2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)

Distance-Based Sampling of Software Configuration Spaces

Christian Kaltenecker*, Alexander Grebhahn*, Norbert Siegmund†, Jianmei Guo‡, Sven Apel*
*University of Passau, Germany
† University of Weimar, Germany
‡ Alibaba Group, China

***Abstract*—Configurable software systems provide a multitude of configuration options to adjust and optimize their functional and non-functional properties. For instance, to find the fastest configuration for a given setting, a brute-force strategy measures the performance of all configurations, which is typically intractable. Addressing this challenge, state-of-the-art strategies rely on machine learning, analyzing only a few configurations (i.e., a sample set) to predict the performance of other configurations. However, to obtain accurate performance predictions, a *representative* sample set of configurations is required. Addressing this task, different sampling strategies have been proposed, which come with different advantages (e.g., covering the configuration space systematically) and disadvantages (e.g., the need to enumerate all configurations). In our experiments, we found that most sampling strategies do not achieve a good coverage of the configuration space with respect to covering relevant performance values. That is, they miss important configurations with distinct performance behavior. Based on this observation, we devise a new sampling strategy, called *distance-based sampling*, that is based on a distance metric and a probability distribution to spread the configurations of the sample set according to a given probability distribution across the configuration space. This way, we cover different kinds of interactions among configuration options in the sample set. To demonstrate the merits of distance-based sampling, we compare it to state-of-the-art sampling strategies, such as t-wise sampling, on 10 real-world configurable software systems. Our results show that distance-based sampling leads to more accurate performance models for medium to large sample sets.**

I. Introduction

Modern software systems can be configured by users to adapt them to specific devices, operating systems, and requirements. Configuration options often have a significant influence on non-functional properties, such as performance or energy consumption. Despite the benefits of configurability, identifying the performance-optimal configuration for a given setting is often a non-trivial task, due to the sheer size of configuration spaces [1] and potential interactions among configuration options [2], [3]. To identify the performance-optimal configuration of a configuration space, one can measure the performance of every valid configuration of the software system in a brute-force manner, which usually does not scale.

To avoid measuring all configurations, machine-learning techniques, such as multiple linear regression [4], [5] and classification and regression trees [6]–[8], have been used to learn a performance model based on a set of valid configurations, called the *sample set*. A performance model allows us to predict the performance of a configuration, and it can be used by an optimizer to determine the performance-optimal configuration [8], [9]. To create an accurate performance model, the sample set must be well-chosen, which is a non-trivial task especially when no domain knowledge is available. In essence, selecting a small, valid, and representative sample set is key to efficiency and accuracy of performance prediction, as performance measurements are usually costly in practice [10]. Several sampling strategies have been proposed, which differ in their methods of selecting the sample set: (1) at random [11], [12], (2) by using an off-the-shelf constraint solver [9], or (3) by aiming at a certain coverage criterion (e.g., selecting each configuration option, at least once) [13]–[15]. Naturally, all sampling strategies come with advantages and disadvantages, as we will discuss in Section II. The main idea is often to cover the configuration space such that one obtains a *representative* sample set, which, ideally includes both influential configuration options and interactions among options relevant to performance, so that accurate performance models can be learned.

In general, a uniform coverage of the configuration space is desirable to obtain a representative sample set when no prior knowledge is available, since it tends to be unbiased when covering the configuration space. However, it is far from trivial to ensure unbiased uniformity if there are non-trivial constraints among configuration options. To achieve the goal in a light-weight way, we propose a new sampling strategy, called *distance-based sampling*, that addresses the shortcomings of existing strategies, as we will discuss next.

- *Random sampling* strives for covering the configuration space uniformly, but current strategies do not scale when constraints exist [12]. In fact, randomly selecting individual configuration options to retrieve a valid configuration is often impractical[1] due to complex constraints among configuration options. Alternatively, one can use a constraint solver to enumerate all valid configurations and randomly draw from this *whole population*, which is intractable either due to the sheer number.
- *Coverage-oriented strategies* focus on specific areas or properties of the configuration space, such as specific kinds of interactions as in t-wise sampling [4]. This might be the optimal way to sample if we would know in

[1]When randomly selecting options of the Linux kernel, there was not a single valid configuration even after one million trials [16], [17].

1558-1225/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSE.2019.00112

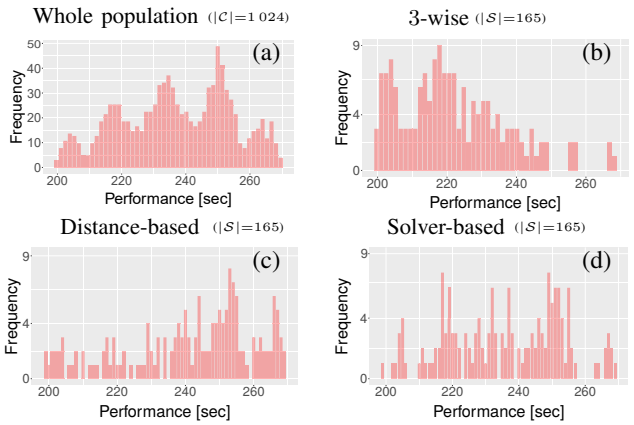

Fig. 1: The performance distribution of (a) the whole population of LLVM (see Section IV-C), the sample set selected (b) by 3-wise sampling, (c) by distance-based sampling, and (d) by solver-based sampling. In 3-wise sampling, some high performance values are missed, leading to a skewed distribution, whereas distance-based sampling resembles the distribution from the whole population better.

advance where to sample, but this is usually not the case for large software systems. For example, Figure 1(a) shows the distribution of performance values (x-axes) for the whole population of LLVM, whereas Figure 1(b) shows the distribution of a sample set obtained by 3-wise sampling. We observe a left-skewed distribution of the sample set that does not match the distribution of the whole population well. In contrast, our distance-based sampling, shown in Figure 1(c), tends to maintain the original distribution, which eases learning a performance model. A better coverage of the performance distribution increases the *representativeness* of the sample set [10].

- Some strategies use an off-the-shelf satisfiability solver for sampling without enumerating all configurations. As we show in Figure 1(d), this results in sample sets that seem to maintain the original distribution, but the produced configurations tend to be locally clustered, which may miss relevant interactions [9]. Though simple and efficient, this strategy provides no guarantees regarding spread of the sample across the configuration space.

The key idea behind distance-based sampling is to produce a sample set that covers the configuration space as uniformly as possible (or following another given probability distribution). To this end, distance-based sampling relies on a *distance metric* and assigns each configuration a *distance value*. It further relies on a *discrete probability distribution* to select configurations according to their distance values from the configuration space. It differs from other sampling strategies in that (1) it spreads the selected configurations across the configuration space according to a given probability distribution and is able to resemble the performance distribution of the whole population, (2) it does not require an analysis on the whole population, and (3) it uses internally a constraint solver for efficiency while avoiding locally-clustered sample sets.

