Performance-Influence Model for Highly Configurable Software with Fourier Learning and Lasso Regression

Huong Ha
The University of Newcastle
Callaghan, Australia
huong.ha@uon.edu.au

Hongyu Zhang
The University of Newcastle
Callaghan, Australia
hongyu.zhang@newcastle.edu.au

***Abstract*—Many software systems are highly configurable, which provide a large number of configuration options for users to choose from. During the maintenance and operation of these configurable systems, it is important to estimate the system performance under any specific configurations and understand the performance-influencing configuration options. However, it is often not feasible to measure the system performance under all the possible configurations as the combination of configurations could be exponential. In this paper, we propose PerLasso, a performance modeling and prediction method based on Fourier Learning and Lasso (Least absolute shrinkage and selection operator) regression techniques. Using a small sample of measured performance values of a configurable system, PerLasso produces a performance-influence model, which can 1) predict system performance under a new configuration; 2) explain the influence of the individual features and their interactions on the software performance. Besides, to reduce the number of Fourier coefficients to be estimated for large-scale systems, we also design a novel dimension reduction algorithm. Our experimental results on four synthetic and six real-world datasets confirm the effectiveness of our approach. Compared to the existing performance-influence models, our models have higher or comparable prediction accuracy.**

I. Introduction

Many software systems are highly configurable. User can customize these systems by selecting a set of configuration options (or features). For example, the popular SQLite database system has 39 configuration options and their combination can reach up to 3 millions [21]. As different configurations may lead to different system performance, it is important to predict the performance of a system under a certain configuration and to understand how different configuration options and their interactions influence system performance. In this way, engineers can make rational configuration decisions that satisfy user's quality requirements during the maintenance and operation of the configurable systems.

Software engineering researchers have proposed to sample a small set of configurations, measure system performance under these configurations, and then build performance-influence models that describe how configuration options and their interactions influence the system performance [20–22, 28]. With the performance-influence models, it becomes easier to understand, debug and optimize highly configurable software systems [20]. Specifically, engineers can use the model to estimate the system performance under any configurations and check if the system behaves as expected. If not, the model can be used for fault diagnosis.

To build the performance-influence model, one approach was suggested in [20–22], namely *SPLConqueror*. The idea is to formulate the system performance value as a linear combination of functions which represent the influence of a single configuration option and the interaction among multiple configuration options. Stepwise linear regression is employed to learn the model from a sample set of measurable configurations and forward-backward feature selection is utilized to reduce the dimensionality problem of handling a very large number of configuration options and their interactions. Besides, several sampling heuristics and experimental designs for configuration options are combined with the suggested learning method to achieve good prediction accuracy.

Recently, Zhang et al. [28] proposed to formulate the software performance function as a Boolean function. Using Fourier transform of the Boolean function, the task of estimating the performance function becomes estimating its associated Fourier coefficients from a small set of samples. The Fourier coefficients are estimated by using the standard formula of computing Fourier coefficients of Boolean function on the sample set. The advantage of this approach is that it can derive a sample size that guarantees a theoretical boundary of the prediction accuracy. The disadvantage is that the number of samples required to achieve a desired accuracy is large (sometimes even more than the whole population of the system), especially for a relatively small system [28].

In this paper, we propose a new approach called *PerLasso*, which formulates the performance-influence model of configurable software using Fourier transformation as proposed in [28]. However, unlike [28], we apply the Lasso regression technique [25] to improve the accuracy of the Fourier estimates. Lasso regression [25], also called L_1 regression, is a regression analysis method that performs both variable selection and regularization in order to enhance the prediction accuracy and interpretability of the statistical model it produces. The

2576-3148/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSME.2019.00080

idea of using Lasso is inspired by the fact that the Fourier coefficients of software performance functions are usually very sparse (i.e., only a small number of configurations have significant impact on system performance) [10, 13, 20, 21, 28]. Applying this domain knowledge to the learning process by incorporating Lasso to the linear regression, we can reduce the estimates errors significantly.

A challenge when forming the performance prediction problem as a linear regression problem is that the number of parameters required to be estimated is exponential [20–22, 28]. Especially, for large configurable software systems, the number of parameters to be estimated can be so large that it causes high computation time. Even worse, the computer memory may not have enough space to do the computation. To address this challenge, we design a novel dimension reduction algorithm to remove the non-contributing Fourier coefficients of the linear regression sequentially. To be more precise, we suggest to split the whole configurations space into small groups, and within each group, we construct a Lasso regression model to determine which configurations have Fourier coefficients influence noticeably to the performance value. After obtaining the list of these configurations, we reconstruct the linear regression model and apply Lasso again to estimate the reduced dimensional Fourier coefficients parameters.

We have implemented *PerLasso* and evaluated it on four synthetic and six real-world datasets. The experimental results show that *PerLasso* can identify the influence of configuration options and their interactions on the system performance. Furthermore, *PerLasso* can achieve higher prediction accuracy compared to other state-of-the-art performance-influence models.

In summary, our contributions are as follows:

1) We propose a new approach to obtain performance-influence models of binary configurable software systems using Fourier transformation and Lasso regression techniques.

2) We design a novel dimension reduction technique such that the proposed approach can work effectively with large-scale configurable software systems.

3) We implement our proposed method, namely *PerLasso*, and evaluate its effectiveness by experiments on four synthetic and six real-world datasets.

The paper is organized as follows. Section II describes the background of the problem. Section III describes the proposed approach PerLasso in detail. Section IV describes our experimental design and results. We discuss our results in Section V and related work in Section VI. Finally, the conclusion is drawn in Section VII.

II. Background

A. Performance modeling for highly configurable software systems

Generally, a performance function of a software system with n binary configurations options can be modelled as follows [28]:

$$f(x_1, x_2, \ldots, x_n) \ : \{0,1\}^n \to \mathbb{R}. \tag{1}$$

where x_i $(i = 1, ..., n)$ is the variable that stores Boolean value indicating if a configuration option i^{th} is selected. Specifically, when the i^{th} option is selected, the value of the variable x_i is 1 and when it is deselected, the value of x_i is 0. The truth table of a performance function $f(x_1, x_2, \ldots, x_n)$ can be described in Table I.

TABLE I
THE TRUTH TABLE OF A PERFORMANCE FUNCTION

x_1	x_2	$\ldots$	x_n	$f(x)$
0	0	$\ldots$	0	y_1
1	0	$\ldots$	0	y_2
.	.	$\ldots$	.	.
1	1	$\ldots$	1	y_{2^n}

To build a cost-effective performance model, it is important that only a small sample of measured performance values are used. Furthermore, such a model should be able to reflect the influence of configuration options on system performance and be able to predict system performance under a new configuration.

