient of variation (the ratio of standard deviation and the arithmetic mean) across all machines was well below ten percent. For commits that did not build, we reported the performance measurement of the most recent commit that did not fail to build.

4.2.2 Operationalization. For the actual experiment, we apply our approach on the performance measurements obtained in the pre-study, with a few adjustments. First, although our study provides a broad and representative sample set of configurations, we cannot arbitrarily sample valid configurations for a couple of reasons. In particular, these include the commits that do not build as well as hidden configuration options and constraints. While we collected configuration options and constraints thoroughly from documentation artifacts, we cannot assure that our selection is complete. We discuss this limitation further in Section 5.

Instead, when acquiring new configurations (cf. Section 3.4), we select configurations from our batch of measurements that (1) have not been used already by our algorithm, and (2) are "closest" to the requested configuration (i.e., with the minimal Hamming distance). The rationale is that this allows us to quickly run rapid repetitions with different initializations of our algorithm. Since we employ multiple sampling strategies in our pre-study, we are confident that our broad batch of measurements is representative. Second, we set the number of measurements per iteration $n_{\text{measurements}}$ to 300. Third, as an initial sample set, we randomly sample 5 configurations from our pre-study set. Last, we initialize our approach with a relative performance threshold θ_τ (cf. Equation 3) of ten percent. In the context of the reported relative variation among performance measurements on the machines used, we consider this a

rather conservative threshold. We repeat the experiments 10 times with different seeds to quantify the robustness of our approach and to account for the randomness in exploring the configuration space. We assess practicality by reporting measurement effort, number of iterations in relation to the pre-study's results, which are based on a vastly greater measurement budget. We qualitatively investigate whether the detected commits and options are actually causes of performance changes. That is, we analyze commit messages of the respective repositories for the identified commits to find occurrences of option names or indicators of performance changes. Moreover, we looked into the code changes (e.g., when the commit message just states 'merge') to rationalize about possible performance affecting code changes.

4.3 Results

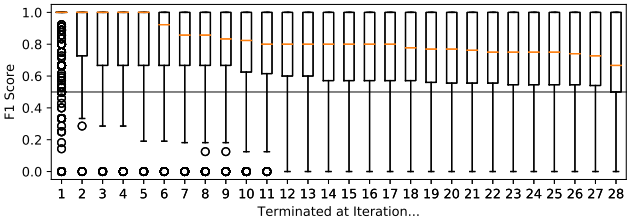
4.3.1 RQ_1 (Controlled Experiment). We illustrate the results of our first set of experiments in Figures 5a, 5b, and 5c. The vast majority of experiments terminated within the limit of 30 iterations, as shown in Figure 5b. A small portion of experiments, however, did not meet our termination criteria (more on that below). From all experiments that terminated in Figure 5a, we depict the F1 score after they have terminated, that is, after their last iteration[2]. For most iterations, the mean F1 score falls around or over 0.7, with the first quartile only slightly being as low as 0.5 (at iteration 29). In the grand scheme of things, the vast majority of experiments terminated with reasonably high F1 score. Regarding measurement effort, for small systems with 8 configuration options, we required up to 50 % of all possible measurements. For greater systems, the required measurement effort was well below 1 %. The high reported measurement effort for smaller systems is due to our choice of the measurements per iterations, which vastly over-approximates. Since we are able handle systems with more configuration options with similar effort, we are confident that a smaller number of measurements per iteration would reduce the relative measurement effort required substantially.

In Figure 5c, we decompose the reported F1 scores to learn how the number of change points or the interaction degree influence our approach. The x-axis shows the highest interaction degree of change points for an experiment run. For interactions of a degree up to 5, the F1 score is mostly above 0.5. We observe a downward trend in the boxplots: the more change points a system contained, the less accurate our algorithm's prediction is.

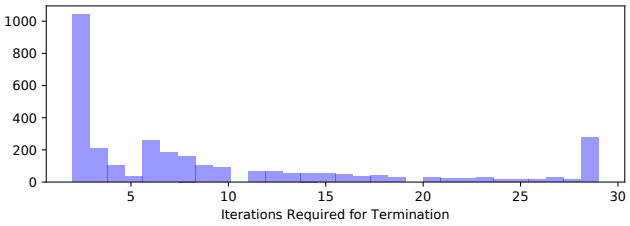
For the experiments not terminating within limit (the rightmost bar in Figure 5b), we conducted an additional analysis. We compared whether and how the parameter setting of Table 1 explains this non-termination. One setting stands out as the cause for not finishing the experiment within 30 iterations: the number of measurements per iterations. The lower the number of measurements ($n_{\text{measurements}}$), the more iterations our approach needs.

