Noname manuscript No.
(will be inserted by the editor)

Tradeoffs in Modeling Performance of Highly-Configurable Software Systems

Sergiy Kolesnikov · Norbert Siegmund · Christian Kästner · Alexander Grebhahn · Sven Apel

Received: date / Accepted: date

Abstract Modeling the performance of a highly-configurable software system requires capturing the influences of its configuration options and their interactions on the system's performance. *Performance-influence models* quantify these influences, explaining this way the performance behavior of a configurable system as a whole. To be useful in practice, a performance-influence model should have a low prediction error, small model size, and reasonable computation time. Because of the inherent tradeoffs among these properties, optimizing for one property may negatively influence the others. It is unclear, though, to what extent these tradeoffs manifest themselves in practice, that is, whether a large configuration space can be described accurately only with large models and significant resource investment.

By means of 10 real-world highly-configurable systems from different domains, we have systematically studied the tradeoffs between the three properties. Surprisingly, we found that the tradeoffs between prediction error and model size and between prediction error and computation time are rather marginal. That is, we can learn accurate and small models in reasonable time, so that one performance-influence model can fit different use cases, such as program comprehension and performance prediction.

We further investigated the reasons for why the tradeoffs are marginal. We found that interactions among four or more configuration options have only a minor influence on the prediction error and that ignoring them when learning a performance-influence model can save a substantial amount of computation time, while keeping the model small without considerably increasing the prediction error. This is an important insight for new sampling and learning techniques as they can focus on specific regions of the configuration space and find a sweet spot between accuracy and effort.

S. Kolesnikov, University of Passau, Germany · N. Siegmund, Bauhaus-University Weimar, Germany · C. Kästner, Carnegie Mellon University, USA · A. Grebhahn, University of Passau, Germany · S. Apel, University of Passau, Germany

We further analyzed the causes for the configuration options and their interactions having the observed influences on the systems' performance. We were able to identify several patterns across subject systems, such as dominant configuration options and data pipelines, that explain the influences of highly influential configuration options and interactions, and give further insights into the domain of highly-configurable systems.

Keywords performance-influence models · highly-configurable software systems · performance prediction · feature interactions · variability · software product lines · machine learning

1 Introduction

Highly-configurable software systems have become ubiquitous in many application domains, such as server software, system tools and utilities, software libraries, and embedded systems. By means of *configuration options*, users can tailor functional and non-functional properties of a system to their needs. For example, the Linux kernel has over 13 000 compile-time configuration options,[1] and can be configured to accomplish a large set of different tasks on different devices, ranging from embedded systems through desktops to clusters. Despite the apparent benefits of configurability, each of the 13 000 configuration options may influence the kernel's performance in unknown ways and, therefore, hinder the understanding of the system's performance behavior.

One way to describe the performance behavior of a configurable software system as whole (and not just as an individual configuration) is to build a *performance-influence model,* which describes how individual configuration options and their interactions influence performance. In previous work, different machine-learning techniques have been used to learn such models from sample measurements (Siegmund et al, 2012; Guo et al, 2013; Siegmund et al, 2015; Sarkar et al, 2015). To be practically useful, an ideal performance-influence model should exhibit the following properties:

1. Low *prediction error* (i.e., be as accurate as possible), such that it accurately describes the system's behavior and can be used as a predictor,
2. Small *model size*, such that it is understandable by developers for a wide variety of tasks involving human judgment, such as debugging, and
3. Short *computation time*, such that constructing the model is feasible in practice.

It is well known in the machine-learning community that there are tradeoffs among prediction error, model size, and computation time (Domingos, 2000; Sammut and Webb, 2011; James et al, 2013). Partly, they are known under the term *bias-variance tradeoff* and *the curse of dimensionality*, which we discuss in more detail in Section 2.2. Ultimately, we cannot generally have an accurate model that is easy to compute and easy to interpret. Hence, optimizing for one property may negatively influence the others. For example, to get more

[1] `http://kernel.org/`

accurate results, we might need to invest more time into learning, which also tends to lead to larger and more complex models. More importantly, different stakeholders may have different priorities: For predicting the performance of a configuration, one might prefer the most accurate model, whereas, for gaining an initial understanding, a quick and approximate overview that characterizes the main effects (but ignores interactions among configuration options) may be sufficient, favoring short computation time and model simplicity. In discussions with four experts from the high-performance computing domain, who intend to use performance-influence models in their daily work, we have identified two common use cases and corresponding preferences for the performance-influence models (see Section 2.1).

The goal of this work is to explore whether the described tradeoffs are practically relevant for performance-influence models in the domain of highly-configurable software systems and how significant they are. This is important, because based on the knowledge about the relevance of the tradeoffs, novel modeling, learning, and sampling approaches can be developed and existing approaches can be improved. Furthermore, we may gain new insights as to whether a large configuration space inevitably leads to large (and complex) models, which are often infeasible to compute due to the exponential complexity of the problem and which are also hard to comprehend if human judgment is required. To this end, we systematically study the tradeoffs among prediction error, model size, and computation time of performance-influence models learned from performance benchmark measurements. Basically, we want to know whether the models can satisfy the use cases identified in our discussions with domain experts (Section 2.1). Technically, we use a state-of-the-art machine-learning algorithm based on *forward feature selection* and *multivariate linear regression* to automatically learn performance-influence models (Siegmund et al, 2015). A particular advantage of linear regression in our setting is that the resulting performance-influence models clearly state the influences of individual configuration options and *their interactions* on systems' performance (Section 2), which we use to explain the tradeoffs and gain further insights into the domain of highly-configurable systems (Section 3.5). To this end, we have studied the properties of models learned for a set of 10 real-world highly-configurable software systems. Based on the results of this study, we analyze the tradeoffs among the properties of the models and discuss their applicability in common practical use cases.

In a nutshell, our results show that, although, the tradeoffs among the different model properties technically exist, their effect is surprisingly low, so that they have *effectively no negative influence for practical purposes*. For most of the subject systems that we studied, we could quickly learn performance-influence models with small model sizes (covering the main effects and including interactions only among few configuration options) that are usable for understanding and debugging tasks. These models tend to be already fairly accurate ($>$ 80-95 % accuracy), and investing further learning effort would reduce the prediction error only marginally, but would significantly increase model size and computation time. These results are important in two ways:

First, they demonstrate that one learning approach can be used for different real-world application scenarios, which is crucial for practicality. Second, they demonstrate that the domain of configurable software systems exhibit specific properties (e.g., the distribution of interactions) that make circumventing the tradeoff problem possible, allowing researchers and practitioners to develop efficient learning approaches by concentrating on a few important configuration options and their low-order interactions (i.e., interactions involving only a small number of configuration options, in our case, two or three).

The contributions of this work are the following:

- Using a machine learning technique based on multivariate linear regression, we systematically studied and analyzed the tradeoffs among prediction error, model size, and computation time of performance-influence models for 10 real-world highly-configurable systems from different domains.
- We found that the low influences of the aforementioned tradeoffs allow us to build models that fit typical use cases, such as program comprehension and performance prediction. Often, the tradeoffs are so marginal that the same model is suitable for both use cases.
- We found that the reason for the marginal tradeoffs lies in the prevalence of low-order interactions among configuration options (i.e., interactions among two or three configuration options) that have a strong influence on performance, which is the case for all our subject systems.
- We investigated the causes for the configuration options and their interactions having the observed influences on the systems' performance, and we identified reoccurring patterns in the systems' architecture and in the dependencies among configuration options that explain these influences.

All experimental data and analysis scripts are available on a supplementary Web site.[2]

2 Motivation and Research Questions

In the domain of highly-configurable systems, we lack empirical understanding of how strong the tradeoffs among prediction error, model size, and computation time of performance-influence models are. That is, while learning performance-influence models for real-world highly-configurable systems, we do not know for which combinations of these properties we are able to effectively optimize. Our goal is to quantify these tradeoffs by means of a series of experiments and to gain insights in to the characteristics of the configuration spaces of highly-configurable software systems, for example, such as the relevance of different kinds of interactions and their influences on performance.

To illustrate the properties of performance-influence models and the tradeoffs among them, we will use two simple models, as shown in Figure 1a. These models describe the request throughput (requests per second, req/s) of the Apache Web server for a fixed standard benchmark. They are slightly

[2] `http://fosd.net/tradeoffs/`

(a) Two performance-influence models for the Apache Web server.