B. Performance modeling with Fourier Learning

Through Fourier transform, any Boolean function $f(x)$ can be rewritten as follows [14]:

$$f(x) \ := \sum_{z\in\{0,1\}^n} \hat{f}(z)\chi_z(x), \tag{2}$$

where $\chi_z(x)$ are computed as,

$$\chi_z(x) \ := \begin{cases} +1 & \text{if } \sum_{i=1}^{n} z_i x_i \bmod 2 = 0 \\ -1 & \text{if } \sum_{i=1}^{n} z_i x_i \bmod 2 = 1 \end{cases} , \tag{3}$$

$\hat{f}(z)$ are called Fourier coefficients and $\chi_z(x)$ are called character functions. As proven in [14], any Boolean function is represented uniquely by its Fourier coefficients, hence, the problem of estimating a function $f(x),\ x \in \{0,1\}^n$ now becomes the problem of estimating its Fourier coefficients $\hat{f}(z),\ z \in \{0,1\}^n$.

Zhang et al. [28] applied this idea to the problem of predicting software performance. For a configurable software, suppose we have a set of configurations $S \subset \{0,1\}^n$ with their corresponding performance function values $f(x),\ x \in S$. The task of predicting software performance of the whole configurations becomes estimating its Fourier coefficients from the sampled data, i.e. $f(x),\ x \in S$. The core idea of their algorithm is that, instead of calculating the Fourier coefficients using the whole dataset as:

$$\hat{f}(z) = \langle f, \chi_z \rangle = \frac{1}{2^n} \sum_{x\in\{0,1\}^n} f(x)\chi_z(x), \tag{4}$$

now $\hat{f}(z)$ can be estimated as:

$$\hat{f}(z) = \langle f, \chi_z \rangle = \frac{1}{|S|} \sum_{x\in S} f(x)\chi_z(x). \tag{5}$$

The main benefit of the algorithm proposed in [28] is to provide theoretical guarantee of prediction accuracy and confidence level. However, as acknowledged in [28], one problem with this algorithm is that it cannot be applied to small datasets as for a small dataset, the required number of samples could be larger than the entire valid space, for any reasonable user-defined accuracy level. Even for large dataset, the number of samples required to achieve a good prediction accuracy is very large.

III. Proposed Approach

In this section, we describe in detail our proposed approach, namely PerLasso. We first describe how a performance model can be constructed from a small set of measurement data using Fourier learning and Lasso regression techniques (Section III.A). To reduce the number of Fourier coefficients to be estimated for large-scale systems, we also design a novel dimension reduction algorithm (Section III.B). The performance model can be used to identify performance-influencing configuration options and their interactions (Section III.C), and to predict performance for a new configuration (Section III.D). Finally, we also describe our tool implementation in Section III.E.

A. Performance modeling with Fourier Learning and Lasso Regression

Given a subset configurations $S = \{x^1, x^2, \dots, x^N\}$ and its performance values $f(x), x \in S$, we can apply (2) to construct the Fourier coefficients estimation problem as a linear regression problem:

$$\begin{bmatrix} f(x^1) \\ f(x^2) \\ \dots \\ f(x^N) \end{bmatrix} = \begin{bmatrix} \chi_{z^1}(x^1) & \chi_{z^2}(x^1) & \dots & \chi_{z^{2^n}}(x^1) \\ \chi_{z^1}(x^2) & \chi_{z^2}(x^2) & \dots & \chi_{z^{2^n}}(x^2) \\ \dots & \dots & \dots & \dots \\ \chi_{z^1}(x^N) & \chi_{z^2}(x^N) & \dots & \chi_{z^{2^n}}(x^N) \end{bmatrix} \begin{bmatrix} \theta^1 \\ \theta^2 \\ \dots \\ \theta^{2^n} \end{bmatrix}, \tag{6}$$

where $\theta = [\theta^1, \theta^2, \dots, \theta^{2^n}]^T = [\hat{f}(z^1), \hat{f}(z^2), \dots, \hat{f}(z^{2^n})]^T$ are the Fourier coefficients and N is the cardinality of the set S. The goal here is to estimate the Fourier coefficients θ based on based on $f(x), x \in S$.

One difficulty in solving (6) is that in reality, we often have $N \ll 2^n$ as the main purpose of the software performance prediction problem is to predict the performance values from a limited number of samples. With this condition, the problem (6) becomes ill-posed, meaning that there is an infinite number of parameters $\hat{\theta}$ that can obtain the perfect fit in (6). However, these estimates normally do not fit well to the new data. The key idea for solving ill-posed problems is to restrict the class of solutions by adding priori knowledge about the parameter θ. That is, among all the solution candidates $\hat{\theta}$ of (6) (i.e. those solutions that satisfy (6)), we pick only those solutions that satisfy our prior knowledge. For software performance functions, their Fourier coefficients are always very sparse, i.e. most of Fourier coefficients $\hat{f}(z), z \in \{0,1\}^n$ are zeros. The reason is that only a small number of configurations have significant impact on system performance [10, 13, 20, 21]. Hence, our goal here is to find a parameter $\hat{\theta}$ that fits perfectly in (6) and it must be sparse.

Up till now, the two most popular techniques to find a sparse estimate for a linear regression problem are the L_1 norm regularization (Lasso) [25] and the nuclear norm regularization [3]. In this paper, we choose to utilize the L_1 norm regularization (Lasso) due to its simplicity and effectiveness [25]. Applying Lasso to the linear regression problem in (6), we can find a sparse estimate of θ by solving the following optimization problem,

$$\min_{\theta} \ \frac{1}{2}\|Y - X\theta\|_2^2 + \lambda\|\theta\|_1, \qquad \lambda \in R^+, \tag{7}$$

where

$$\begin{aligned} Y &= [f(x^1),\ f(x^2),\ \dots,\ f(x^N)]^T, \\ X &= \begin{bmatrix} \chi_{z^1}(x^1) & \chi_{z^2}(x^1) & \dots & \chi_{z^{2^n}}(x^1) \\ \chi_{z^1}(x^2) & \chi_{z^2}(x^2) & \dots & \chi_{z^{2^n}}(x^2) \\ \dots & \dots & \dots & \dots \\ \chi_{z^1}(x^N) & \chi_{z^2}(x^N) & \dots & \chi_{z^{2^n}}(x^N) \end{bmatrix} \\ \theta &= [\theta^1,\ \theta^2,\ \dots,\ \theta^{2^n}]^T, \end{aligned} \tag{8}$$

and λ is a user-defined parameter.

The accuracy of the performance prediction depends mostly on the choice of λ, which is not known as a priori. A too small value of λ will make the effect of the L_1 norm regularization to be minimal, hence not effective. For example, when λ is equal to 0, the solution of the problem (7) is simply the least squares solutions. Vice versa, a too large value of λ will shrink the estimated parameters θ to be 0. To automatically find a suitable λ that gives a high prediction accuracy, we use the cross validation technique, i.e.,

1) Split the training dataset into two parts: estimation and validation (67%-33%).
2) Use estimation dataset to estimate $\hat{\theta}_\lambda$ using Eq. (7) with different values of λ.
3) For each estimate $\hat{\theta}_\lambda$ obtained in Step 2, compute the sum of squared error between the measured output and the predicted model output on the validation dataset. Finally, choose the value of λ that minimizes the errors on the validation dataset.