In summary, our contributions are as follows:

- We define a new distance-based sampling strategy that is based on a given discrete probability distribution and a distance metric for configurations of software systems.
- We perform an empirical study to compare our sampling strategy with other widely-used sampling strategies learning performance models for 10 popular real-world software systems. We find that distance-based sampling achieves better results in terms of prediction accuracy and robustness than other sampling strategies.
- To further improve the diversity of the sample set, we devise an optimization of distance-based sampling by iteratively forcing the selection of the least frequently selected configuration option for configurations with the same distance. The optimization leads to a significantly higher robustness and significantly better prediction accuracy of distance-based sampling.
- We show that our strategy incurs a lower computing effort than other sampling strategies, and that it is more flexible in that the probability distribution it relies on can be exchanged on demand.

All experiment and replication data are available on a supplementary website[2].

II. Preliminaries

In this section, we introduce basic concepts of configurable software systems. Furthermore, we provide an overview of state-of-the-art sampling strategies, from which we derive the motivation for distance-based sampling.

A. Configurable Software Systems

A configurable software system offers a set $\mathcal{O}$ of configuration options. In this work, we concentrate on binary configuration options, which take only the values 0 (deselected) and 1 (selected). The set of all valid configurations is denoted as $\mathcal{C}$, which we will refer to as *whole population*. A configuration $c \in \mathcal{C}$ is a function $c : \mathcal{O} \to \{0,1\}$, which assigns either 1 (if option o is selected) or 0 (if option o is not selected) to each configuration option $o \in \mathcal{O}$. We call a configuration option $o \in \mathcal{O}$ mandatory if $\forall c \in \mathcal{C} : c(o) = 1$, that is, the configuration option is selected in every configuration.

In practice, not all combinations of configuration options are valid (i.e., $|\mathcal{C}| < 2^{|\mathcal{O}|}$), due to constraints among configuration options. Constraints can be expressed in terms of propositional formulas over the set of configuration options. For instance, in a compression library offering two compression algorithms, *rar* and *zip*, exactly one of these compression algorithms has to be selected to obtain a valid configuration. The corresponding Boolean expression would be: $(zip \wedge \neg rar) \vee (\neg zip \wedge rar)$. The constraints among configuration options are often combined in a variability model [18]. For complex systems, such constraints are the reason that the acquisition of the whole population is time consuming. Hence, selecting only few instead of all configurations is advisable, as we describe next.

[2]https://github.com/se-passau/Distance-Based_Data/

B. Sampling

Sampling is the process of selecting a subset $\mathcal{S} \subseteq \mathcal{C}$ of all valid configurations $\mathcal{C}$ of a given configurable software system. There are different strategies for this: random sampling, solver-based sampling, and coverage-oriented sampling.

Random sampling: One way to create a sample set is by randomly assigning either 0 or 1 to each configuration option for each configuration [6]. However, it is very likely that many invalid configurations are selected this way due to unsatisfied constraints, which makes this strategy inefficient. Chakraborty et al. [12] use hash functions to split the configuration space recursively in multiple regions, and they select configurations from each of the regions. Still, this strategy produces many invalid configurations. Chen et al. [19] use a distance metric to find different test inputs for methods to uniformly cover the configuration space. However, they do not consider constraints among the input variables and, thus, produce many invalid configurations. Oh et al. [20] encode a system's configuration space using a binary decision diagram. This way, they can represent and enumerate all configurations in a compact way, such that they can randomly draw configurations. However, construction time and memory consumption of binary decision diagrams are high, and they do not scale to the largest configurable software systems [21]. Gogate and Dechter [11] propose a random sampling strategy that uniformly selects configurations without enumerating all configurations using the Monte-Carlo method. This strategy also selects invalid configurations, though.

Solver-based sampling: Many strategies use an off-the-shelf constraint solver, such as SAT4J[3], for sampling. Naturally, these strategies do not guarantee true randomness [9] as in random sampling. Often the sample set consists only of the first k solutions provided by the constraint solver [12], and the internal solver strategy is typically to search in the "neighborhood" of an already found solution. Hence, the result is a locally clustered set of configurations. To weaken the locality drawback of solver-based sampling, Henard et al. [9] change the order of configuration options, constraints, and values in each solver run. This strategy, which we call henceforth *randomized solver-based sampling*, increases diversity of configurations, but it cannot give any guarantees about randomness or coverage. As we will show in our evaluation, this strategy requires to rebuild the entire solver model from scratch at each solver call (i.e. selection of one configuration), which is computationally expensive.

Coverage-oriented sampling: Coverage-oriented sampling strategies optimize the sample set according to a specific coverage criterion. One prominent example is *t-wise* sampling [13]–[15]. This sampling strategy selects configurations to cover all combinations of t configuration options being selected. For instance, pair-wise (t=2) sampling covers all pair-wise combinations of configuration options being selected. To identify the influence of pairs of configuration options

[3]https://www.sat4j.org/

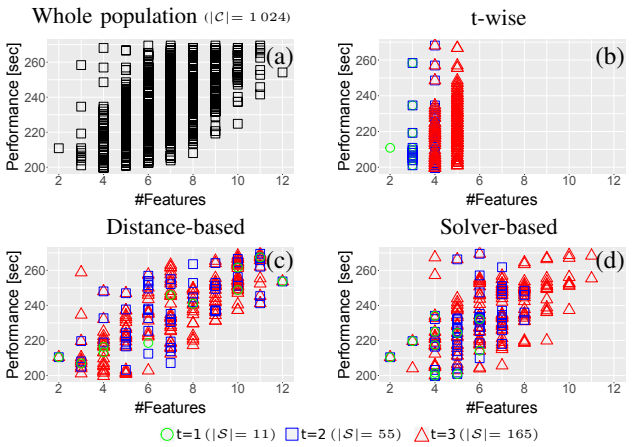

Fig. 2: Distribution of configurations of LLVM (see Section IV-C) based on their distance value and their performance, for (a) the whole population, a sample set selected (b) by t-wise sampling, (c) by distance-based sampling, and (d) by solver-based sampling with the same sample size as t-wise with t=1, t=2, and t=3, respectively.

and not to be affected by influences of other configuration options, Siegmund et al. [4] improves *t-wise* sampling by additionally minimizing the number of other selected configuration options in each configuration. Another strategy aims at a balanced selection and deselection of all configuration options in the sample set. Sarkar et al. showed that such a frequency-based sampling further improves the accuracy of performance models learned based on the sample set [22]. Other coverage-oriented sampling strategies are, for example, statement-coverage sampling [23] or most-enabled-disabled sampling [24]. Statement-coverage sampling is a white-box strategy, in which the configurations are selected such that every block of optional code from the software system is selected, at least, once; whereas most-enabled-disabled sampling selects just one configuration where all configuration options are selected and one where all configuration options are deselected. The main problem of these strategies is that they require prior knowledge to select a proper coverage criterion. Thus, depending on the coverage criterion, specific regions of the configuration space are emphasized, as shown in Figure 2: We see that a higher number of selected configuration options leads to higher performance values and contains information from interactions among multiple configuration options. Since t-wise sampling focuses only on a specific part of the configuration space (e.g., configurations with distances between 2 and 5), we miss certain performance values in the sample set (e.g., greater than 250 seconds). By contrast, a sample set that covers all performance values is more representative, as it resembles the whole population better. Moreover, not every configuration option interacts with any other, so not all pairs are relevant. So, the sample set is likely unnecessarily large.

III. Distance-Based Sampling

Distance-based sampling aims at covering the configuration space by uniformly (or according to another given probability distribution) selecting configurations with different distance values (and therewith interaction degrees) without relying on a whole-population analysis. In Section III-A, we describe the basic algorithm; in Section III-B, we present an optimization that further increases the diversity of the sample set.

