Identifying Software Performance Changes Across Variants and Versions

Anonymous Author(s)

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 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 – and propose a novel approach to identify configuration-dependent performance changes. In a nutshell, we iteratively sample pairs of configurations and versions and measure the respective performance that help us 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 configuration options and interactions as well as commits introducing change points with high accuracy and at scale. Our experiments on three real-world systems confirm the effectiveness and practicality of our approach.

ACM Reference Format:
Anonymous Author(s). 2020. Identifying Software Performance Changes Across Variants and Versions. In *ESEC/FSE '20: IEEE/ACM International Conference on Automated Software Engineering, September 21–25, 2020, Melbourne, Australia.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Software performance plays a crucial role in users' perception of software quality. Overly long 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 to customize 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 the configuration decisions.

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 [10]. 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 [7].

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. What is often overlooked is the fact that software system is 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 identify changes in the performance histories 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. In essence, we face a huge combinatorial problem for performance-change detection: code changes across software versions and behavioral changes across software variants. Our problem arises along two dimensions: time (versions) and configuration space (variants). *First*, we aim at finding performance changes in software performance from one version to another. The detection of such changes is referred to as *change point detection* and is a recurring theme in temporal analysis. The main limiting factor is that exhaustive measurements across the configuration space are neither available nor feasible [?]. change point detection techniques with both exhaustive measurement [3] as well as limited data availability [20] have been successfully applied to identify performance changes. *Second*, we aim at associating changes in performance (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 [8, 25, 28, 30]. 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?

Our main idea is as follows: We address the configuration complexity of this problem by selecting representative sample sets of configurations. Next, we uniformly sample a constant number of commits for each configuration and conduct respective performance measurements. Based on these measurement data, we assemble a prediction model. For each configuration, we estimate the

ASE '20, September 21–25, 2020, Melbourne, Australia
© 2020 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

probability 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, for each change point, we obtain 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. configuration-dependent change points.

In summary, we offer the following contributions:

- A technique to effectively identify shifts in performance of configurable software system. We are able to pin point 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 two configurable software systems. Synthetic data lets us assess our model and approach conceptually and at scale, whereas we are able to show their the effectiveness with real-world data.
- An additional website providing material including a reference implementation of our approach and performance measurement data[1].

2 CONFIGURATION-DEPENDENT CHANGE POINTS

Performance as a property, such as execution time, 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 [30]. 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 hypothetical 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. This is because 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 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 *performance-influence models* [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. A formal representation is as follows.

In Equation 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. Each term is described by a coefficient β_f and a configuration parameter c_f. These 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. The singleton subsets correspond to individual features, compound subsets to interactions of features.

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

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, $c_{\text{Compr}\wedge\text{Encrypt}} = c_{\text{Encrypt}} c_{\text{Compr}}$. In the last term, we omit the configuration parameter $c_\emptyset$, as this term represents invariable functionality.

$$\begin{aligned} \Pi_{\text{DBMS}}(c) = {} & \beta_{\text{Encrpyt}} \cdot c_{\text{Encrpyt}} + \beta_{\text{Compr}} \cdot c_{\text{Compr}} \\ & + \beta_{\text{Compr} \wedge \text{Encrpyt}} \cdot c_{\text{Compr} \wedge \text{Encrpyt}} + \beta_\emptyset \end{aligned} \tag{2}$$

Prediction models of configuration-dependent performance of this form are not the only possible representation [6, 8], but linear models are easily explainable because they represent the performance-influence of features and interactions as the linear model's coefficients. We review extensions and alternatives to linear performance-influence models in Section 6.

2.2 Evolution of Performance Influences

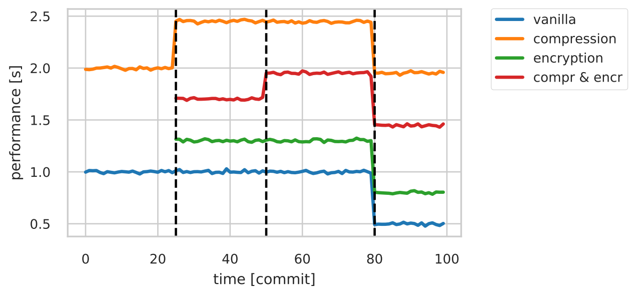
Performance-influence models describe how features shape performance, but they do not allow for understanding configuration-dependent performance changes over time. We expand our DBMS example with a temporal dimension, considering a development history of hundred commits. In particular, we assume the following three events:

(1) At commit 25, feature Compression has been modified, increasing the execution time; feature Encryption is introduced and can be combined with Compression.
(2) At commit 50, the interaction of Compression and Encryption changes, resulting in an increased execution time.