Summary (RQ_1): We are able to identify and pinpoint configuration-specific performance change points accurately and at scale. The number of measurements per iteration is the main factor influencing how fast our algorithm terminates.

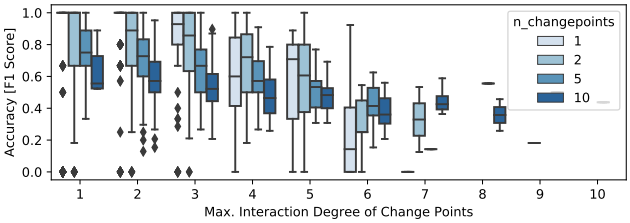
[2]Due to space limitations we report precision and recall on the paper's companion Web site.



(a) F1 score of experiments terminating at different iterations.



(b) Frequency of experiments terminating at different iterations.



(c) Influence of the number of change points and interaction degree on accuracy

Figure 5: Result for the synthetic experiment

4.3.2 RQ_2 (Pre-Study). For our three subject systems, we have manually identified a number of commits, for which performance changed substantially. We have found 7 change points for LRZIP, 2 for XZ, and 2 for OGGENC. For LRZIP, 5 of 7, and for XZ, 2 of 2 performance changes affect multiple configurations. By contrast, 2 change points for LRZIP affect all measured configurations. All of the measured configurations for OGGENC show a shift in performance. That is, the identified change points for OGGENC as well as the two for LRZIP are likely not configuration-specific. To further understand possible relations of change points with configuration options, we searched in commit messages for clues. We discuss two notable findings.

For XZ, the commit messages for both change points referenced three particular configuration options (hc3, hc4, MatchFinder). For LRZIP, one commit message references configuration option (zpac). Four of seven commit messages contained keywords relating to performance. Two examples are the following:

> "`liblzma: Adjust default depth calculation for` **`HC3`** `and` **`HC4`**`. [...]`" (revision 626 of XZ)

> "`Use ffsl for a` **`faster`** `lesser_bitness function.`" (revision 521 of LRZIP)

Of the remaining 6 commits for LRZIP, we identified one as a successor to a merge commit. Within the related pull request[3], we discovered that the configuration LZMA was set to be ignored. This plausibly corresponds to an observed performance shift for configurations with this option enabled. However, this orphan configuration option results in an inconsistent configuration interface, which we discovered with our approach.

4.3.3 RQ_2 (Real-World Performance Data). Across the repetitions of our experiments, we found a number of candidate solutions close to the two change points commits for XZ and OGGENC, respectively, on average after, 3 to 5 iterations. After more than five iterations, the majority of repetitions pinpointed commits correctly within our narrow 5 commit interval. By contrast, for LRZIP, we required up to fifteen iterations to reach this temporal precision. Note that, for LRZIP, the number of change points is more than three times greater than for the other two software systems. In addition, three pairs of change points are 10 commits or less apart, which led to falsely identified commits.

The association of commits to configurability, with the exception of OGGENC, is inconclusive. For OGGENC, our approach consistently reported both change points as not configuration-related. For LRZIP, the reported associations included a variety of possible options, but did not converge and report solutions in line with our pre-study. For XZ, the majority of association configuration options were „match finder" options referenced in the commit messages.

We attribute the inconclusive results for LRZIP in part to the inconsistency of the configuration interface. For both XZ and LRZIP, many configuration options are alternatives. That is, for a functionality such as the compression algorithm, only one of the available options can be selected. In our approach, we employ Lasso regression for automated feature selection. This regression model might, given only a small configuration sample, have difficulties with attributing a change point to multiple alternative configuration options and rather opt for one instead.

Summary (RQ_2): Our approach is able to precisely link change points to commits with real-world data. Manually associating commits with configuration options worked in many cases, whereas the automated analysis was inconclusive due to incomplete data in the repositories. We observed, at least, one inconsistency in the configuration interface of one subject system.