A. Basic Algorithm

The key idea to spread the sample set across the configuration space to increase diversity in the sample set is to use a *distance metric* in combination with a *discrete probability distribution* (e.g., a uniform distribution or a binomial distribution).

Algorithm 1: *Distance-based sampling*

```
   Input: VM, numSamples, probabilityDistr
   Output: sampleSet
1  sampleSet ← ∅
2  while otherSolutionsExist(VM, sampleSet) and size(sampleSet) < numSamples do
3  |   d ← selectDistance(probabilityDistr)
4  |   c ← searchConfigWithDistance(VM, d) ▷ Search for configuration c with
   |     exactly d configuration options selected
5  |   if c ≠ ∅ then
6  |   |   sampleSet ← sampleSet ∪ {c}
7  |   end
8  end
9  return sampleSet
```

In Algorithm 1, we describe the algorithm behind distance-based sampling. It receives three parameters as input: the variability model (*VM*), the number of configurations to be selected (*numSamples*), and the probability distribution to use (*probabilityDistr*). Internally, we use a constraint solver that uses the variability model to determine the valid configurations. We assume that the solver is globally available to the algorithm.

The algorithm selects a distance d based on the probability distribution (*probabilityDistr*). The distance is passed as an additional numeric constraint (i.e., in addition to the constraints of the variability model) to the constraint solver, which searches for a solution with exactly d configuration options selected (Line 4). If a solution (i.e., a valid configuration) is found, it is included into the sample set (Line 6). If not, another distance d is selected until a valid configuration with this distance is found. This process is repeated until the sample set contains the desired number of valid configurations or there are no more solutions (Line 2).

In what follows, we define the distance metric and the discrete probability distribution that we use, and we describe the selection of a valid configuration in more detail.

Distance metric (selectDistance): Figure 2 illustrates that t-wise sampling covers only specific intervals in the range of possible distances. In fact, t-wise sampling misses information on interactions among more than t configuration options. By using a distance metric to diversify the sample set, we cover more regions of the configuration space, which leads to a more diverse sample set. In what follows, we use the Manhattan distance [25] of a configuration to the origin of the configuration space c_0 ($\forall o \in O : c_0(o) = 0$) as distance; another reference point would be possible, though.

So, let dist: $\mathcal{C} \to \mathbb{N}$ be the distance metric defined as follows:

$$\text{dist}(c) = \sum_{o \in \mathcal{O}} c(o) \tag{1}$$

where $c \in \mathcal{C}$ is a valid configuration. Let $\mathcal{D}$ be the set of all distances:

$$\mathcal{D} = \{\text{dist}(c) \mid \forall c \in C\} \tag{2}$$

Note that, in the case of having only binary configuration options and using the Manhattan distance, the distance is just the number of selected configuration options of the configuration. For instance, for d=2, we will search for a configuration that has exactly two configuration options selected and all remaining configuration options deselected.

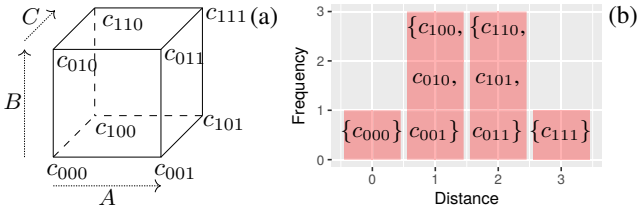

Fig. 3: Example for applying the distance function to a software system with three configuration options A, B, and C without any constraints. In (a), we show the configuration space and in (b) we illustrate the distribution of distances.

Probability distribution (*probabilityDistr*): In Figure 3, we show the distances and the number of configurations (frequency) per distance for a system with three configuration options A, B, and C, without any constraints. In this small example, the distance distribution is easy to compute: We derive all valid configurations and apply the distance metric to each of them. However, deriving all valid configurations (i.e., the whole population) is infeasible for complex systems. Fortunately, it turns out that we do not need a data set following an exact probability distribution. Instead, we use pre-defined discrete probability distributions (e.g., uniform, geometric, binomial) to define the desired distribution of our sample set. These discrete probability distributions (*probabilityDistr*) are used to express the likelihood of choosing a certain value $u \in U$. In every discrete probability distribution P, we have $\sum_{u \in U} P(X = u) = 1$, where X is a random variable. In what follows, we use the discrete *uniform* distribution. So, we uniformly draw a distance $d \in \mathcal{D}$ to derive a configuration with d configuration options selected. Thus, we obtain:

$$P(X = d) = \frac{1}{|\mathcal{D}|} \tag{3}$$

In other words, each distance d equally likely to be picked. While it is possible to use another discrete probability distribution, such as the geometric distribution or the binomial distribution, we fix this degree of freedom for now, to keep

the discussion focused and the experiment design tractable. Nevertheless, we performed experiments with different distributions, which we discuss in Section V-C.

Without computing the whole population or understanding all constraints in the variability model, we have no knowledge about the value domain of $\mathcal{D}$ and thus about $P(X = d)$. We can approximate the lower and the upper bound for $\mathcal{D}$, though, by using the number of mandatory configuration options and the number of all configuration options, respectively.

$$\max(\mathcal{D}) = \text{card}(\{o \mid o \in \mathcal{O}\}) \tag{4}$$

$$\min(\mathcal{D}) = \text{card}(\{o \mid o \in \mathcal{O} \wedge o \text{ is mandatory}\}) \tag{5}$$

where card is the cardinality function. Mandatory configuration options can be easily computed based on the given constraints of the variability model [26].

Configuration selection (searchConfigWithDistance): After choosing a certain distance d, we select a configuration that has a distance of d (d options selected in our setting), for which we use a constraint solver. To this end, we add an additional constraint to the solver describing that exactly d options have to be selected in the configuration. If there is no more configuration with exactly d selected configuration options, another distance d is selected. This process is repeated until the sample set contains a given number of configurations or no further configurations are found. In cases where we select several times the same distance value, the constraint solver could generate locally clustered solutions. To address this problem, we propose an optimization to further increase diversity of our sample set, as we describe in Section III-B. Note that we do not minimize the number of selected configuration options, as in t-wise sampling, which would be computationally expensive and does not pay off (see Section V).

Time complexity: The most costly function in Algorithm 1 is searchConfigWithDistance, whose complexity is dominated by the time of computing a feasible solution by the constraint solver. Theoretically, a constraint solver, such as a SAT solver, has an exponential time complexity to solve a satisfiability problem [27]. Practically, state-of-the-art constraint solvers are able to handle thousands of variables and constraints efficiently [28]. In our experiments, the constraint solver considered up to 54 variables and up to 216 000 constraints, which resulted in less than 0.3 seconds to find a solution (i.e., a valid configuration).

B. Increasing Diversity

In preliminary experiments with Algorithm 1, we noticed that the produced sample sets may lack diversity in that some configuration options are selected in many configurations and some only in few or even none. Hence, to increase diversity of the sample set, we refine the configuration selection procedure of distance-based sampling by adding configurations that contain the least frequently selected configuration options. This way, we reduce the possibility of missing or underrepresenting certain configuration options in the sampling process. Technically, we determine a ranking over the frequency of configuration options, which is defined as follows:

$$\forall o \in \mathcal{O} : \text{card}(\{c \mid c \in \mathcal{S} \wedge c(o) = 1\}) \tag{6}$$

If there is no valid configuration with the given configuration option and distance, we select the next option in the ranking and so on.