Model A:

$$1000 - 250 \cdot \underline{A}ccessLog - 150 \cdot \underline{H}ostnameLookups$$

Model B:

$$1000 - 250 \cdot \underline{A}ccessLog - 150 \cdot \underline{H}ostnameLookups + 100 \cdot \underline{A}ccessLog \cdot \underline{H}ostnameLookups + \ldots$$
$$+ 2 \cdot \underline{A}ccessLog \cdot \underline{E}nableSendfile \cdot \underline{K}eepAlive + 1 \cdot \underline{E}nableSendfile \cdot \underline{F}ollowSymLinks \cdot Ha\underline{n}dle$$

(b) Performance values predicted by the models.

#	Configuration	Measured Value	Predicted Value Model A	Predicted Value Model B
1	A	750	750	750
2	H	850	850	850
3	H, ***T***	850	850	850
4	H, ***I***	950	850	850
5	A, H	700	600	700
6	A, E, K	752	750	752
7	A, E, F, n	751	750	751
⋮	⋮	⋮	⋮	⋮

Fig. 1: Two examples of performance-influence models for the Apache Web server and the corresponding predicted performance values. The underlined letters in the option names are used as abbreviations in the table (e.g., A stands for $\underline{A}ccessLog$). The slanted letters in the table denote configuration options that are not covered by either model. The predicted values that match the actually measured values are shaded in green, those that do not match are shaded in red.

simplified versions of the real models that we learned during our evaluation. The variables in the models represent configuration options of the Web server ($\underline{A}ccessLog$, $\underline{H}ostnameLookups$, etc.), which can be either enabled or disabled (values 0 or 1). To predict the request throughput for a given configuration we set the variables to 0 or 1 according to the configuration and evaluate the expression. For example, if *AccessLog* is enabled and *HostnameLookups* is disabled then for Model A we get the expression $1000 - 250 \cdot 1 - 150 \cdot 0$, which evaluates to 750 (req/s).

It is important to note that both models in Figure 1a describe the same system, but have different size and prediction error. The table in Figure 1b lists the actual performance measurements of the Web server next to the predictions for the corresponding configurations using either Model A or Model B. Both models describe the main effects of the configuration options strongly influencing the system: In its default configuration (with all options disabled), the server can process 1000 req/s. However, with option $\underline{A}ccessLog$ enabled, the throughput is decreased by 250 req/s. Enabling option $\underline{H}ostnameLookups$ decreases the throughput by further 150 req/s. Both models accurately describe the performance of the first two configurations with one or the other option

enabled (configurations 1 and 2 in Figure 1b). The third configuration contains the configuration option *TypeConfig*, which is not covered by the two models. Nonetheless, both models predict the configuration's performance accurately, because the configuration option has no measurable influence on performance. In contrast, configuration option *InMemory* in the fourth configuration has a substantial influence on the performance, and its absence in both models leads to prediction errors.

The two models differ in how they characterize minor variations and interactions among configuration options. By the individual influences of 250 req/s and 150 req/s of the options *AccessLog* and *HostnameLookups*, we could expect a combined performance penalty of 400 req/s, when both options are enabled, and in fact the first, simpler model assumes that. In practice though, we observed that both options interact, and their combined penalty is only 300 req/s (see Figure 1b, configuration number 5). This is an example of a positive interaction between configuration options (Apel et al, 2013). By studying the system's documentation, we found that both options partially use the same data retrieved from a request, so the data are retrieved only once, but used by both options, and this reuse results in a higher throughput. While the first model produces an inaccurate prediction by ignoring this interaction, the second model covers this interaction (term *AccessLog*·*HostnameLookups*) and yields accurate predictions for more configurations, but at the cost of a more complex model (5 model terms instead of 2) and increased computation time.

Clearly, interactions can be important for prediction accuracy, but not all interactions may have a substantial influence on performance. The second model includes several interaction terms that only slightly alter the predicted performance by 2 or 1 req/s (e.g., *EnableSendfile*·*KeepAlive*), resulting in a relative accuracy improvement of at most 0.3 % and 0.1 % (compare the predicted values for configurations number 6 and 7). If we are fine with accepting such small prediction errors, we could ignore these interactions and work with smaller and simpler models. Note that such small-influence terms may be an indication for model overfitting, that is, the model describes measurement noise more than actual influences, at the cost of significant computation time and complexity of models.

2.1 Use Cases of Influence Models – A Discussion with HPC Experts

In an attempt to better understand requirements and use cases of performance-influence models regarding prediction error, model size, and computation time, we had several discussions with four of our collaborators from the HPC domain. The discussions constitute a basis for a lightweight explanatory analysis rather than a deep study in itself. Still, the discussions are informative enough to guide our analysis. All four HPC experts develop, analyze, and work with performance-critical applications on an everyday basis in areas, such as image and signal processing, automatic code generation, and differential-equations solvers, having 10 to 20 years of experience in the corresponding

areas. They are working with the following systems: DUNE (Bastian et al, 2006), HSMGP (Kuckuk et al, 2013), HIPA^{CC} (Membarth et al, 2012), and SaC (Grelck and Scholz, 2006) (DUNE and HSMGP being also subject systems in our experiments; see Section 3).

To anchor our discussions with concrete data, we learned performance-influence models for the systems with which the experts were deeply familiar. Specifically, we took models at an early, intermediate, and late stage of the incremental learning process (described in Section 3.1): The early models were smaller, but more inaccurate (like Model A in Figure 1) than those in the later stages of the learning process (like Model B in Figure 1). We presented the models[3] to the experts explaining their general structure and asked the following questions:

1. What are use cases for the presented performance-influence models that you can think of?
2. What are acceptable tradeoffs among prediction error, model size, and computation time of a model with respect to these use cases?

The use cases mentioned by the experts can be grouped in two categories:

1. *Performance prediction.* Performance-influence models can help stakeholders of a system find the system's optimal configuration (for a give setting). For example, the SaC compiler has a default configuration that may have suboptimal performance for some hardware platforms. Learning a performance-influence model for each target platform allows developers to find the optimal configuration for each of them.
2. *Program comprehension and debugging.* Among the important program comprehension tasks, the experts named (1) confirming or disproving existing assumptions about influences of individual configuration options and (2) gaining new insights and deeper understanding of the performance of the system. For example, the HIPA^{CC} expert was surprised to see that *pixelsPerThread* configuration option had only a small influence on system performance.

All experts stated that the decision about which model to chose or which property to optimize would depend on the given use case. For example, a model with the lowest prediction error is needed for performance prediction, whereas the model size and the computation time would be less important. For confirming a theoretical assumption about the influence of a certain configuration option, a simple model without interaction terms that is fast to learn could suffice. All experts were ready to accept the model with the highest prediction error (among the provided models), which was smaller and therefore easier to comprehend. For gaining deeper insights, such as finding and debugging unexpected interactions, all experts said that they required a model with interaction terms, but still of a tractable size. For debugging purposes, computation time becomes important too, and the experts were ready to accept

[3] The models used in the discussions can be found on the supplementary Web site.

a model with a high prediction error if they could save a considerable amount of computation time during debugging.

The use cases identified in our discussions map very well to the model properties that we study in this work. Therefore, we are confident that exploring the tradeoff space spanned by these properties has an immediate practical use.

2.2 Tradeoffs in Machine Learning

The tradeoffs between prediction error, model size, and computation time are well known in the machine-learning community: A key concept is the bias-variance tradeoff (Domingos, 2000; Sammut and Webb, 2011), which refers to the tradeoff between the size and prediction error of a model. Bias refers to the prediction error one encounters for a model with a fixed size and all data that is available. That is, for a small and simple model, the bias error may be high, because the model potentially does not explain the observed data to a full extent. Variance refers to the sensitivity of the model to the noise in the training data (such as measurement errors). More complex and larger models tend to fit the noise in the learning set, so that one may encounter large prediction errors when the model is applied to new data. So, learning a larger model may reduce its prediction error, but, at the same time, may complicate its understandability, simply because of its large size. Therefore, one of the main goals in machine learning is to find the sweet spot between underfitting (i.e., too simplistic models) and overfitting (i.e., too complicated models). However, often the search for this sweet spot is primarily driven by the minimization of the prediction error and does not take the comprehensibility of the resulting model into account.