5 DISCUSSION

Efficiency. Among all experiments conducted with our synthetic setup, our approach yielded reasonable to high accuracy across most parameter settings. A subsequent analysis of parameter settings showed that our algorithm is not limited by the size of a problem space, but rather the number of measurements per iteration. Our algorithm found multiple change points across settings of up to 2,500 commits and and 2^{64} configurations, resulting in a search space of about $4.6 \cdot 10^{22}$ possible measurements. In the context of not more than 5,000 measurements per iteration, the maximum number of measurements (after 30 iterations) added to the initial set of measurements is 150,000. For a setting with 2,500 commits, this represents around $1.4 \cdot 10^{-6}$ % and $3.2 \cdot 10^{-16}$ % of possible measurements for settings with 32 and 64 options, respectively. The vast majority of cases terminated in early iterations. Hence, we are confident that our approach scales and can efficiently approximate change points for large software systems. Notably, we are able to reproduce this finding in real-world settings when identifying the temporal location of change points. For our controlled experiment setup, we have not tuned the algorithms' parameters to reach optimal efficiency or accuracy. However, even without tuning, we demonstrated the feasibility and efficiency of our approach. Project-specific fine tuning of several parameters is a possible extension to our approach and can further improve our results.

Pinpointing Configuration Options. The synthetic setup has shown that our algorithm can conceptually handle both time and space. We were able to precisely identify commits for which performance changed substantially. By contrast, we were not able to fully reproduce this in a setting with real-world data when attempting to pinpoint change points to configuration options. Possible explanations in the case of LRZIP include a partly inconsistent configuration interface. The reason for that is not a limitation of our approach, but missing information in the real-world data (which makes the synthetic setup so important). So, we have documented that, given a precisely identified commit, we can link the affected files to configuration options with little effort. In turn, given a set of code segments of a commit, matching this with the overall code base has been extensively studied before under the umbrella of feature location [9]. Inconsistencies in the documentation of configuration options is a well known and prevalent problem [23]. Our approach might be a means to complement existing feature location techniques with information about configuration-dependent performance changes. In our latter experiment, instead of directly sampling new arbitrary configurations, we used a finite set of configurations as a proxy (cf. Section 4.2.2). This does not reflect the original layout of our algorithm, but lets us efficiently explore some limitations of this approach. We have learned that, for interpreting the results of our approach, domain knowledge is advantageous as we have identified incomplete and inconsistent configuration interfaces as possible limiting factors. We acknowledge that, while our approach performs well as an automated pipeline in a controlled environment, for real-world applications, a semi-automated setup with additional manual measurement prioritization and deeper knowledge of configuration constraints would improve the applicability of our approach. This is a promising further avenue of future work.

Threats to Validity. Threats to *internal validity* include measurement noise that may distort change point approximation. We mitigate this threat by repeating each measurement five times and reporting the mean. With respect to the mean performance, the reported variation was below 10%. For RQ2, we considered only performance changes of, at least, 10%. We conducted a pre-study, where we manually identified change points. Hence, we are confident that our raw data are robust against outliers. The setup in RQ2 relies on a limited set of previously sampled configurations instead of actively acquiring new ones. We mitigate this limitation by selecting a broad set of configurations in our pre-study with different sampling strategies. We sampled as many randomly selected configurations as there

[3] https://github.com/ole-tange/lrzip/commit/1203a1853e892300f79da19f14aca11152b5b890

are pairs of configuration options, resulting in a reasonably high coverage of the configuration space.
Regarding *external validity*, we cannot claim that we have identified all limitations possibly arising in a practical setting. However, we selected popular software systems that are well optimized for performance and often make use of configuration options to tune performance. Although we have shown possible limitations in a practical setting, our results, in conjunction with our exhaustive synthetic study, are reproducible, and we were able to test corner cases and assess scalability so that we believe that our results hold for many practical use cases.