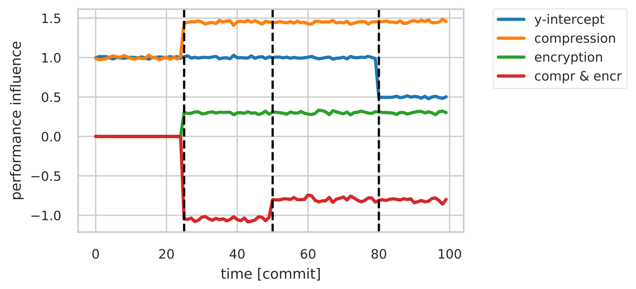
[1]https://github.com/submission-398/changepoints/

(3) At commit 80, the core functionality of the DBMS is refactored, which decreased execution time for all variants.

We illustrate the three events in Figure 1. Our DBMS has four valid variants: one vanilla variant with no feature selected, two variants with one feature enabled, respectively, and one variant with both features enabled. We show the performance history of all variants in Figure 1a and how the influence of features and interactions evolves accordingly in Figure 1b.



(a) Performance of 4 variants with 3 change points



(b) Influence of 2 features, 1 interaction, and y-intercept.

Figure 1: Performance histories and influences for a DBMS with change points at commits 25, 50, and 80

The performance histories of the four variants in Figure 1a exhibit three change points at 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. We decompose the performance histories to a history of performance-influences in Figure 1b. The influence of configuration options and interactions changes only once per influence[2]. At commit 25, the Compression influence increases and only affects one corresponding variant. Similarly, at commit 50, the change in influence of the interaction of configuration options Compression and Encrpytion affects only one variant. At commit 80, the influence of the invariable behavior changes and results in decreasing performance across all variants. That is, from a set of configurations and corresponding performance histories, we can associate performance changes with specific configuration options.

[2]We do not consider the change of performance-influence of Encrpytion and Compression ∧ Encryption in Figure 1b at commit 25 since Encryption is not selectable (and therefore influential) before.

2.3 Taming Complexity

A naive approach for identifying change points in the influence of different options and interactions on performance over time is to combine existing work on performance-prediction over time and across software variants.

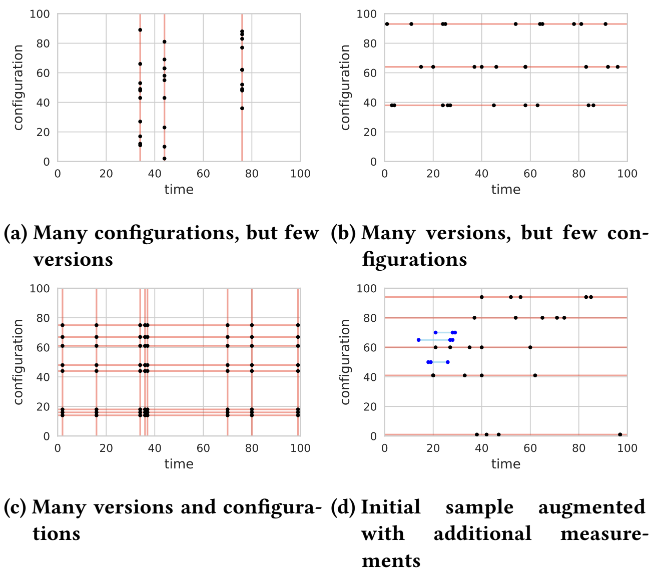
For our example, we could measure all variants for each commit building a performance-influence model per commit. Having 4 configurations and 100 commits, this would result in 400 measurements. Since the number of configurations grows exponentially with the number of features, we end up with 2^n times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

We illustrate the measurement effort in Figure 2c, where each dot represents a pair of configuration and version. To reduce measurement effort, one could limit the measurements to either a few configurations (cf. Figure 2a) or a few commits (cf. Figure 2b), but at the cost of information loss along the respective complementary dimension. This would leave us with either accurate performance-influence models for few versions, resulting in poor temporal granularity, or with accurate performance histories, but only for a few configurations. Both do not provide accurate estimations of changes in the performance influence.

To obtain accurate estimations with a limited budget of measurements, we require a different approach. In Figure 3, we illustrate the trade-offs between measurement budget and accuracy for both dimensions (configuration space and time). Starting with a design similar to Figure 2c, we opt to measure only few versions per configuration and then repeatedly augment the set of measurements with pairs of configurations and versions that are likely to increase relevant information. This is illustrated by the additional blue dots in Figure 2d as well as the segmented arrow in Figure 3. The latter describes, in fact, an iterative and adaptive sampling approach. That is, we use an initial, but sparse sample set to explore the problem space and then increase the level of granularity at promising regions and dimensions (history segments and individual features and interactions, in our case). Adaptive sampling techniques have been successfully applied to obtain both performance-influence models [25, 30] and performance histories [20] before. However, it is unclear whether far less measurements are sufficient to assess if, and if so, where the performance-influence of configuration options and interactions changes? This is what we aim to accomplish with our work.