Researchers in software engineering often apply machine learning without specifically considering the possible effects of the tradeoffs, or they just optimize for one criterion (e.g., prediction error) until other criteria leave the acceptable value ranges. For example, genetic algorithms have been used for multi-objective optimization to find configurations of configurable systems that satisfy multiple quality requirements (Sayyad et al, 2013). However, they trade computation time for prediction error, because most of these configurations are not valid. Other approaches aim solely at reducing the prediction error using classification and regression trees (Guo et al, 2013; Sarkar et al, 2015), but produce models that are hard to comprehend for humans. The goal of the mentioned approaches was never to balance or even explore the tradeoffs, but to optimize only for one property and ignore the others. As pointed out by our experts (Section 2.1), such approaches are only of limited practicality, because a different use case may require a different approach with yet another tool. We aim at filling this gap.

Notably, recent research in the performance-engineering community recognized the importance of the tradeoffs. In their recent work, Brosig et al (2015) explore alternative stochastic performance-modeling approaches regarding several low-level properties, such as the capability of handling loops in the ana-

lyzed software system. While we concentrate on regression models and more general properties, their work clearly connects to ours in showing that different stochastic models are suitable for different use cases and that it is important to have this information before performance analysis.

2.3 Research Questions

Our overarching goal is to explore the tradeoffs during the learning process of performance-influence models and gain insights into the performance behavior of highly-configurable systems. For the purpose of our study, we use a state-of-the-art learning technique that is based on multivariate linear regression learning and forward feature selection (Chandrashekar and Sahin, 2014). We specifically aim at answering two research questions:

- **RQ1:** How significant are the tradeoffs among prediction error, model size, and computation time of the performance-influence models of real-world highly-configurable systems?
- **RQ2:** Can these tradeoffs be balanced, such that the resulting models can be applied in different use cases, as identified by our discussion with experts?

3 Empirical Study

To answer our two research questions, we conducted an empirical study in which we created and compared different performance-influence models for 10 real-world highly-configurable software systems. Next, we describe how we learned performance-influence models for our subject systems and how we analyzed their properties.

3.1 Learning Performance-Influence Models

For our experiments, we use a learning algorithm based on multivariate linear regression and forward feature selection, which we developed and evaluated in our previous work. It has proved to be accurate and effective for learning performance-influence models of real-world highly-configurable systems (Siegmund et al, 2015). During the learning process, we learn increasingly accurate models and keep track of the prediction error, model size, and computation time of each intermediate model, so that we can study how the properties evolve and how significant the tradeoffs among them are. Note that it is not our primary goal and contribution to invent a new technique for learning performance-influence models, but to use an established technique to study and leverage the tradeoffs among the three properties.

In a nutshell, the algorithm begins with calculating a set of candidates that can be included in the model to reduce its prediction error. A *candidate*

is either a single or a combination of configuration options.[4] For example, *AccessLog* (a single configuration option) and $AccessLog \cdot HostnameLookups$ (an interaction between two configuration options) in Figure 1a are two candidates that have been eventually added to the model as new terms. The algorithm iterates over the set of candidates and selects the one that explains the performance variation in the measurements best; that is, the candidate that, when incorporated into the model, yields the model's lowest prediction error. The accuracy improvement for each candidate (i.e., the *score* of a candidate) is calculated by refitting a linear model over all model terms, including the new candidate, and then comparing the model's prediction error with the model's prediction error without the candidate. The prediction error of a model is calculated by comparing the predicted performance values for the system configurations with the values that we actually measured on a sample set of configurations. The selection of candidates continues until either the accuracy, specified by a parameter, is reached or all candidates that could reduce the prediction error of the model have been considered. For a detailed description of the algorithm, we refer the reader to Siegmund et al (2015).

3.2 Measurement Procedure

To answer our research questions, we need to quantify the prediction error, size, and computation time of performance-influence models and tradeoffs among them. For this purpose, we define a number of measures.

3.2.1 Measuring Model Properties

The *prediction error* of a performance-influence model Π is the mean relative prediction error over the set of system configurations C:

$$error(\Pi, C) = \frac{1}{|C|} \sum_{c \in C} \left| \frac{\Pi(c) - measure(c)}{measure(c)} \right|,$$

where $c \in C$ is a system configuration, $measure(c)$ is the performance of the configuration actually measured, and $\Pi(c)$ is the performance of the configuration predicted by the model Π. For example, for Model B of Figure 1a and the set of configurations number 1 through 7 in Figure 1b, the prediction error is 0.03 (or 3 %), mainly because the model wrongly predicts the performance value for the configuration number 4.

We define the *model size* as the number of configuration options in every term of the model. The model size of a performance-influence model Π and its set $terms(\Pi)$ of terms is defined as follows:

$$modelSize(\Pi) = \sum_{t \in terms(\Pi)} size(t),$$

[4] Only valid combinations (i.e., those that respect dependencies among configuration options) are considered.

where $t \in terms(\Pi)$ is a term of the model Π and $size(t)$ is the number of configuration options in t. For example, Model B in Figure 1a has a size of 2, because it contains two terms and each term consists of only one configuration option.

The *computation time* of a model is equal to the CPU time used by the algorithm to learn the model.

3.2.2 Measuring Tradeoffs

To characterize the tradeoffs between the three properties quantitatively, we use the *Area Under the Curve* (AUC) measure. To calculate the AUC for a tradeoff between two properties, we plot one property against another and calculate the integral of the resulting curve. The integral value is the corresponding AUC. We normalize the property values in the range $[0, 1]$ before calculation, therefore, the corresponding AUC is a value in the same range.

Figure 2 illustrates three example tradeoff curves and the corresponding AUC values for different kinds of tradeoffs between computation time and prediction error properties.[5] If the two properties are in inverse relationship (Figure 2a), then a relatively large (small) positive change in one property always results in a relatively large (small) negative change in the other property. That is, the tradeoff between these two properties is balanced. The AUC value for a balanced tradeoff like this is close to 0.5.

AUC values that are smaller that 0.5 indicate a shift to a marginal tradeoff (Figure 2b), which is favorable in our setting: A small initial increase in computation time already leads to a large initial decrease of prediction error. Conversely, a large initial decrease in prediction error requires only a small increase in computation time. A marginal tradeoff would allow us to learn smaller and more accurate models faster.

AUC values that are larger than 0.5 indicate a shift to a significant tradeoff (Figure 2c), which is unfavorable in our setting: A large initial increase in computation time would lead to a small initial decrease in prediction error.

[5] The tradeoffs for other property pairs are calculated in the same way.

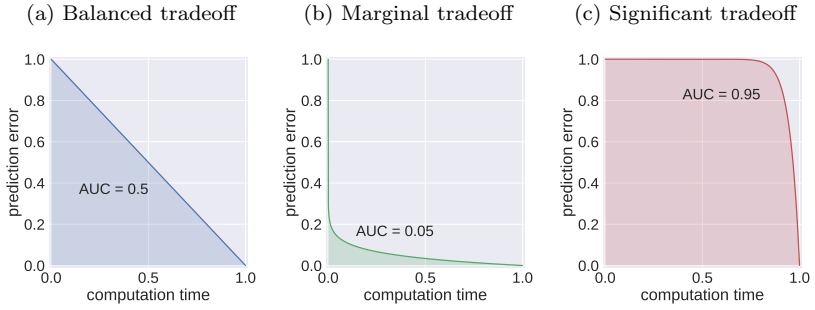

Fig. 2: Example tradeoff curves and corresponding AUC values.

A significant tradeoff means that we would have to invest much computation time or accept large model sizes if we want to learn a model with low prediction error.

By calculating the AUC values for each subject system and each pair of model properties, we can determine which kinds of tradeoffs—balanced, marginal, or significant—are present in the subject systems.

3.3 Subject Systems and Experimental Setup

As subject systems, we selected 10 real-world highly-configurable software systems of different sizes, complexities, and from different application domains, as summarized in Table 1. The systems differ in the number of configuration options as well as in the number of resulting configurations. They are implemented in different programming languages and support configuration at compile time, load time, or both. We used the systems' documentation to determine which configuration options may have influence on performance.