The range to search for λ in Step 2 is also important, as we do not want to spend a lot of time to search for λ from 0 to $+\infty$. Here, based on our empirical experiment, we suggest to search for λ within the range from 0 to $0.1\|X^TY\|_\infty$ (in the literature, the value $\lambda_{\max} = \|X^TY\|_\infty$ is called "critical value", which makes the solution of the Lasso problem to be 0). In addition, to solve (7) faster with the desired level of accuracy, we scale X and Y to a factor of $2^n/\max(Y)$.

B. A Dimension Reduction Lasso Fourier Learning Algorithm

1) Reducing the number of coefficients: Solving (7) can give a good Fourier coefficient estimate of software performance, however, the number of coefficients in the model is normally very large (2^n coefficients). In practice, a small model is more helpful for users to understand the influence of individual configuration options and their interactions on software performance. As mentioned in Section I and Section III-A, for software performance function, many Fourier coefficients are zeros or very close to zeros. Hence, we can remove these coefficients since they do not contribute to the prediction accuracy of the model. Fig. 1 shows an example of how the prediction errors look like when we remove 0 to 1024 coefficients out of the model. It can be seen that in this case, we can remove around 1000 coefficients as they do not help to reduce the prediction error.

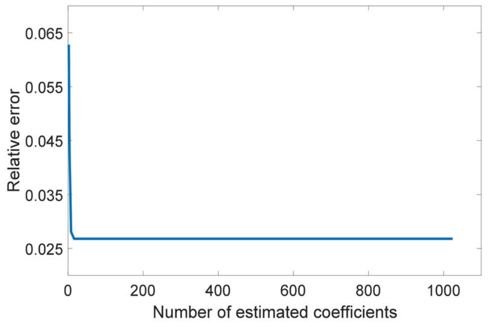

Fig. 1. Elbow graph to remove non-contributing Fourier coefficients (LLVM).

To remove all the non-contributing Fourier coefficients (i.e. the coefficients that do not improve the prediction accuracy when they are included in the model), we use the elbow method [24] to select the right number of Fourier coefficients to be reduced. We first solve the optimization problem in (7) to find 2^n estimated Fourier coefficients. We then remove any coefficients whose presence in the model does not give much better modelling of the data. To be more specific, a coefficient is removed if the difference in accuracy between the model without such a coefficient and the model with all coefficients is less than 1%. Having selected the list of contributing Fourier coefficients, we re-construct the linear regression model, and apply Lasso one more time to find the final estimate.

2) Dimension Reduction for Large Configurable Systems: For small or medium configurable systems (i.e. systems with less than 15 configuration options), any optimization solver is capable of solving (7) within a reasonable amount of time. However, when the number of configuration options is large (i.e. when the number of configuration options n is more than 15), the number of parameters need to be estimated increases exponentially. Solving (7) directly will then require enormous amount of computation time or memory. Hence, it is necessary to design an algorithm that can solve (7) when the number of configuration options n is large. Our core idea is to split the 2^n configurations into 2^{n-p} groups, each group has 2^p configurations, $p \leq n$. We then construct a linear regression problem in each group and solve it with Lasso to find out which configurations in each group contribute the most to the performance value. Specifically, for group i, we solve the following optimization problem:

$$\min_{\theta} \; \frac{1}{2}\|Y - X_i\theta_i\|_2^2 + \lambda_i\|\theta_i\|_1, \qquad \lambda \in R^+, \tag{9}$$

where

$$\begin{aligned} Y &= [f(x^1),\ f(x^2),\ \ldots,\ f(x^N)]^T, \\ X_i &= \begin{bmatrix} \chi_{z^{i,1}}(x^1) & \chi_{z^{i,2}}(x^1) & \ldots & \chi_{z^{i,2^p}}(x^1) \\ \chi_{z^{i,1}}(x^2) & \chi_{z^{i,2}}(x^2) & \ldots & \chi_{z^{i,2^p}}(x^2) \\ \ldots & \ldots & \ldots & \ldots \\ \chi_{z^{i,1}}(x^N) & \chi_{z^{i,2}}(x^N) & \ldots & \chi_{z^{i,2^p}}(x^N) \end{bmatrix}, \\ \theta_i &= [\theta^1,\ \theta^2,\ \ldots,\ \theta^{2^p}]^T, \end{aligned} \tag{10}$$

and $z^{i,h}$ $(h = 1, ..., 2^p)$ is the h^{th} configuration in group i. Note that the output vector of the optimization problem in (9) is the same for all groups and equal to the output vector of the linear regression problem in (6). This is to ensure that the configurations we pick in each group is truly due to their Fourier coefficients contributing the most to the performance value. Then by solving (9) for 2^{n-p} groups, and use the elbow method described in Section III.B.1, we can find the contributing Fourier coefficients in each group. Combining these 2^{n-p} lists gives the list of all contributing Fourier coefficients. Finally, using this list, we can reconstruct the linear regression and apply Lasso to produce the final estimate. With this splitting strategy, at one time, we only need to estimate 2^p $(p \leq n)$ number of Fourier coefficients. To ensure that our algorithm can run on an ordinary desktop computer, we suggest p to be between 0 and a maximum value MAX_P (MAX_P $\leq n$). In our experiments, based on our computer memory capacity, we set MAX_P to 14 and set p to be equal to MAX_P.

In addition, when the number of configuration options n increases, the number of the Lasso regressions (9) needs to be solved also increases. For example, with a software system having 39 configuration options, and p is chosen as 14, then we need to solve 2^{39-14} Lasso regressions. In this case, the computation time becomes very expensive. As mentioned in Section I and Section III.A, for software performance function, many Fourier coefficients are zeros, thus there are many groups whose Fourier coefficients are also zeros. For these groups, we do not need to solve the Lasso regressions (9) to find the estimated Fourier coefficients. Therefore, to reduce the computation time of the algorithm, we suggest to solve (9) for only a few groups that contain the most contributing configurations.

These most contributing configurations can be found by analyzing the property of the Fourier transformation of the performance function. Specifically, using Fourier learning, the performance function of a software system becomes [28],

$$\begin{aligned} f(x) &= \sum_{z\in\{0,1\}^n} \hat{f}(z)\chi_z(x), \\ &= \sum_{z\in C_0} \hat{f}(z)\chi_z(x) + \sum_{z\in C_1} \hat{f}(z)\chi_z(x) + \ldots \end{aligned} \tag{11}$$

where $\chi_z(x)$ is computed using (3), C_i is the group of configurations whose i options are set to 1 and $(n-i)$ options are set to 0. Since $\chi_z(x) = -2 \times \text{mod}(\sum_{j=1}^{n} z_j x_j, 2) + 1$, with $\text{mod}(y, 2) = y$ mod 2 and z_j, x_j being the j^{th} options of the configurations z and x respectively. Hence (11) can be rewritten as,