3 AN ALGORITHM FOR CHANGE POINT DETECTION

We propose an algorithm to detect substantial shifts in performance of configurations and associate them to individual commits as well as configuration options or interactions. We lay out our approach as an iterative search across the commit history and configuration space.



(a) **Many configurations, but few versions**

(b) **Many versions, but few configurations**

(c) **Many versions and configurations**

(d) **Initial sample augmented with additional measurements**

Figure 2: Four different sampling strategies across time and configuration space.

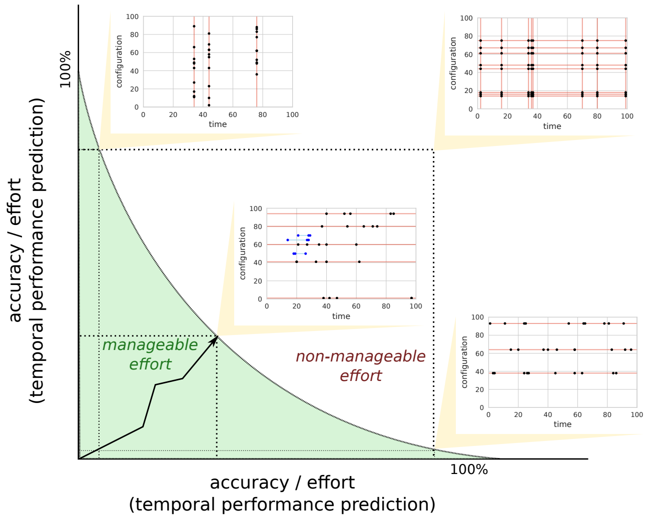


Figure 3: Complexity

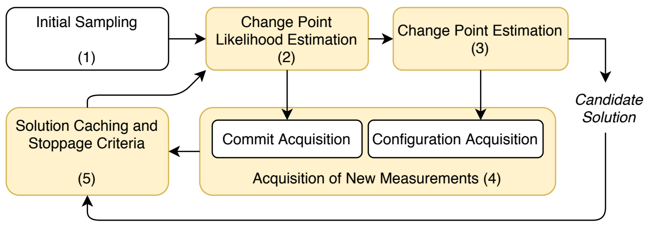
We provide an overview of our approach in Figure 4. It starts with a small initial sample set of measurements (i.e., performance observations of varying configurations and varying commits). Based on this sample set, it calculates for each configuration the likelihood of each commit being a change point.

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 influence. 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 in a performance influence arising from an interaction among two or more options. In the following, 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 a change point or options falsely associated with a commit). That is, for the latter objective, we make an informed decision of what measurements are to be included, whereas the first objective 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 follows, we present the steps of our approach in detail.



Initial Sampling (1)
Change Point Likelihood Estimation (2)
Change Point Estimation (3)
Candidate Solution
Solution Caching and Stoppage Criteria (5)
Commit Acquisition
Configuration Acquisition
Acquisition of New Measurements (4)

Figure 4: Overview of the five steps of our approach: ① Selecting training data across configuration space and time, 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 determining whether the algorithm terminates.

3.1 Initialization and Sampling

The first step in 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 search space and refine the temporal and spatial (i.e., with respect to the configuration space) resolution where necessary.

For the initial configuration sampling, we use distance-based sampling [16], 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 probabilistically 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 each configuration to be sufficiently dissimilar from each other configuration 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 orders (i.e., interactions involving two or more configuration options) are possible, but relatively rare among configurable software systems [17, 18]. Therefore, instead of exhaustive sampling with respect to higher-order interactions, 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 review different techniques to identify a performance change in Section 6.

We encode the threshold in a step function θ_τ in Equation 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.

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

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 proportional 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 density function as well as its approximation are illustrated in Figure 5, where each change point corresponds to a peak in the probability distribution.

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. To this end, we first estimate candidate commits based on the change point likelihood from the previous step. Then, we associate these candidate commits with configuration options.

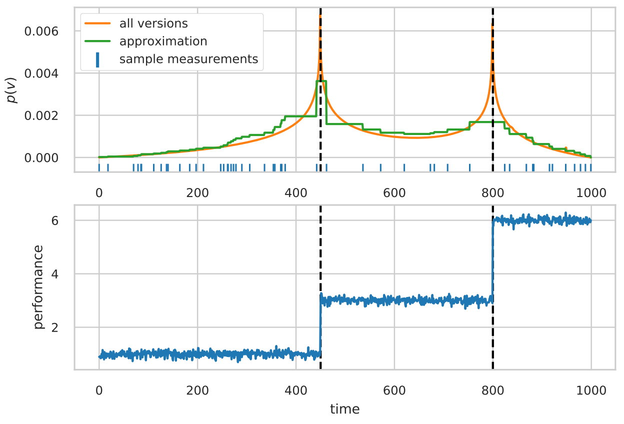


Figure 5: Performance history with 1.000 commits and two change points (bottom); Ground truth and approximated change point likelihood $p(v)$ (top)

3.3.1 Candidate Commits. For each configuration, we compute an approximation of the change point likelihood over commits (cf. Figure 5) to identify local maxima (peaks). We select such peaks under the condition that the peak change point likelihood is greater than a threshold t_{CPL}, as defined in Equation 6. This threshold is the average change point likelihood over all commits plus a factor N_{CPL} times its corresponding standard deviation over all commits.

