Performance-Influence Models for Highly Configurable Systems

Norbert Siegmund[†], Alexander Grebhahn[†], Sven Apel[†], Christian Kästner[‡]
[†]University of Passau, Germany [‡]Carnegie Mellon University, USA

ABSTRACT

Almost every complex software system today is configurable. While configurability has many benefits, it challenges performance prediction, optimization, and debugging. Often, the influences of individual configuration options on performance are unknown. Worse, configuration options may interact, giving rise to a configuration space of possibly exponential size. Addressing this challenge, we propose an approach that derives a *performance-influence model* for a given configurable system, describing all *relevant* influences of configuration options and their interactions. Our approach combines machine-learning and sampling heuristics in a novel way. It improves over standard techniques in that it (1) represents influences of options and their interactions explicitly (which eases debugging), (2) smoothly integrates binary and numeric configuration options for the first time, (3) incorporates domain knowledge, if available (which eases learning and increases accuracy), (4) considers complex constraints among options, and (5) systematically reduces the solution space to a tractable size. A series of experiments demonstrates the feasibility of our approach in terms of the accuracy of the models learned as well as the accuracy of the performance predictions one can make with them.

Categories and Subject Descriptors: C.4 [**Performance of Systems**]: Measurement techniques; D.2.13 [**Reusable Software**]: Domain engineering

Keywords: Performance-influence models, sampling, machine learning

1. INTRODUCTION

End-users, developers, and administrators are often overwhelmed with the possibilities to *configure* a software system. In most systems today, including databases, Web servers, video encoders, and compilers, hundreds of configuration options can be combined, each potentially with distinct functionality and different effects on quality attributes. The sheer size of the configuration space and complex constraints among configuration options make it difficult to find a configuration that performs as desired, with the consequence that many users stick to default configurations or only try changing an option here or there. This way, the significant optimization potential already built in many of our modern software systems remains untapped. Even domain experts and the developers themselves often do not (fully) understand the performance influences of all configuration options and their combined influence when they interact.

Our goal is to build performance-influence models (and models of other measurable quality attributes, such as energy consumption) that describe how configuration options and their interactions influence the performance of a system (e.g., throughput or execution time of a benchmark). Performance-influence models are meant to ease *understanding, debugging, and optimization* of highly configurable software systems. For example, an end user may use an optimizer to identify the best performing configuration under certain constraints (e.g., encryption needs to be enabled) from the model; a database administrator may use it to determine the influence of certain configuration options and how they interact; and a developer may compare an inferred performance-influence model with her own mental model to check whether the system behaves as expected.

Our approach is to infer a performance-influence model for a given configurable system in a black-box manner from a series of *measurements* of a set of *sample* configurations using *machine learning*. That is, we benchmark a given system multiple times in different configurations and learn the influence of individual configuration options and their interactions from the differences among the measurements. Our approach addresses several challenges:

- We are facing huge configuration spaces that explode with the number of configuration options. At the same time, we can sample and measure only a relatively small number of configurations (several hundred or thousand measurements), so we better select our *sample* purposefully.
- Often, configuration spaces are highly constrained, such that already random sampling is challenging, because most random samples do not satisfy the constraints.
- Binary and numeric options in configurable systems typically have different characteristics that require dedicated sampling and learning strategies. Of course, binary and numeric options can interact, too [1, 15].
- If available, domain knowledge should be exploited for sampling and learning. For example, if we know that performance likely decreases quadratically with a certain nu-

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3675-8/15/08...
http://dx.doi.org/10.1145/2786805.2786845

meric option, we would like to incorporate that knowledge for creating better models with fewer measurements.

To address these challenges, we propose a novel approach of sampling and learning that takes the characteristics of binary and numeric configuration options and their interactions into account. Specifically, we propose a hybrid sampling strategy that uses different experimental designs for numeric options and sampling heuristics tailored for binary options, grounded in their respective characteristics in practical systems. Furthermore, we use a learning mechanism that can cope with both and that additionally incorporates domain knowledge about an option's influence, if available. Our approach differs from prior work on performance optimization in highly configurable systems in that (1) it supports both binary and numeric options, (2) it relies on sampling heuristics that respect constraints among configuration options, and (3) it learns models that can explain performance behavior in terms of individual influences of individual options and their interactions, allowing applications beyond mere local or global optimization.

Our approach is able to build reasonably accurate performance models of configuration spaces of real-world systems, including compilers, multi-grid solvers, and video encoders. In a series of experiments with configurable systems with up to 10^{31} configurations, we demonstrate that few measurements are sufficient to build fairly accurate models (19 % prediction error, on average). The performance-influence models learned by our approach can explain the performance variation among configurations with a few dozen terms describing the influence of individual options and another dozen terms describing interactions. Finally, while accuracy is important, simple models are important, too. Views on a performance-influence model can be used to isolate influences of individual options and their interactions. This way, we found two performance bugs in the POLLY extension of the LLVM compiler framework.[1]

In summary, our contributions are the following:

- We introduce a novel kind of performance-influence models describing the influence of binary and numeric configuration options and their interactions on performance.
- We propose a learning algorithm based on stepwise feature selection that learns performance models focusing on the most important factors influencing performance.
- We develop a sampling approach that smoothly integrates binary sampling using heuristics and numeric sampling using experimental designs.
- We demonstrate practicality and feasibility of our approach with several synthetic benchmarks as well as an empirical evaluation on six real-world systems.
- We make an implementation of our approach on top of the tool SPL CONQUEROR as well as all measurement results and configuration models available online, including data of several months of performance measurements, supporting others in replicating our experiments and evaluating related approaches: `http://www.fosd.de/SPLConqueror/`.

2. PERFORMANCE-INFLUENCE MODELS

A performance-influence model consists of several terms that describe the performance of a configuration based on the values of configuration options. Individual terms may refer to a single option, describing the influence of that option, or to multiple options, describing an interaction. Before we explain how influence models are obtained by sampling and learning in Section 3, let us outline the underlying concepts and how we unify binary and numeric options.

Assume $\mathcal{O}$ is the set of all configuration options, including numeric and binary options, and $\mathcal{C}$ the set of all configurations. We model a configuration $c \in \mathcal{C}$ as a function $c : \mathcal{O} \to \mathbb{R}$ assigning a (user-)selected value to every option. For a binary option o, $c(o) = 1$ if the corresponding option is selected and $c(o) = 0$ otherwise. For a numeric option o, $c(o)$ returns a number in the value range of that option. Without loss of generality, we normalize the value range of all numeric configuration options to the interval [0..1], as common in machine learning.