For each subject system, we measured performance of *all valid*[6] *configurations* (whole-population analysis) using standard benchmarks for the respective domain. We repeated the measurements multiple times to control for the measurement noise (see Section 3.6, for more detail). Based on the benchmark data, we learned performance-influence models of the subject systems using the machine-learning algorithm described in Section 3.1. In practice, one would not measure all configurations but only a sample, due to time constraints when gathering a learning set for the machine-learning process, which we demonstrated in prior work (Siegmund et al, 2015). However, for the purpose of our study, we were specifically interested in exploring the full range of tradeoffs, meaning that we were also interested in the maximum possible accuracy of the

[6] Not all combinations of configuration options are valid system configurations, because of dependencies among the configuration options.

Table 1: Subject systems; $|O|$: number of configuration options, $|C|$: number of configurations. The number of configurations is less than $|O|^2$ because of dependencies among configuration options.

System	**Domain**	$\lvert O\rvert$	$\lvert C\rvert$	**Performance metric**
AJStats	Static analysis	20	30 256	Analysis time
Apache	Web server	9	192	Response rate
BDB-C	DBMS	18	2 560	I/O time
BDB-J	DBMS	26	180	I/O time
Clasp	ASP solver	19	700	Solving time
DUNE	Stencil code	31	2 304	Solving time
HSMGP	Stencil code	32	3 456	Solving time
LLVM	Compiler	11	1 024	Optimization time
Lrzip	Archiving tool	19	432	Compression time
x264	Video codec	16	1 152	Encoding time

resulting performance-influence models (to see the maximum possible extent of the corresponding tradeoffs). So, we used the benchmark results for all configurations as the learning set. The usage of the largest possible learning set also neutralizes one of the possible reasons for overfitting: non-representative sampling of the learning set.

The learning procedure was conducted on a dedicated server with an Intel Xeon E5-2609, 2.5 GHz and 128 GB RAM, running Ubuntu 14.04. To obtain accurate models, but not to run the computation indefinitely, we terminated the learning procedure as soon as the score of the current candidate fell below 0.05 (see Section 3.1). From our experience, this ensures that we learn all actually existing performance influences, but largely avoid measurement errors manifesting in the model (i.e., overfitting). If we had continued learning, we would have essentially learned the measurement error.

After each iteration of the learning algorithm, we saved the current model (see Section 3.1) to study the evolution of the model properties. For each model, we calculated the prediction error, its size, and the computation time. To rule out the time measurement bias caused by warm-up effects and computation-setup overhead, we subtracted the time of the first learning round from the elapsed-time measurement. Considering that the initial learning rounds are the fastest, this subtraction does not introduce any relevant deviation from the actual computation time.

3.4 Results

For most subject systems, we obtained highly accurate models at the end of the learning procedure. The largest prediction error is 6.25 % for BDB-C. In Figure 3, we show how the prediction errors evolve during the learning process (the solid blue line). The AUC lies in the range between 0.07 and 0.29, indicating a marginal tradeoff between computation time and prediction error. That is, we may be able to to learn more accurate models faster.

The size of the learned models varies substantially from system to system: Among the models with the highest prediction accuracy, the smallest model has the size of 12 (BDB-J) and the largest model has the size of 544 (DUNE). In Figure 4, we show how the model size evolves for each system during learning.

Due to the dependency between computation time and model size, the tradeoff between model size and prediction error is similar to the tradeoff between computation time and prediction error, as we show in Figure 5, with similar AUC values between 0.13 and 0.29. As one would expect, more accurate models have larger sizes and, conversely, smaller models have a higher prediction error.

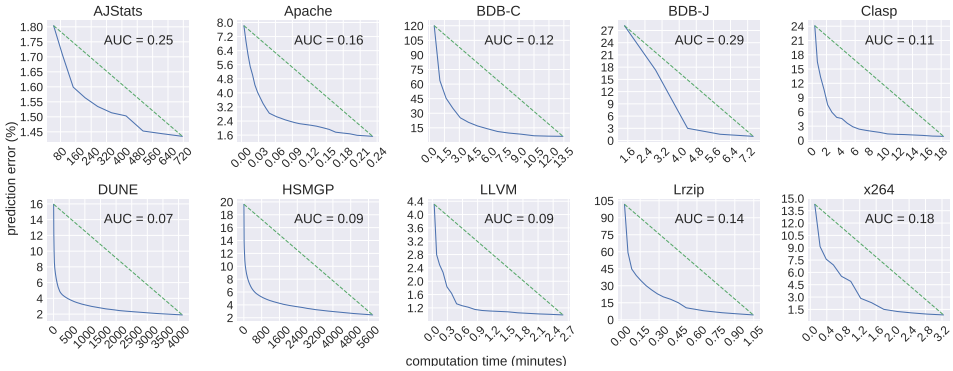

Fig. 3: Time–error tradeoff.

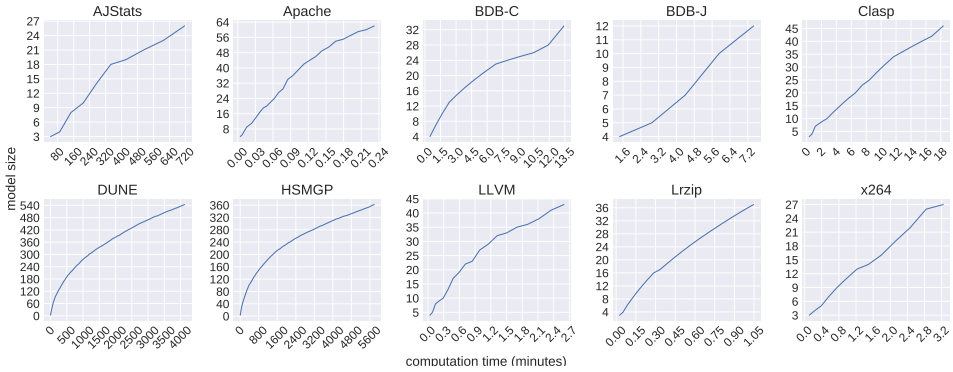

Fig. 4: Time–size dependency.

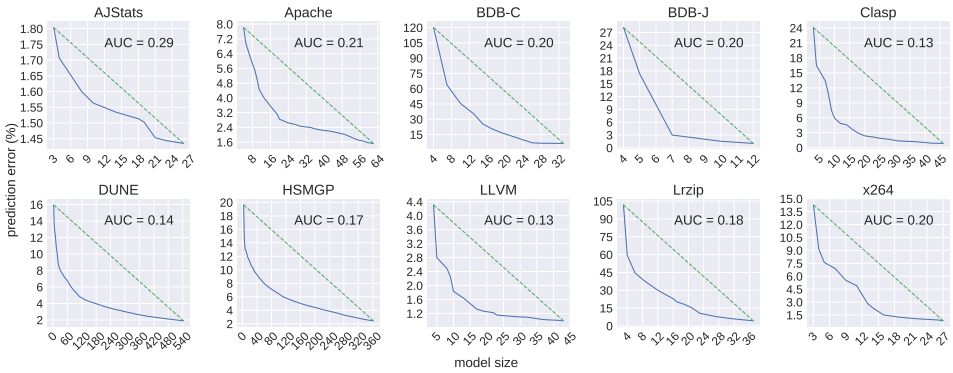

Fig. 5: Size–error tradeoff.

3.5 Discussion

3.5.1 Research Questions

Our results confirm that there is, as expected, a tradeoff between computation time and prediction error: investing more time reduces the prediction error. However, our results also show that this tradeoff is rather marginal, with AUC smaller than 0.3, for all systems. This insight is surprising and is good news for the domain of highly-configurable systems, because it means that it is possible to efficiently learn relatively accurate models. In Figure 3, we can observe how the prediction error drops quickly to a certain level early on in the learning process, whereas later the accuracy improvement saturates.

Between model size and computation time, we observe a strong positive dependency instead of a tradeoff. This result was to be expected due to the incremental nature of our algorithm, which monotonically increases model size by learning an additional term in each round. In fact, most machine learning mechanisms operate iteratively to incrementally approximate an optimal solution, simply out of necessity to handle the complexity of the huge search space (for $|o|$ options, there are $|o|$ possible main influences, $|o| \cdot (|o| - 1)/2$ possible pairwise interactions, and an exponential number of higher-order interactions among more than two options). With this huge search space, it is generally not feasible to use an exact, analytical approach.

Due to the strong positive dependency between computation time and model size, we see also a marginal tradeoff between model size and prediction error, much like we saw it between computation time and prediction error. While, again, we can learn more accurate models that are larger, the increased accuracy benefits are small. The fairly small models, early in the learning process, can characterize the performance of highly-configurable software systems already fairly accurately.