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

The factor N_{CPL} 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.

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 (sensitivity analysis). The core idea is to train a linear model $M : [0, 1]^{n_{\text{options}}} \longrightarrow [0, 1]$, 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 regression). In addition to the ordinary least squares penalty, this technique favors solutions with more parameters coefficients set to zero. This technique is commonly used to help prevent overfitting, decrease model complexity, and eliminate non-influential model parameters (configuration options, in our case) [?]. 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 $t_{\text{influence}}$, as defined in Equation 7.

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

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 that is to increase commit coverage among already sampled configurations. 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 aim at avoiding 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) [16]. Each configuration is required to be sufficiently dissimilar from previously sampled configurations. That is, we define distance constraints for each existing configuration (in addition to the variability constraints among configuration options) and select a number of new configurations using a satisfiabiltiy solver.
The exploitation of new configurations requires additional constraints, but follows the sample principle. In addition to the constraints mentioned above, we define constraints exploiting the candidate solution's associations. For each commit in the candidate

solution, we perform exploitation in two directions. For the first direction, all configuration options associated with a commit are required to be selected, whereas, for the second direction, all configuration options not associated with a commit are required to be selected. This way, we limit variation among configuration options. The first direction aims at enabling additional configuration options to search for previously unseen, but missing associations (under-approximation). The second direction aims at the opposite: We limit variation to those configuration options already associated with a commit and therefore search for configuration options that are falsely associated (over-approximation).

We limit the total number of new measurements per iteration with a parameter $n_{\text{measurements}}$. The ratio of measurements for exploration and exploitation is defined as $r_{\text{exploration}}$. This ratio is decreased by a factor r_{shift} in each iteration (similar to simulated annealing) [?]. As the algorithm proceeds, more measurements are used for exploitation. We fix the number of configurations for exploration $n_{\text{exploration}}$ as well as the number of configurations for exploitation per candidate commit and iteration, $n_{\text{exploitation}}$. For the assignment of values to parameters, see Section 3.6.

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 a weight 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 with the following three steps. First, all associations that are either newly included or have been seen before have their weight increased by a constant factor $w_{\text{increase}} = 1$. Second, the weights of all associations in the solution cache are multiplied by a constant decrease factor $e_{\text{decrease}} \in]0,1[$. 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 $k \in \mathbb{N}$ 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.

3.6 Parameter Setting

Each step in our approach has a number of parameters that have not been specified so far. We present an overview of all parameters along with their explanation in Table 1. We have assigned values to each parameter and use them for the following evaluation with both synthetic data as well as real-world systems. These assignments do not necessarily represent optimal values, however, yield acceptable results for synthetic and real-world systems across different domains. As (configurable) software systems vary in terms of number of configuration options and commits, fine-tuning parameters in

Table 1: Parameterization of our algorithm. The step number corresponds to the algorithm steps outlined the overview workflow in Figure 4.

Step	Parameter	Purpose	Value
(1)	n_{configs}	Initial number of configurations	50
	r_{commits}	Initial percentage of commits sampled per configuration	0.05
(2)	τ	Relative threshold for performance shifts to be considered substantial	0.05
(3)	N_{CPL}	Peaks with a change point likelihood greater than average plus N_{CPL} times the standard deviation over commits are considered to be clustered into candidate commits	3
	$N_{\text{influence}}$	Configuration options with an influence greater than average plus $N_{\text{influence}}$ times the standard deviation over all options are considered influential	3
(4)	$n_{\text{measurements}}$	Total number of new measurements per iteration	200
	$r_{\text{exploration}}$	Percentage of measurements for exploration	0.5
	r_{shift}	Factor by which $r_{\text{exploration}}$ is multiplied/shifted towards exploitation each iteration	0.95
	$n_{\text{exploration}}$	Number of new configurations for exploration per iteration	5
	$n_{\text{exploitation}}$	Number of new configurations for exploitation per candidate commit and iteration	2
(5)	e_{decrease}	Factor by which association weights are decreased in each iteration	0.3
	k	Number of iterations after which (a) an association is dropped if only visited once, and (b) the algorithm terminates if no association is dropped from the solution cache	3

a practitioner's setting is recommended. Our selection represents a proof of concept. We discuss possible tuning guidelines in the result section of the following evaluation (cf. Section ??).