Conceptually, a performance-influence model is simply a function from configurations to a performance measure $\Pi : \mathcal{C} \to \mathbb{R}$, where performance can be any measurable property that produces interval-scaled data. The model is described as a sum of terms over configuration values. Individual terms of the performance model can have different shapes, including $n \cdot c(X)$, $n \cdot c(X)^2$, or $n \cdot \sqrt{c(X)} \cdot c(Y)$. For illustration, consider a configurable database management system with the options *encryption (E)*, *compression (C)*, *statistics (S)*, *page size (P)*, and *DB size (D)* and a corresponding performance-influence model:

$$\Pi(c) = 50 + \overbrace{20{\cdot}c(E)}^{\phi_E} + \overbrace{15{\cdot}c(C)}^{\phi_C} + \overbrace{5{\cdot}c(S)}^{\phi_S} - \overbrace{0.5{\cdot}c(P)}^{\phi_P} + \overbrace{1.5{\cdot}c(D)^2}^{\phi_D} - \underbrace{10{\cdot}c(E){\cdot}c(C)}_{\Phi_{E,C}} + \underbrace{0.3{\cdot}c(E){\cdot}c(P)}_{\Phi_{E,P}} + \underbrace{2.5{\cdot}c(E){\cdot}c(C){\cdot}c(D)}_{\Phi_{E,C,D}}$$

In general, we distinguish between terms that refer to a single option o, denoted as ϕ_o, and terms that refer to multiple options $i..j$, denoted as $\Phi_{i..j}$. The former describes the performance influence of an option (e.g., $\phi_E = 20{\cdot}c(E)$ denotes the influence of encryption), the latter describes the influence of a *performance interaction* among multiple options (e.g., $\Phi_{E,C} = 10{\cdot}c(E){\cdot}c(C)$ denotes the influence of the interaction between encryption and compression).

All performance-influence models are of the following form:

$$\Pi(c) = \beta_0 + \sum_{i \in \mathcal{O}} \phi_i(c(i)) + \sum_{i..j \in \mathcal{O}} \Phi_{i..j}(c(i)..c(j)) \quad (1)$$

β_0 represents a minimum, constant base performance shared by all configurations, as determined during learning; $\sum_{i \in \mathcal{O}} \phi_i(c(i))$ represents the sum of the influences of all individual options; $\sum_{i..j \in \mathcal{O}} \Phi_{i..j}(c(i)..c(j))$ is the sum of the influences of all interactions among all options.

This structure allows us to easily see the influence of an individual option from the model. In our example, we can see that encryption (E), if enabled, slows down our example system by 20 (seconds) and that encryption interacts with compression (i.e., their combined slowdown is less severe than expected from their individual effects, since compressed data are quicker to encrypt, which illustrates that performance interactions are not necessarily bad).

Combining Binary and Numeric Options. Our approach is flexible enough to incorporate both binary and numeric options, but still allows us to handle them separately. Although we could discretize the values of a numeric option into a number of mutually exclusive auxiliary binary options, this way, we would loose the relation between the binary op-

[1] `http://polly.llvm.org/`

tions that make up the corresponding numeric option, such that we cannot fit a function and interpolate. By contrast, converting a binary option into a numeric option, we would interpret a categorical variable as an interval variable, assigning a possibly wrong meaning to the range between 0 and 1 and invalidating the mathematical methods we use in our approach. Next, we explain how to obtain the ϕ and Φ terms of Equation 1, by means of learning and sampling.

3. LEARNING INFLUENCE MODELS

Although performance-influence models are simple in their structure, it is challenging to actually determine the relevant influencing factors (i.e., terms) with a *reasonable* number of measurements. The key problem is that we cannot learn the whole influence model in a single step, since it contains a potentially exponential number of terms. As a solution, we incrementally select only the strongest influences regarding prediction accuracy in each iteration of the learning process. A key contribution of our work is thus not to develop a new learning technique, but to select a suitable one and adapt it to the requirements and specifics of configurable systems.

3.1 Overview

We use *stepwise linear regression* to learn the function of a performance-influence model from a sample set of measured configurations. To reduce the dimensionality problem of handling a very large number of options and interactions, we use *forward and backward feature selection* to incrementally learn the model.[2]

We use linear regression because it fits the requirements of the domain of configurable software systems and the practicality requirements for end users. In contrast to many alternative approaches, including classification and regression trees, neuronal nets, and support vector machines, linear regression allows us to learn a formula that can be understood by humans. It also makes it easy to incorporate domain knowledge on the influence of certain options, if available, which can be an effective means to avoid overfitting and underfitting [5]. Finally, we do not need any internal information about the system, but can apply the learning approach in a black-box fashion onto sampled benchmark results.

Linear regression is a common approach to learn how a dependent variable y depends on a number of independent variables x_i in the form

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n + \epsilon$$

Given a learning set of observations for $y, x_1, ..., x_n$, linear regression fits the regression coefficients β_i such that the overall error is reduced, this way, learning a function that explains the observations. We use the measured performance of a configuration as dependent variable y and the configuration values $c(i)$ as the independent variables, thus learning performance models of the form $\beta_0 + \beta_1 c(o_1) + \ldots + \beta_n c(o_n)$ —a performance model of basic ϕ terms without any interactions or nonlinear behavior.

Linear regression can also be used to learn linear coefficients for more complex terms (i.e., to learn non-linear functions) by using computed values as independent variables. For example, we can use $c(p) \cdot c(q)$ as independent variable to learn interactions between p and q, or $c(o)^2$ to learn coefficient for a quadratic function. That is, if we know which single-option terms, interaction terms, or function terms might appear in the performance-influence model, we can learn linear coefficients for all of them, yielding performance-influence functions as in our initial example in Section 2.

[2]Note that the term feature selection used here, is a machine-learning term and must not be confused by the process of selecting features in product-line engineering.

Algorithm 1: Stepwise feature selection

```
   Data: measurements, O
   Result: model
 1 featureSet = ∅, error = ∞
 2 repeat
 3 |   lastError = error
 4 |   bestCandidate = ⊥
 5 |   candidates = generateCandidates(featureSet, O);
 6 |   foreach feature in candidates do
 7 |   |   model = learnFunction(featureSet ∪ {feature},
 8 |   |                         measurements)
 9 |   |   modelError = computeError(model, measurements)
10 |   |   if modelError < error then
11 |   |   |   error = modelError, bestCandidate = candidate
12 |   |   end
13 |   end
14 |   if bestCandidate ≠ ⊥ then
15 |   |   featureSet = featureSet ∪ {feature}
16 |   end
17 until (lastError − error < margin) ∨ (error < threshold);
18 featureSet = backwardStep(featureSet, measurements)
19 return learnFunction(featureSet, measurements);
```

The key challenge of using linear regression is, hence, to identify the relevant terms to be used as independent variables. Next to $|\mathcal{O}|$ basic option terms ($x = c(o)$), there is an exponential number of basic interaction terms ($x = c(o_1) \cdot \ldots \cdot c(o_n)$) and an infinite number of additional function terms over one or multiple options (e.g., polynomials). The problem of too many possible independent variables is known as the curse of dimensionality [5].

Given a set of measurements selected by a specific sampling heuristics (see Section 4), our learning process proceeds iteratively in a process known as *forward feature selection* [4], learning one term at a time, until further improvement (minimizing the error term ϵ) becomes negligible. In each iterative step, we try a number of candidate terms and select the term that provides the biggest improvement for the model. We select candidate terms based on heuristics and domain knowledge, if available, as we will explain in Section 3.3.

