Variability-Aware Performance Prediction: A Statistical Learning Approach

Jianmei Guo*, Krzysztof Czarnecki*, Sven Apel†, Norbert Siegmund†, and Andrzej Wąsowski‡
*University of Waterloo, Canada
†University of Passau, Germany
‡IT University of Copenhagen, Denmark

***Abstract*—Configurable software systems allow stakeholders to derive program variants by selecting features. Understanding the correlation between feature selections and performance is important for stakeholders to be able to derive a program variant that meets their requirements. A major challenge in practice is to accurately predict performance based on a small sample of measured variants, especially when features interact. We propose a variability-aware approach to performance prediction via statistical learning. The approach works progressively with random samples, without additional effort to detect feature interactions. Empirical results on six real-world case studies demonstrate an average of 94 % prediction accuracy based on small random samples. Furthermore, we investigate why the approach works by a comparative analysis of performance distributions. Finally, we compare our approach to an existing technique and guide users to choose one or the other in practice.**

I. Introduction

Many software systems provide configuration options for users to tailor their functional behavior as well as non-functional properties (e.g., performance, cost, and energy consumption). Configuration options relevant to users are often called *features* [2], [3], [7], [8], [13], [16]. Each *variant* derived from a configurable software system can be represented as a *selection* of features, called a *configuration*.

Performance (e.g., response time or throughput) is one of the most important non-functional properties, because it directly affects user perception and cost [25]. Finding an optimal configuration to meet a specific performance goal is a fundamental task for developers and system administrators. Performance of a software system is often subject to a wide variety of influencing factors [24]. Understanding trade-offs between influencing factors and performance is non-trivial. In this paper, we focus on how to determine the influence of feature selections on performance. Considering a configurable software system as a black box, we investigate and exploit the correlation between feature selections and performance for performance prediction.

A straightforward approach to reveal such correlation is to measure the performance of all configurations of a software system and then provide direct answers (e.g., which configuration is the fastest). However, such a brute-force approach is usually infeasible, because even a small-scale configurable system can give rise to an exponential number of configurations, due to feature combinatorics, and the cost of measurement may be high (e.g., executing a complex benchmark). Therefore, in practice, often only a limited set of configurations can be measured, either by simulation or by monitoring in the field [24]. We denote these configurations along with their performance measurements as a *sample*, and all configurations of a software system along with their performance as the *whole population*. The challenge is how to use a *small* (e.g., linear in the number of features) sample to predict the performance of other configurations in the whole population, with a *high* accuracy (e.g., above 90 %).

Quantifying the performance influence of each individual feature is not sufficient in most cases, as *feature interactions* may cause unpredictable performance anomalies [19]. That is, the performance influence of two features, both appearing in a configuration, may not be easily deducible from the performance influence of each feature without the other. Siegmund et al. [19] addressed this issue by introducing a measurement-based prediction approach, called SPLConqueror, which detects performance-relevant feature interactions using specific sampling heuristics that meet different feature-coverage criteria. However, in practice, the configurations that we can measure or that we already have at our disposal may not meet any feature-coverage criterion. Thus, we pose the following research question: *Is it feasible to use small random samples as a basis for accurate performance prediction?*

To answer this question, we formalize the problem of variability-aware performance prediction and reduce it to a *non-linear regression* problem. We use a statistical learning technique, *Classification And Regression Trees (CART)* [5], to address the problem and to model the correlation between feature selections and performance.

Compared to existing methods [19], [20], [22]–[25], our approach works *automatically* and *progressively* with random samples, such that one can use it to produce predictions, starting with a small random sample, and subsequently extend it when further measurements are available; it considers *all* features of a system and identifies the performance-relevant ones; it treats the selected and deselected features in a configuration *equally*, to describe the correlation between feature selections and performance; and it can be *easily* implemented and deployed in practice, without additional effort to detect feature interactions—an inherently challenging task [19].

In summary, we make the following contributions:

- We propose a progressive and variability-aware approach that predicts a configuration's performance based on random samples. The approach builds an explicit performance model to specify the correlation between feature selections and performance, to be used for performance prediction.

- We implement the approach and demonstrate its practicality and generality by experiments on six real-world configurable software systems. The results show that the approach produces an average prediction accuracy of $94\,\%$, based on only small random samples. Moreover, we observe a desirable increasing trend of prediction accuracy when the sample size increases.
- We conduct a comparative analysis of performance distributions on the evaluated case studies and empirically explore why the approach works with small random samples.[1] A key finding is that it works well when the sample it uses has a performance distribution similar to the whole population.
- We compare our method with an existing technique that relies on heuristics and feature coverage [19]. In particular, we discuss the strengths and weaknesses of the two approaches and guide users to choose one or the other in practice.

The implementation of the approach and all experimental data are available at `http://cpm.googlecode.com`.

II. Motivating Example

We use the configurable tool X264 as an example to motivate our approach. X264 is a command-line tool to encode video streams into the H.264/MPEG-4 AVC format.[2] In this example, we consider 16 encoder features of X264, such as parallel encoding on multiple processors or encoding with multiple reference frames. Users can configure X264 by selecting different features to encode a video. We use the encoding time to indicate the performance of X264 in different configurations. Even such a simple case with only 16 features gives rise to $1,152$ configurations. Given that we measure the performance of a limited set of configurations as a sample, how can we determine the performance of other configurations?

To address this issue, previous work on SPLConqueror [19] focuses on selecting a specific sample to detect performance-relevant feature interactions. That is, following a certain feature-coverage criterion, SPLConqueror selects a fixed set of specific configurations and then measures their performance, which is then the input for predicting the performance of other configurations. Two fundamental feature-coverage criteria are *feature-wise* and *pair-wise*. The feature-wise measurement quantifies an individual feature's performance influence by calculating the performance delta of two minimal configurations with and without the feature. The pair-wise heuristic selects and measures additionally a specific set of configurations to detect all pair-wise feature interactions. SPLConqueror also provides heuristics for the detection of higher-order feature interactions.

[1]A performance distribution denotes the frequency distribution of all performance values in a sample or in the whole population.

[2]http://www.videolan.org/developers/x264.html

An important point is that, in practice, the configurations that we can measure or that we already have are often arbitrarily selected; they may not meet any feature-coverage criterion. Moreover, the number of available configurations may vary and is usually very limited due to the high cost of performance measurement. For example, Table I lists a sample of 16 randomly-selected configurations of X264 and corresponding performance measurements. The question is, can we predict performance of all other configurations accurately with such a limited number of measurements? Next, we formally reduce this question to a non-linear regression problem, and then we present our approach to address the problem.

III. Problem Formalization and Reduction

In this section, we formalize the problem of variability-aware performance prediction and reduce it to a non-linear regression problem.

We represent all features of a configurable software system as a set X of binary decision variables. If a feature is selected in a configuration, then the corresponding variable x is equal to 1, and 0 otherwise. We denote the number of all features in a system as N, i.e., $X = \{x_1, x_2, ..., x_N\}$. Then, we represent each configuration of a system as an N-tuple, assigning value 1 or 0 to each variable in X. For example, X264 has 16 features in total, as listed in Columns x_1 to x_{16} in Table I; thus, each configuration of X264 is represented by a 16-tuple, e.g., $\mathbf{x}_1 = (x_1 = 1, x_2 = 1, x_3 = 0, ..., x_{16} = 1)$. We denote all valid configurations in a system as set $\mathbf{X}$.