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 become internally valid, the assessment of our approach with respect to accuracy requires prior knowledge of change points as ground truth, which is hardly obtainable as it would require exhaustive performance measurements across commits and configurations. Moreover, to assess aspects of external validity, such as scalability and sensitivity, we require a great degree of variation among subject system metrics (e.g., number of commits and options, domains of subject systems, etc.). That is, for a fair assessment, we require not only a large set of subject systems, but also the respective ground truth performance measurements. In brief, it is practically impossible to conduct such exhaustive performance measurements in the large, which is the main reason for proposing our approach in the first place.

To mitigate this trade-off in our evaluation, we conduct two separate experiments, based on synthetic and real-world performance

measurements. The first experiments uses synthesized performance data and allows for a controlled experimental setup to assess scalability, accuracy, and efficiency at low cost while simultaneously be able to simulate different scenarios by varying the number of change points and affected configurations in the synthesized data. The second experiment uses real-world performance measurements along with a non-exhaustive ground truth performance measurements to answer whether our algorithm is practically applicable to real-world systems.
This split experiment setup allows for assessing accuracy, efficiency, and scalability in-depth without the pitfall of exhaustive measurements. At the same time, it enables us to validate the approach in a real-world setting. The first and second experiment answer the following two research questions, respectively:

RQ_1) Can we *accurately* and *efficiently* identify configuration-specific performance change points?
RQ_2) Can we *practically* identify configuration-specific change points *in a real-world setting*?

4.1 Controlled Experiment Setup

We lay out a controlled experiment in which we construct synthetic performance-influence models and derive from these synthesized performance values for a specific commit. Using a synthetic setup allows us to investigate the strengths and weaknesses as well as limitations of our algorithm at scale at reasonable cost. Therefore, 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 [20]; similarly, 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 parameters in Table 2.

4.1.2 Influence of Change Point Properties. A change point might correspond to a single option or an interaction involving 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 [20]. We sample the degree of interactions from a geometric distribution (i.e., 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 [18, 19] that has shown that, by far, most performance issues are related to only single options, or interactions of low degree. We present the concrete ranges of the two parameters in Table 2.

Table 2: Parameter ranges for the synthetic experiment.

	Parameter	Range
Synthesized Systems	n_{options}	$2^3, \ldots, 2^6$
	n_{commits}	1000,5000,10000
	$n_{\text{changepoints}}$	1, 2, 5, 10
	$p_{\text{interaction}}$	0.8, 0.85, 0.9
Initia-lization	N_{initial}	2, 5, 10
	$n_{\text{measurements}}$	100, 200, 500

4.1.3 Influence of Measurement Effort. We want to understand how the invested measurement effort influences the accuracy and efficiency. An initial sample set chosen too small or large might fail to cover change points to exploit or be unnecessarily large. 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, fails 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 [20]. We present the ranges of the two parameters in Table 2.

4.1.4 Operationalization. We synthesize performance data by initializing each option with a randomly selected influence. In addition, we randomly sample a number of options interactions. We assign to each interaction a real-valued influence using random uniform sampling from the range [-1, 1]. For the six parameter ranges (cf. Table 2), we construct the Cartesian product. 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 all synthetic experiments, we specify a maximum number of 30 iterations.
For each parameter combination, we record the number of measurements at each iteration, the required number of iteration for termination, as well as each iteration‘s candidate solution. To assess the accuracy of a parameter combination‘s outcome (as well as intermediate iterations), we use the F_1 score, a combination of the metrics 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 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 this 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 practical challenges and to determine 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 an exhaustive ground truth data of our selected subject systems since the search space for such problem is exponential in the number of configuration options.
However, to provide some context for interpretation of our algorithm‘s results we measured a representative subset of valid configurations across all commits. To be precise, we sampled configurations using feature-wise, negative feature-wise, pair-wise, and also uniform random sampling strategies [2], which have been successfully applied to learn performance influences before [28].
As a workload for LRZIP and XZ, we compressed the Silesia corpus[3] which contains over 200 MB of files of different movie "Sintel"[4]) of over 700MB . For OGGENC, we encoded a WAVE audio file of over 60 MB from the Wikimedia Commons collection[5]. The workloads for XZ and LRZIP have been studied in previous work [20, 27]. For all subject systems, we assess performance by reporting the execution time.

Table 3: Properties of our three subject systems

Name	$\|O\|$	$\|C\|$
XZ	x	y
LRZIP	x	y
OGGENC	x	y

$|O|$ = number of options, $|C|$ = number of commits

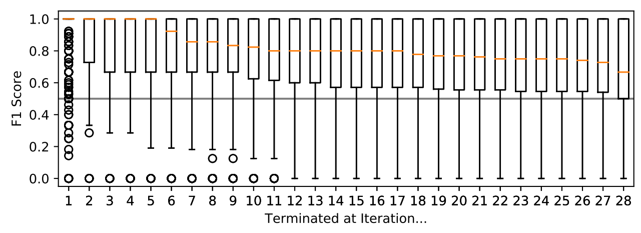
All performance measurements were conducted on commercial off-the-shelf clusters with identical hard- and software setups. In particular, XZ and LRZIP ran on Ubuntu machines with Intel Core 2 Quad CPUs (2.83 GHz) and 8 GB of RAM, OGGENC on machines with the same CPUs, but 16 GB of RAM. To mitgate measurement bias, we repeated each measurement five times. The coefficient of variation (standard deviation divided by the arithmetic mean) reported on all machines was well below ten percent.