3.2 Incremental Learning Algorithm

In Algorithm 1, we sketch the learning algorithm that computes the function representing the performance-influence model from a set of measurements (representing selected configurations and corresponding measurement results). Conceptually, we incrementally compute the relevant feature set, in which each feature represents a term over one or multiple configuration values $c(o)$. Throughout the learning process, the feature set holds the features that have been identified to improve the error rate of the learned model with regard to the measured learning set. For our example model in Section 2, the feature set would eventually include $c(E)$, $c(D)^2$, $c(E) \cdot c(C)$, and so forth.

Starting with an empty feature set, the algorithm selects one feature in each iteration until improvements of model accuracy become marginal or a threshold for expected accuracy is reached (`repeat` loop, Line 2–17). The feature to be added stems from a number of candidate features (their

selection is described in Section 3.3). Using linear regression to compute a model with every candidate (Line 12), we select the candidate that reduces the error rate most.

The algorithm concludes with a backward learning step (backward feature selection; not shown for brevity), in which every feature in the feature set is tested for whether its removal would decrease model accuracy. This can happen if initially a single feature is selected because it best explains the measurements, but it becomes obsolete by other features (e.g., representing interactions) later in the learning process.

3.3 Selecting Candidate Features

A final, critical step is to select candidate features that are explored during the learning process. Much like it is infeasible to learn a model with all features at once, it is infeasible to try all features in each iteration—it is typically not even possible to enumerate all features due to their sheer number. Hence, we have to apply heuristics to select these candidates that are likely to influence performance. A key innovation of our approach is the procedure of determining which candidates we select for learning. We use separate heuristics for the two main factors that result in the huge number of candidate features: interaction and function terms.

Hierarchical Interactions. Conceptually, any combination of options may cause a distinct performance interaction [25], which would render any learning approach useless as there is no common pattern. In practice, however, performance behavior is usually more tractable in that only few interactions contribute substantially to the overall performance. In our previous work, we found that relevant interactions do not emerge randomly among configuration options, but form a hierarchy [25, 26]. That is, three-way interactions (i.e., interactions among three options) build on corresponding two-way interactions among the same set of options. Furthermore, we found, at most, five-way interactions, and the most common interactions were two-way. Thus, based on our experience and following standard practices of machine learning [2], we perform our learning hierarchically: We first start adding only the individual option influences and then add interactions as candidates containing options that have been found already to contribute to performance.

Function Learning. For numeric options, we learn functions over their data ranges. If we know the kind of the function or the polynomial degree, then we generate the corresponding candidates. For example, if we know that the performance influence of a numeric option o is logarithmically, we generate only the feature $\phi_o(\log(c(o)))$ rather than a set of polynomial candidates. Without domain knowledge, we use an array of standard functions (linear, quadratic, logarithmic) that can be activated by the user of our approach. Again, our approach can incrementally increase the complexity of these functions by building new functions from combinations of already learned ones.

4. SAMPLING CONFIGURATION SPACES

Sampling heuristics must satisfy two requirements: First, they have to produce a reasonable number of measurements; heuristics requiring millions of measurements are certainly infeasible. Second, a heuristic must select the configurations that incorporate most of the relevant interactions. That is, we want to find a sweet spot between prediction accuracy and measurement effort. There are several invariants that need to be considered when constructing the learning set:

- **Random sampling:** Choosing configurations randomly from the configuration space is still an open problem for highly-configurable systems. The presence of constraints prohibit off-the-shelf solutions, such as experimental designs or random selection and filtering [17]. Using SAT solving to find valid configurations usually produces only locally clustered solutions in the configuration space [23], although recent attempts try to mitigate that problem [11].
- **Binary and numeric options:** Binary and numeric options have substantially different value ranges resulting in huge differences in the number of measurement points we should select for them. For instance, a numeric option might have a quadratic performance influence when varying its value, which certainly requires more measurements than the constant influence of a binary option when switched on and off.

We address these problems by dividing the configuration space along the two types of configuration options and by applying dedicated sampling heuristics to them.

4.1 Binary-Option Sampling

For binary-option sampling, we use heuristics that we developed in previous work [25]. The goal of these heuristics is to pick configurations to learn the basic influence of each individual binary option and, subsequently, to pick configurations that exhibit two-way interactions.

Option-Wise Sampling (OW). Option-wise sampling selects configurations such that it purposefully avoids interactions. We use a constraint-satisfaction-problem (CSP) solver to find, for each option o, a configuration with as many options disabled as possible under the constraint $c(o) = 1$ and the given constraints among the options. The process is repeated until all options have been included in the learning set, which requires a number of measurements that is linear in the number of binary options.

Negative Option-Wise Sampling (nOW). Instead of minimizing the number of options, negative option-wise sampling aims at maximizing the number of options in a configuration to maximize the number of possible interactions. That is, for each option o, we search a configuration with as many options enabled as possible under the constraints $c(o) = 0$ and the constraints between the options. Much like OW, nOW requires a linear number of measurements.

Pair-Wise Sampling (PW). For pair-wise sampling, we construct a learning set that includes a minimal set of configurations, in which all two-way interactions are present and not confounded with other interactions. That is, for each pair of options q and p, we determine a configuration with as many options disabled as possible under the constraints $c(q) = 1 \wedge c(p) = 1$ and the constraints between the options. Without constraints, PW requires a number of measurements that is quadratic in the number of binary options.

4.2 Numeric-Option Sampling

The science of choosing an appropriate learning set in the presence of numeric options has a long history, and many approaches have been proposed under the umbrella of the

Design of Experiments (or *Experimental Designs*) [22]. The goal of an experimental design is to generate an experimental plan by assigning values to independent variables (numeric configuration options, in our case), such that a given hypothesis can be properly tested.

As a preparation for this work, we surveyed eight standard experimental designs and sorted out those that do not meet our requirements.[3] For example, the D-optimal design requires to enumerate all possible combinations, or other designs, such as the full factorial and the 2^{k-1} design, require a number of measurements that is infeasible in practice. After this preselection, we selected the following experimental designs for our empirical evaluation: Box-Behnken, Plackett-Burman, Central Composite, and Randomization with seed [18]. Due to space constraints, we report only on the best performing designs (Plackett-Burman and Random); the complete set of evaluation results are available on our supplementary Web site.

Plackett-Burman Design. In 1946, Plackett and Burman proposed an experimental design that aims at determining the main effects of an experiment [20]. It minimizes the variance of the estimates of the independent variables while using a limited number of measurements. Wang and Wu extended it to incorporate two-way interactions [28].