So, with respect to the first research question (RQ1), we conclude that, for learning performance-influence models for highly-configurable software systems, the tradeoffs between computation time and prediction error and between model size and prediction error are marginal; furthermore, model size and computation time have a strong positive dependency. What this means for practice is that learning simple models can be suitable to serve multiple or even all use cases, as identified in our discussions with experts (RQ2). This is an important insight for the research community: Although, the tradeoffs are known in the machine-learning literature, it was previously unclear to what extend they affect learning performance-influence models for software systems, that is, whether a large configuration space can only be accurately described with complex models learned with significant resource investment. Fortunately, we were able to show that this worst-case scenario is not the rule for real-world software systems.

3.5.2 Understanding the Tradeoffs: Influence of Interactions

A followup question that arises from our results is why the tradeoffs are so marginal. That is, why are accurate models also simple and can be learned in feasible time in this domain? To answer this question, we additionally analyzed our experiment's data for the role of interactions. A hypothesis is that the tradeoffs are marginal because performance in software systems can be described with few main effects, whereas many options and most potential interactions do not affect performance much.

Specifically, we analyzed what kinds of terms are learned in each round and how do they contribute to the accuracy of the model. We distinguish between influences of individual options (term size 1), influences of interactions between two options (term size 2), influences of interactions among three options (term size 3), and so forth. We plot our observations in Figure 6. Each plot shows how with additional time (left to right) additional terms are learned and how the prediction error is reduced. We specifically distinguish terms of different sizes using different background colors. For example, for AJSTATS, the model with the prediction error of 1.7 % (bottom x-axis) contains four model terms representing the influence of only individual options (i.e., four model terms of terms size 1 each). Then, during the learning process, a fifth term is added describing an interaction among four options, decreasing the prediction error to 1.6 %. Note that prediction error and computation time share an axis, but the scales are independent: prediction error reduces linearly, but computation time grows superlinearly. So, the final marginal reductions in prediction errors typically require significant investment in computation time.

Figure 6 reveals that few mostly small model terms are sufficient to build relatively accurate models. Considering interactions among options is important to achieve accuracy, but high accuracy can be reached without considering a huge number of interactions among many configuration options. For the most of the systems, 10 model terms with size 3 or lower are sufficient to build a model with a prediction error of under 5 %. Adding more interaction terms of larger size later in the learning process results in marginal improvements only. This explains why we did not observe strong tradeoffs among prediction error and computation time earlier. In fact, the substantial increase of the share of larger model terms and the simultaneous growth of the total number of model terms needed for very high accuracy may be an indication for the overfitting effect. These additional model terms may describe measurement noise rather than the actual performance behavior of the system.

Note that the measured computation times should be considered in relation to the corresponding prediction errors and not as absolute values. Consider the APACHE case study, which is one of the smallest in terms of configuration options (9) and in terms of configurations (192). To calculate a performance-influence model with 3 % prediction error for this system takes about 3 seconds. But calculating a slightly more inaccurate model with 7 % prediction error is 6 times faster. So the developer can save relatively much time by stopping the learning process at earlier stages. We have a similar picture for one of the

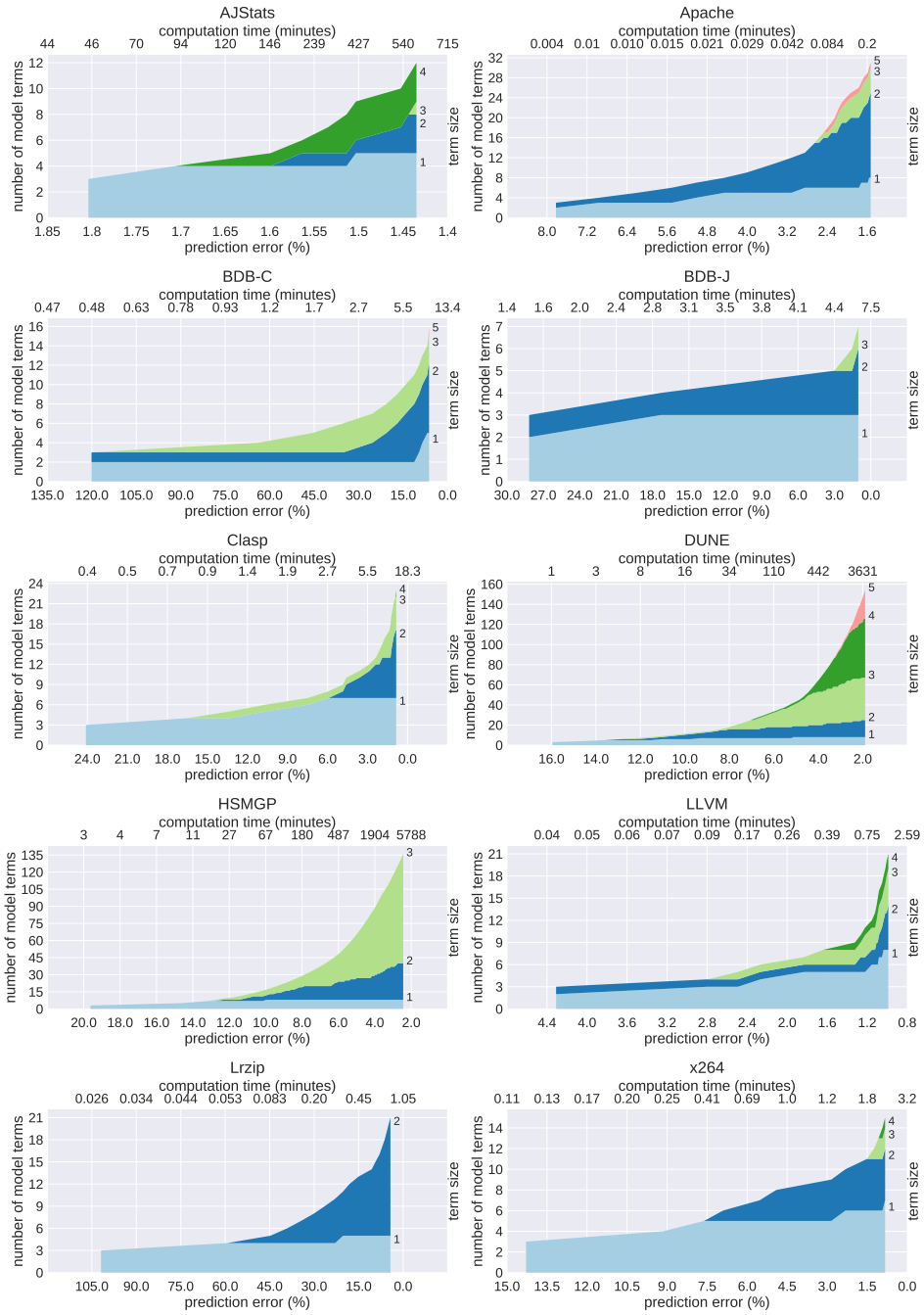

Fig. 6: Shares of interactions of different size in performance-influence models and their influence on the prediction error.

largest systems, HSMGP, with 32 configuration options and 3456 configurations. For this system, we can compute a performance-influence model with 7 % prediction error 13 times faster than a model with 3 % prediction error (which needs 3 days to compute). The same pattern applies to all our subject systems irrespective of their size: The time needed to achieve an acceptable prediction accuracy of the model is always multiple times less than the time needed for further marginal increases of the prediction accuracy. These results suggest that the same may apply for very large configuration spaces.

We conclude that the marginality of the tradeoffs can be explained by the fact that interactions among three or more configuration options have only a low influence on the performance of highly-configurable software systems. Note that our result regarding interactions primarily describes a characteristic of performance in highly-configurable software systems, not of machine learning in general.