4.2.2 Operationalization. For the experiment, we run our approach on the performance measurements obtained in our 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. Instead, when acquiring new configurations (cf. Sec. 3.4), we select configurations that (1) have not already been used by our algorithm, and (2) are closest to the requested configuration (i.e., with the minimal Hamming distance). The rationale is, that this allows for rapid repetitions with different initializations. We are confident that our broad pre-study is representative since we employed multiple sampling strategies. We discuss this decision in the threats to validity. Second, we set the number of measurements per iteration to 300. Third, as an initial configuration 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 assess robustness of our approach and 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 assess practicality not only by reporting the measurement effort and iterations, but also by qualitatively investigating 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 tunings / degradations. Moreover, we further looked into the code changes (e.g.„ when the commit message just states 'merge') to rationalize about possible performance affecting code changes.

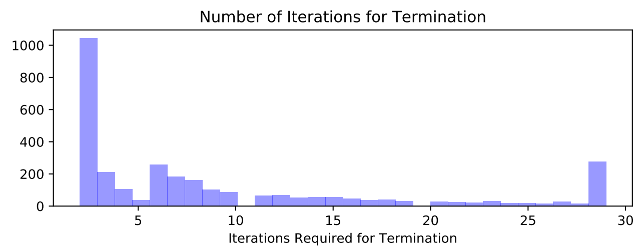
4.3 Results

4.3.1 RQ_1: Controlled Experiment Setup. We illustrate the results of our first series of experiments in Figures 6a, 6b, 6c, and 6d. The vast majority of experiments terminated within the limit of 30 iterations, especially at early iterations, as shown in Figure 6b. A small portion of experiments, however, did not meet our termination criteria (more on that below). From all experiments that terminated in Figure 6a, we depict the F1 score after they have terminated, that is, after their last iteration. For most iterations, the mean F1 score falls around or over 0.7 with the first quartile only slightly reach 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. In addition, we report the visualization of precision and recall on this paper's companion website.
The measurement effort, i.e., the number of measurement required for our approach to terminate is set constant per iteration and, thus, only dependent from the number of iterations. In Figure **??**, we present the measurement effort in relation to the number of possible measurements (experiment runs are color-coded by their number of configuration options) after termination. For small systems with 8 configuration options, we required up to 50 % of all possible measurements. By contrast, for all greater systems, the required measurement effort was well below 1 % (note, that the dots in Figure **??** for systems with 64 configuration options are overlapped by other colors). The high reported measurement effort for systems with 8 configuration options 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 6d, we decompose the reported F1 scores to find out how the number of change points or the interaction degree influence our approach. The x-axis depicts the highest interaction degree of change points for a experiment run. For interactions of a degree up

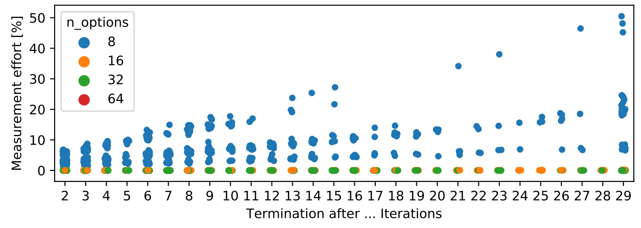
[3]link to silesia corpus
[4]link to sintel trailer
[5]link to WAVE benchmark



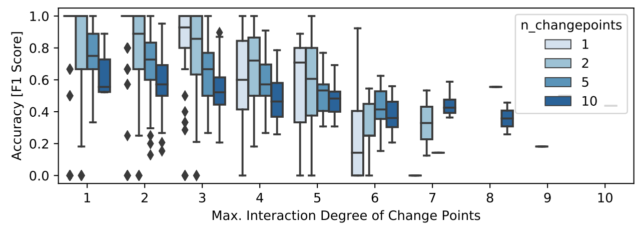
(a) F1 score of experiments terminating at different iterations.



(b) Frequency of experiments terminating at different iterations.



(c) Relative measurement effort for experiments after termination



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

Figure 6: Result description for our synthetic experimental setup

to of 5, the F1 score is mostly above 0.5. We observe a similar downward trend within the boxplots groupings: the more change points a system contained, the less accurate our algorithm's prediction is. For the experiments not terminating within our iteration limit, we conducted an additional analysis. We compared whether and how the parameter setting of (cf. Table 2) explains the 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, the more iterations our approach would need. This finding is in line with previous runs, where the number of measurements per iteration was the most substantial limiting factor. Moreover, this agrees with the huge confuguration and commit spaces we are covering in our experiment. For example, we use settings covering at maximum 2 % of all configurations and commits to as low as 0.0002 %, which are the experiments that would require more iterations.