A Plackett-Burman design is a specific type of fractional factorial design, which are often used for combinatorial testing [15]. The design specifies *seeds* depending on the number of experiments to be conducted and the number of levels of the input variables. The level specifies the number of distinct values of an independent variable, and the seed defines the pattern at which the different values are varied, which affects also the number of measurements. Since numeric configuration options can have a large number of values, we sample the value range uniformly depending on the chosen level. For a three-level design, we take the minimum, maximum, and center point of the value range. Next, we have to determine the number of measurements to be performed. When we are interested only in the main factors (i.e., only ϕ), we need $n+1$ measurements for n numeric configuration options. If domain knowledge is available and we know that there might be interactions between configuration options, we chose the seed such that more configurations are measured. In Table 1, we show a Plackett-Burman design for a seed ($n = 9$, $l = 3$) defining 9 experiments for 8 independent variables with 3 levels. The first row represents the seed, and each row below is computed by a right shift of the row above, except for the last row, which sets all numeric options to 0.

	o_1	o_2	o_3	o_4	o_5	o_6	o_7	o_8
c_1	0	1	1	2	0	2	2	1
c_2	1	0	1	1	2	0	2	2
c_3	2	1	0	1	1	2	0	2
c_4	2	2	1	0	1	1	2	0
c_5	0	2	2	1	0	1	1	2
c_6	2	0	2	2	1	0	1	1
c_7	1	2	0	2	2	1	0	1
c_8	1	1	2	0	2	2	1	0
c_9	0	0	0	0	0	0	0	0

Figure 1: Plackett-Burman design defining 9 experiments for 8 numeric options using the minimum (0), maximum (2), and midpoint as levels (1).

[3]Central Composite, 2^{k-1}, D-optimal, Plackett-Burman, Box-Behnken, one factor at a time, full factorial, hyper sampling (form of gridding)

Random Design. Unlike binary-option sampling, we can easily determine random values from numeric options, because we know the minimum and maximum value as well as the step size, as these must be given to obtain a meaningful configuration. Furthermore, constraints among numeric options appear rarely in configurable systems. It is very unusual that numeric options have value ranges with undefined or invalid holes or that setting a numeric option's value prohibits certain value ranges of other options. To ensure reproducibility of our experiments, we use random values with seeds (available on the supplementary Web site).

4.3 Combining Binary and Numeric Sampling

So far, we have explained how we sample the configuration space regarding binary and numeric options individually. The remaining task is to combine the two. In a first step, we determine configurations using binary-option sampling, resulting in a set of *partial configurations*. In a second step, for each partial configuration, we determine a set of complete configurations based on numeric-option sampling. The rationale behind this ordering is that the majority of domain constraints are usually between binary options, so this part of the configuration space is more challenging for finding valid configurations. Furthermore, in real-world applications, such as database and web servers, compilers, and enterprise applications, it is often the case that a binary option activates a piece of functionality and numeric options adjust existing functionality (e.g., by setting input bounds, specifying the workload, or controlling the output quality). As a consequence, numeric options often depend on binary options. Overall, this combined approach yields a learning set comprising $n*r$ measurements, where n is the number of binary partial configurations and r is the number of numeric partial configurations.

5. EVALUATION

A key question is whether our approach can learn accurate performance-influence models for highly-configurable real-world systems with a reasonably small number of measurements. More specifically, we aim at answering the following three research questions:

- **RQ1**: What is the range of prediction errors per sampling heuristics we get with our approach?
- **RQ2**: Which combinations of binary and numeric sampling technique give rise to Pareto-optimal solutions with respect to measurement effort and prediction accuracy?
- **RQ3**: Is our learning approach accurate in the sense that we learn actually existing influences and interactions?

For the purpose of our evaluation, we operationalize accuracy as follows: Given a sample set, we compute a performance-influence model, which we subsequently use to *predict* the performance of a large number of configurations in an evaluation set. As said previously, the actual performance metric may vary and depends on the subject systems. We then measure the configurations of the evaluation set and calculate the error rate by $\frac{|\text{measured - predicted}|}{\text{measured}}$, averaged over all configurations. Ideally, we would use the whole configuration space of a system as evaluation set, but the measurement effort would be prohibitively high for most real-world systems. Therefore, we perform the evaluation in two separate experiments with different goals.

In a first experiment, we use synthetic performance models. We start with a known, realistic formula of a perfor-

mance-influence model as ground truth and derive the measurements for both sample set and evaluation set from this formula. Since there are no measurement costs and no measurement bias, we can perform accurate and large-scale experiments with high internal validity, using the entire configuration space as evaluation set. We use this first experiment primarily as a sanity check to determine whether we can accurately learn a performance-influence model (RQ1 and 3) and as means to explore the tradeoffs of different sampling strategies (RQ2). In a second experiment, we assess the feasibility of our approach by building performance-influence models for six real-world software systems from a number of different domains (RQ1). That is, we execute their actual benchmarks and measure execution time over a large number of configurations in both sample and evaluation sets.

5.1 Experiment #1: Correctness and Accuracy

Standard learning approaches are typically sensitive to even slight variations in the learning set. This is undesired, because we cannot trust the resulting performance-influence model when analyzing the system, although it might give reasonable performance estimates. Thus, we aim at checking whether our approach learns the actually existing influences, answering RQ3. For this purpose, we create measurements from a number of ground-truth models, from which we learn performance-influence models and compare them against the ground truth to check whether the learned influences and interactions are similar to the given ones. A second goal of this experiment is to filter out infeasible sampling heuristics regarding prediction accuracy and measurement effort, because evaluating all combinations of sampling heuristics on real-world systems is computationally infeasible.

Setup. Overall, we compare three sampling heuristics for binary options, option-wise (OW), negative option-wise (nOW), and pair-wise (PW), their combinations, and several experimental designs, from which we report here only the Plackett-Burman and Random Design with 100 randomly selected numeric configurations and five seeds.

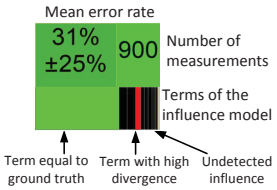
As ground-truth performance models, we use 7 formulas (Π) along the lines of our motivating example in Section 2. But, instead of creating random models, we create models that represent performance variations in real software systems by deriving them from actual performance models extracted in prior work [25] using a different approach (see Section 5.3). Since the original models contained only binary options, we enhance them with numeric options as follows: (a) we add one to four numeric options with different value ranges (100 to 1000), (b) we vary the number of numeric-option interactions (1 to 3) and their kinds (i.e., binary–numeric and numeric–numeric), and (c) we use different shapes of functions (linear, quadratic, linear combined with quadratic). This way, we consider a wide range of different influence functions and interactions. To get an impression of how our given models look, we refer the reader to Section 5.4.

To distinguish the ground-truth models, we name them after the configurable systems from which we obtained the binary options. Table 5.1 shows all ground-truth models including their configuration spaces, the application domain of the original performance models, and the number of constraints among options. We provide all models as well as the intermediate models of the learning process on the supplementary Web site.

Table 1: Overview of the ground-truth performance-influence models.

Model	Domain	# Bin	#Num	#Const	$\lvert\mathcal{C}\rvert$
AJStats	Analysis tool	20	1	3	10^7
Apache	Apache Web server	9	2	2	10^6
BDB C	Berkeley DB C	7	2	0	10^5
BDB J	Berkeley DB Java	26	4	31	10^{14}
Clasp	Answer set solver	20	4	64	10^{15}
LLVM	Compiler infrastructure	11	2	1	10^7
lrzip	Compression library	19	4	107	10^{14}

#Bin: Number of binary options; #Num: number of numeric options; #Const: number of constraints; $\lvert\mathcal{C}\rvert$: number of config.