Finally, the results of our analysis have implications for sampling and learning algorithms. For example, if an algorithm considers interactions among configuration options, it can concentrate on interactions among three configuration options and fewer. Excluding interaction of larger size reduces the search space and may improve the performance of the algorithm without sacrificing accuracy. The reduction of the search space may be substantial if we consider that the number of potential interactions grows exponentially with the number of configuration options. Already for 10 optional configuration options without additional constraints we get 1018 ($2^{10}-6$) potential interactions. Considering the emergent nature of interactions, we have to admit that any of these potential interactions may actually exist. But, as our results show, for real-world highly-configurable systems, the number of actually relevant (i.e., those that have influence on performance) interactions is much smaller than the number of potential interactions. For example, for the smallest subject system Apache with 9 configuration options (which, according to Apache's documentation, all may have influence on performance), the number of potential interactions is 124 (considering the dependencies among the configuration options). By considering only 3 interactions of 124, we already can learn a relatively accurate performance-influence model with only 4 % prediction error. That is, only 2 % of all potential interactions do actually exist and have relevant influence on performance of the Apache system.

3.5.3 Analysis of the Influence of Configuration Options and their Interactions

To identify commonalities among the influential configuration options and their interactions across the subject systems, we conducted an exploratory analysis by reading the systems' documentation and the source code to build hypotheses, and talking to the systems' developers in unclear cases. We proceeded iteratively until we were able to explain the influences. Finally, we formulated the commonalities as patterns that explain the influences, such as, *dominant configuration option*, *data pipeline*, and *workload tuning*. A complete

overview of the analyzed configuration options and their interactions is given in Appendix A on page 25.

Dominant Configuration Option. During our analysis, we found that the most influential configuration option for Apache is *KeepAlive*, which has an influence of 876.61, on average. This value denotes that enabling *KeepAlive* increases the response rate of the Web server by, on average, 876.61 responses per second (cf. performance metric in Table 1). That is, enabling this configuration option increases the performance of the Web server. The performance increases, because this configuration option enables the *persistence connection* functionality of the HTTP 1.1 protocol, which enables sending multiple requests over the same TCP connection. This functionality saves the overhead of establishing a separate connection for each request. Note that the influence of *KeepAlive* is larger than the sum of absolute influences of the next three most influential options. So, *KeepAlive* is a *dominant* configuration option, which largely determines the performance of the system and, consequently, the prediction error of the corresponding performance influence model. We found that such dominant configuration options are also present in other subject systems: *S1MiB* in BDB-J, *heuristicUnit* in Clasp, *Smoother_GSACBE* in HSMGP, etc. We also observed that the dominant configuration options interact with other configuration options in highly influential interactions in all subject systems (except for x264). Some dominant configuration options can be identified based on domain knowledge and documentation. For example, the Apache documentation states that enabling *KeepAlive* can result in almost 50 % speedup.[7] As our data suggests, knowing the dominant configuration options from the documentation, we can also assume that they interact with other configuration options.

Data Pipeline. Regarding the most influential interactions, we found that, for Clasp, DUNE, HSMGP, LLVM, and x264, the interactions arise due to the architecture of the systems that prescribes which system modules/algorithms (enabled through the configuration options) supply input data to other system modules/algorithms. That is, the architecture constitutes a *data pipeline*, and the parts of this pipeline are determined by configuration options. For example, in the Clasp solver, the options *eq* and *satPreproYes* enable preprocessing steps that can reduce the initial problem, such that the solving algorithm can find a solution for this problem faster. Therefore, the most influential interactions for this system are among the preprocessing options and solver heuristics, for example, $eq \cdot heuristicUnit$.

We observe a similar picture for DUNE and HSMGP. These systems are built such that the input data are preprocessed before they reach a solver, and the output of the solver is post-processed. The corresponding configuration options (*pre**[8] and *post** for DUNE; *numPre** and *numPost** for *HSMGP*)

[7] `https://httpd.apache.org/docs/2.4/mod/core.html#keepalive`

[8] *pre** denotes all configuration options starting with "pre".

define the number of these pre- and post-processing steps (e.g., if *pre1* is enabled, one preprocessing step is made). Each pre- and post-processing step introduces a computational overhead, which increases the solution time. Therefore, we observe that the most influential interactions of these systems include pre-, post-processing and solver-related (or smoother-related in the case of HSMGP) configuration options.

Data pipelines also explain why larger interactions include partly the same configurations options as smaller interactions. That is, why for a set of interacting options there exist interactions for its respective subsets. The reason is that smaller interactions describe smaller parts of the pipelines and larger interactions include these smaller parts. For example, this is the case in DUNE with *cells**, *pre**, and *post** options, and configuration options for solvers, which build up data pipelines (and, consequently, interactions) of size up to 5.

We assume that, in the case of a data-pipeline architecture, developers can deduce from the system's architecture which configuration options are likely to interact. Therefore, identifying these interactions using performance-influence models can be seen as a sanity check or regression test if parts of the architecture are changed. This use case corresponds to program comprehension and debugging as described in Section 2.1.

Workload Tuning. Furthermore, we found configuration options that adjust the workload by tuning the main data processing algorithm of the system. These tuning configuration options often interact with configuration options denoting processing algorithms. Configuration options *level** in LRZIP, for example, determine the compression level for the data compression algorithms. With a growing compression level the compression time grows too. Therefore, interactions among configuration options that specify the compression level and the compression algorithm arise.

Domain-Specific Interactions. Other interactions that we analyzed had a more domain-specific nature. For example, the interaction between *inline* and *licm* configuration options, which enable code optimizations in the LLVM compiler. The *inline* optimization inlines code of methods at the call sites and *licm* moves code out of loops. The peculiarity of LLVM is that these optimizations can be executed in arbitrary order (determined by the order of the corresponding command line parameters). If inlining is performed before loop optimization (which was the case in our experiment) there may be some code that gets inlined into loops. Consequently, the loop optimization has more code to process and, consequently, requires more computation time. As a result, we observe a performance interaction between *inline* and *licm*.

Another example of a domain-specific interaction is an interaction in X264 between *no_fast_pskip*, which disables Fast-P-Skip optimization, and *ref_9*, which sets the number of reference frames. Both configuration options tune the main encoding algorithm, but Fast-P-Skip optimization is more effective with more reference frames. This dependency between the two configuration options induces an interaction between them.

The two examples of the interactions show that their domain-specific nature does not allow us to describe them in general terms (like data pipelines) and requires deep understanding of the system's workings to explain their influences.

Summary. Based on domain knowledge, systems' documentation, and information provided to us by systems' developers, we were able to explain the most influential configuration options and interactions of the subject system. Furthermore, we identified several interaction patterns across multiple subject systems providing further insights in the domain of highly-configurable systems.

3.6 Threats to Validity

Internal Validity. To learn a performance-influence model, we rely on benchmark measurements that are susceptible to measurement errors. There is a threat that these measurement errors may bias the results of the learning procedure, such that the resulting model may not properly characterize the actual performance of the system. To investigate the potential influence of measurement errors on the prediction error of a performance-influence model, we conducted a separate experiment. We added random noise (representing measurement errors) to the original measurements for our subject systems[9] and repeated the learning process. Then, we compared the prediction error of the noisy models to the prediction error of the original models to see the potential influence of measurement errors.

A noise value was computed for each original measurement value by randomly sampling a value from a normal distribution. The parameter σ of the normal distribution specified the standard deviation of the noise values: the larger σ, the larger the noise value can be (i.e., the larger the measurement error). We set the initial value of σ to 0.75 (the average standard deviation of the original measurement values of our subject systems). For each subject system, we doubled σ and repeated the learning process five times to simulate the influence of increasing measurement errors.

Analyzing the noisy models, we found that most of them had approximately the same prediction error as the original models until σ (i.e., potential magnitude of the simulated measurement errors) reached a value of 6 (i.e., the potential errors were 8 times larger, than the errors of our original measurement). From this result, we can conclude that the learning algorithm that we used is robust against realistic measurement errors. The data of the experiment is available on the supplementary Web site.

Our simple model size measure (Section 3.2.1) could be further refined to reflect the complexity of the model more accurately, for example, by considering the number of the interaction terms. We decided against this refinement,

[9] We excluded HSMGP, because conducting this additional experiment with the system would have taken several months of computation time.

because we do not have enough empirical evidence to quantify the influence of the interaction terms on the complexity of a model. Still, our interviews (Section 2.1) indicate that our simple model size measure quantifies the complexity of the models rather well.

External Validity. Our results are not automatically transferable to larger models or other subject systems. However, to increase the external validity of our study, we collected 10 real-world systems of different sizes, complexities, and from different application domains. Furthermore, the subject systems differ in the number of configuration options and in the number of resulting configurations. They are implemented in different programming languages and support configuration at compile or load time, or both. As we observed differing results regarding the number of interactions, but found a similar picture regarding the model properties, we gain some confidence that our results are general to a certain extent, because our selection of subject systems covers heterogeneous systems of important domains.