$$\begin{aligned} f(x) &= \sum_{z\in C_0} \hat{f}(z)(-2\times \text{mod}(\sum_{j=1}^{n} z_j x_j, 2)+1) \\ &\quad + \sum_{z\in C_1} \hat{f}(z)(-2\times \text{mod}(\sum_{j=1}^{n} z_j x_j, 2)+1) + \dots \\ &= \sum_{z\in\{0,1\}^n} \hat{f}(z) - 2\sum_{i=1}^{n} \hat{f}(z^{[i]})\text{mod}(x_i, 2) \\ &\quad - 2\sum_{i,j=1}^{i,j=n} \hat{f}(z^{[i,j]})\text{mod}(x_i + x_j, 2) - \dots, \end{aligned} \tag{12}$$

where $z^{[i]}$ is the configuration with the i^{th} option being 1 (all other options are 0); $z^{[i,j]}$ is the configuration with the i^{th} and j^{th} options being 1 (other options are 0). From (12), we can see that $\hat{f}(z^{[i]})$ represents the influence of the option i^{th} on the performance while $\hat{f}(z^{[i,j]})$ represents the interactions between the options i^{th} and j^{th}. Based on our empirical experiment on some subject systems, $\hat{f}(z^{[i]})_{i=1}^{n}$ usually have the largest absolutes values, followed by $\hat{f}(z^{[i,j]})_{i,j=1}^{n}$ and other subsequent $\hat{f}(z)$. This means the most contributing configurations belong to the first few C_i groups (where C_is are defined in (11)). Using this observation, we only need to solve the Lasso regression (9) for the configurations belonging to the first few C_i groups, with i ranging from 0 to MAX_GROUP (MAX_GROUP $\leq n$). Note that the larger the value of MAX_GROUP, the higher the prediction accuracy of the model. In our experiments, to achieve a reasonable computation time with good prediction accuracy, we empirically set MAX_GROUP to 6 when n ranges from 15 to 20 and MAX_GROUP to 4 when $n > 20$. When $n < 15$, we deem the system a small or medium configurable system so no dimension reduction is performed. The full description of our algorithm is presented in **Algorithm 1**.

C. Identifying performance-influencing configuration options and their interactions

To identify the influencing configuration options and their interactions, Equation (12) can be utilized. Specifically, it can be proved[1] that, when $\{x_{m_i}\}_{i=1}^{n} \in \{0,1\}$, we have:

$$\begin{aligned} &\text{mod}(x_{m_1} + x_{m_2} + \dots + x_{m_n}, 2) \\ &= \sum_{i=m_1}^{i=m_n} x_i - 2^1 \sum_{\substack{i,j=m_1 \\ i\neq j}}^{i,j=m_n} x_i x_j + 2^2 \sum_{\substack{i,j,k=m_1 \\ i\neq j\neq k}}^{i,j,k=m_n} x_i x_j x_k \\ &\quad + \dots + (-1)^{m_n-2} 2^{m_n-2} \sum_{\substack{i_1,\dots,i_{m_n-1}=m_1 \\ i_1\neq\dots\neq i_{m_n-1}}}^{i_1,\dots,i_{m_n-1}=m_n} x_{i_1}\dots x_{i_{m_n-1}} \\ &\quad + (-1)^{m_n-1} 2^{m_n-1} x_{m_1} x_{m_2} \dots x_{m_n}. \end{aligned} \tag{13}$$

[1]The proof is available at https://bit.ly/2IpBZ34

ALGORITHM 1: DIMENSION REDUCTION LASSO FOURIER LEARNING ALGORITHM (PERLASSO)

```
1) From the space of all configurations, select the configurations that belong to the groups $\{C_m\}_{m=0}^{\text{MAX\_GROUP}}$. Split all these configurations into different groups, where each group has $2^p$ configurations, using the methodology described in Section III.B.2. The default setting for $p$ is $\min(n,\text{MAX\_P})$. Denote the number of groups (that has $2^p$ configurations in each group) by NUM_GROUP.
2) for i=1:NUM_GROUP
    a) Solve the following Lasso linear regression problem to find the Fourier coefficients estimates $\theta_i$ of group $i$ (the cross validation method described in Section III.A is used to find the regularization parameter $\lambda_i$),
       $\min_{\theta} \frac{1}{2}\|Y - X_i\theta_i\|_2^2 + \lambda_i\|\theta_i\|_1, \quad \lambda \in R^+,$
       where $Y, X_i, \theta_i$ satisfy (9).
    b) Sort the estimated Fourier coefficients $\theta_i$ by its absolute values. Use the elbow method described in Section III-B to find the configurations with contributing Fourier coefficients.
3) Generate a new regression matrix $X_r$, where
   $\begin{bmatrix} \chi_{z'^1}(x^1) & \chi_{z'^2}(x^1) & ... & \chi_{z'^m}(x^1) \\ \chi_{z'^1}(x^2) & \chi_{z'^2}(x^2) & ... & \chi_{z'^m}(x^2) \\ ... & ... & ... & ... \\ \chi_{z'^1}(x^N) & \chi_{z'^2}(x^N) & ... & \chi_{z'^m}(x^N) \end{bmatrix},$
   where $z'^i$ is a configuration found in Step 2.
4) Solve the following optimization problem using the steps listed in Step 2,
   $\min_{\theta} \frac{1}{2}\|Y - X_r\theta_r\|_2^2 + \lambda\|\theta_r\|_1, \quad \lambda \in R^+.$
5) Remove all the coefficients whose absolute values are smaller than 1% of the maximum coefficient.
```

Combining Equations (12) and (13), we have a simple regression formula of the learned model as follows,

$$f(x) = a_0 + \sum_{i=1}^{i=n} a_i x_i + \sum_{\substack{i,j=1 \\ i\neq j}}^{i,j=n} a_{i,j} x_i x_j + \dots + a_{1,\dots,n} \prod_{i=1}^{i=n} x_i,$$

where,

$$a_0 = \sum_{z\in\{0,1\}^n} \hat{f}(z),$$

$$a_i = -2\Big(\hat{f}(z^{[i]}) + \sum_{\substack{j=1 \\ j\neq i}}^{n} \hat{f}(z^{[i,j]}) + \dots + \sum_{\substack{j_k=1 \\ j_k\neq i}}^{n} \hat{f}(z^{[i,j_1,\dots,j_{n-1}]})\Big),$$

$$\dots\dots\dots$$

$$a_{1,\dots,n} = -2\hat{f}(z^{[1,2,\dots,n]}),$$

$$x = [x_1, x_2, \dots, x_n],$$

with $z^{[i]}$ being the configuration with the i^{th} option being 1 (all other options are 0), $z^{[i,j]}$ being the configuration with the i^{th} and j^{th} options being 1 (other options are 0), etc. Using this model, we can see that the influence of a configuration option x_i on the performance function can be represented by the coefficients a_i, e.g. the larger the absolute value of a_i, the more x_i influences the performance. The interaction between configuration options x_i and x_j can be represented by the coefficients $a_{i,j}$, e.g. the larger the absolute value of $a_{i,j}$, the more the interaction influences the performance.