To characterize the accuracy of a learned model for a specific sampling strategy, we use a compact visual representation, as depicted to the right, covering several aspects: mean error rate over predictions of all configurations, the standard deviation of the error when repeating sampling and learning (for random sampling), the number of measurements required by the sampling heuristic, and a compact representation of all terms in the model (lower part: boxes denote individual terms weighted in size by their effect strength (coefficients); accuracy per term is color-coded, from green for the same value to red for a relative difference larger than 100%, and black for terms that are missing in the learned model).

Results. Figure 2 summarizes our experimental results for all binary-option samplings and Plackett-Burman and Random Design. Again, we show only the best combinations of sampling heuristics and refer the interested reader to our supplementary Web site for all results. Overall, we observe a good degree of similarity between the learned models and the ground-truth models, indicated by the green rectangles. Only when binary-option interactions exist and no pair-wise sampling was used, we observe some undetected influences (black bars). There are also models, such as for the synthetic *Clasp* model, for which we obtain a prediction error of 1 %, on average, but we miss 3 to 5 influence functions (row PW PBD). This observation suggests that missing a single term likely affects only a small fraction of the configuration space, such that the overall prediction error remains small. These results suggest that we indeed learn the actual existing influences in most cases, which is useful for performance debugging and interactive configuration tools [19].

To analyze the tradeoff among prediction accuracy and measurement effort (RQ2), we plot the Pareto front (dashed line) of the combination of binary-option and numeric-option sampling heuristics in Figure 3. The Plackett-Burman design ($n = 49$, $l = 7$) outperforms all designs no matter which binary sampling heuristics is used, which applies also to combinations not shown. For binary-option sampling, pair-wise sampling has the highest accuracy in combination with Plackett-Burman, 1.6 %, on average, (even better in combination with option-wise and negative option-wise sampling, 0.3 %, on average). However, it comes at the cost of an increased number of measurements compared to the pure

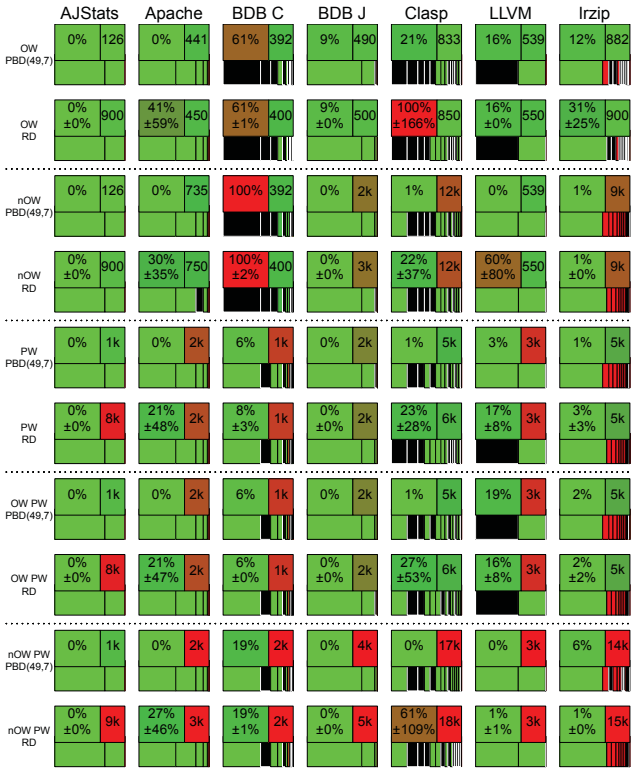


Figure 2: Comparison of learned and ground-truth models in terms of mean prediction error, number of measurements, and existence and similarity of model terms.

option-wise approach (which in turn has a mean prediction error of about 18 %). Random sampling is not competitive at sample sizes comparable to the Plackett-Burman design, and we observed a strong fluctuation in the error rate when repeating learning with a fresh random sample.

5.2 Experiment #2: Effort and Accuracy

In our second experiment, we evaluate whether our learning and sampling approach is feasible in practice. Although partly parallelized, we invested *more than two months (24/7)* for measuring configurations of six subject systems to obtain a huge data basis (including learning and evaluation sets). However, using our approach in practice would require much less measurements, as most measurements were meant to evaluate and analyze our approach.

Setup. With a focus on external validity, we selected six highly configurable systems from different domains, written in different programming languages, with varying numbers of binary and numeric options. Some systems support configuration at compile time, others at load time. We purposefully selected some systems for which we can perform a whole population analysis (Dune MGS, HIPA^{cc}, HSMGP) being able to reliably quantify prediction accuracy as well as systems that are highly configurable to evaluate the scalability of our approach (see Table 2 for an overview).

- Dune MGS is a geometric multi-grid solver based on the Dune framework [3]. The framework provides algorithms for smoothing and solving Poisson equations on structured grids. Binary options include several smoother and solver algorithms. Numeric options include different grid sizes and pre- and post-smoothing steps. We measured the time to solve Poisson's equation on a Dell OptiPlex-9020 with an Intel i5-4570 Quad Code and 32 GB RAM (Ubuntu 13.4).
- HIPA^{cc} is an image processing acceleration framework, which generates efficient low-level code from a high-level specification. Binary options are, among others, the kind of memory to be used (e.g., texture vs. local). The number of pixels calculated per thread is an example of a numeric option. We measured the time needed for solving a test set of partial differential equations on an nVidia Tesla K20 card with 5GB RAM and 2496 cores (Ubuntu 14.04).
- HSMGP is a highly scalable multi-grid solver for large-scale data sets. Binary options include in-place conjugate gradient and in-place algebraic multi-grid solvers. Numeric options include the number of smoothing steps and the number of nodes used for computing the solution. As a benchmark, we performed a multi-grid iteration of solving Poisson's equation. We executed the benchmark runs on JuQueen, a Blue Gene/Q system, located at the Jülich Supercomputing Center, Germany.
- JavaGC is the Java garbage collector (version 7) with several options for adaptive garbage-collection boundary and size policies. For measurement, we executed the DaCapo benchmark suite on a computing cluster consisting of 16 nodes each equipped with an Intel Xeon E5-2690 Ivy Bridge having 10 cores and 64 GB RAM (Ubuntu 14.04).
- SaC is a variant of C for high-performance computing based on stateless arrays. The SaC compiler implements a large number of high-level and low-level optimizations to tune high-level programs for efficient parallel executions. The compiler is highly configurable, allowing users to select various optimizations and to customize the optimization effort (e.g., optimization cycles and loop-unrolling threshold). As benchmark, we compile and execute an n-body simulation shipped with the compiler, measuring the execution time of the simulation at different optimization levels. We executed all benchmarks on an 8 core Intel i7-2720QM machine with 8 GB RAM (Ubuntu 12.04).
- x264 is a video encoder that encodes raw videos into the H.264 compressed format. Configuration options configure output quality, encoder types, and encoding heuristics. As benchmark, we measured the time needed to encode the Sintel trailer (734 MB) using on an Intel Core2 Q6600 with 4GB RAM (Ubuntu 14.04).