Algorithm 2: *Diversified distance-based sampling*

Input: *VM*, *numSamples*, *probabilityDistr*
Output: *sampleSet*

```
1  candidates ← getAllOptions(VM)  ▷ Generates a list of candidates, one candidate
     for each configuration option
2  sampleSet ← ∅
3  while otherSolutionsExist(VM, sampleSet) and size(sampleSet) < numSamples do
4      d ← selectDistance(probabilityDistr); c ← ∅
5      while candidatesExist(candidates, d) and c = ∅ do
6          candidate ← getLeastFrequentCandidate(candidates, d)
7          c ← searchConfigWithDistance(VM, candidate, d)
8          if c = ∅ then removeCandidate(candidates, candidate, d)
9      end
10     if c ≠ ∅ then
11         sampleSet ← sampleSet ∪ {c}
12         updateCandidateMap(c, d, candidates)
13     end
14 end
15 return sampleSet
```

In Algorithm 2, we show the optimized version of distance-based sampling, which we call *diversified distance-based sampling*. The novelty here is that we count the number of selections of each configuration option for each distance $d \in \mathcal{D}$. To this end, we define one map in Line 1 for each distance $d \in \mathcal{D}$ and update the map in Line 12 when a new configuration is added to the sample set.

The least frequent configuration option of the current distance d is selected using the map in Line 6 and used to retrieve the next configuration in Line 7. If there is no more configuration with the given distance d that contains the candidate, this candidate is removed from the distance's candidate map in Line 8 and the next configuration option is used. As we show in Line 5, another least frequent candidate is repeatedly chosen until a valid configuration is found.

IV. Experiment Setup

In this section, we introduce our research questions regarding the comparison of distance-based sampling with other state-of-the-art sampling strategies. Furthermore, we describe how we attempt to answer the research questions and the software systems we use for the comparison.

A. Research Questions

The prediction accuracy of machine-learning techniques largely depends on the data set, which is defined by the sampling strategy, in our setting. Naturally, some sampling strategies, such as random sampling, are affected by randomness, which can have a considerable influence on the sample set and, consequently, on the prediction accuracy. Hence, we consider both the prediction accuracy and its robustness when comparing sampling strategies. To this end, we aim at answering two research questions:

- *RQ1:* What is the influence of using distance-based, diversified distance-based, random, solver-based, randomized solver-based, and t-wise sampling on the accuracy of performance predictions?
- *RQ2:* What is the influence of randomness of using distance-based, diversified distance-based, solver-based, randomized solver-based, and random sampling on the robustness of prediction accuracy?

Note that we have excluded t-wise sampling from RQ_2, as it is deterministic in our setting and does not lead to variations.

B. Operationalization

To answer our research questions, we apply a state-of-the-art machine-learning technique that relies on multiple linear regression and feature-forward selection [5] to learn performance models based on the sample sets defined by the different sampling strategies.

To answer RQ_1, we use the resulting performance models to predict the performance of the whole population of each of our subject systems. We quantify the difference between the predicted performance and the measured performance by means of the *error rate* for all configurations $c \in \mathcal{C}$ as follows:

$$error_c = \frac{|measured_c - predicted_c|}{measured_c} \tag{7}$$

where $predicted_c$ is the predicted performance of configuration c and $measured_c$ the measured performance of configuration c. Lower error rates indicate a higher prediction accuracy and, thus, are better. We further determine the mean error rate of the whole population:

$$\overline{error} = \frac{\sum_{c \in \mathcal{C}} error_c}{|\mathcal{C}|} \tag{8}$$

Note that we compute the error rate based on predictions for the whole population, including configurations from the sample set. The background is that we use regression learning as machine-learning technique, which may produce imperfect predictions even for configurations from the sample set, and we would like to take that into account.

Further note that, initially, we compared the performance distributions of the sample sets and the whole population. But, as the similarity of distributions of a sample set and the whole population does not necessarily imply good predictions, we refrain from this evaluation method and decided for the more definitive method of comparing error rates.

To answer RQ_2, we perform the sampling and machine-learning procedures 100 times per experiment run using different seeds for the random number generator, and we compute the variance across the error rates:

$$\widetilde{error} = \mathrm{Var}(\{error_c \mid c \in \mathcal{C}\}) \tag{9}$$

A lower variance indicates a higher robustness (i.e., is better). So, in our experiments, the *independent variables* are the subject systems, the sample sizes, the sampling strategies, and the random seeds for the random number generator. To rule out influences of different sample sizes, we selected the same sample sizes for (diversified) distance-based, (randomized) solver-based, and random sampling such that their size equals the size for t-wise sampling with t=1, t=2, and t=3[4]. The *dependent variables* are, for RQ_1, the mean error rates of the performance predictions (i.e., $\overline{error}$) and, for RQ_2, the variance of the error rates of the performance predictions on the whole population (i.e., $\widetilde{error}$).

For both research questions, we perform a standardization on the error rates when considering different subject systems, to be able to answer the research questions without considering each subject system separately. For RQ_1, we use a Kruskal-Wallis test [29] to identify for every sample size t=1, t=2, and t=3 if the error rates of, at least, two sampling strategies differ significantly ($p < 0.05$). As proposed by Arcuri and Briand [30], we then perform pair-wise and one-sided Mann-Whitney U tests [31] to identify which sampling strategy leads to significant lower error rates than others. In addition to testing for statistical significance, we determine the effect size using the $\hat{A}_{12}$ measure by Vargha and Delaney [32]. Values of $\hat{A}_{12}$ of more than 0.56, 0.64, and 0.71 indicate small, medium, and large effect sizes, respectively.

To answer RQ_2, we use Levene's test [33] to identify whether the variances of, at least, two sampling strategies differ significantly from each other. If this is the case, we perform a pair-wise comparison using one-sided F-tests [34] to identify the sampling strategy with the lower variance.

Technically, we implemented the (diversified) distance-based sampling strategy on top of the tool SPL CONQUEROR[5] and compared it with the implementations of t-wise sampling, (randomized) solver-based sampling, and random sampling of SPL CONQUEROR. t-wise sampling corresponds to the optimized t-wise strategy by Siegmund et al. [4]. For random sampling, SPL CONQUEROR selects randomly distributed configurations from the whole population, which guarantees a uniform distribution of configurations across the configuration space. That is, for the purpose of computing a baseline (for $\overline{error}$ and $\widetilde{error}$), we follow the non-scalable random sampling: we derive the whole population (i.e., all valid configurations) first, which is necessary to answer RQ1. Then, we randomly draw configurations from the whole population to the sample set. Other random sampling strategies such as the Monte Carlo method or BDD-based sampling are not suitable, because of the disadvantages mentioned in Section II-B. This design decision allows us to maximize internal validity, but requires to acquire the whole population. For larger subject systems, we required in total more than a week of measurement per subject system. For (randomized) solver-based sampling, we used the Z3 solver [35], which allows us to set a random seed. Specifying different random seeds influences the variable-selection heuristics and, thus, determines the location of the sample set in the configuration space. We performed each sampling 100 times with different random seeds from 1 to 100.