D. Predicting performance for a new configuration

After getting the estimated Fourier coefficients $\hat{\theta}_r$ using Algorithm *PerLasso*, the predicted performance value of a new configuration x_{new} can be computed by,

$$f(x^{\text{new}}) = [\chi_{z'^1}(x^{\text{new}}), \chi_{z'^2}(x^{\text{new}}) \dots \chi_{z'^{2^{n-p_{m_{\text{rank}}}}}}(x^{\text{new}})]\hat{\theta}_r.$$

E. Tool Implementation

We implement our algorithms using Matlab R2017a. We use the ADMM Lasso code provided in [2] to solve the local Lasso problems. For the Dimension Reduction Lasso Fourier Learning algorithm (*PerLasso*) described in this section, we implement it by ourselves. The source code is publicly available at: ***https://bit.ly/2IpBZ34***.

IV. Evaluation

A. Experimental Design

In this section, we aim to answer the following research questions:

RQ1: How accurate is the proposed approach in identifying performance-influencing configuration options?

RQ2: How accurate is the proposed approach in predicting system performance?

The detailed setup for each experiment will be described in the subsequent sections. In general, we use the training dataset (sample) to generate a performance-influence model for each method, and then use this model to understand the influence/interaction of configuration options and predict performance values of configurations in the testing dataset.

B. RQ1: How accurate is the proposed approach in identifying performance-influencing configuration options?

In this section, we aim to evaluate if the performance-influence model obtained by *PerLasso* is accurate, i.e. if the learned model is similar to the ground truth.

1) Setup: We use 4 synthetic datasets to evaluate the ability of our model in identifying performance-influencing configuration options. Here we use a similar experiment setup in [20]. We use the ground-truth performance models described in [20] and provided by the SPLConqueror project[2]. The ground-truth performance models are realistic formulas and can represent performance variations in real software systems. To generate a synthetic dataset, we apply the ground-truth model on the whole configuration space and generate the measurements of all the configurations. The descriptions of these 4 synthetic datasets are shown in Table II. For each synthetic dataset, we *randomly* select a certain number of configurations and their corresponding performance values to construct the training dataset. We use different sizes for the training datasets of each synthetic dataset: $60, 80, 100, 120$. Besides, to evaluate the consistency and stability of our approach, we repeat the random sampling, training and building the performance-influence models process 30 times.

In the learned regression model, each term (except the constant term) indicates a configuration option or an interaction among the configuration options. To evaluate the accuracy of a learned model compared to the ground-truth, we compute: the number of terms both the learned model and the ground truth have (i.e. *correct_terms*), the number of terms the learned model does not have but the ground truth has (i.e. *missing_terms*), and the number of terms the learned model has but the ground truth does not have (i.e. *extra_terms*). We then adopt the widely-used *Precision* and *Recall* metrics to compute the degree of similarity between the learned model obtained by *PerLasso* and the ground truth model. Specifically, we define these two metrics as:

$$\mathit{Precision} = \frac{\mathit{correct\_terms}}{\mathit{correct\_terms} + \mathit{extra\_terms}},$$
$$\mathit{Recall} = \frac{\mathit{correct\_terms}}{\mathit{correct\_terms} + \mathit{missing\_terms}}.$$

With this definition, the *Precision* metric represents the percentage of correct terms in the learned model while the *Recall* metric shows the percentage of correct terms the model can recall from the ground truth.

TABLE II
THE SYNTHETIC GROUND-TRUTH PERFORMANCE-INFLUENCE MODELS

Model	Domain	Language	$\|D\|$	n
Apache-S	Web Server	C	512	9
LLVM-S	Compiler	C++	2048	11
BDBC-S	Database	C	512	9
BDBJ-S	Database	Java	10^6	18

n is the number of configuration options.
$|D|$ is the number of configurations.

2) Results: In Table III, we show the accuracy of the learned models obtained by *PerLasso* for the 4 synthetic datasets with different sample sizes. It can be seen that for all the synthetic datasets, there is strong similarity between the learned models and the ground truth models. For most of the synthetic datasets (Apache-S, BDBJ-S, LLVM-S), *PerLasso* accurately identifies all the influencing options and their interaction. Its precision is also very high, nearly 1 for all sample sizes, which means it does not wrongly identify any extra configuration options or interactions. For the synthetic BDBC-S dataset, larger sample is required to build an accurate model. When the sample size ranges from 100 to 160, *PerLasso* can correctly identify more than 60% of the terms in the ground truth model.

To demonstrate how the performance-influence model obtained by *PerLasso* looks like, we show a concrete model

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

TABLE III
COMPARISON BETWEEN PERLASSO MODELS AND THE ACTUAL MODELS

Subject System	Sample Size	Precision	Recall	MRE
Apache-S	60	0.90	0.84	16.44
	80	1	0.92	9.50
	100	1	0.97	3.71
	120	1	1	0.52
BDBJ-S	60	0.64	0.93	27.66
	80	0.94	1	3.71
	100	0.98	1	1.18
	120	1	1	0.64
LLVM-S	60	0.89	0.96	3.11
	80	1	1	0.00
	100	1	1	0.00
	120	1	1	0.00
BDBC-S	100	0.65	0.48	43.86
	120	0.63	0.51	40.79
	140	0.63	0.54	36.75
	160	0.63	0.58	30.69

Precision: the average precision of 30 experiments. Recall: the average recall of 30 experiments. MRE: the average MRE of 30 experiments.

obtained by *PerLasso* using a random sample of 100 performance values from the synthetic BDBJ-S dataset:

$$\hat{f}(x) = -353.44 + 98234.75 \times x_1 + 237128.64 \times x_3 - 46091.74 \times x_2 - 188926.97 \times x_2 \times x_3.$$

The model says that for the system BDBJ-S, features $x_1(root)$, $x_3(Finest)$, $x_2(S100MiB)$ have major influence on system performance. The selection of the configuration options $root$ and $Finest$ could reduce the system performance, while the selection of $S100MiB$ could increase the performance. The interaction between $Finest$ and $S100MiB$ (denoted as $x_2 \times x_3$) also influences the performance.

Note that the ground truth model for the synthetic BDBJ-S dataset is:

$$f(x) = 98599.59 \times x_1 + 237630.81 \times x_3 - 46072.03 \times x_2 - 189466.12 \times x_2 \times x_3.$$

It can be seen that the performance-influence model obtained by *PerLasso* is very similar to the ground truth. It can identify accurately all the performance-influencing configuration options (*root*, *Finest* and *S100MiB*) and their interactions (between *S100MiB* and *Finest*). The ability to understand the influencing configuration options and their interaction can help maintenance, debugging, and optimization of highly configurable software systems.

C. RQ2: How accurate is the proposed approach in predicting system performance?

In this section, we compare *PerLasso* with the state-of-the-art performance-influence models, namely *SPLConqueror* [20, 21] and *FourierLearning* [28], which are described in Section I.