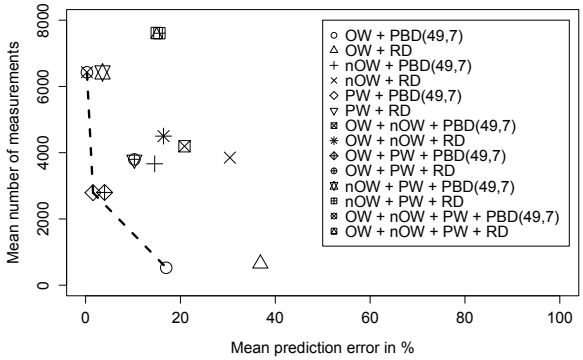


Figure 3: Pareto front (dashed line) of combinations of sampling heuristics.

Table 2: Overview of the real-world subject systems.

System	Domain	#Bin	#Num	#Const	$\|\mathcal{C}\|$
Dune MGS	Multi-Grid Solver	8	3	20	2 304
HIPA^{cc}	Image Processing	31	2	416	13 485
HSMGP	Stencil-Grid Solver	11	3	45	3 456
JavaGC	Runtime Env.	12	23	4	10^{31}
SaC	Compiler	53	7	10	10^{23}
x264	Video Encoder	8	13	0	10^{27}

#Bin: Number of binary options; #Num: number of numeric options; #Const: number of constraints; $|\mathcal{C}|$: number of config.

We started our experiments by determining sample sets to following the Plackett-Burman and Random Sampling (as they performed best in our first experiment; see Sec. 5.1) as well as our binary-sampling heuristics (OW, PW). We do not report on the results for nOW, because our first experiment (Sec. 5.1) showed that it increases measurement effort considerably for only a limited gain in accuracy. As evaluation set, we selected either the whole population (Dune MGS, HSMGP, HIPA^{cc}) or a large random set of configurations (more than 10 000 randomly selected configurations).

Results. In Table 3, we present the results for the six subject systems. As expected, the error rate is larger than for the synthetic models, since we cannot fully control for confounding factors, such as measurement bias, which made up to 10 % of deviations within multiple repetitions of measuring the same configuration. Putting this into perspective, we observe still a comparatively high prediction accuracy for most of the systems. We yield for every subject system a performance-influence model whose error rate is below 19 %. For SaC, the PW heuristic requires more than 160 000 measurements, which is infeasible.[4]

Surprisingly, the OW heuristic performs often equally well as the PW heuristic (for Dune MGS, HIPA^{cc}, HSMGP, SaC, x264), although requiring a substantially lower number of measurements. For numeric-option sampling, we see similar results as in our first experiment: The Plackett-Burman design is often superior to the Random Design. The measurement effort for Plackett-Burman is higher with constrained configuration spaces (e.g., Dune MGS or HIPA^{cc}). For systems with few constraints and many options, Plackett-Burman requires less measurements.

Learning a model required 1 to 5 hours, depending on the size of the learning set and the size of the models. Remarkably, the resulting models are compact with only few terms explaining most of the performance variations; particularly, we observe usually a larger number of influences from individual options (i.e., ϕ_i) and only a low number of interactions (i.e., $\Phi_{i..j}$), considering that there is potentially an exponential number of interactions. Furthermore, we see an increase in the number of interactions when PW sampling is used compared to OW sampling, and also Plackett-Burman results in larger numbers of identified interactions compared to Random Design. In Section 5.4, we discuss these and other results and put them into perspective.

[4]We use an active-learning approach, in which we first evaluated the performance-influence model produced with the OW heuristic, to determine which binary options have an influence at all. Then, we applied the PW heuristic to these options. The corresponding results are marked with * in Table 3. We discuss this solution more in Section 5.4

Table 3: Results for the six subject systems and combinations of option-wise (OW) and pair-wise (PW) sampling. We consider terms with an absolute coefficient of > 0.01 only.

	OW		PW	
	$\bar{e}/\|\mathcal{C}\|$	$\phi_i\|\Phi_{i..j}$	$\bar{e}/\|\mathcal{C}\|$	$\phi_i\|\Phi_{i..j}$
Dune MGS				
RD	20.1%/49	5|0	22.1%/78	8|8
PBD(49,7)	10.6%/240	6|24	11%/384	6|18
PBD(125,5)	8.8%/375	8|16	8.3%/600	8|20
HIPA^{cc}				
RD	14.2%/261	16|13	13.9%/1281	11|16
PBD(49,7)	13.9%/736	18|8	10.6%/3645	12|19
PBD(125,5)	13.8%/528	18|9	10.7%/2631	12|19
HSMGP				
RD	4.5%/77	11|14	2.8%/173	9|13
PBD(49,7)	2.2%/384	9|13	1.6%/864	10|13
PBD(125,5)	1.7%/480	11|13	1.5%/1080	11|14
JavaGC				
RD	31.3%/534	4|0	24.6%/3032	5|7
PBD(49,7)	37.4%/423	5|0	28.2%/2571	9|14
PBD(125,5)	21.9%/855	3|0	18.8%/5312	5|21
SaC				
RD	21.1%/2060	14|5	30.7%/3261*	7|9
PBD(49,7)	16%/2499	14|11	25%/4704*	8|13
PBD(125,5)	20.3%/2295	14|5	27%/4320*	6|19
x264				
RD	14.2%/304	4|3	13.5%/1000	3|7
PBD(49,7)	36.7%/216	5|4	12.5%/636	6|9
PBD(125,5)	21.2%/339	4|1	15%/1046	4|6

PBD: Plackett-Burman Design; RD: Random Design; $\bar{e}$: mean prediction error; $|\mathcal{C}|$: size of sample set; ϕ_i: number of terms representing the influence of individual options; $\Phi_{i..j}$: number of terms representing the influence of interactions

5.3 Threats to Validity

Internal Validity. To rule out conceptual and implementation errors of our approach, we conducted a high-internal validity experiment, in which we used ground-truth models and aimed at learning these models using different sampling heuristics. Note that the ground-truth models have been learned in prior work with linear programming [25], which is important, as it would be invalid to compare models that have been created with the same learning technique. Furthermore, we do not only computationally compared the models, but manually reviewed them to avoid errors in the evaluation. This way, we mitigate the threat that we learn models with similar prediction accuracy, but with different influences and detected interactions, which might be a result of overfitting. In the second experiment, we repeated all measurements several times and used averages to control measurement bias. Although we parallelized the measurement on equal machines, they might slightly differ in their performance, but these effects are usually small and, due to repeated measurements, likely cancel each other out.

External Validity. As highly configurable systems exist in a broad range of application domains, with different configuration spaces, we selected six real-world subject systems in our second experiment. These systems have varying numbers of options and are from different application domains,

which increases external validity. Naturally, we cannot guarantee that our approach works for all possibly configurable software systems, but we invested several months of measurement to gather and analyze a substantial evaluation set.

5.4 Discussion

Research Questions. Our results show that building performance-influence models is (a) computationally tractable (a few hours of learning and measurement), (b) our approach finds actual influences and represents them directly, and (c) we attain a reasonable prediction accuracy.