The use of a particular machine-learning technique, namely multivariate linear regression with forward feature selection, may limit the generalizability of our results.

4 Related Work

Our goal was not to propose a certain machine-learning technique for learning performance-influence models of configurable systems, but to explore the design space of performance-influence models with respect to prediction error, computation time, and model size. Next, we discuss learning techniques, feature-interaction-detection approaches, and model-size definitions related to our study.

Learning. There are a number of machine-learning techniques that can be used to learn performance-influence models. Classification and regression trees represent a successful method to learn prediction models from a learning set (Steinberg and Colla, 2009). Guo et al (2013) applied this technique to highly-configurable software systems. They required only a limited computation time and achieved a high prediction accuracy. However, decision trees and the related forests (Liaw and Wiener, 2002) have two drawbacks: First, they model variants and not configuration options and their interactions, which hinders comprehension in that the influence of individual configuration options and their interactions on performance is not explicitly denoted; Second, decision trees are unstable in that even small changes in the training set can lead to vastly different models (in contrast, we did not observe the instability problem with linear regression in our experimental setting). Other learning techniques using support vector machines (Steinwart and Christmann, 2008), Bayesian networks (Ben-Gal, 2007), evolutionary algorithms (Simon, 2013),

or Fourier transforms (Zhang et al, 2015) trade off even more comprehensibility of the underlying prediction models in return for prediction accuracy or focus more on finding the fastest configuration or reducing the number of samples instead of quantifying the influence of individual configuration options and their interactions on performance. Hence, there is only a limited choice of techniques that let us explore the tradeoffs among prediction accuracy, computation time, and model size of performance-influence models. Brosig et al (2015) study stochastic performance models acquired with different model generation approaches, but the main focus lies on the accuracy and efficiency of the generation approaches and the corresponding tradeoffs. Furthermore, the applicability of the models with regard to their complexity is not considered.

Performance Engineering. The field of performance engineering aims at modeling non-functional properties of a system to evaluate if these properties satisfy a given set of requirements (Balsamo et al, 2004; Brunnert et al, 2013; Pooley, 2000). We do not focus on a single system, but on a potentially exponential number (in the number of configuration options) of system configurations, which lifts the problem to a higher level of complexity. But what is more important, we are primarily concerned with presenting the influence of the system's configuration options and their interactions on the system's performance to a user in a concise and understandable way without sacrificing the model's prediction accuracy.

Feature Interactions. Interactions among configurations options are the key to learning accurate performance-influence models. Nhlabatsi et al (2008) and others as well as Calder and Miller (2006) surveyed detection mechanisms for feature interactions. Other approaches focus on properties such as semantic correctness (Classen et al, 2010; Apel et al, 2011) and global system behavior (Prehofer, 2004) in the presence of feature interactions. Zhang et al (2016) propose a mathematical model of performance-relevant feature interactions and describe two algorithms to automatically detect them and quantify their influence. In our own previous work, we proposed a number of techniques for finding performance interactions using sampling heuristics in combination with linear programming (Siegmund et al, 2012, 2013). Here, we focus not on detecting interactions, but on the effect of interactions on model size, computation time, and prediction error of the learned performance-influence model. We are not aware of any other studies that explore the properties of performance-influence models of highly-configurable systems in this way.

Model Complexity. There is no single accepted definition and measure for model complexity. One approach is to define model complexity through its size. The larger the model the more difficult it is to comprehend and to use. Several measures have been proposed to measure such model size: Schruben and Yucesan (1993) proposed a measure based on McCabe's software complexity measure, Wallace (1987) defined a similar measure. Although the given

definition of complexity is similar to ours, the proposed measures cannot be applied to our models, because they have been developed for graph-based model representations. Another approach is to define the size of a model through its susceptibility for overfitting, that is, the more complex a model, the higher its ability to fit random noise in the data (Myung and Pitt, 2004). Although this definition describes an important property of a model, it does not fit the research questions that we addressed in this work.

5 Conclusion

Performance-influence models help developers and users to better understand performance characteristics of complex configurable software systems. An ideal performance-influence model should have low prediction error, short computation time, and small model size. However, there are usually tradeoffs between these properties that do not allow to optimize for all of them at once. In our discussions with four domain experts, we identified two important practical use cases for performance-influence models: performance prediction and program comprehension. Performance prediction would require a model with the lowest possible prediction error; program comprehension would require a model of small size and short computation time.

Since it is unclear to what extent the tradeoffs among prediction error, model size, and computation time affect the applicability of performance-influence models in these two use cases, we conducted an empirical study with the goal of systematically exploring the properties of the configuration spaces of 10 real-world highly-configurable software systems. Our results show that there are indeed tradeoffs between prediction error and model size and between prediction error and computation time. However, we found that these tradeoffs are rather marginal, such that accurate and also simple performance-influence models can be learned in feasible time, which is surprising and good news.

To further understand why efficient learning is possible, we analyzed the learned performance-influence models regarding the influences they capture. We found that individual configuration options and interactions between, at most, three options explain most of the performance variances. That is, identifying and learning the influence of interactions between more than three options will likely improve the prediction accuracy only by a tiny fraction, but will still increase computation time and model size considerably. Our research provides a foundation for many approaches in the area of performance-influence modeling and learning of configurable software systems, as it shows, for the first time, how the important properties of accuracy, model size, and computation time influence each other and that learning methods concentrating only on a subset of options and interactions are viable.

To gain further insights into the domain of highly-configurable systems we investigated why the systems' configuration options and their interactions have that particular influence on performance which we observed in the ex-

periments. We traced the reasons for the observed influences back to the architecture of the systems and interdependencies among system components. Therefrom, we extracted general patterns, such as dominant configuration option and data pipeline, which, when discovered in a systems, hint at which configuration options of this system are likely to interact. Identifying these expected interactions using performance-influence models can be also seen as a sanity check or regression test if parts of the architecture are changed.

Acknowledgements Kolesnikov's, Grebhahn's, and Apel's work has been supported by the German Research Foundation (AP 206/5, AP 206/6, AP 206/7, AP 206/11) and by the Austrian Federal Ministry of Transport, Innovation and Technology (BMVIT) project No. 849928. Siegmund's work has been supported by the German Research Foundation under the contracts SI 2171/2 and SI 2171/3. Kästner's work has been supported in part by the National Science Foundation (awards 1318808, 1552944, and 1717022), the Science of Security Lablet (H9823014C0140), and AFRL and DARPA (FA8750-16-2-0042).

Appendices

A Influence of Configuration Options and their Interactions

Table 2: A list of the most influential configuration options and interactions grouped by subject system. For each configuration option and interaction, we indicate its influence on performance and give a description. We also denote if the configuration option (if enabled) or interaction (if present) increases (↑) or decreases (↓) the performance of the system.

№	Config. Option/ Interaction	Influence	Description
	Apache		
1	*KeepAlive*	876.61↑	Allow multiple requests over the same TCP connection; speeds up transmission
2	*HostnameLookups*	-233.61↓	Perform a DNS lookup for every request; causes communication overhead
3	*InMemory*	197.48↑	Copy specified files into RAM on startup; reduces I/O
4	*AccessLog*	-116.80↓	Log every request in the logfile on disk; causes I/O overhead
5	*InMemory* · *KeepAlive*	166.87↑	Files cached in RAM are served over the same connection; speeds up transmission
6	*AccessLog* · *KeepAlive*	-157.49↓	Disk I/O induced by logging reduces performance even if *KeepAlive* is enabled
	AJStats		
1	*CodeFormatter*	3048.44↓	Preprocess code for parsing; avoids unnecessary parsing
2	*Interfaces*	-304.99↑	Disable interfaces statistics; speeds up processing
3	*ClassMethods*	-198.27↑	Disable calls-methods statistics; speeds up processing