1) Setup: Here we compare the prediction accuracy of the three performance-influence models on six real-world subjects systems. They are the same as the subject systems used in [21] and five of them are used in [28]. These systems have different characteristics and are from different application domains, e.g. web server, database library, compiler, etc. They are also of different sizes (45 thousands to more than 300 thousand lines of code) and written in different languages (Java, C, and C++). The number of configuration options ranges from 8 to 39 while the number of valid configurations range from 180 to nearly 4 millions. The overview of these six subject systems are described in Table IV. For each subject system, from all the measurements, we randomly select a certain number of configurations and their corresponding performance values to construct the training set. All the remaining measurements are used as the testing set. For each subject system, with each sampling size, we repeat this prediction and evaluation process 30 times. To evaluate the prediction accuracy, we use the mean relative error (MRE), which is computed as,

$$MRE = \frac{1}{|C|} \sum_{c \in V} \frac{|predicted_c - actual_c|}{actual_c} \times 100, \qquad (14)$$

where V is the testing dataset, $predicted_c$ is the predicted performance value of configuration c, $actual_c$ is the actual performance value of configuration c. We choose this metric as it is widely used to measure the accuracy of prediction models [4, 12, 20].

TABLE IV
THE REAL-WORLD SOFTWARE SYSTEMS

System	Domain	Language	LOC	$\|D\|$	n
Apache	Web Server	C	230,277	192	9
x264	Encoder	C	45,743	1152	16
LLVM	Compiler	C++	47,549	1024	11
BDBC	Database	C	219,811	2560	18
BDBJ	Database	Java	42,596	180	26
SQLite	Database	C	312,625	3,932,160	39

LOC is the number of lines of codes
$|D|$ is the number of valid configurations
n is the number of configuration options (features).

2) Results: Fig. 2 shows the prediction accuracy of *PerLasso* and *SPLConqueror* with different sample sizes. As described in [21], *SPLConqueror* is suggested to be used with five different sampling heuristics: Feature-Wise (FW), Pair-Wise (PW), Higher-Order (HO), Hot-spot (HS) and Brute-Force (BF). Among these five heuristics, the PW, HO and HS heuristics achieve the best prediction accuracy using the smallest sample. Hence, we only plot the MREs of *SPLConqueror* models when the training dataset is constructed using these three heuristics. We use directly the results published in their paper [21]. From Fig. 2, we can see that for the subject systems x264, LLVM, BDBJ and SQLite, *PerLasso* outperforms *SPLConqueror* for all sample sizes. For the system Apache, there are sample sizes that *SPLConqueror* are better than *PerLasso* and there is sample size that *SPLConqueror* and

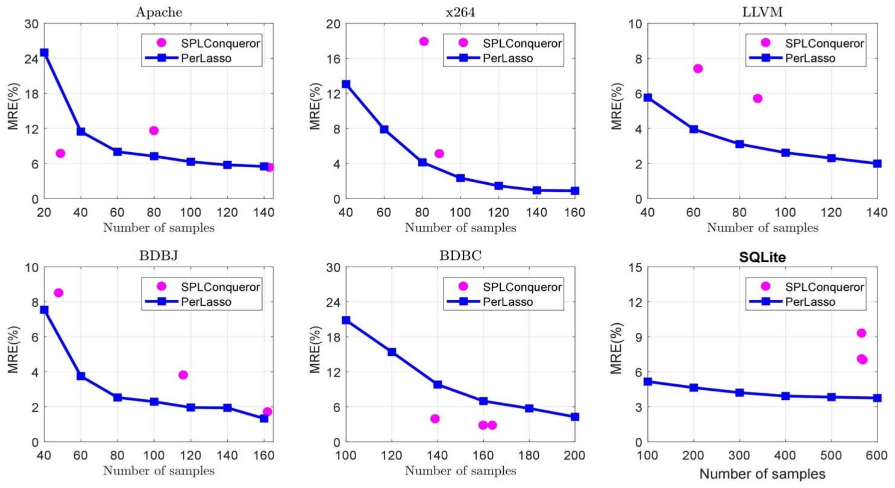

Fig. 2. MRE versus sample size.

PerLasso perform similarly. For the subject system BDBC, *SPLConqeror* is slightly better than *PerLasso*.

Besides, we utilize the sample size suggested in [18] and perform performance prediction using *PerLasso* and *Fourier-Learning*. These sample sizes were chosen using the projective sampling strategy suggested in [18] and the acceptable prediction accuracy is set to 95%. Here we do not include the performance of *SPLConqueror* as their sample sizes are very different (ranging from 29 to 566) thus are difficult to be compared with. This is because *SPLConqueror* uses heuristics to obtain samples while *FourierLearning* and *PerLasso* use random sampling. All the statistics related to the prediction accuracy of the two learning methods are shown in Table V. From this table, it can be seen that *FourierLearning* does not perform well compared to *PerLasso*. As acknowledged in [28], for the studied subject systems, the number of samples required by *FourierLearning* is more than the entire valid domain of the systems for any reasonable accuracy level. It has been observed that although *FourierLearning* can guarantee a theoretical boundary of the prediction accuracy, but this approach still needs thousands to hundreds of thousands of executions of sample configurations [16].

In summary, our experiments show that *PerLasso* can achieve higher or comparable prediction accuracy, when compared to existing performance-influence models.

In our experiments, we use a Windows 10 computer with Intel Core i7-7600U CPU 2.80GHz with 16GB RAM and Matlab R2017a. For all the subject systems described in Table IV and for all the sample sizes investigated in RQ2, *PerLasso* takes a few seconds to 1 minute to build a performance model. The computation time of *SPLConqueror* is similar to that of *PerLasso*. The computation time of *FourierLearning* is similar to that of *PerLasso* for small software systems (systems with a small number of configuration options) and two or three times higher than that of *PerLasso* for large software systems.

TABLE V
COMPARISON BETWEEN PERLASSO AND FOURIERLEARNING USING THE SAMPLE SIZE SUGGESTED IN [18]

System	Approach	Mean(MRE)	STDev(MRE)	N
Apache	PerLasso	8.2	1.7	55
	FourierLearning	100	0.1	55
x264	PerLasso	2.76	1.16	93
	FourierLearning	100	0.1	93
LLVM	PerLasso	4	0.86	62
	FourierLearning	100	0.1	62
BDBJ	PerLasso	6.2	4.55	48
	FourierLearning	100	0.1	48
BDBC	PerLasso	4.88	5.09	191
	FourierLearning	100	0.1	191
SQLite	PerLasso	4.2	0.4	925
	FourierLearning	100	0.1	925

Mean MRE is the mean of the prediction relative errors seen in 30 repeats. STDev is the standard deviation of the MREs from these 30 repeats. N is the number of samples for training data.

Note that both *PerLasso* and *FourierLearning* need to estimate 2^n parameters, however, *PerLasso* requires less time than *FourierLearning* because of the dimension reduction strategy proposed in Section III.B.

V. DISCUSSION

A. Why does the proposed algorithm work?

As discussed in Section III.A, by writing the Fourier coefficients estimation problem as a linear regression problem (6), the task of predicting performance value becomes the task of finding the true Fourier coefficients parameter θ_0 of the performance function. The more accurate the Fourier coefficients are estimated, the better the performance prediction is.