Regarding RQ1 (prediction accuracy), in the first experiment, we almost always learned performance-influence models with nearly perfect prediction accuracy. For the real-world systems, we observe average error rates of 10 % to 19 %, which are only a slightly higher than the measurement bias. For some subject systems and sampling heuristics, we observe larger prediction errors. A possible explanation is that these subject systems contain three-way or more complex interactions, which cannot be detected by OW and PW sampling. Another possible reason is that many weak influences may sum up to larger prediction errors. We purposefully do not search for weak influences as they complicate the model to an extent that is impracticable for performance debugging or comprehension.

Regarding RQ2 (tradeoff between measurement effort and prediction accuracy), we found that more configurations do not necessarily lead to more accurate predictions. It depends on which configurations are selected and how they are distributed in the configuration space, to cover the relevant interactions. For both experiments, Plackett-Burman designs yield the best tradeoff. Although random sampling is very effective in learning accurate models with comparable sampling sizes, it incurs substantial fluctuations, such that an additional design should be applied that acts as a base layer to cover all numeric options and their respective data ranges. For binary-option sampling, we observe a mixed picture. We found that the PW heuristic is superior to OW and nOW for the ground-truth models (in which the binary-option part originates from real-world performance models), whereas without OW sampling, we miss some influences leading to higher error rates for some real-world systems. Hence, also in this case, a combination of the OW and PW heuristic might be a better solution.

RQ3 aimed at determining whether we learned the actual, relevant options and interactions. The results of the first experiment provide us the clear picture that we, indeed, find most of the original terms of a given performance-influence model. Moreover, when considering the size of the learned performance-influence models of the real-world systems (number of terms), we conclude that we learn simple models (at the cost of some prediction accuracy) with both individual options and interactions. The ratio of $\frac{|\phi_i|}{|\mathcal{O}|}$ over all subject systems for OW with Plackett-Burman sampling is 0.31, indicating that one third of the options significantly contribute to the performance of a system. When considering interactions, $\frac{|\Phi_{i..j}|}{|\mathcal{O}|}$ is only 0.34, such that the number of determined interactions is similar the the number of relevant influences of individual options.

Comprehension and Debugging. An issue not addressed by our evaluation is to what extent performance-influence models help developers in their everyday development and debugging tasks. While we strive for simple models that contain only the most important factors for performance, they still may get considerably large. Nevertheless, based on such models, we are able to provide views such that isolated influences of configuration options or interactions become immediately apparent. Let us consider again the excerpt of the performance-influence model of Berkeley DB C:

$$0.05 \cdot PageSize - 0.12 \cdot CacheSize + 2.4 \cdot Hash + 0.12 \cdot Statistics + 29.88 \cdot Crypto \cdot Hash - 8.04 \cdot Hash \cdot Verify \cdot Statistics + 0.59 \cdot Crypto \cdot Hash \cdot Replication - 0.15 \cdot CacheSize \cdot Crypto + \ldots$$

If we are interested in understanding the influence of cryptography on the overall execution time, we can project out all other influences yielding the following simpler formula:

$$29.88 \cdot Crypto \cdot Hash + 0.59 \cdot Crypto \cdot Hash \cdot Replication - 0.15 \cdot CacheSize$$

This view suggests that the cryptography feature should not be used in combination with the hash search index, because it degrades performance; when used in combination with replication, the degradation is even worse, but, increasing the cache size limits the negative influence of cryptography on the execution time.

Actually, using our approach, we found an unexpected slowdown in the Polly extension of the LLVM compiler framework (`trmm.c`). By incident, this was observed around the same time by others.[5] Moreover, for the configuration options *ignore-aliasing* and *no-runtime-alias-checks*, we found differing performance influences, for which a developer of Polly expected that both should have a similar effect.

We already propagated some performance-influence models back to domain experts (e.g., the HPC domain) to guide and improve the development and configuration, which goes beyond performance tuning of highly-configurable software systems. While these examples nicely illustrate the power of (views on) performance-influence models for program comprehension and debugging, it will be imperative to conduct a comprehensive user study in further work.

Limitations. Our approach rests on several assumptions. First, we use regression analysis to learn influence functions. If a configuration option has an unsteady performance behavior or has a very complex (nearly chaotic) behavior, we cannot learn, but only approximate its performance influence. Furthermore, we need the configurable system to have a deterministic performance behavior. If two equal runs of the same program lead to largely different performances, we cannot reliably learn influences and hardly predict performance. Finally, our approach has its limits regarding the number of options and the size of the learning set. Although we already tried to minimize the learning set (and there is room for further improvement), it is still an infeasible problem to support systems with thousands of options (in terms of constrainedness and performance variability). Still, our evaluation demonstrated that our solution scales to problems with up to 10^{31} configurations, making it feasible for a sufficient number of real-world systems.

[5] `https://groups.google.com/forum/#!topic/isl-development/Dm0bJS7jsCY`

Perspectives. Beside various facets of performance, performance-influence models may be beneficial to reason about other non-functional properties and quality attributes, most notably, energy consumption. Moreover, we can supply the models we learned to other performance-modeling and optimization tools, such as Clafer [19] and EPOAL [9].

Technically, our approach could be extended to support active learning. That is, we could evaluate the performance-influence models in an intermediate step to decide whether additional measurements should be applied. We found that, for SAC, applying OW sampling with a Plackett-Burman design resulted in a performance-influence model, in which only 10 of 53 binary options have a relevant influence. Based on this result, it is advisable to use the PW heuristic only for the 10 binary options, which would reduce the required measurements from 2809 to 100 (or from 280,900 to 10,000 when combined with 100 random numeric-option samples).

In general, our notion of performance-influence models is conceptually independent of the concrete learning technique. That is, the concrete technique is hidden behind the ϕ_is and $\Phi_{i..j}$ terms of the model. Thus, our approach is complementary to existing approaches of performance modeling. We made a number of decisions to support program comprehension and debugging performance of configurable systems. However, when prediction accuracy or optimization is the single most important aspect, then other techniques, such as support vector machines, could be used.

6. RELATED WORK

Learning. Our approach aims at determining the individual influences of configuration options and their interactions, which has several use cases, such as performance-bug detection or configuration optimization. There are many successful approaches that aim at finding optimal configurations *without* pinpointing the influence of configuration options explicitly [7, 12, 13]. More closely related to our work are standard machine-learning techniques, such as support-vector machines, Bayesian nets, and evolutionary algorithms. These approaches trade simplicity and understandability of the learned models for predictive power. Software configuration, however, usually involves humans in the reasoning process, since not a single, but a number of objectives need to be satisfied. Hence, we need to understand how individual options influence performance and which interact.

There are a number of approaches that use profiling data to create performance models [16]. For instance, Jovic and others analyze samplings of call stacks of deployed versions of a program to find performance bugs [14]. Grechanik and others propose to learn rules for the generation of workloads that reveal program paths with suboptimal performance [6]. However, these approaches concentrate on workload variability rather than software-system configurability.

Sampling. Although a proper sampling heuristic is a critical success factor for determining the influence of configuration options and finding optimal configurations, there is only little work done so far. Important sampling approaches have been developed in statistics, in which experimental designs have been developed to ensure certain statistical properties. We used these designs, such as Central Composite, Box Behnken, and Plackett Burman, to determine configurations of numeric options [18]. One simple approach is *Gridding*, which computes a grid over the space of the input parameters. It was used for sampling configurations of Berkeley DB [27]. However, due to its exponential complexity, Sullivan and others could consider only four options in a reasonable amount of time [27].