Each configuration $\mathbf{x}$ of a system has an actual performance value y. We indicate the actual performance of all configurations of a system as Y. Suppose that we acquire a set of configurations $\mathbf{X}_S \subset \mathbf{X}$ and measure their actual performance Y_S, together forming sample S. For example, Table I lists a sample of 16 randomly-selected configurations of X264 (Rows $\mathbf{x}_1$ to $\mathbf{x}_{16}$) and their performance values (the rightmost column). Thus, the problem of variability-aware performance prediction is how to predict the performance of other unmeasured configurations in $\mathbf{X} \setminus \mathbf{X}_S$ based on the measured sample S.

Since we focus on the influence of feature selections on performance, we consider all variables in X as *predictors* and a configuration's actual performance value y as the *response*. In essence, we try to find a function to relate the tuple $\mathbf{x}$ of predictors to the quantitative response y, which is a typical regression problem [11]. Given a sample S, the problem is to find a function f that reveals the correlation between $\mathbf{X}_S$ and Y_S and that makes each configuration's predicted performance $f(\mathbf{x})$ as close as possible to its actual performance y, i.e.:

$$f : \mathbf{X} \to \mathbb{R} \text{ such that } \sum_{\mathbf{x}, y \in S} L(y, f(\mathbf{x})) \text{ is minimal} \qquad (1)$$

where L is a loss function to penalize errors in prediction. An assumption of our approach is that the sample S and the whole population exhibit the same or similar correlation between feature selections and performance. Thus, we can use

TABLE I
A SAMPLE OF 16 RANDOMLY-SELECTED CONFIGURATIONS OF X264 AND CORRESPONDING PERFORMANCE MEASUREMENTS (SECONDS)

Conf.	Features																Perf. (s)
$\mathbf{x}_i$	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}	x_{13}	x_{14}	x_{15}	x_{16}	y_i
$\mathbf{x}_1$	1	1	0	1	1	1	1	0	1	0	0	1	1	0	0	1	651
$\mathbf{x}_2$	1	1	1	1	1	1	0	1	1	1	0	0	1	0	1	0	536
$\mathbf{x}_3$	1	1	1	1	0	0	0	0	1	1	0	0	1	0	0	1	581
$\mathbf{x}_4$	1	0	0	0	0	0	1	0	1	1	0	0	1	0	1	0	381
$\mathbf{x}_5$	1	1	0	1	0	0	0	1	1	1	0	0	1	0	1	0	424
$\mathbf{x}_6$	1	1	0	0	1	0	1	1	1	1	0	0	1	0	0	1	615
$\mathbf{x}_7$	1	0	1	0	1	1	1	0	1	1	0	0	1	0	1	0	477
$\mathbf{x}_8$	1	0	1	0	0	0	0	1	1	0	0	1	1	1	0	0	263
$\mathbf{x}_9$	1	0	0	0	0	0	1	1	1	0	0	1	1	1	0	0	272
$\mathbf{x}_{10}$	1	1	1	1	0	0	0	1	1	0	0	1	1	1	0	0	247
$\mathbf{x}_{11}$	1	0	0	0	0	0	0	0	1	0	1	0	1	0	0	1	612
$\mathbf{x}_{12}$	1	0	1	1	1	0	0	0	1	0	0	1	1	0	1	0	510
$\mathbf{x}_{13}$	1	1	1	1	0	1	1	0	1	0	1	0	1	0	0	1	555
$\mathbf{x}_{14}$	1	1	0	0	1	0	1	1	1	0	0	1	1	1	0	0	264
$\mathbf{x}_{15}$	1	0	1	0	0	1	1	1	1	0	0	1	1	0	0	1	576
$\mathbf{x}_{16}$	1	0	1	0	1	0	1	1	1	0	1	0	1	1	0	0	268

the function f, built on the sample S, to predict performance of other configurations in $\mathbf{X} \setminus \mathbf{X}_S$.

Note that we do not presume which features are actually relevant to performance, but we consider all features of a system. Moreover, we cannot linearly deduce the performance of a configuration from the performance influence of each individual feature in separation due to feature interactions [19]. Therefore, the problem of variability-aware performance prediction is a non-linear regression problem, i.e., the prediction function depends non-linearly on one or more predictors [11].

IV. VARIABILITY-AWARE PERFORMANCE PREDICTION

This section presents our progressive and variability-aware approach to performance prediction via statistical learning.

A. Overview of the Approach

Figure 1 illustrates our approach, which consists of two iterative processes. The first process predicts performance based on a set of rules. After configuring a system A and deriving a new, previously unmeasured configuration $\mathbf{x}$, users want to know the performance y if system A uses the configuration $\mathbf{x}$. Our approach returns the quantitative prediction (i.e., $f(\mathbf{x})$) after retrieving the corresponding decision rule for configuration $\mathbf{x}$. Each decision rule specifies the predicted performance value of a configuration when the configuration has the same feature selections (i.e., selected and deselected features) as the rule defines.

The second process includes performance modeling and validation. As shown in the dotted box in Figure 1, performance modeling starts with a random sample. We use the sample to build a performance model automatically by statistical learning. From the performance model, we derive a set of decision rules to enable fast, direct question answering for performance prediction. To validate the current performance model, users can measure the actual performance of configuration $\mathbf{x}$ and then compare its actual performance measurement with its performance prediction. Next, configuration $\mathbf{x}$ and its actual performance measurement can be reused to expand the sample and then to rebuild the performance model. Thus, the approach works in a progressive way and improves performance predictions based on updated samples.

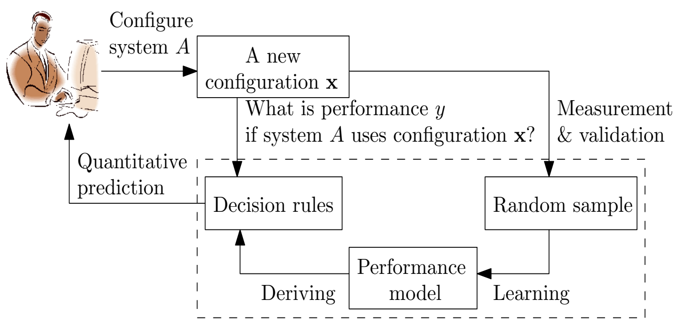


Figure 1. Overview of the approach

B. CART-Based Performance Modeling

In this section, we explain the process of variability-aware performance modeling via statistical learning in detail, as illustrated in the dotted box in Figure 1. As explained in Section III, the problem is to find a function f that predicts the performance value y for a configuration $\mathbf{x}$ based on a sample S. We use CART [5], [11] to address this problem. The basic idea is as follows. We *recursively partition* the sample into smaller *segments* until we can fit a simple *local* prediction model into each segment; and finally we organize all the local models into a *global* prediction model, which is represented as a binary decision tree.