[4]In t-wise sampling, the size can be chosen only by using t, whereas in distance-based, solver-based, and random sampling, any positive number in $[1, |\mathcal{C}|]$ can be used for the specification of the sample size.

[5]http://www.fosd.de/SPLConqueror/

TABLE I: Overview of the subject systems including domain, number of valid configurations ($|\mathcal{C}|$), number of configuration options ($|\mathcal{O}|$), and the performance metric to be predicted.

	Domain	$\lvert\mathcal{C}\rvert$	$\lvert\mathcal{O}\rvert$	Performance
7Z	File archive utility	68 640	44	Compression time
BDB-C	Embedded database	2 560	18	Response time
DUNE	Multigrid solver	2 304	32	Solving time
HIPA^{cc}	Image processing	13 485	54	Solving time
JAVAGC	Garbage collector	193 536	39	Time
LLVM	Compiler infrastructure	1 024	11	Compilation time
LRZIP	File archive utility	432	19	Compression time
POLLY	Code optimizer	60 000	40	Runtime
VP9	Video encoder	216 000	42	Encoding time
X264	Video encoder	1 152	16	Encoding time

C. Subject Systems

In our experiments, we consider 10 real-world configurable software systems from different domains and of different sizes. We measured all configurations of all subject systems (i.e., the whole population) between 5 to 10 times until reaching a standard deviation of less than 10 %, to control measurement bias. In total, the measurements took multiple years of CPU time. In Table I, we provide an overview of the subject systems. Note, as we needed the whole population for every subject system to perform random sampling, the sizes of configuration spaces of potential subject systems was limited, which is not a limitation of distance-based sampling. We provide the variability models and the measurements of the subject systems on our supplementary website. Next, we describe the subject systems in more detail.

7-ZIP (7Z) is a file archiver written in C++. Configuration options include different compression methods, different sizes of the dictionary, and the use of single- or multithreading. We used version 9.20 of 7-ZIP and measured the compression time of the Canterbury corpus[6] on an Intel Xeon E5-2690 and 64 GB RAM (Ubuntu 16.04).

BERKELEYDB-C (BDB-C) is an embedded database engine written in C. We consider configuration options defining, for example, the page and cache size or the use of encryption. We measured the time of version 4.4.20 to answer different read and write queries on a machine with an Intel Core 2 Quad CPU 2.66 GHz and 4 GB RAM (Windows Vista).

DUNE MGS (DUNE) is a geometric multigrid solver for partial differential equations based on the DUNE framework [36]. As configuration options, we consider different algorithms for smoothing and different numbers of pre-smoothing and post-smoothing steps to solve Poisson's equation. We performed all measurements with version 2.2 on an Intel i5-4570 and 32 GB RAM (Ubuntu 13.04).

HIPA^{cc} SOLVER (HIPA^{cc}) is an image processing framework written in C++. We included, for instance, different numbers of pixels calculated per thread and different types of memory (e.g., texture, local) as configuration options. We measured the runtime for solving partial differential equations on an nVidia Tesla K20 with 5 GB RAM and 2 496 cores (Ubuntu 14.04).

[6]https://corpus.canterbury.ac.nz/

JAVAGC is the garbage collector of the Java VM, which provides several configuration options, such as disabling the explicit garbage collection call, modifying the adaptive garbage collection boundary, and modifying the policy size. We measured the garbage collection time of Java 1.8 to execute the DaCapo benchmark suite[7] on a cluster with an Intel Xeon E5-2690 and 64 GB RAM (Ubuntu 14.04).

LLVM is a popular compiler infrastructure written in C++. Configuration options that we considered concern code optimization, such as enabling inlining, jump threading, and dead code elimination. We measured the compile time (using the Clang frontend) of version 2.7 for executing the opt-tool benchmark on an AMD Athlon64 Dual Core, 2 GB RAM (Debian GNU/Linux 6).

LRZIP is a file compression tool. We consider configuration options that define, for instance, the compression level and the use of encryption. We used the uiq2[8] generator to generate a file (632 MB), and we measured the time for compressing this file with version 0.600 on a machine with AMD Athlon64 Dual Core, 2 GB RAM (Debian GNU/Linux 6).

POLLY is a loop optimizer that rests on top of LLVM. POLLY provides various configuration options that define, for example, whether code should be parallelized or the choice of the tile size. We used POLLY version 3.9, LLVM version 4.0.0, and Clang version 4.0.0. As benchmark, we used the `gemm` program from `polybench` and measured its runtime on an Intel Xeon E5-2690 and 64 GB RAM (Ubuntu 16.04).

VPXENC (VP9) is a video encoder that uses the VP9 video coding format. It offers different configuration options, such as adjusting the quality, the bitrate of the coded video, and the number of threads to use. We measured the encoding time of 2 seconds from the Big Buck Bunny trailer on an Intel Xeon E5-2690 and 64 GB RAM (Ubuntu 16.04).

X264 is a video encoder for the H.264 compression format. Relevant configuration options included the number of reference frames, enabling or disabling the default entropy encoder, and the number of frames for ratecontrol and lookahead. We have measured the time to encode the Sintel trailer (734 MB) on an Intel Core Q6600 with 4 GB RAM (Ubuntu 14.04).

V. RESULTS

In Section V-A, we present the results regarding RQ_1 and in Section V-B the results regarding RQ_2. In Section V-C, we discuss further findings, the computation effort, and our optimization. In Section V-D, we discuss threats to validity.

A. Results RQ_1—Prediction Accuracy

In Table II, we show the mean error rates for the different sampling strategies. We show the results of random sampling in the rightmost column. Again, random sampling requires the computation of the whole population and does not scale, but it serves as a base line for our experiments. We mark for each sample-set size the lowest, statistically significant error rate in green. That is, if two strategies perform similarly and have

[7]http://dacapobench.sourceforge.net/

[8]http://mattmahoney.net/dc/uiq/

TABLE II: Error rates of t-wise, (randomized) solver-based, (diversified) distance-based, and random sampling for all 10 subject systems. The bottom row contains the mean value across all subject systems. The best results per subject system and sample set size are highlighted in bold and green iff the Mann-Whitney U test reported a significant difference ($p < 0.05$).