4	*ClassConstructors* · *CodeFormatter*	-664.49↑	Disable constructor statistics; the effect is increased in the presence of *CodeFormatter*
	BDB-C		
1	*PS16K*	-1.10↑	Set page size to 16 K; read longer portions of data from the disk and speed up data retrieval
2	*PS32K*	-1.06↑	Same as 1
3	*HAVE_CRYPTO* · *HAVE_HASH* · *PS32K*	43.29↓	Hash data structure performs poorly if the stored data is encrypted
4	*HAVE_CRYPTO* · *HAVE_HASH* · *PS16K*	16.73↓	Same as 3, but the performance decrease is smaller with smaller page size
	BDB-J		
1	*S1MiB*	44078↓	Sets recovery log size to 1 MB (default is 100 MB); increases I/O overhead
2	*Finest* · *S1MiB*	222790↓	Save maximum possible recovery information in multiple small recovery log files; increases I/O overhead
	Clasp		
1	*heuristicUnit*	345493↓	Enable Unit heuristic with an expensive Lookahead operation
2	*eq*	-92677↑	Enable preprocessing that may reduce the problem and speed up solving
3	*heuristic*	-35218↑	Enable Berkmin-Heuristic; faster than Unit-heuristic
4	*satPreproYes*	19959↓	Enable SatElite-like preprocessing that may reduce the problem; preprocessing introduces overhead
5	*eq* · *heuristicUnit*	-163900↑	The heuristic works on the reduced problem and can solve it faster
6	*heuristicUnit* · *satPreproYes*	-148980↑	The problem is reduced through preprocessing and solved faster by the Unit heuristic
	DUNE		
1	*post0*	2793.76↓	Disable postprocessing steps; without postprocessing the main algorithm requires more time to calculate a solution, which results in a decreased performance
2	*cells50*	-1605.82↑	Set the size of the computation domain (workload) to 50 (the smallest workload)
3	*CGSolver* · *post0* · *pre1*	17946.17↓	The interaction describes a data pipeline including a solver, a post, and a pre-processing step
4	*CGSolver* · *cells55* · *post0* · *pre1*	12810.57↓	The interaction describes a data pipeline as in 3 plus the *cells55* option, which sets the highest workload for the pipeline
	HSMGP		
1	*Smoother_GSACBE*	4669.89↓	Enable the GSACBE smoother, which is an essential part of the multigrid algorithm; GSACBE is one of the slower smoothers
2	*Smoother_GS*	-513.12↑	Enable the GS smoother, which is faster than GSACBE
3	*Smoother_GSACBE* · *numPost_0* · *numPre_1*	-4976.29↑	The interaction denotes a data pipeline with a pre, post-processing, and a smoother
4	*Smoother_GSACBE* · *numPost_0* · *numPre_2*	-3377.14↑	Same as in 3, but increasing the number of preprocessing steps reduces performance compared to 3

	LLVM		
1	*gvn*	24.27↓	Enable Global Value Numbering optimization; introduces computational overhead
2	*instcombine*	17.61↓	Enable combining of redundant instructions; introduces computational overhead
3	*inline*	10.67↓	Enable code inlining; introduces computational overhead
4	*inline · licm*	31.57↓	More code is inlined in the loops that are processed by *licm*, introducing more computation overhead and decreasing performance
	Lrzip		
1	*compressionZpaq*	2032161.26↓	Enable ZPAQ compression (slower than the default BZip2)
2	*compression*	193262.66↓	Enable the default BZip2 compression
3	*compressionZpaq · level9*	3433850.93↓	The interaction describes the influence of ZPAQ algorithm with the highest compression level 9 on performance
4	*compressionZpaq · level8*	3415670.93↓	Same as in 3, but a lower compression level results in a slightly increased performance compared to 3
	x264		
1	*ref_1*	-349.61↑	Set the number of reference frames to 1 (the default is 9); less reference frames means less workload and higher performance
2	*ref_5*	-178.68↑	Same as in Line 1, but the increase in performance is twice as less as in Line 1 because of more reference frames
3	*no_fast_pskip · ref_9*	110.22↓	Fast-P-Skip can increase encoding speed; it is more effective with more reference frames; disabling Fast-P-Skip with 9 reference frames decreases performance

References

Apel S, Speidel H, Wendler P, von A Rhein, Beyer D (2011) Detection of feature interactions using feature-aware verification. In: Proceedings of the International Conference on Automated Software Engineering (ASE), IEEE, pp 372–375

Apel S, Kolesnikov S, Siegmund N, Kästner C, Garvin B (2013) Exploring feature interactions in the wild: The new feature-interaction challenge. In: Proceedings of the 5th International Workshop on Feature-Oriented Software Development (FOSD), ACM, pp 1–8

Balsamo S, Di Marco A, Inverardi P, Simeoni M (2004) Model-based performance prediction in software development: A survey. IEEE Transactions on Software Engineering 30(5):295–310

Bastian P, Blatt M, Engwer C, Dedner A, Kuttanikkad S, Ohlberger M, Sander O (2006) The distributed and unified numerics environment (Dune). In: Proceedings of the Symposium on Simulation Technique in Hannover, pp 12–14

Ben-Gal I (2007) Bayesian networks, Wiley Online Library

Brosig F, Meier P, Becker S, Koziolek A, Koziolek H, Kounev S (2015) Quantitative evaluation of model-driven performance analysis and simulation of component-based architectures. IEEE Transactions on Software Engineering 41(2):157–175

Brunnert A, Vögele C, Krcmar H (2013) Automatic Performance Model Generation for Java Enterprise Edition (EE) Applications, Springer, pp 74–88

Calder M, Miller A (2006) Feature interaction detection by pairwise analysis of LTL properties—A case study. Formal Methods in System Design 28(3):213–261

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

Classen A, Heymans P, Schobbens P, Legay A, Raskin J (2010) Model checking lots of systems: Efficient verification of temporal properties in software product lines. In: Proceedings of the International Conference on Software Engineering (ICSE), ACM, pp 335–344

Domingos P (2000) A unified bias-variance decomposition. In: Proceedings of International Conference on Machine Learning (ICML), Morgan Kaufmann, pp 231–238

Grelck C, Scholz SB (2006) SaC—A functional array language for efficient multi-threaded execution. International Journal of Parallel Programming 34(4):383–427

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

James G, Witten D, Hastie T, Tibshirani R (2013) An introduction to statistical learning, vol 112. Springer

Kuckuk S, Gmeiner B, Köstler H, Rüde U (2013) A generic prototype to benchmark algorithms and data structures for hierarchical hybrid grids. In: Parallel Computing: Accelerating Computational Science and Engineering (CSE), IOS Press, pp 813–822

Liaw A, Wiener M (2002) Classification and regression by randomForest. R News 2(3):18–22

Membarth R, Hannig F, Teich J, Körner M, Eckert W (2012) Generating device-specific GPU code for local operators in medical imaging. In: Proceedings of the International Parallel & Distributed Processing Symposium (IPDPS), IEEE, pp 569–581

Myung J, Pitt M (2004) Model comparison methods. Methods in Enzymology 383:351–366

Nhlabatsi A, Laney R, Nuseibeh B (2008) Feature interaction: The security threat from within software systems. Progress in Informatics 5:75–89

Pooley R (2000) Software engineering and performance: A roadmap. In: Proceedings of the Conference on The Future of Software Engineering, ACM, pp 189–199

Prehofer C (2004) Plug-and-play composition of features and feature interactions with statechart diagrams. Software and Systems Modeling 3(3):221–234

Sammut C, Webb GI (2011) Encyclopedia of machine learning. Springer

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

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

Schruben L, Yucesan E (1993) Complexity of simulation models a graph theoretic approach. In: Proceedings of the Conference on Winter Simulation, ACM, pp 641–649

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

Siegmund N, Rosenmüller M, Kästner C, Giarrusso P, Apel S, Kolesnikov S (2013) Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption. Information and Software Technology 55(3):491–507

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

Simon D (2013) Evolutionary optimization algorithms. John Wiley & Sons

Steinberg D, Colla P (2009) CART: Classification and regression trees. The top ten algorithms in data mining 9:179

Steinwart I, Christmann A (2008) Support Vector Machines. Springer

Wallace J (1987) The control and transformation metric: Toward the measurement of simulation model complexity. In: Proceedings of the Conference on Winter Simulation, ACM, pp 597–603

Zhang Y, Guo J, Blais E, Czarnecki K (2015) Performance prediction of configurable software systems by Fourier learning. In: Proceedings of the International Conference on Automated Software Engineering (ASE), IEEE/ACM, pp 365–373

Zhang Y, Guo J, Blais E, Czarnecki K, Yu H (2016) A mathematical model of performance-relevant feature interactions. In: Proceedings of the International Systems and Software Product Line Conference, ACM, pp 25–34