Figure 2 shows a performance model generated by CART based on the X264 sample in Table I. CART starts with the sample S that contains 16 configurations $\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_{16}$ and their performance measurements $y_1, y_2, ..., y_{16}$. Then, CART partitions the sample S into two segments S_L and S_R by

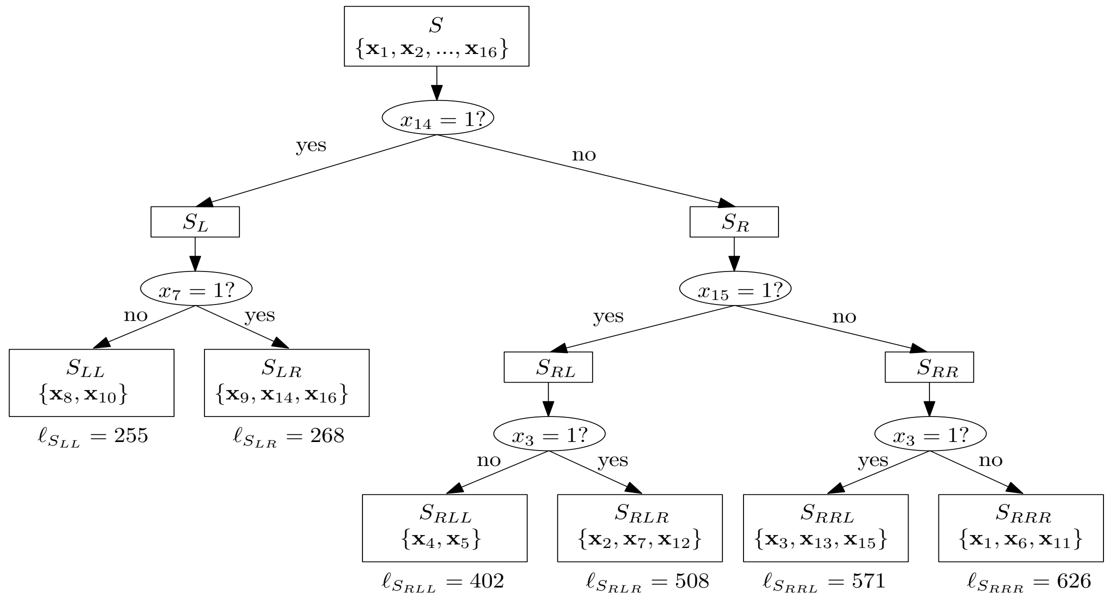

S
$\{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_{16}\}$
$x_{14} = 1?$
yes
no
S_L
S_R
$x_7 = 1?$
no
yes
S_{LL} $\{\mathbf{x}_8, \mathbf{x}_{10}\}$
S_{LR} $\{\mathbf{x}_9, \mathbf{x}_{14}, \mathbf{x}_{16}\}$
$\ell_{S_{LL}} = 255$
$\ell_{S_{LR}} = 268$
$x_{15} = 1?$
yes
no
S_{RL}
S_{RR}
$x_3 = 1?$
no
yes
$x_3 = 1?$
yes
no
S_{RLL} $\{\mathbf{x}_4, \mathbf{x}_5\}$
S_{RLR} $\{\mathbf{x}_2, \mathbf{x}_7, \mathbf{x}_{12}\}$
S_{RRL} $\{\mathbf{x}_3, \mathbf{x}_{13}, \mathbf{x}_{15}\}$
S_{RRR} $\{\mathbf{x}_1, \mathbf{x}_6, \mathbf{x}_{11}\}$
$\ell_{S_{RLL}} = 402$
$\ell_{S_{RLR}} = 508$
$\ell_{S_{RRL}} = 571$
$\ell_{S_{RRR}} = 626$

Figure 2. Example performance model of X264 generated by CART based on the random sample of Table I

exhaustively searching over all feature-selection variables in X for the best *split* that minimizes the total prediction errors in its two resulting segments. For example, as shown in Figure 2, the first best split for the X264 sample S is the feature-selection variable x_{14}, because choosing x_{14} to partition S produces the minimal total prediction errors in the two resulting segments S_L and S_R. After partitioning, configurations with $x_{14} = 1$ go to the left segment S_L, and configurations with $x_{14} = 0$ go to the right segment S_R. Each segment is partitioned recursively by further splits, such as the variables x_7, x_{15}, and x_3.

For each segment S_i, we use the *sample mean* of the actual performance measurements as the local prediction model of the segment to make prediction fast [4]:

$$\ell_{S_i} = \frac{1}{|S_i|} \sum_{y_j \in S_i} y_j \tag{2}$$

The local model of each segment identifies the common feature selections (the corresponding branch from the first split to the current split) and the average performance of the configurations contained in the segment. For example, the local model of the leftmost leaf in Figure 2 indicates the common feature selections $(x_{14} = 1, x_7 = 0)$ and the average performance $\ell_{S_{LL}} = \frac{1}{2}(y_8 + y_{10}) = 255$s for the two configurations $\mathbf{x}_8, \mathbf{x}_{10}$.

To penalize the prediction errors in each segment S_i that uses the corresponding local model ℓ_{S_i}, we adopt the most common and convenient loss function, the sum of *squared error loss* [11]:

$$\sum_{y_j \in S_i} L(y_j, \ell_{S_i}) = \sum_{y_j \in S_i} (y_j - \ell_{S_i})^2 \tag{3}$$

Thus, the best split for each segment S_i is determined to partition S_i into two segments S_{iL} and S_{iR} such that:

$$\sum_{y_j \in S_{iL}} L(y_j, \ell_{S_{iL}}) + \sum_{y_j \in S_{iR}} L(y_j, \ell_{S_{iR}}) \textit{ is minimal} \tag{4}$$

To prevent *underfitting* the input sample, we may expect that each final segment (i.e., *leaf*) is small enough to produce as small prediction errors as possible; but excessive partitioning may give rise to *overfitting* the input sample and thus compromise prediction accuracy for other configurations [4], [11].[3] Hence, determining when is the best time to stop the recursive partitioning process is an empirical activity to trade-off underfitting and overfitting, and it often involves a manual, iterative process of parameter tuning [4], [24]. For our case studies, we use two important parameters and define a set of empirically-determined parameter settings to automatically control the termination of the recursive partition process, as we explain in Section V.

Suppose that there are q leaves in the tree structure of a performance model; we organize all the local models of these leaves into a global model as follows:

$$f(\mathbf{x}) = \sum_{i=1}^{q} \ell_{S_i} I(\mathbf{x} \in S_i) \tag{5}$$

where $I(\mathbf{x} \in S_i)$ is an indicator function to denote if configuration $\mathbf{x}$ belongs to a leaf S_i. To determine to which leaf a configuration $\mathbf{x}$ belongs, we match the feature selections of a configuration with the corresponding branch in the tree, from the first split to a leaf. For example, in the tree shown in Figure 2, if a configuration $\mathbf{x}$ satisfies $(x_{14} = 1, x_7 = 0)$,

[3]If an algorithm works poorly even with the existing data, then the algorithm *underfits* the existing data. If an algorithm works well with the existing data, but not with new data, then the algorithm *overfits* the existing data.

which is consistent with the feature selections of the leftmost branch, then this configuration falls into the leftmost leaf S_{LL}. The global model for the tree shown in Figure 2 is specified as follows:

$$\begin{aligned} f(\mathbf{x}) = 255 &* I(x_{14}=1, x_7=0) \\ + 268 &* I(x_{14}=1, x_7=1) \\ + 402 &* I(x_{14}=0, x_{15}=1, x_3=0) \\ + 508 &* I(x_{14}=0, x_{15}=1, x_3=1) \\ + 571 &* I(x_{14}=0, x_{15}=0, x_3=1) \\ + 626 &* I(x_{14}=0, x_{15}=0, x_3=0) \end{aligned}$$

We can derive a set of decision rules from a global performance model to provide direct performance predictions for users. Each branch in the tree structure of a performance model indicates a decision rule. For example, we can derive the following if-then decision rule from the leftmost branch of the tree shown in Figure 2: if a configuration satisfies $(x_{14}=1, x_7=0)$, then its predicted performance is 255s.