6 RELATED WORK

Performance-relevant Commits. Identifying performance-changing commits is a prevalent challenge in regression testing, often applying change point detection algorithms on batches of performance observations [2, 3]. A way to reduce testing overhead is to employ only a fraction of performance observations or to prioritize commits. Alcocer et al. assess performance for uniformly selected commits to estimate the risk of a performance change introduced by unobserved commits [24, 25]. Huang et al. estimate the risk of performance changes solely based on static analysis of individual commits to prioritize commits for regression testing without assessing performance [10]. Mühlbauer et al. use Gaussian Processes and iterative sampling to learn performance histories with few performance observations [19]. They extract performance changing commits by applying change point detection algorithms to learned performance histories. All of the above approaches reduce testing overhead, but do not address the performance across different configurations of software systems. Our approach incorporates performance changes across different configurations by leveraging similarities in related configuration's performance histories.

Performance of Configurable Software Systems. Associating configuration options and interactions with their influence on performance is a conceptual basis for our work. There exists extensive work on modeling configuration options as features for machine learning, such as classification and regression trees [4, 6, 20, 26], multivariable regression [28], and deep neural networks [7]. Predictive models for software performance can be used to find optimal software configurations [21]. In contrast to learning performance for arbitrary configurations, our approach aims at finding performance changes, and subsequently, pinpointing them to configuration options and interactions. A similar scenario to ours has been studied by Jamshidi et al. [11–13]. Between software versions, the performance influence of only few configuration options changes. The authors propose two approaches for learning a transfer function between performance models of different environments, such as versions, based on Gaussian Processes [13] and progressive shrinking the configuration space [12]. We see great potential for future work in applying Gaussian Processes to our problem.

7 CONCLUSION

Changes in the observed performance of a software system can be configuration-specific and difficult to pinpoint to configuration options and commits. We combine both two orthogonal perspectives, the commit history and configurability of software systems to pinpoint performance changes to individual commits and configuration options. We propose an iterative sampling approach that leverages similarities in performance histories of related configurations to identify performance shifts precisely and, subsequently, pinpoint them to configuration options and interactions. We have shown with a synthetic data set that our approach is sound and able to identify performance change points accurately and at scale. Experiments with three configurable software systems confirm the temporal precision of our approach and have confirmed prevalent practical challenges when studying real world-software systems. It is an avenue for further research to incorporate further information, such as commit artifacts, when pinpointing performance changes.

8 ACKNOLEDGEMENTS

Apel's work has been supported by the Deutsche Forschungsgemeinschaft (DFG) under the contract AP 206/11-1. Siegmund's work has been supported by the DFG under the contracts SI 2171/2 and SI 2171/3-1.

REFERENCES

[1] Juliana Alves Pereira, Mathieu Acher, Hugo Martin, and Jean-Marc Jézéquel. 2020. Sampling Effect on Performance Prediction of Configurable Systems: A Case Study. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE)*. ACM, 277–288.

[2] Jürgen Cito, Dritan Suljoti, Philipp Leitner, and Schahram Dustdar. 2014. Identifying Root Causes of Web Performance Degradation Using Changepoint Analysis. In *Proceedings of the International Conference on Web Engineering (ICWE)*. Springer, 181–199.

[3] David Daly, William Brown, Henrik Ingo, Jim O'Leary, and David Bradford. 2020. Industry Paper: The Use of Change Point Detection to Identify Software Performance Regressions in a Continuous Integration System. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE)*. ACM, 67–75.

[4] Jianmei Guo, Krzysztof Czarnecki, Sven Apely, Norbert Siegmund, and Andrzej Wasowski. 2013. Variability-aware Performance Prediction: A Statistical Learning Approach. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE, 301–311.

[5] Jianmei Guo, Jules White, Guangxin Wang, Jian Li, and Yinglin Wang. 2011. A Genetic Algorithm for Optimized Feature Selection with Resource Constraints in Software Product Lines. *Journal of Systems and Software (JSS)* 84, 12 (2011), 2208–2221.

[6] Jianmei Guo, Dingyu Yang, Norbert Siegmund, Sven Apel, Atrisha Sarkar, Pavel Valov, Krzysztof Czarnecki, Andrzej Wasowski, and Huiqun Yu. 2018. Data-efficient performance learning for configurable systems. *Empirical Software Engineering* 23, 3 (2018), 1826–1867.

[7] Huong Ha and Hongyu Zhang. 2019. DeepPerf: Performance Prediction for Configurable Software with Deep Sparse Neural Network. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 1095–1106.

[8] Xue Han and Tingting Yu. 2016. An Empirical Study on Performance Bugs for Highly Configurable Software Systems. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ACM, 1–10.