To increase the prediction accuracy when the sample size is small, we can incorporate our prior knowledge into the prediction problem. That is, among those potential candidates, we only pick the candidates that satisfy our belief about the

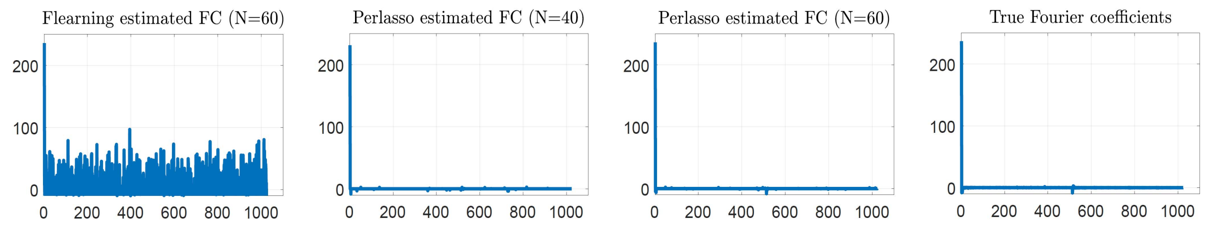

Fig. 3. Estimated (True) Fourier coefficients (FC) of the system LLVM. FLearning stands for Fourier Learning. The X-axis represents the number of FCs and the Y-axis represents the value of the FCs.

true parameter θ_0. By doing this, we can reduce the number of potential candidates $\hat{\theta}$ and hence, there is higher chance that we can find the true parameter or we can approximate it close enough. In other words, if we use our domain knowledge and apply it to the learning process, even when the sample size is small, we can still produce a set of possible candidates that is similar to the set of possible candidates when we have much larger sample size but no knowledge about the true parameter. In our proposed algorithm, by applying LASSO into (6), we pick parameters that a) fit the best to the regression model (6) and b) must be sparse as it is our belief that only a small number of configurations significantly affect system performance. Each value of regularized parameter λ in (7) will result in a possible candidate. By choosing the right value of λ, we can find the right Fourier coefficient parameter θ_0 of the performance value.

Due to the space limit, we only present the estimated Fourier coefficients of the LLVM system here. Among the six benchmark systems, we choose LLVM because it is the only system that has the performance value of the whole configurations. Hence, we can compute the true Fourier coefficients of LLVM using (4). Fig. (3) shows the true Fourier coefficients of LLVM, its estimates computed using the formula in (5), and its estimates using our proposed algorithm. We can see that the true Fourier coefficients of the system LLVM is very sparse, however, if we use the formula in (5) to approximate the coefficients, the estimates are very noisy. When using *PerLasso*, even with the sample size of 40, the estimated Fourier coefficients replicates the true Fourier coefficients quite well. When we have more training data, the cross validation process and the training process are more accurate, hence, the Fourier coefficients estimates are closer to the true Fourier coefficients.

B. Strengths and limitations of PerLasso

A strength of *PerLasso* is that from the training sample, it can build a performance-influence model, i.e. model that explicitly represents the influence of configuration options and their interaction, with higher prediction accuracy compared to other approaches. The second strength of *PerLasso* is that it uses random sampling so it is more flexible when construct the training dataset compared to *SPLConqueror*. *FourierLearning* also uses random sampling, however, the prediction accuracy of the model obtained by *FourierLearning* is very low compared to *PerLasso*. Lastly, *PerLasso* is a progressive algorithm, hence, users can always achieve a much higher model accuracy when having more training data.

A limitation of *PerLasso* is that currently it only works with binary configurable software systems, i.e. software systems with binary configuration options. Similarly, *FourierLearning* can only work with binary configurable software systems. *SPLConqueror* can work with binary-numeric software systems, i.e. software systems with both binary and numeric configuration options. In the future, we will improve our approach so that it can work with binary-numeric configurable systems. Another limitation of *PerLasso* is that it does not provide theoretical boundary of the prediction accuracy given a sample size. *FourierLearning* can provide this theoretical boundary while *SPLConqueror* does not guarantee any theoretical bound on the model prediction accuracy either.

C. Threats to Validity

To evaluate the accuracy of the performance-influence models obtained by the proposed method, we used four synthetic datasets whose performance models are known. These ground-truth models are obtained in prior work with linear programming [21, 22], which is a completely different technique compared to our proposed method.

Furthermore, to increase the internal validity our experiment results, for each subject system, we evaluate the performance of our proposed approach and other state-of-the-art approaches with various different sample sizes. For each sample size, we randomly repeat the sampling, model construction and evaluation process 30 times. For each process, the model is evaluated on a test dataset which does not include any part of the training dataset. To compute the similarity between our models and the ground truth models, we use the *Precision* and *Recall* metrics. These metrics have been utilized intensively in the machine learning area. To evaluate the prediction accuracy, we use the mean relative error (MRE) as it is a widely-used metric in the literature for evaluating the effectiveness of performance prediction algorithm and it is also used to evaluate other approaches we compared. We are aware that the relative error sometimes receives criticism, however, as the focus of our paper is to suggest an algorithm that produces performance-influence model with high prediction accuracy, so it makes more sense to use the relative error instead of other metrics.

For external validity, we evaluate the algorithms using six public real-world datasets with different characteristics, domains, languages etc. These subject systems have a large range of configuration options and have been used extensively in the literature to evaluate the effectiveness of performance prediction algorithms.

VI. Related Work

In recent years, a large body of work has been conducted to model the performance function of large-scale configurable software systems. The core idea is to measure performance of a limited set of configurations, build a performance model, and use the model to predict the performance of the system under any new configurations or to understand the influence of the configuration options and their interactions.

There are various performance prediction models proposed in the literature. *CART* [5] is a regression-tree based technique that partitions features into subsets, and the performance value of each subset is computed as the average performance values of sampled configurations within the subset. Recently, *CART* was further improved by utilizing various resampling and automatic hyperparameter tuning techniques, and became a learning method with higher prediction accuracy, namely *DECART* [7]. *CART/DECART* have been utilized in many frameworks for software performance prediction [16, 18]. Recently, Ha and Zhang proposed *DeepPerf* [8], which models highly configurable software system using a deep feedforward neural network combined with the L_1 regularization. They also designed a practical search strategy for automatically tuning the network hyperparameters. However, a limitation of these approaches is that their learned models are not performance-influence models. To be more specific, we cannot explain the influence of configuration options and their interactions from the *CART/DECART/DeepPerf* models.

Some performance-influence models have been proposed to describe the individual influences of configuration options and their interactions on system performance. *SPL Conqueror* [21] is a measurement-based prediction approach. The idea is to produce a set of configurations that differ in a single feature, compute the delta difference in the performance, and use these deltas to generate approximations of features and feature interactions. Later, in their improved version [20], Siegmund et al. used stepwise linear regression to learn the function of a performance-influence model from a sample set of measured configurations. To reduce the dimensionality problem, they used forward and backward feature selection to incrementally learn the model. *Fourier Learning* is an approach that was first suggested in [28]. The key contribution of this technique is that it works with random sampling and provides a theoretical boundary of the prediction accuracy but it requires a very large training data in order to predict with a reasonable accuracy. As shown in Section IV, our proposed approach can yield a performance-influence model with much higher prediction accuracy compared to these approaches.