V. Implementation

We implemented our approach using R 2.15.1 and JAVA (ECLIPSE 4.2 with JVM 1.7). R is a language and environment for statistical computing and graphics.[4] We used the R packages RATTLE and RPART to implement CART and to generate the performance models [26]. We developed a rule generator to parse the built performance models and generate decision rules. We also experimented with two CART variants, *Random Forests* and *Boosting* [26], which try to enhance the prediction effects of CART, but we observed similar prediction improvements on our evaluated case studies through parameter tuning. Thus, we choose a simple solution that uses only CART for our case studies.

As mentioned in Section IV-B, we use two important parameters to control the recursive partitioning process of CART: $minbucket$ is the minimum sample size for any leaf of the tree structure of a performance model; and $minsplit$ is the minimum sample size for any segment in the tree before the segment is considered for further partitioning. A segment is not considered for partitioning if its sample size is less than $minsplit$. We performed a set of preliminary experimental tests to identify parameter settings that trade-off underfitting and overfitting for our case studies. Moreover, to implement a fully-automated process of performance modeling by CART, we aim at setting the two parameters automatically in terms of the size of the input sample, i.e., $|S|$. Since the size of most of the samples used in our case studies is less than 100, we set the threshold of 100 to distinguish random samples of different sizes. Finally, we use the following empirically-determined parameter settings to achieve automated performance modeling and reasonable prediction accuracy for our case studies: if $|S| \leq 100$, then $minbucket = \lfloor \frac{|S|}{10} + \frac{1}{2} \rfloor$ and $minsplit = 2 * minbucket$; if $|S| > 100$, then $minsplit = \lfloor \frac{|S|}{10} + \frac{1}{2} \rfloor$ and $minbucket = \lfloor \frac{minsplit}{2} \rfloor$; the minimum of $minbucket$ is 2; and the minimum of $minsplit$ is 4.[5]

[4]http://www.r-project.org/

[5]$\lfloor\ \rfloor$ indicates rounding down, i.e., $\lfloor x \rfloor = max\{n \in \mathbb{Z} | n \leq x\}$.

VI. Evaluation

We conducted a series of case studies to evaluate our approach. We aim at answering the following research questions:

RQ 1: How accurate is the approach of variability-aware performance prediction? (Section VI-C)
RQ 2: Can the prediction process be progressive? (Section VI-C)
RQ 3: How fast is the prediction process? (Section VI-D)
RQ 4: Is it possible to make accurate predictions using only small random samples? (Section VI-E)
RQ 5: What are the strengths and weaknesses of our approach compared to existing techniques? (Section VI-G)

A. Subject Systems

We performed our case studies on a publicly-available dataset, deployed with the SPLCONQUEROR tool.[6] The dataset covers a reasonable spectrum of practical application scenarios. As shown in Table II, there are six existing real-world configurable systems with different characteristics: different sizes (42 thousand to 300 thousand lines of code, 192 to millions of configurations), different implementation languages (C, C++, and JAVA), and different configuration mechanisms (conditional compilation, configuration files, and command-line options). Moreover, the dataset contains the whole population of each system, i.e., all configurations of each system and their performance measurements (the exception is SQLITE, for which the dataset contains $4,553$ configurations for prediction modeling and 100 additional random configurations for prediction evaluation [19]). For each system, the performance has been measured using a standard benchmark, either delivered by its vendor (e.g., ORACLE's standard benchmark for BERKELEY DB) or used widely in its application domain (e.g., AUTOBENCH and HTTPERF for the APACHE Web Server).

B. Experimental Setup

In our experiments, the *independent variables* are the subject system and the size of the input sample. The prediction fault rate and the time cost of building a performance model are measured as the *dependent variables*. The prediction fault rate is the relative difference between the *actual* performance and the *predicted* performance, i.e., FR=$\frac{|actual-predicted|}{actual}$. Correspondingly, the prediction accuracy is $1-$FR.

To reduce the fluctuations of the dependent variables caused by random generation, we performed five repetitions for each combination of the independent variables. That is, for each subject system, we repeated five times generating a random sample of a certain size and subsequently measured the dependent variables after applying our approach to the sample. We took only the average of these measurements for analysis. We performed all measurements on the same Windows 7 machine with Intel Core i5 CPU 2.5 GHz and 8 GB RAM.

[6]The dataset is available at http://fosd.de/SPLConqueror.

TABLE II
OVERVIEW OF THE SIX SUBJECT SYSTEMS; LANG.—LANGUAGE; LOC—LINES OF CODE; $|\mathbf{X}|$—NUMBER OF ALL VALID CONFIGURATIONS; N—NUMBER OF ALL FEATURES; M—NUMBER OF CONFIGURATIONS REQUIRED BY THE PAIR-WISE HEURISTIC OF SPLCONQUEROR

	System	Domain	Lang.	LOC	$\|\mathbf{X}\|$	N	M
1	APACHE	Web Server	C	230,277	192	9	29
2	LLVM	Compiler	C++	47,549	1,024	11	62
3	X264	Encoder	C	45,743	1,152	16	81
4	BERKELEY DB	Database	C	219,811	2,560	18	139
5	BERKELEY DB	Database	JAVA	42,596	400	26	48
6	SQLITE	Database	C	312,625	3,932,160	39	566

For each subject system, we randomly selected a certain number of configurations from the whole population as the *training sample* for prediction modeling and all remaining as the *test sample* for prediction evaluation. Take the X264 system as an example, if we select 16 configurations as the training sample, then the remaining $1,136$ configurations form the test sample.