	Coverage-based			Solver-based			Randomized solver-based			Distance-based			Diversified distance-based			Random		
	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$
7z	51.2 %	33.8 %	22.6 %	65.4 %	58.2 %	25.2 %	55.1 %	37.2 %	16.7 %	85.9 %	27.3 %	16.6 %	74.3 %	**16.3 %**	17.2 %	58.2 %	15.1 %	9.9 %
BDB-C	122.9 %	29.0 %	26.5 %	49.5 %	46.8 %	42.0 %	**45.1 %**	46.1 %	18.1 %	320.0 %	75.1 %	15.0 %	237.0 %	**12.7 %**	**9.3 %**	121.3 %	39.1 %	12.2 %
Dune	**15.5 %**	12.5 %	11.4 %	23.6 %	15.1 %	11.8 %	43.3 %	16.8 %	11.2 %	24.4 %	15.2 %	11.4 %	21.5 %	**11.8 %**	**11.0 %**	17.6 %	11.5 %	11.3 %
Hipacc	**26.2 %**	20.5 %	20.5 %	44.8 %	17.2 %	14.7 %	31.9 %	15.7 %	14.2 %	27.9 %	19.0 %	15.3 %	31.5 %	**14.5 %**	**14.0 %**	19.9 %	13.9 %	13.4 %
JavaGC	**36.7 %**	32.1 %	23.7 %	54.2 %	59.3 %	35.8 %	41.9 %	37.8 %	30.2 %	72.9 %	43.8 %	28.2 %	56.0 %	29.9 %	**13.2 %**	55.8 %	13.9 %	12.3 %
LLVM	6.2 %	6.2 %	5.8 %	9.5 %	5.5 %	5.2 %	5.6 %	5.2 %	5.4 %	5.8 %	**5.2 %**	5.3 %	5.9 %	5.3 %	5.2 %	5.6 %	5.2 %	5.2 %
lrzip	**27.2 %**	28.2 %	**13.4 %**	47.3 %	27.3 %	23.9 %	91.5 %	36.0 %	25.0 %	162.5 %	39.7 %	21.9 %	134.2 %	**25.1 %**	18.2 %	62.7 %	18.3 %	15.6 %
Polly	**19.7 %**	12.7 %	**7.3 %**	20.3 %	16.1 %	15.5 %	20.0 %	13.6 %	14.0 %	23.3 %	14.2 %	14.9 %	25.8 %	**10.5 %**	11.8 %	25.1 %	13.0 %	10.3 %
VP9	**100.3 %**	96.3 %	45.3 %	413.0 %	224.2 %	80.8 %	470.2 %	389.1 %	94.5 %	721.9 %	125.0 %	84.5 %	189.8 %	**66.5 %**	**32.0 %**	80.6 %	27.2 %	23.3 %
x264	20.9 %	11.9 %	10.9 %	26.2 %	40.4 %	42.2 %	18.5 %	22.2 %	33.2 %	14.7 %	10.0 %	9.4 %	**12.6 %**	**8.8 %**	**9.0 %**	13.5 %	9.2 %	9.1 %
Mean	**42.7 %**	28.3 %	18.7 %	75.4 %	51.0 %	29.7 %	82.3 %	62.0 %	26.2 %	145.9 %	37.4 %	22.2 %	78.9 %	**20.1 %**	**14.1 %**	46.0 %	16.6 %	12.3 %

no statistically significant difference, we do not mark them. Additionally, we provide the mean error rate (bottom row) over all subject systems.

There are several observations: Diversified distance-based sampling performs best or similar to all other sampling strategies for t=2 and t=3. Distance-based sampling without optimization produces partially good results for some systems (e.g., for 7Z and LLVM), but is outperformed for other systems (e.g., JAVAGC, LRZIP, and VP9).

Solver-based sampling results in inaccurate performance models for most subject systems and sample-set sizes. Randomized solver-based sampling performs overall better than solver-based sampling; t-wise sampling perform best when only a very limited number of samples are considered (i.e., t=1).

When we compare the results to random sampling, we make two observations. First, it seems that a diverse coverage of the configuration (by random selection) yields most accurate performance models, especially for systems with many configurations (e.g., 7Z, JAVAGC, and VP9). Second, we observe that the error rates of diversified distance-based sampling often come close to the base line of random sampling, especially when the size of the sample set increases.

When performing Kruskal-Wallis tests for all sample sizes (t=1, t=2, and t=3), we observe p values less than 0.05 (shown on our supplementary website), indicating that, at least, two sampling strategies differ significantly for each sample size. To identify these sampling strategies, we apply one-sided Mann-Whitney U tests pair-wisely and, if significant ($p < 0.05$), report the effect sizes in Table III. Specifically, we test whether the sampling strategy of the row in Table III has a significantly lower error rate than the sampling strategy of the column. The first row shows that t-wise sampling leads to significantly lower error rates than solver-based sampling, with a small effect size for all sample sizes, and to significantly lower error rates than distance-based sampling for t=1. In the fourth row, we see that distance-based sampling leads to lower error rates than t-wise sampling for t=2 and t=3, with a small effect sizes. Distance-based sampling has also lower error rates than solver-based sampling for t=2 and t=3 with a small effect size and t=3 with a medium effect size. Solver-based sampling performs significantly better than distance-based sampling for t=1, which is also negligible due to the small effect size. Randomized solver-based sampling performs significantly better than distance-based sampling for t=1 with small effect size.

When comparing the error rates of diversified distance-based sampling with t-wise sampling, randomized solver-based sampling, and solver-based sampling, we see that t-wise sampling, solver-based sampling, and randomized solver-based sampling lead to higher error rates for t=2 and t=3. The effect size in comparison to t-wise sampling is large for diversified distance-based sampling. Moreover, diversified distance-based sampling performs better than solver-based and randomized solver-based sampling with large effect sizes. Comparing diversified distance-based sampling to random sampling, we see that random sampling has significantly lower error rates with small to medium effect sizes. This result indicates that we can reach nearly the same low error rates using distance-based sampling as the computationally intractable random sampling.

> *Summary*: Diversified distance-based sampling outperforms all other sampling strategies for t=2 and t=3, almost reaching the accuracy of the base line of random sampling, but without relying on the whole population. For small sample sets (t=1), t-wise sampling is superior.

B. Results RQ_2—Robustness

Based on 100 runs per experiment, we obtained a distribution of mean error rates for each sampling strategy, which we further aggregated to compute their variances, $\widetilde{error}$. We compared the variances as follows: First, we performed Levene's test (shown on our supplementary website), which checks the existence of significantly different variances between, at least, two sampling strategies over all sampling sizes. Then, we performed pair-wisely one-sided F-tests. We show the results in Table IV. In the second row, we can see that randomized solver-based sampling has a significantly lower variance than

TABLE III: p values from a one-sided pair-wise Mann-Whitney U test, where we tested pair-wisely whether the population from the row is smaller than the population from the column for different sample sizes after standardization. The effect size is included for every significant result ($p < 0.05$), where we consider differences as small, medium, and large when $\hat{A}_{12}$ is over 0.56, 0.64, and 0.71, respectively.

Mann-Whitney U test [p value ($\hat{A}_{12}$)]																		
	Coverage-based			Solver-based			Randomized solver-based			Distance-based			Diversified distance-based			Random		
	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$
Coverage-based				10^{-25} (0.63)	10^{-12} (0.59)	10^{-09} (0.57)	10^{-11} (0.59)	10^{-04} (0.54)	10^{-06} (0.56)	10^{-55} (0.70)			10^{-28} (0.64)					
Solver-based										10^{-04} (0.54)								
Randomized solver-based				10^{-06} (0.56)	10^{-06} (0.56)	10^{-06} (0.56)				10^{-17} (0.61)			10^{-05} (0.55)					
Distance-based		10^{-07} (0.56)	10^{-11} (0.58)		10^{-19} (0.61)	10^{-30} (0.65)		10^{-10} (0.58)	10^{-25} (0.63)									
Diversified distance-based		10^{-151} (0.84)	10^{-107} (0.78)		10^{-147} (0.83)	10^{-119} (0.80)		10^{-141} (0.83)	10^{-146} (0.83)	10^{-07} (0.57)	10^{-78} (0.74)	10^{-49} (0.69)						
Random	10^{-03} (0.54)	10^{-196} (0.89)	10^{-155} (0.84)	10^{-30} (0.65)	10^{-175} (0.86)	10^{-151} (0.84)	10^{-15} (0.60)	10^{-175} (0.86)	10^{-187} (0.88)	10^{-50} (0.69)	10^{-119} (0.80)	10^{-83} (0.75)	10^{-27} (0.64)	10^{-11} (0.58)	10^{-10} (0.58)			

TABLE IV: p values from a one-sided pair-wise F-test, where we tested pair-wisely whether the variances of the population from the row is smaller than the one from the column, for different sample sizes after standardization.