Summary: We are able to identify and pinpoint configuration-specific performance change points accurately and at scale. The allowed number of measurements per iteration influences how fast our algorithm terminates.

4.3.2 RQ_2: Pre-Study. For the three software 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 a variety of configurations. By contrast, 2 of change points for LRZIP affected 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 a possible relations of change points to configuration options, we searched in commit messages for clues. We illustrate two qualitative findings in the following.

For XZ, the commit messages for both change points referenced three particular configuration options (hc3, hc4, MatchFinder). For LRZIP, one commit message references a configuration option (zpac). Four of seven commit messages contained keywords relating to a 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[6], 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 interface for configuration options provided. Hence, we may have identified a configuration error in this system.

4.3.3 RQ_2: Real-World Performance Data. Across the repetitions of our experiments, we have found 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 repetitions, 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 configuration, with the exception of OGGENC, however, has not been conclusive. 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

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

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 identified. In addition to this, for both XZ and LRZIP, many configuration options are „alternatives", i.e., 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. Across the repetitions of our experiments, we have found 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 repetitions, 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. This finding is in line with our observations from our synthetic experiment. The association of commits to configuration, with the exception of OGGENC, however, has not been conclusive. For OGGENC, our approach consistently reported both change points as not configuration-related For XZ and LRZIP, the reported associations included a variety of possible options, but did not converge and report solutions in line with our pre-study.

Summary: We are able to precisely identify change point to commits with real-world data. Manual associating commits with configuration options worked in many cases as the automated analysis was inconclusive due to incomplete data in the repositories. We observed at least one inconsistency in the configurability of one system across a segment of commits.

5 DISCUSSION

5.1 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 used 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 $3.2 \cdot 10^{-16}$ % and $1.4 \cdot 10^{-6}$ % 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. In addition, we are able to reproduce this finding with real-world performance when identifying the temporal location of change points.

5.1.1 Pinpointing Configuration Options. The synthetic setup has shown that our algorithm can conceptually handle both time and space. By contrast, we were not are to fully reproduce this in a setting with real-world data regarding pinpointing change points to configuration options. Possible explanations for 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, given a precisely identified commit, we have demonstrated that with little effort we can link the affected files to configuration options. 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 techniques [11]. The inconsistencies in the documentation of configuration options is a well known and prevalent problem among open source software systems [23] and, although not intended in the first place, our approach might be a means to complement existing feature location techniques with information about configuration-dependent performance changes. This is a promising further direction that we will investigate in future work.

5.2 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 the experiments of RQ2, we considered only performance changes of at least 10%. We additionally conducted a pre-study, where we manually identified change points. Hence, we are confident, that our raw data is robust against outliers. The setup in RQ2 used a limited set of previously sampled configurations instead of actively acquiring new ones, as proposed by our approach. We mitigate this threat by selecting a broad set of configurations in our pre-study with different sampling strategies. We sampled as many randomly selected configurations as there 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 in a practical setting. However, we selected highly used systems that are well optimized for performance and often make use of configuration options to tune performance. Moreover, by additionally performing an exhaustive synthetic study, 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

The approach presented in this paper combines finding performance-relevant commits with the attribution of changes to configuration options and interactions. In the following, we discuss work that addresses these aspects.

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 [4, 5]. A way to reduce testing overhead is to employ only a fraction of performance observations or prioritize commits. Sandoval Alcocer et al. assess performance for uniformly selected commits to estimate the risk of a performance change introduced by unobserved commits [1, 24]. 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 [12]. Mühlbauer et al. use Gaussian Processes (GP) and iterative sampling to learn performance histories with few performance observations [20]. 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 performance 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 [6, 21, 26], multi-variable regression [29], and deep neural networks [9]. In contrast to learning performance for arbitrary configurations, our approach aims at explaining performance change, and subsequently, a change in a configuration option‘s or interaction‘s influence on performance over time. A similar scenario to ours has been studied by Jamshidi et al. [13–15]. 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 GPs [15] and progressive shrinking the configuration space [14]. We see great potential for future work in applying GPs to our problem statement as they have been successfully applied for the two orthogonal perspectives of time and configuration space.

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 and interactions. 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 is able 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, emphasizing the importance of experimental setups for evaluating algorithms. It is an avenue for further research to incorporate information, such as commit artifacts, into pinpointing performance changes as we were able to identify plausible relations of commits to configuration options to most commits with manually with little effort.