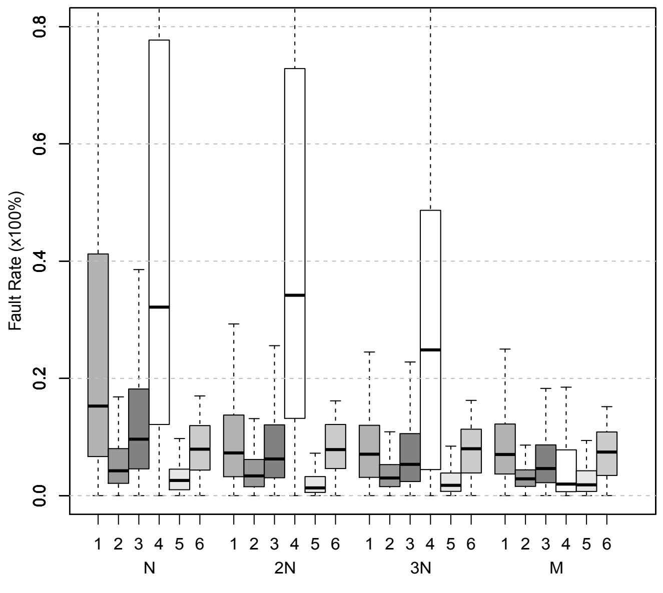
To assess the effectiveness of our approach working with random samples of different sizes, we use four sizes for the training sample of each subject system: N, $2N$, $3N$, and M, where N is the number of all features of each system, and M is the number of all specific configurations required by the pair-wise heuristic of SPLCONQUEROR. We list the concrete values of N and M for each system in the rightmost two columns in Table II. We choose size N, $2N$, and $3N$, because measuring a sample whose size is linear in the number of all features is likely feasible and reasonable in practice, given the high cost of performance measurement. For example, the number of even one percent of all configurations of X264 (i.e., 115) is still much more than the triple fold of the number of all features (i.e., 48). We choose size M, so that we can compare our approach to SPLCONQUEROR.

C. Experiment on Prediction Fault Rate

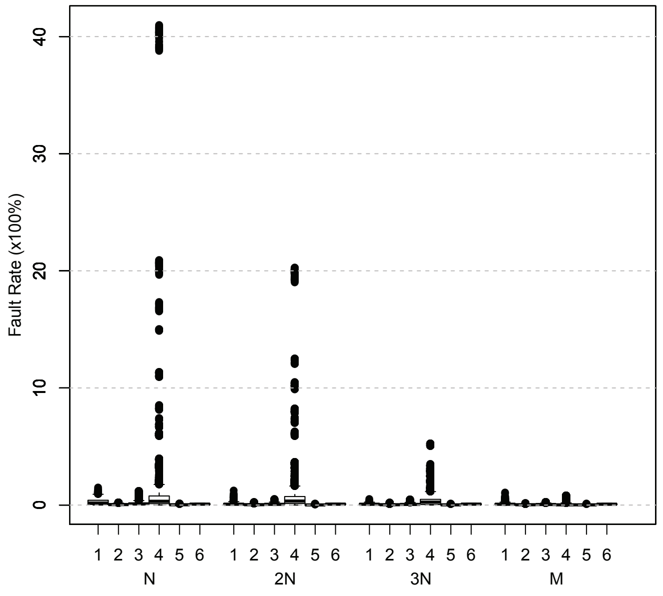
CART has been proved effective for many non-linear regression problems [4], [5]. Moreover, most statistical learning techniques can make more accurate predictions when more data are available [11]. Hence, the hypotheses of this experiment are as follows.

1) Hypotheses: Our CART-based approach is effective for variability-aware performance prediction (for RQ 1). Furthermore, it works progressively and improves the prediction accuracy when a larger sample is available (for RQ 2).

2) Results: We measured the prediction fault rate for the six systems listed in Table II and the four sample sizes (N, $2N$, $3N$, and M). We present the experimental results using different statistical measures. Figure 3a shows the *boxplots* of the results excluding *outliers*, such that other statistical measures such as the *median* and *quartiles* can be shown clearly.[7] Figure 3b includes all outliers. Table III (Column



(a) Excluding outliers



(b) Including outliers

Figure 3. Boxplots of the prediction fault rates for the six systems (1 to 6 listed in Table II) and the four sample sizes (N to M listed in Table II)

"Fault Rate") lists the *mean* and *standard deviation* of the prediction fault rate for each system and each sample size.

As shown in Figure 3 and Table III, for any statistical measure (the mean, standard deviation, median, or outlier), we observe a robust decreasing trend of the prediction fault rate when the sample size increases from N to M for each system.[8] As listed in Table III, based on a random sample of size N (Column "Fault Rate – N"), the fault rate is $8\,\%$ or

[7]A boxplot represents statistical data on a plot, in which a rectangle is drawn to represent the second and third quartiles, usually with a vertical line inside to indicate the median value. The lower and upper quartiles are shown as horizontal lines either side of the rectangle. An outlier is one that appears to deviate markedly from other members of the sample in which it occurs.

[8]Note that M depends on the number of features and on the configuration constraints among features in a case study. In most cases, M is greater than $3N$; the exception is BERKELEY DB JAVA where $M = 48$ and $3N = 78$.

TABLE III
MEAN±STANDARD DEVIATION OF THE PREDICTION FAULT RATE (%) AND TIME COST (ms) FOR THE SIX SYSTEMS (1 TO 6 LISTED IN TABLE II) AND THE FOUR SAMPLE SIZES (N TO M LISTED IN TABLE II)

	Fault Rate (%)				Time Cost (ms)			
	N	$2N$	$3N$	M	N	$2N$	$3N$	M
1	26.9±28.4	11.6±14.4	8.4±6.7	9.7±10.8	26±5	30±12	24±5	24±5
2	5.7±4.9	4.5±4.2	4.0±3.6	3.3±2.4	34±5	42±13	36±9	34±9
3	15.1±18.1	8.5±7.5	7.2±6.4	6.4±5.7	24±5	20±0	26±5	22±4
4	112.4±354.6	98.3±243.1	46.8±70.7	7.8±13.2	26±5	42±8	26±5	34±5
5	3.2±2.6	2.2±2.3	2.6±2.5	2.7±2.5	34±9	36±9	34±5	32±4
6	8.0±4.5	8.1±4.4	7.6±4.4	7.2±4.2	44±5	42±4	44±9	90±0

less for three subject systems (LLVM, BERKELEY DB JAVA, and SQLITE, in Rows 2, 5 and 6). Based on a random sample of size M, we achieve a mean prediction fault rate of 6.2 %, on average, for all six subject systems, i.e., the average of all mean prediction fault rates listed in Column "Fault Rate – M".

3) Discussion: An average prediction accuracy of 93.8 % on small random samples of size M for six real-world configurable systems demonstrates the effectiveness of our approach for variability-aware performance prediction. For three subject systems, the approach even produces a prediction accuracy of 92 % or higher, based on small random samples of size N. Moreover, our approach does show a robust increasing trend of prediction accuracy with the increasing sample size. Thus, the experiment confirms that our approach can work progressively with random samples of any user-defined size (i.e., N or larger) and improve the prediction accuracy when more data are available.

D. Experiment on Time Cost

For RQ 3, the time consumed by the prediction process of our approach mainly stems from performance modeling using CART. Once the performance model is built, as explained in Section IV, deriving a decision rule and providing the prediction result for users are instantaneous. Furthermore, as CART has been widely used in statistics and data-mining applications, and it has been shown fast and reliable [4], [11], the hypothesis of this experiment is as follows.

1) Hypothesis: The time of building a performance model by CART is reasonable.

2) Results: We measured the time of building a performance model in the same experimental context as the experiment of Section VI-C. The results are listed in Table III (Column "Time Cost"). For five of the six systems, the time of building a performance model on any random sample of size from N to M is approximately 42 milliseconds (ms). Only for SQLITE and the sample size $M = 566$, the time cost reaches a high of 90ms, which is still a reasonable amount of time.