There are also some work on selecting an optimal set of configurations (or features) for configurable software systems. For example, Zhang et al. [26, 27] proposed a Bayesian Belief Network (BBN) based approach, which allows users to configure features to best satisfy quality requirements through qualitative analysis. Guo et al. [6] proposed a genetic algorithm based technique for optimizing feature selection in the face of resource constraints. Sayyad et al. [19] utilized evolutionary algorithms to select optimal features regarding multiple objectives. Nair et al. [16] proposed to select a small number of configurations based on a distance matrix between the configurations. In [15], a rank-based approach is proposed to reduce the measurement cost as well as the time required to build performance models. Oh et al. [17] randomly sample and recursively search a configuration space directly to find near-optimal configurations without constructing a prediction model. Jamshidi et al. [10, 11] proposed a transfer learning based approach to performance prediction: instead of taking the measurements from the target system, they learn the model using samples from other sources. Han et al. [9] manually analyzed bug reports and found that exposing performance bugs often requires combinations of multiple input parameters and certain input parameters are frequently involved in exposing the bugs. There are also much work on checking inter-dependencies among configurations [1, 23]. Our work and these related methods can be combined to further improve the effectiveness of performance modeling and prediction.

VII. Conclusion

In this paper, we have proposed *PerLasso*, which can model the influence of configuration options and their interactions on the performance of highly configurable software systems, using a small sample of measured performance data. *PerLasso* is based on Fourier Learning and Lasso regression, and incorporates a novel dimensional reduction algorithm. The models constructed with *PerLasso* can help understand the performance-influencing configuration options and can be used to accurately predict the system performance with a new configuration. Our experimental results confirm the effectiveness of the proposed approach. Compared with existing performance-influence models, our model has higher or comparable prediction accuracy.

Our tool and experimental data are publicly available at: ***https://bit.ly/2IpBZ34***.

VIII. Acknowledgment

We thank the anonymous reviewers for the constructive comments. This work is supported by NSFC grant 61828201.

References

[1] D. Batory, D. Benavides, and A. Ruiz-Cortes. Automated analysis of feature models: Challenges ahead. *Commun. ACM*, 49(12):45–47, December 2006.

[2] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends in Machine Learning*, 3(1):1–122, 2011.

[3] M. Fazel, H. Hindi, and S. Boyd. A rank minimization heuristic with application to minimum order system

approximation. In *Proceedings 2001 American Control Conference*, volume 6, pages 4734–4739, 2001.

[4] T. Foss, E. Stensrud, B. Kitchenham, and I. Myrtveit. A simulation study of the model evaluation criterion mmre. *IEEE Transactions on Software Engineering*, 29(11):985–995, 2003.

[5] J. Guo, K. Czarnecki, S. Apel, N. Siegmund, and A. Wsowski. Variability-aware performance prediction: A statistical learning approach. In *International Conference on Automated Software Engineering (ASE)*, pages 301–311, 2013.

[6] 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*, 84(12):2208 – 2221, 2011.

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

[8] H. Ha and H. Zhang. Deepperf: Performance prediction for configurable software with deep sparse neural network. In *International Conference on Software Engineering (ICSE)*, pages 1095–1106, 2019.

[9] X. Han, T. Yu, and D. Lo. Perflearner: Learning from bug reports to understand and generate performance test frames. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, pages 17–28, 2018.

[10] P. Jamshidi, N. Siegmund, M. Velez, C. Kästner, A. Patel, and Y. Agarwal. Transfer learning for performance modeling of configurable systems: An exploratory analysis. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 497–508, 2017.

[11] P. Jamshidi, N. Siegmund, M. Velez, C. Kästner, A. Patel, and Y. Agarwal. Transfer learning for performance modeling of configurable systems: An exploratory analysis. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 497–508, 2017.

[12] M. Jorgensen and M. Shepperd. A systematic review of software development cost estimation studies. *IEEE Transactions on Software Engineering*, 33(1):33–53, 2007.

[13] D.R. Kuhn, R.N. Kacker, and Y. Lei. *Introduction to combinatorial testing*. CRC Press, 2013.

[14] Y. Mansour. Learning boolean functions via the fourier transform. In A. Orlitsky V. Roychowdhury, K.Y. Siu, editor, *Theoretical Advances in Neural Computation and Learning*, pages 391–424. Springer, 1994.

[15] V. Nair, T. Menzies, N. Siegmund, and S. Apel. Using bad learners to find good configurations. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 257–267.

[16] V. Nair, T. Menzies, N. Siegmund, and S. Apel. Faster discovery of faster system configurations with spectral learning. *Automated Software Engineering*, Aug 2017.

[17] J. Oh, D. Batory, M. Myers, and N. Siegmund. Finding near-optimal configurations in product lines by random sampling. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 61–71, New York, NY, USA, 2017. ACM.

[18] A. Sarkar, J. Guo, N. Siegmund, S. Apel, and K. Czarnecki. Cost-efficient sampling for performance prediction of configurable systems (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 342–352, Nov 2015.

[19] A. S. 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 *2013 35th International Conference on Software Engineering (ICSE)*, pages 492–501, May 2013.

[20] N. Siegmund, A. Grebhahn, S. Apel, and C. Kästner. Performance-influence models for highly configurable systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 284–294, 2015.

[21] N. Siegmund, S. S. Kolesnikov, C. Kstner, S. Apel, D. Batory, M. Rosenmller, and G. Saake. Predicting performance via automated feature-interaction detection. In *International Conference on Software Engineering (ICSE)*, pages 167–177, 2012.

[22] N. Siegmund, M. Rosenmuller, C. Kastner, P. G. Giarrusso, S. Apel, and S. S. Kolesnikov. Scalable prediction of non-functional properties in software product lines. In *International Software Product Line Conference*, pages 160–169, Aug 2011.

[23] J. Sun, H. Zhang, Y. Fang, and L.H. Wang. Formal semantics and verification for feature modeling. In *IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)*, pages 303–312, June 2005.

[24] R.L. Thorndike. Who belongs in the family? *Psychometrika*, 18(4):267–276, 1953.

[25] R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society, Series B*, 58:267–288, 1996.

[26] H. Zhang and S. Jarzabek. A bayesian network approach to rational architectural design. *International Journal of Software Engineering and Knowledge Engineering*, 15(4):695–718, 2005.

[27] H. Zhang, S. Jarzabek, and B. Yang. Quality prediction and assessment for product lines. In *Advanced Information Systems Engineering, 15th International Conference, CAiSE 2003, Klagenfurt, Austria, June 16-18, 2003, Proceedings*, pages 681–695, 2003.

[28] Y. Zhang, J. Guo, E. Blais, and K. Czarnecki. Performance prediction of configurable software systems by fourier learning (t). In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 365–373, Lincoln, NE, 2015.