REFERENCES

[1] Juan Pablo Sandoval Alcocer, Alexandre Bergel, and Marco Tulio Valente. 2020. Prioritizing Versions for Performance Regression Testing: The Pharo Case. *Science of Computer Programming* (Feb. 2020), 102415. https://doi.org/10.1016/j.scico.2020.102415

[2] 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* (Edmonton AB, Canada) *(ICPE '20)*. Association for Computing Machinery, New York, NY, USA, 277–288. https://doi.org/10.1145/3358960.3379137

[3] Jürgen Cito, Dritan Suljoti, Philipp Leitner, and Schahram Dustdar. 2014. Identifying Root Causes of Web Performance Degradation Using Changepoint Analysis. In *Web Engineering*, Sven Casteleyn, Gustavo Rossi, and Marco Winckler (Eds.). Vol. 8541. Springer International Publishing, Cham, 181–199. https://doi.org/10.1007/978-3-319-08245-5_11

[4] 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.

[5] 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 *11th ACM/SPEC International Conference on Performance Engineering*. ACM, Edmonton, Canada, 9.

[6] Jianmei Guo, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, and Andrzej Wasowski. 2013. Variability-Aware Performance Prediction: A Statistical Learning Approach. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Silicon Valley, CA, USA, 301–311. https://doi.org/10.1109/ASE.2013.6693089

[7] 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* 84, 12 (Dec. 2011), 2208–2221. https://doi.org/10.1016/j.jss.2011.06.026

[8] Huong Ha and Hongyu Zhang. 2019. DeepPerf: Performance Prediction for Configurable Software with Deep Sparse Neural Network. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, Montreal, QC, Canada, 1095–1106. https://doi.org/10.1109/ICSE.2019.00113

[9] 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.

[10] Xue Han and Tingting Yu. 2016. An Empirical Study on Performance Bugs for Highly Configurable Software Systems. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '16)*. Association for Computing Machinery, Ciudad Real, Spain, 1–10. https://doi.org/10.1145/2961111.2962602

[11] Emily Hill, Alberto Bacchelli, Dave Binkley, Bogdan Dit, Dawn Lawrie, and Rocco Oliveto. 2013. Which Feature Location Technique is Better?. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance (ICSM '13)*. IEEE Computer Society, USA, 408–411. https://doi.org/10.1109/ICSM.2013.59

[12] Peng Huang, Xiao Ma, Dongcai Shen, and Yuanyuan Zhou. 2014. Performance Regression Testing Target Prioritization via Performance Risk Analysis. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 60–71. https://doi.org/10.1145/2568225.2568232

[13] 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 32nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE 2017)*. IEEE Press, 497–508.

[14] 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 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 71–82. https://doi.org/10.1145/3236024.3236074

[15] 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 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems* (Buenos Aires, Argentina) *(SEAMS '17)*. IEEE Press, 31–41. https://doi.org/10.1109/SEAMS.2017.11

[16] 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.

[17] 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* 24, 4 (Aug. 2019), 2410–2437. https://doi.org/10.1007/s10664-019-09705-w

[18] Sergiy Kolesnikov, Norbert Siegmund, Christian Kästner, Alexander Grebhahn, and Sven Apel. 2019. Tradeoffs in Modeling Performance of Highly Configurable Software Systems. *Software & Systems Modeling* 18, 3 (June 2019), 2265–2283. https://doi.org/10.1007/s10270-018-0662-9

[19] 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 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 643–654. https://doi.org/10.1145/2884781.2884793

[20] Stefan Mühlbauer, Sven Apel, and Norbert Siegmund. 2019. Accurate Modeling of Performance Histories for Evolving Software Systems. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, San Diego, CA, USA, 640–652. https://doi.org/10.1109/ASE.2019.00065

[21] 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.

[22] John Ousterhout. 2018. Always Measure One Level Deeper. *Commun. ACM* 61, 7 (June 2018), 74–83. https://doi.org/10.1145/3213770

[23] Ariel Rabkin and Randy Katz. 2011. Static Extraction of Program Configuration Options. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. Association for Computing Machinery, New York, NY, USA, 131–140. https://doi.org/10.1145/1985793.1985812

[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 7th ACM/SPEC on International Conference on Performance Engineering - ICPE '16*. ACM Press, Delft, The Netherlands, 37–48. https://doi.org/10.1145/2851553.2851571

[25] Atri Sarkar, Jianmei Guo, Norbert Siegmund, Sven Apel, and Krzysztof Czarnecki. [n.d.]. Cost-Efficient Sampling for Performance Prediction of Configurable Systems. ([n. d.]), 11.

[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 *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. IEEE, Florence, Italy, 9–19. https://doi.org/10.1109/ICSE.2015.24

[28] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. 2015. Performance-Influence Models for Highly Configurable Systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*. ACM Press, Bergamo, Italy, 284–294. https://doi.org/10.1145/2786805.2786845

[29] 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.

[30] Norbert Siegmund, Sergiy S. Kolesnikov, Christian Kästner, Sven Apel, Don Batory, Marko Rosenmuller, and Gunter Saake. 2012. Predicting Performance via Automated Feature-Interaction Detection. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, Zurich, 167–177. https://doi.org/10.1109/ICSE.2012.6227196