3) Discussion: Although we perform CART with an exhaustive search over all feature-selection variables for the best split that minimizes the total prediction errors, as explained in Section IV-B, the constant local model defined in Equation 2 makes the search process fast, because there is no complicated calculation for the total prediction errors defined in Equation 4. Moreover, the time of at most 90ms needed for all six subject systems and for any sample size from N to M demonstrates that our approach is highly efficient for variability-aware performance prediction.

E. Comparative Analysis of Performance Distributions

The previous experiments demonstrate the effectiveness of our approach, however, we still want to give evidence why the approach works with small random samples (for RQ 4). Since the approach depends on CART to address a non-linear regression problem, a general explanation from the statistical-learning theory is that CART works well when the problem it addresses or the data it evaluates does fit the regressive pattern it builds [4]. Moreover, as explained in Section IV-B, CART builds a tree-like prediction model that recursively partitions a sample and renders the total prediction errors in each partition minimal; this way, the prediction model always fits the sample well. If the sample can represent the whole population or reflect the important characteristics of the whole population, then the prediction model built on the sample also fits the whole population well and makes accurate predictions.

Since our prediction targets numeric performance values, the performance distribution is an important characteristic for performance prediction. Thus, we conducted a comparative analysis of performance distributions between random samples and the corresponding whole populations, guided by the following hypothesis.

1) Hypothesis: Our approach works well with a small random sample when the sample has a similar performance distribution as the whole population.

2) Results: For each subject system, we collected all the random samples generated in the previous experiments. Then, we visualized the average performance distribution for each sample size from N to M, to mitigate the influence of a specific performance distribution of a certain random sample. Furthermore, we visualized the performance distribution of the whole population of each system, to be able to compare it to the performance distribution of each random sample of size from N to M.

Due to the space limit, we present the experimental results for only one subject system here.[9] Figure 4 shows the *his-*

[9]The experimental results of all six subject systems (1 to 6 listed in Table II) are available in a technical report at http://gsd.uwaterloo.ca/node/527.

tograms of the performance distributions for the four sample sizes (N to M) and the whole population of X264.[10]

As shown in Figure 4e, the performance distribution of the whole population of X264 is roughly a distribution with two peaks. The performance distribution of the random sample of size N (Figure 4a) identifies the two peaks, but misses the correct locations of these peaks. The performance distributions for the sample size from $2N$ (Figure 4b), $3N$ (Figure 4c), to M (Figure 4d) gradually move and form the two peaks approximate to the precise locations, as shown in Figure 4e. With such a gradual process that generates a more similar performance distribution as the whole population, the prediction fault rate shows a robust decreasing trend from $15.1\,\%$, $8.5\,\%$, $7.2\,\%$ to $6.4\,\%$ when the sample size increases from N, $2N$, $3N$ to M.

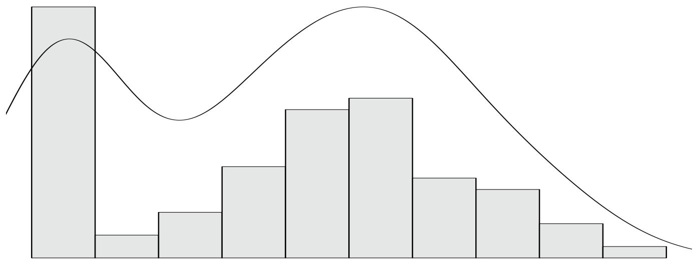
Similarly, for each of the other five subject systems, the random sample of size M always exhibits a very similar performance distribution as the whole population. For the three systems (LLVM, BERKELEY DB JAVA, and SQLITE) with $92\,\%$ or higher prediction accuracy based on a random sample of size N, a similar performance distribution as the whole population can be found on the random sample of size N.

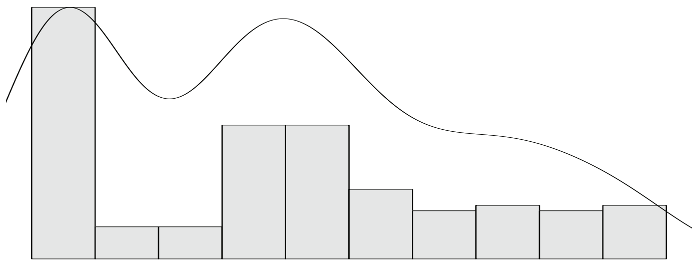
3) Discussion: The comparative analysis of performance distributions between random samples and the whole populations reveals that our approach works well with a small random sample when the sample has a similar performance distribution as the whole population. In fact, we found explicit evidence that a sample does reflect some important characteristics of the whole population when we can produce accurate predictions based on it. However, we are aware that the performance distribution may be just one of the relevant characteristics for performance prediction. These characteristics may involve, for example, the number and dispersion of distinct values as well as the feature coverage. A quantitative study on the similarity between a random sample and the whole population, involving more characteristics for performance prediction, shall be conducted in future work.

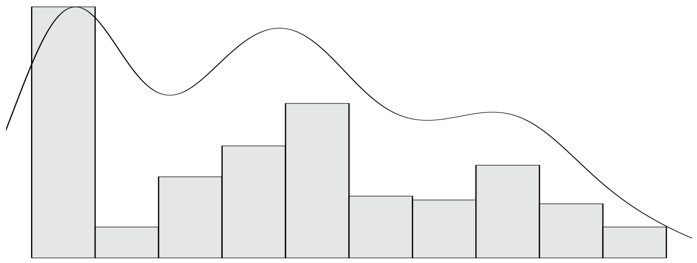
F. Threats to Validity

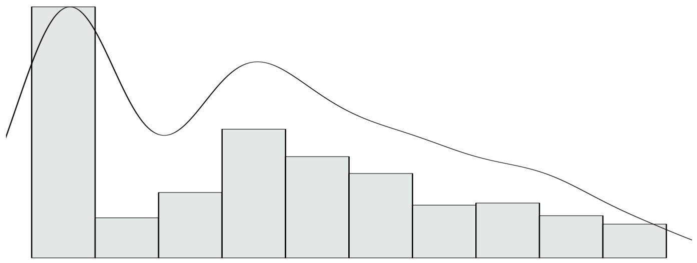
To enhance internal validity, we performed automated random sampling avoiding misleading effects of specific-selected training samples and test samples. We randomly selected samples of four sizes (N to M) respectively from the whole population of each subject system as the training sample, and all of the rest as the test sample. We repeated each random sampling five times with freshly generated training samples and test samples of the same size. The exception is the test sample of SQLITE, in which the original authors could not measure all valid configurations in reasonable time; thus, to mitigate the possible effects of missing some important configurations, they sampled 100 additional random configurations for prediction evaluation [19].

[10]A histogram provides a quick and intuitive visualization of the distribution of the data [26]; it consists of two parts: the *vertical bars*, each of which displays the frequency of each value range; and the *density estimate curve*, which shows a more accurate display of the distribution of the data.