For binary-option sampling, several approaches tackle the problem of finding valid configurations [8]. Especially, evolutionary algorithms have been proposed for this task [24, 23, 11]. Pohl and others found that determining a valid configuration based on a variability model increases response time exponentially with respect to the number of features [21].

The sampling heuristics and experimental designs we use to identify interactions are related to the heuristics used in combinatorial testing [10, 15]. The difference is that we do not focus on functional correctness, but on performance, which allows us to learn performance-influence models using linear regression. This would be considerably harder when applying it to defect prediction, as defects are much more singular events in a program's execution than the observable performance profile.

7. CONCLUSION

Today, most contemporary systems are configurable, which makes performance prediction, optimization, and debugging difficult. We address this challenge by proposing an approach that derives a *performance-influence model* for a given configurable system, describing all *relevant* influences of individual configuration options and their interactions. To this end, we select and adapt a suitable machine-learning technique and combine it with sampling heuristics for binary and numeric configuration options in a novel way. Our approach rests on an algorithm that iteratively learns a performance-influence model using a small set of candidate features representing relevant performance influences. To derive learning sets of tractable sizes, we combine heuristics for binary-option sampling with experimental designs for numeric-option sampling.

By means of a first experiment on (partially) synthetic, ground-truth models, we could show that our hierarchical learning strategy finds the actually relevant influencing options and interactions and yields a mean prediction error of 1 %. In a second experiment, we applied our approach to six real-world software systems, in which we measured performance in terms of the execution time of a given benchmark. Our results confirm the first experiment for both accuracy and measurement effort. A major insight is that the Plackett-Burman design is superior to all other numeric-option sampling heuristics regarding the tradeoff between measurement effort and prediction accuracy, with an average prediction error below 19 %, which is only slightly above the measurement bias. Furthermore, we found that our approach is feasible for finding performance bugs in real-world systems, which is a promising avenue of further research.

8. ACKNOWLEDGMENTS

We thank Z. Kolter, Y. Agarwal, and D. Batory for comments on earlier drafts of this paper, A. Simbürger for his help with the measurements, and the Jülich Supercomputing Center for providing access to the supercomputer JuQueen. This work has been supported by the DFG grants AP 206/4, AP 206/6, and AP 206/7 and by the NSF award 1318808.

9. REFERENCES

[1] S. Apel, S. Kolesnikov, N. Siegmund, C. Kästner, and B. Garvin. Exploring feature interactions in the wild: The new feature-interaction challenge. In *Proceedings of the International Workshop on Feature-Oriented Software Development (FOSD)*, pages 1–8. ACM, 2013.

[2] J. Bien, J. Taylor, and R. Tibshirani. A lasso for hierarchical interactions. *The Annals of Statistics*, 41(3):1111–1141, 2013.

[3] M. Blatt and P. Bastian. The iterative solver template library. In *Applied Parallel Computing. State of the Art in Scientific Computing*, pages 666–675. Springer, 2007.

[4] G. Chandrashekar and F. Sahin. A survey on feature selection methods. *Computers & Electrical Engineering*, 40(1):16–28, 2014.

[5] P. Domingos. A few useful things to know about machine learning. *Communications of the ACM*, 55(10):78–87, 2012.

[6] M. Grechanik, C. Fu, and Q. Xie. Automatically finding performance problems with feedback-directed learning software testing. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 156–166. IEEE, 2012.

[7] 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)*, pages 301–311. IEEE, 2013.

[8] J. Guo, J. White, G. Wang, J. Li, and Y. Wang. A genetic algorithm for optimized feature selection with resource constraints in software product lines. *J. Systems and Software*, 84(12):2208–2221, 2011.

[9] J. Guo, E. Zulkoski, R. Olaechea, D. Rayside, K. Czarnecki, S. Apel, and J. Atlee. Scaling exact multi-objective combinatorial optimization by parallelization. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 409–420. ACM, 2014.

[10] A. Hartman and L. Raskin. Problems and algorithms for covering arrays. *Discrete Mathematics*, 284(1-3):149–156, 2004.

[11] C. Henard, M. Papadakis, M. Harman, and Y. Traon. Combining multi-objective search and constraint solving for configuring large software product lines. In *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 2015.

[12] F. Hutter, H. Hoos, K. Leyton-Brown, and T. Stützle. Paramils: An automatic algorithm configuration framework. *J. Artificial Intelligence Research*, 36(1):267–306, 2009.

[13] F. Hutter, L. Xu, H. Hoos, and K. Leyton-Brown. Algorithm runtime prediction: Methods & evaluation. *Artificial Intelligence*, 206:79–111, 2014.

[14] M. Jovic, A. Adamoli, and M. Hauswirth. Catch me if you can: Performance bug detection in the wild. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 155–170. ACM, 2011.

[15] R. Kuhn, R. Kacker, and Y. Lei. *Introduction to Combinatorial Testing*. Chapman & Hall, 2013.

[16] Y. Kwon, S. Lee, H. Yi, D. Kwon, S. Yang, B.-G. Chun, L. Huang, P. Maniatis, M. Naik, and Y. Paek. Automatic generation of efficient performance predictors for smartphone applications. In *Proceedings of the USENIX Annual Technical Conference*, pages 297–308. Usenix Association, 2013.

[17] 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)*, pages 81–91. ACM, 2013.

[18] D. Montgomery. *Design and Analysis of Experiments*. John Wiley & Sons, 2006.

[19] A. Murashkin, M. Antkiewicz, D. Rayside, and K. Czarnecki. Visualization and exploration of optimal variants in product line engineering. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 111–115. ACM, 2013.

[20] R. Plackett and J. Burman. The design of optimum multifactorial experiments. *Biometrika*, 33(4):305–325, 1946.

[21] R. Pohl, V. Stricker, and K. Pohl. Measuring the structural complexity of feature models. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 454–464. IEEE, 2013.

[22] F. Pukelsheim. *Optimal Design of Experiments*. Classics in Applied Mathematics. Society for Industrial and Applied Mathematics, 2006.

[23] A. Sayyad, J. Ingram, T. Menzies, and H. Ammar. Scalable product line configuration: A straw to break the camel's back. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 465–474. IEEE, 2013.

[24] A. Sayyad, T. Menzies, and H. Ammar. On the value of user preferences in search-based software engineering: A case study in software product lines. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 492–501. IEEE, 2013.

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

[26] N. Siegmund, A. von Rhein, and S. Apel. Family-based performance measurement. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 95–104. ACM, 2013.

[27] D. Sullivan, M. Seltzer, and A. Pfeffer. Using probabilistic reasoning to automate software tuning. *ACM SIGMETRICS Performance Evaluation Review*, 32(1):404–405, 2004.

[28] J. Wang and C. Wu. A hidden projection property of Plackett-Burman and related designs. *Statistica Sinica*, 5:235–250, 1995.