F-test (p value)															
	Solver-based			Randomized solver-based			Distance-based			Diversified distance-based			Random		
	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$
Solver-based															
Randomized solver-based	10^{-20}	10^{-46}	10^{-45}				10^{-09}	10^{-16}	10^{-25}						
Distance-based	10^{-04}	10^{-10}	10^{-05}												
Diversified distance-based	10^{-36}	10^{-195}	10^{-114}	10^{-04}	10^{-67}	10^{-21}	10^{-21}	10^{-132}	10^{-81}					10^{-09}	
Random	10^{-100}	10^{-137}	10^{-141}	10^{-37}	10^{-32}	10^{-34}	10^{-74}	10^{-83}	10^{-105}	10^{-21}		10^{-03}			

distance-based sampling. In the third row, we can see that distance-based sampling has a significantly lower variance than solver-based sampling on all sample sizes. The last row shows that random sampling has the lowest variance. When it comes to the diversified variant of distance-based sampling, it leads to a significantly lower variance compared to plain distance-based sampling. The optimization has a significantly lower error rate than random sampling for t=2, which requires, however, a whole-population analysis. Regarding randomized solver-based sampling, the variance of diversified distance-based sampling is significantly lower for all sample sizes.

We explain these observations as follows: Solver-based sampling either relies also on a random seed or produces a deterministic set of configurations (not considered here). In the case of random seeds, the seed defines the first solution found by the solver. Since neighboring solutions are produced very likely in subsequent solver calls, the sample set will be locally clustered around the first solution. Hence, for different seeds, different clusters are sampled such that high variations occur in the error rate depending on the representativeness of the cluster. Randomized solver-based sampling avoids building clusters, and thus the variance is significantly lower than solver-based sampling. Distance-based sampling uses also a solver to obtain configurations, but only with a certain distance. Having multiple configurations with the same distance might lead to clusters similar to solver-based sampling. This is why we observe a lower variance than for solver-based sampling (we might obtain one cluster per distance, but not a single cluster in total), but a higher variance than random sampling, randomized solver-based sampling, and diversified distance-based sampling. Reducing clustering, diversified distance-based sampling yields even lower variances. As diversified distance-based sampling avoids clusters such as randomized solver-based sampling, the variances are rather similar. Hence, we conclude that our optimization of distance-based sampling is effective to increase the variety of configurations and thus lowers the variance of the prediction error.

> *Summary*: Diversified distance-based sampling is more robust than other sampling strategies except for random sampling, but at the benefit of lower computational effort.

C. Discussion

Computational effort: The computational effort of distance-based sampling and its diversified variant is lower than the effort of random sampling because we include every configuration found by the solver into the sample set instead of enumerating all valid configurations and discarding later a large part (i.e., configurations not used in the sample set) of it (see Section II-B). Moreover, to reduce computational effort,

we do not perform an expensive optimization as in t-wise sampling (i.e., minimizing the set of selected configuration options that are not of interest in the current configuration), but rather include additional constraints to the constraint solver that define, for example, the number of selected configuration options. The computational effort of randomized solver-based sampling is high, since the solver has to be reinitialized from scratch to permute (1) the constraints, (2) the literals, and (3) initial assignment. Whereas random sampling and the randomized solver-based sampling need more than 10 hours to acquire the sample set, the other sampling strategies are some orders of magnitude faster. We provide all times for computing all sample sets on the supplementary website.

Probability distributions: In our experiments, we used exclusively the discrete uniform distribution for selecting distances, but our algorithm can be parameterized (*probabilityDistr*). Preliminary experiments with binomial and geometric distributions suggest that a uniform coverage of the configuration space is superior, though (see the supplementary website). Testing further distributions is an avenue of further work.

Diversity: Comparing plain distance-based sampling with diversified distance-based sampling, we observe that optimizing for diversity indeed pays off in terms of prediction accuracy and robustness. Aiming at diversity is optimal in a black-box strategy, where no domain knowledge is available. However, diversified distance-based sampling only pays off with larger sample sizes, because the smaller sample sizes do not suffice to cover all configuration options, at least, once for each distance.

D. Threats to Validity

Internal validity: To rule out errors in our implementation of (diversified) distance-based sampling, we have thoroughly tested it. We verified that the produced sample set follows the given distribution of the configuration distances. We found deviations only when all configurations of a specific distance were already selected, which occurred only in few cases.

External validity: To increase external validity, we have selected software systems from different domains. We consider software systems ranging from systems with 432 configurations to systems with 216 000 configurations. We have excluded larger systems because it would be computationally infeasible for t-wise sampling and random sampling [24].

The selection of the machine-learning technique to learn a performance model may threaten external validity. We used deliberately the same machine-learning technique for all experiments to increase internal validity. But, other machine-learning techniques have other strategies to derive information from the sample set and, thus, may lead to different results. In a parallel line of experiments, we compared six different machine-learning techniques and observed that multiple regression is often as accurate as classification and regression trees and random forests[9], which are often used for learning performance models of configurable software systems. So, we are confident that our results generalize to other machine-learning techniques.

Another threat to validity is the choice of the constraint solver, as different solvers adopt different search heuristics. This, however, might represent a further reason not to rely on solver-based sampling, as this way the generation of the sample set remains intransparent. For instance, we observed even worse results for solver-based sampling when using the solver of the Microsoft solver foundation.

[9] https://github.com/se-passau/Distance-Based_Data/

VI. Conclusion

Measuring every configuration of a software system to identify the performance-optimal configuration is often unfeasible due to the sheer size of the configuration space. Addressing this problem, machine learning is used to predict the performance of individual (or all) configurations by deriving information from a small and representative sample set. Finding a tractably small and representative set of configurations is an important but difficult task. To this end, different sampling strategies, such as t-wise sampling, solver-based sampling, and random sampling have been proposed, which focus on different aspects with different strengths and weaknesses. To address the weaknesses, we propose distance-based sampling, which is based on a user-defined discrete probability distribution and a distance metric. The key idea is that distance-based sampling spreads the selected configurations across the configuration space based on a given probability distribution while not relying on an expensive analysis of the whole population. To compare distance-based sampling with the state of the art, we learn performance models for 10 real-world software systems using 6 different sampling strategies and compare the accuracy of the performance models.

Our results demonstrate that distance-based sampling, when used in combination with a diversity optimization, leads to significantly lower error rates than state-of-the-art strategies, especially for larger sample sizes (t=2, t=3), and the predictions are more stable than solver-based sampling with respect to multiple runs using different random seeds. Our results demonstrate that, based on a distance metric and a probability distribution, we can effectively sample diverse configurations across the configuration space and without the need for a whole-population analysis, which makes random sampling unfeasible for highly configurable software systems. This work provides a new view on sampling based on probability distributions and paves the way for further research in this area. For instance, using other metrics or distributions could lead more accurate predictions or improve the prediction robustness.