(a) Random sample of size N


(b) Random sample of size $2N$


(c) Random sample of size $3N$


(d) Random sample of size M

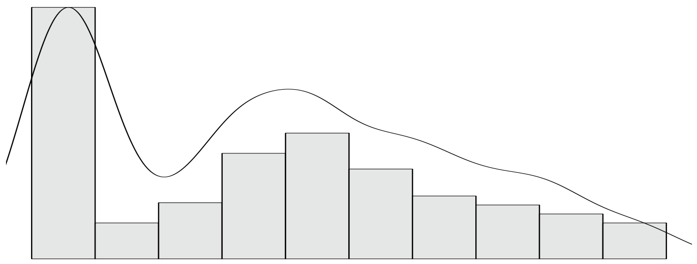

(e) Whole population

Figure 4. Histograms of the performance distributions of the random samples of four sizes (N to M listed in Table II) and of the whole population of X264 (X-axis: Performance (seconds); Y-axis: Relative Frequency)

To automate the process of performance modeling by CART, we use two important parameters ($minbucket$ and $minsplit$) and fix others provided by the R packages to control the recursive partitioning process of CART. For each case study, we followed the same parameter settings to generate the performance models automatically. We cannot guarantee that the prediction fault rate and the time cost of performance modeling obtained in our experiments depend on certain shapes of the performance models built by CART. However, to avoid misleading effects of specially-shaped performance models, we generated all performance models automatically, repeated each measurement (the prediction fault rate or the time cost) for each system and each sample size five times, and took only the average of these measurements for analysis.

To increase external validity, we used a public dataset with six systems spanning different domains, with different sizes, different configuration mechanisms, and different implementation languages. All systems have been deployed and used in real-world scenarios. Moreover, the performance is measured by standard benchmarks in the respective application domain. However, we are aware that the results of our experiments are not automatically transferable to all other configurable systems, but we are confident that we controlled this threat sufficiently.

G. Strengths and Weaknesses of the Approach

To answer RQ 5, we compared our approach to SPLConqueror, a most recent approach to performance prediction for configurable software systems [19]. We summarize the strengths and weaknesses of the two approaches, which guides users to choose one or the other in practice.

1) Prediction Fault Rate: In our experimental setup, N is the number of all features of a system, and M is the number of configurations required by the pair-wise heuristic of SPLConqueror. Moreover, N is similar to the number of configurations required by the feature-wise measurement of SPLConqueror. Thus, we can compare the two approaches according to these sample sizes. Table IV lists the prediction fault rates produced by the two approaches when we apply them to the (random or specific) samples of size N and M for the six subject systems listed in Table II.

Our approach produces a prediction fault rate of 8 % or less based on a random sample of size N for three systems (Rows 2, 5, and 6 in Table IV); and it produces an average of 6.2 % prediction fault rate for all six systems, when the sample size reaches M. By comparison, SPLConqueror produces a prediction fault rate of 7.8 % using the feature-wise measurement on a specific sample of size N for two systems (Rows 2 and 6); and it produces an average of 9.1 % prediction fault rate using the pair-wise heuristic on a specific sample of size M for all six systems. When other heuristics for higher-order feature interactions are considered, SPLConqueror can produce an average of 5 % prediction fault rate for all six systems (not listed in Table IV) [19], which is more accurate than our approach, but requires additional measurements.

2) Prediction Effort: The higher prediction accuracy of SPLConqueror comes at a cost: SPLConqueror needs additional effort to select specific configurations and to detect performance-relevant feature interactions. When we encounter a large-scale system with a great number of features, such

TABLE IV
Mean±Standard Deviation of the prediction fault rate (%) for the six systems (1 to 6 listed in Table II) and the two sample sizes (N and M listed in Table II) using our approach and SPLConqueror; The number in bold indicates the best case in each row

	Our approach		SPLConqueror	
	N	M	N	M
1	26.9±28.4	9.7±10.8	14.9±24.8	**7.7**±11.2
2	5.7±4.9	**3.3**±2.4	7.8±9.0	7.4±10.2
3	15.1±18.1	**6.4**±5.7	29.6±22.0	17.9±27.2
4	112.4±354.6	7.8±13.2	44.1±42.3	**3.9**±5.3
5	3.2±2.6	**2.7**±2.5	17.7±19.6	8.5±9.6
6	8.0±4.5	**7.2**±4.2	7.8±9.2	9.3±12.5
Avg.	28.6±68.9	**6.2**±6.5	20.3±21.2	9.1±12.7

effort can be quite expensive [14]. In contrast, our approach supports random sampling and progressive performance prediction, which makes it more pragmatic. That is, our approach can work with random samples and make more accurate predictions progressively when more data are available. Furthermore, as shown in Section VI-D, our approach is highly efficient and the prediction process often takes only little time.

3) Sample Dependence: SPLConqueror works because it relies on specific samples selected by heuristics to detect performance-relevant feature interactions; our approach works because (1) the evaluated dataset fits well in the non-linear regression model we use, and (2) the random sample we use reflects the important characteristics of the whole population. As demonstrated in Section VI-E, our approach works well with a small random sample, provided it has a similar performance distribution as the whole population. However, if such a random sample happens to be skewed to some undesirable characteristics, the prediction effects of our approach might be affected. For example, users may prefer some specific configurations with certain features and always miss some other features; based on such a sample, our approach may not produce accurate predictions for the configurations selecting the missed features.[11]

4) Application Scope: A clear advantage of a regression technique is that it can support not only *Boolean* (e.g., a "selected" or "deselected" feature), but also *numeric* feature selections (e.g., the heap size or the CPU speed [24], [25]), which makes our approach applicable in wider domains. In contrast, the existing techniques for feature-interaction detection support only Boolean feature selections.

5) Summary: Both approaches have strengths and weaknesses. We expect that a combination of both approaches is beneficial to further increase prediction accuracy and reduce prediction effort in wider application domains (e.g., by combining heuristics used in SPLConqueror to identify suitable samples for CART). In particular, we provide the following guidance for users to choose between the two approaches in practice. (1) If a software system has a small number of features, such that the cost of detecting feature interactions is

[11]An exploratory experiment on missing features and skewed configurations is available in a technical report at http://gsd.uwaterloo.ca/node/484.

acceptable, users shall choose SPLConqueror, due to its higher prediction accuracy; otherwise, our proposed approach is superior, due to its reduced prediction effort and reasonable prediction accuracy. (2) If there is already a sample available, one has to check whether the sample satisfies certain feature-coverage criteria, before using SPLConqueror; whereas one can produce predictions directly based on the sample using our approach.

VII. Related Work

A. Model-Based Prediction

CART and its variants, such as Random Forests and Boosting, have been widely used in statistics and data mining, because CART's algorithm is fast and reliable, and its tree structure can provide insight into the relevant input variables for prediction [4], [11].

Thereska et al. [24] proposed a practical performance model based on CART for interactive client applications, such as Microsoft Office and Mozilla. They focused on a range of deployment parameters from the users' application environment, such as CPU speed and memory size; instead, we consider the configuration options of a software system. Moreover, our approach targets all kinds of configurable software systems, as long as the valid configurations can be derived.

Westermann et al. [25] presented an approach to the automated improvement of performance-prediction functions by three measurement-point-selection strategies based on the prediction accuracy. They constructed the prediction functions by statistical inference techniques, including CART. This approach, however, assumes that all input variables of the prediction function are already relevant to performance; while our approach does not have such a restriction, but considers all features of a software system.

Even though the above studies have demonstrated the effects of CART on performance prediction for different case studies [24], [25], they did not explicitly provide evidence for why CART does or does not work. We found that our approach works well with a small random sample when the sample has a similar performance distribution as the whole population.