[9] Emily Hill, Alberto Bacchelli, Dave Binkley, Bogdan Dit, Dawn Lawrie, and Rocco Oliveto. 2013. Which Feature Location Technique is Better? IEEE, 408–411.

[10] Peng Huang, Xiao Ma, Dongcai Shen, and Yuanyuan Zhou. 2014. Performance Regression Testing Target Prioritization via Performance Risk Analysis. In *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 60–71.

[11] Pooyan Jamshidi, Norbert Siegmund, Miguel Velez, Christian Kästner, Akshay Patel, and Yuvraj Agarwal. 2017. Transfer Learning for Performance Modeling of Configurable Systems: An Exploratory Analysis. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE, 497–508.

[12] Pooyan Jamshidi, Miguel Velez, Christian Kästner, and Norbert Siegmund. 2018. 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, 71–82.

[13] Pooyan Jamshidi, Miguel Velez, Christian Kästner, Norbert Siegmund, and Prasad Kawthekar. 2017. 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, 31–41.

[14] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, Jianmei Guo, and Sven Apel. 2019. Distance-based Sampling of Software Configuration Spaces. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 1084–1094.

[15] Sergiy Kolesnikov, Norbert Siegmund, Christian Kästner, and Sven Apel. 2019. On the Relation of Control-Flow and Performance Feature Interactions: A Case Study. *Empirical Software Engineering (ESE)* 24, 4 (2019), 2410–2437.

[16] Sergiy Kolesnikov, Norbert Siegmund, Christian Kästner, Alexander Grebhahn, and Sven Apel. 2019. Tradeoffs in Modeling Performance of Highly Configurable Software Systems. *Software and Systems Modeling (SoSyM)* 18, 3 (2019), 2265–2283.

[17] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 643–654.

[18] Ian Molyneaux. 2015. *The Art of Application Performance Testing* (2nd ed.). O'Reilly, Beijing.

[19] Stefan Mühlbauer, Sven Apel, and Norbert Siegmund. 2019. Accurate Modeling of Performance Histories for Evolving Software Systems. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE, 640–652.

[20] Vivek Nair, Tim Menzies, Norbert Siegmund, and Sven Apel. 2017. 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, 257–267.

[21] V. Nair, Z. Yu, T. Menzies, N. Siegmund, and S. Apel. 2020. Finding Faster Configurations Using FLASH. *IEEE Transactions on Software Engineering (TSE)* 46, 7 (2020), 794–811.

[22] John Ousterhout. 2018. Always Measure One Level Deeper. *Communications of the ACM* 61 (2018), 74–83.

[23] Ariel Rabkin and Randy Katz. 2011. Static Extraction of Program Configuration Options. In *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 131–140.

[24] Juan Pablo Sandoval Alcocer, Alexandre Bergel, and Marco Tulio Valente. 2016. Learning from Source Code History to Identify Performance Failures. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE)*. ACM, 37–48.

[25] Juan Pablo Sandoval Alcocer, Alexandre Bergel, and Marco Tulio Valente. 2020. Prioritizing Versions for Performance Regression Testing: The Pharo Case. *Science of Computer Programming* 191 (2020), 102415.

[26] Atri Sarkar, Jianmei Guo, Norbert Siegmund, Sven Apel, and Krzysztof Czarnecki. 2015. Cost-Efficient Sampling for Performance Prediction of Configurable Systems (T). In *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE, 342–352.

[27] Janet Siegmund, Norbert Siegmund, and Sven Apel. 2015. Views on Internal and External Validity in Empirical Software Engineering. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 9–19.

[28] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. 2015. 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, 284–294.

[29] Norbert Siegmund, Sergiy S. Kolesnikov, Christian Kastner, Sven Apel, Don Batory, Marko Rosenmuller, and Gunter Saake. 2012. Predicting Performance via Automated Feature-Interaction Detection. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 167–177.

[30] Robert Tibshirani. 1996. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)* 58 (1996), 267–288.

[31] Jules White, Brian Dougherty, and Douglas C. Schmidt. 2009. Selecting Highly Optimal Architectural Feature Sets with Filtered Cartesian Flattening. *Journal of Systems and Software (JSS)* 82 (2009), 1268–1284.