VII. Acknowledgments

Grebhahn's work is supported by the DFG under the contract AP 206/7. Apel's work is supported by the DFG under the contracts AP 206/6, AP 206/7, and AP 206/11. Siegmund's work is supported by the DFG under the contracts SI 2171/2 and SI 2171/3-1. Guos work is supported by NSFC (61772200), Shanghai Pujiang Talent Program (17PJ1401900), Shanghai NSF (17ZR1406900), and Alibaba Group.

REFERENCES

[1] T. Xu, L. Jin, X. Fan, Y. Zhou, S. Pasupathy, and R. Talwadker, "Hey, you have given me too many knobs!: Understanding and dealing with over-designed configuration in system software," in *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2015, pp. 307–319.

[2] C. H. P. Kim, C. Kästner, and D. S. Batory, "On the modularity of feature interactions," in *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*. ACM, 2008, pp. 23–34.

[3] S. Kolesnikov, N. Siegmund, C. Kstner, A. Grebhahn, and S. Apel, "Tradeoffs in modeling performance of highly-configurable software systems," *Software and Systems Modeling*, 2018, online first: http://rdcu.be/GzLq.

[4] N. Siegmund, S. S. Kolesnikov, C. Kästner, S. Apel, D. S. Batory, M. Rosenmüller, and G. Saake, "Predicting performance via automated feature-interaction detection," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 167–177.

[5] N. Siegmund, A. Grebhahn, S. Apel, and C. Kästner, "Performance-influence models for highly configurable systems," in *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2015, pp. 284–294.

[6] J. Guo, K. Czarnecki, S. Apel, N. Siegmund, and A. Wasowski, "Variability-aware performance prediction: A statistical learning approach," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 301–311.

[7] V. Nair, T. Menzies, N. Siegmund, and S. Apel, "Using bad learners to find good configurations," in *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2017, pp. 257–267.

[8] V. Nair, Z. Yu, T. Menzies, N. Siegmund, and S. Apel, "Finding faster configurations using FLASH," *IEEE Transactions on Software Engineering*, 2018, online first: https://arxiv.org/abs/1801.02175/.

[9] C. Henard, M. Papadakis, M. Harman, and Y. Le Traon, "Combining multi-objective search and constraint solving for configuring large software product lines," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 2015, pp. 517–528.

[10] J. Guo, D. Yang, N. Siegmund, S. Apel, A. Sarkar, P. Valov, K. Czarnecki, A. Wasowski, and H. Yu, "Data-efficient performance learning for configurable systems," *Empirical Software Engineering*, vol. 23, no. 3, pp. 1826–1867, 2018.

[11] V. Gogate and R. Dechter, "A new algorithm for sampling CSP solutions uniformly at random," in *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*. Springer, 2006, pp. 711–715.

[12] S. Chakraborty, D. J. Fremont, K. S. Meel, S. A. Seshia, and M. Y. Vardi, "Distribution-aware sampling and weighted model counting for sat," in *Proceedings of the Association for the Advancement of Artificial Intelligence (AAAI)*. AAAI Press, 2014, pp. 1722–1730.

[13] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "IPOG/IPOG-D: Efficient test generation for multi-way combinatorial testing," *Software Testing, Verification and Reliability*, vol. 18, no. 3, pp. 125–148, 2008.

[14] M. F. Johansen, Ø. Haugen, and F. Fleurey, "An algorithm for generating t-wise covering arrays from large feature models," in *Proceedings of the International Software Product Line Conference (SPLC)*. ACM, 2012, pp. 46–55.

[15] D. Marijan, A. Gotlieb, S. Sen, and A. Hervieu, "Practical pairwise testing for software product lines," in *Proceedings of the International Software Product Line Conference (SPLC)*. ACM, 2013, pp. 227–235.

[16] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer, "Scalable analysis of variable software," in *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2013, pp. 81–91.

[17] A. von Rhein, J. Liebig, A. Janker, C. Kästner, and S. Apel, "Variability-aware static analysis at scale: An empirical study," *ACM Transactions on Software Engineering and Methodology*, vol. 27, no. 4, pp. 18:1–18:33, 2018.

[18] S. Apel, D. S. Batory, C. Kästner, and G. Saake, *Feature-Oriented Software Product Lines – Concepts and Implementation*. Springer, 2013.

[19] T. Y. Chen, H. Leung, and I. K. Mak, "Adaptive random testing," in *Proceedings of the Asian Computing Science Conference (ASIAN)*. Springer, 2004, pp. 320–329.

[20] J. Oh, D. S. Batory, M. Myers, and N. Siegmund, "Finding near-optimal configurations in product lines by random sampling," in *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2017, pp. 61–71.

[21] S. She, "Feature model synthesis," Ph.D. dissertation, University of Waterloo, Canada, 2013.

[22] A. Sarkar, J. Guo, N. Siegmund, S. Apel, and K. Czarnecki, "Cost-efficient sampling for performance prediction of configurable systems," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 342–352.

[23] R. Tartler, D. Lohmann, C. Dietrich, C. Egger, and J. Sincero, "Configuration coverage in the analysis of large-scale system software," in *Proceedings of the Workshop on Programming Languages and Operating Systems (PLOS)*. ACM, 2011, pp. 2:1–2:5.

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

[25] E. F. Krause, *Taxicab Geometry: An Adventure in Non-Euclidean Geometry*. Courier Corporation, 1986.

[26] D. Benavides, S. Segura, and A. R. Cortés, "Automated analysis of feature models 20 years later: A literature review," *Information Systems*, vol. 35, no. 6, pp. 615–636, 2010.

[27] A. Biere, M. Heule, and H. van Maaren, *Handbook of Satisfiability*. IOS press, 2009, vol. 185.

[28] J. Liang, V. Ganesh, K. Czarnecki, and V. Raman, "SAT-based analysis of large real-world feature models is easy," in *Proceedings of the International Software Product Line Conference (SPLC)*, 2015, pp. 91–100.

[29] W. H. Kruskal and W. A. Wallis, "Use of ranks in one-criterion variance analysis," *Journal of the American Statistical Association*, vol. 47, no. 260, pp. 583–621, 1952.

[30] A. Arcuri and L. C. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Proceedings of the International Conference on Software Engineering, (ICSE)*. ACM, 2011, pp. 1–10.

[31] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *Annals of Mathematical Statististics*, vol. 18, pp. 50–60, 1947.

[32] A. Vargha and H. D. Delaney, "A critique and improvement of the "CL" common language effect size statistics of mcgraw and wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

[33] H. Levene, "Robust tests for equality of variances," in *Contributions to Probability and Statistics. Essays in Honor of Harold Hotelling*. Stanford University Press, 1961, pp. 279–292.

[34] G. W. C. Snedecor and G. William, "Statistical methods," International Statistical Institute, Tech. Rep., 1989.

[35] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2008, pp. 337–340.

[36] M. Blatt and P. Bastian, "The iterative solver template library," in *Proceedings of the Workshop on State-Of-The-Art in Scientific and Parallel Computing (PARA)*. Springer, 2007, pp. 666–675.