Happe et al. [10] proposed a compositional reasoning approach based on component specifications with resource demands and predicted execution time. Their approach is restricted to component-based systems, whereas our approach is applicable to all configurable systems, once their configurable options are abstracted as features.

Tawhid and Petriu [23] presented a model-driven approach to deriving a performance model from an extended feature model with performance-analysis information. The approach requires detailed up-front knowledge from a domain-specific performance analysis, which makes tuning prediction for accuracy difficult. Our approach avoids these problems by directly working with performance measurements.

Ramirez and Cheng [17] presented an approach that leverages goal-based models to facilitate the automatic derivation of utility functions at the requirements level; our approach works at the level of actual program variants.

B. Measurement-Based Prediction

A most recent measurement-based prediction technique is SPLConqueror [19], [21]. We have compared it to our approach and discussed the strengths and weaknesses of both approaches in Section VI-G.

Sincero et al. [22] used existing configurations and measurements to predict a configuration's non-functional properties. They designed the Feedback approach to find the correlation between feature selections and measurements and to provide qualitative information about how a feature influences a non-functional property during the configuration process. In contrast to our approach, their approach does not actually predict a performance value quantitatively.

Chen et al. [6] combined benchmarking and profiling to predict the performance of component-based applications. In contrast, our approach correlates performance measurements with configurations and can work with any set of configurations measured by simulation or by monitoring in the field.

VIII. Conclusion

We proposed a progressive and variability-aware approach to performance prediction for configurable software systems based on random samples. The approach uses the statistical learning technique CART to build an explicit performance model that represents the correlation between feature selections and performance. We demonstrated the feasibility and effectiveness of our approach on six real-world systems, spanning different domains, implementation languages, and configuration mechanisms. Our empirical results show that the approach produces a prediction accuracy of $94\,\%$, on average, based on small random samples. Moreover, our approach shows a robust increasing trend of prediction accuracy as the sample size increases. A comparative analysis of performance distributions revealed that our approach works well when the corresponding sample has a similar performance distribution as the whole population. We compared our approach to a state-of-the-art technique, called SPLConqueror, and explored the strengths and weaknesses of the two approaches, to guide users to choose one or the other in practice.

Our approach has the potential of wide application to help users make trade-offs between feature selections and performance and to guide the configuration process [15]. In future work, we aim at performing systematic parameter tuning for CART and trying other regression techniques (e.g., Support Vector Machines [4]). Moreover, we aim at quantifying the similarity between a sample and the whole population, involving several characteristics for performance prediction. In addition, we will explore the potential of using our approach for configuration optimization [1], [9], [18], test generation [27], and bug prediction [12].

Acknowledgments

This work has been partially supported by NECSIS, Ontario Research Fund - Research Excellence Project on Model-Based Development, the German Research Foundation (AP 206/4 and AP 206/5), and MT-LAB (a VKR Centre of Excellence).

References

[1] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, and I. Meedeniya, "Software Architecture Optimization Methods: A Systematic Literature Review," *IEEE Transactions on Software Engineering*, vol. 39, no. 5, pp. 658–683, 2013.

[2] S. Apel and C. Kästner, "An Overview of Feature-Oriented Software Development," *Journal of Object Technology*, vol. 8, no. 5, pp. 49–84, 2009.

[3] D. Batory, D. Benavides, and A. Ruiz-Cortes, "Automated Analyses of Feature Models: Challenges Ahead," *Communications of the ACM*, vol. 49, no. 12, pp. 45–47, 2006.

[4] R. Berk, *Statistical Learning from a Regression Perspective*. Springer, 2008.

[5] L. Breiman, J. Friedman, C. Stone, and R. Olshen, *Classication and Regression Trees*. Wadsworth and Brooks, 1984.

[6] S. Chen, Y. Liu, I. Gorton, and A. Liu, "Performance Prediction of Component-Based Applications," *Journal of Systems and Software*, vol. 74, no. 1, pp. 35–43, 2005.

[7] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[8] K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[9] 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," *Journal of Systems and Software*, vol. 84, no. 12, pp. 2208–2221, 2011.

[10] J. Happe, H. Koziolek, and R. Reussner, "Facilitating Performance Predictions Using Software Components," *IEEE Software*, vol. 28, no. 3, pp. 27–33, 2011.

[11] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, 2nd ed. Springer, 2009.

[12] K. Herzig, S. Just, and A. Zeller, "It's not a Bug, it's a Feature: How Misclassification Impacts Bug Prediction," in *Proc. ICSE*. IEEE, 2013.

[13] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," SEI, CMU, Tech. Rep. SEI-90-TR-021, 1990.

[14] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer, "Scalable Analysis of Variable Software," in *Proc. ESEC/FSE*. ACM, 2013.

[15] A. Murashkin, M. Antkiewicz, D. Rayside, and K. Czarnecki, "Visualization and Exploration of Optimal Variants in Product Line Engineering," in *Proc. SPLC*. ACM, 2013.

[16] K. Pohl, G. Bockle, and F. van der Linden, *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer-Verlag, 2005.

[17] A. Ramirez and B. Cheng, "Automatic Derivation of Utility Functions for Monitoring Software Requirements," in *Proc. MODELS*. IEEE, 2011.

[18] 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 *Proc. ICSE*. IEEE, 2013.

[19] N. Siegmund, S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, and G. Saake, "Predicting Performance via Automated Feature-Interaction Detection," in *Proc. ICSE*. IEEE, 2012.

[20] N. Siegmund, M. Rosenmüller, C. Kästner, P. Giarrusso, S. Apel, and S. Kolesnikov, "Scalable Prediction of Non-functional Properties in Software Product Lines: Footprint and Memory Consumption," *Information and Software Technology*, vol. 55, no. 3, pp. 491–507, 2013.

[21] N. Siegmund, M. Rosenmüller, M. Kuhlemann, C. Kästner, S. Apel, and G. Saake, "SPL Conqueror: Toward Optimization of Non-Functional Properties in Software Product Lines," *Software Quality Journal*, vol. 20, no. 3-4, pp. 487–517, 2012.

[22] J. Sincero, W. Schröder-Preikschat, and O. Spinczyk, "Approaching Non-functional Properties of Software Product Lines: Learning from Products," in *Proc. APSEC*. IEEE, 2010.

[23] R. Tawhid and D. Petriu, "Automatic Derivation of a Product Performance Model from a Software Product Line Model," in *Proc. SPLC*. IEEE, 2011.

[24] E. Thereska, B. Doebel, A. Zheng, and P. Nobel, "Practical Performance Models for Complex, Popular Applications," in *Proc. SIGMETRICS*. ACM, 2010.

[25] D. Westermann, J. Happe, R. Krebs, and R. Farahbod, "Automated Inference of Goal-Oriented Performance Prediction Functions," in *Proc. ASE*. ACM, 2012.

[26] G. Williams, *Data Mining with Rattle and R: The Art of Excavating Data for Knowledge Discovery*. Springer, 2011.

[27] S. Yoo and M. Harman, "Regression Testing Minimization, Selection and Prioritization: A Survey," *Software Testing, Verification & Reliability*, vol. 22, no. 2, pp. 67–120, 2012.