Cost-Efficient Construction of Performance Models

Larissa Schmid
Karlsruhe Institute of Technology
Germany

Timur Sağlam
Karlsruhe Institute of Technology
Germany

Michael Selzer
Karlsruhe Institute of Technology
Germany

Anne Koziolek
Karlsruhe Institute of Technology
Germany

ABSTRACT

Modern high-performance applications are highly-configurable systems that provide hundreds of configuration options. Performance models offer insights into the performance of these applications and help users understand the impact of these options. Yet, crafting models for such applications proves costly due to the many configuration options and their unknown performance impacts that need to be modeled. However, some options are performance-irrelevant, and removing them can reduce construction costs without compromising accuracy. This paper explores an approach to automatically identify performance-irrelevant configuration options empirically. By leveraging established performance modeling methods, we devise cost-efficient preliminary prediction models that rely on fewer samples and analyze them to identify such options. We evaluate our approach using a real-world HPC application to demonstrate our method's effectiveness in recognizing performance-irrelevant options and the potential to save costs for performance modeling.

CCS CONCEPTS

• **Software and its engineering → Software performance**.

KEYWORDS

automatic performance modeling, empirical performance modeling, configurable systems, sampling

ACM Reference Format:
Larissa Schmid, Timur Sağlam, Michael Selzer, and Anne Koziolek. 2024. Cost-Efficient Construction of Performance Models. In *4th Workshop on Performance EngineeRing, Modelling, Analysis, and VisualizatiOn STrategy (PERMAVOST '24), June 3–4, 2024, Pisa, Italy.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3660317.3660322

1 INTRODUCTION

Many software systems are configurable, allowing the user to set functional and non-functional properties according to their needs. For example, in a materials simulation [12], users select which properties to simulate (functional) and which algorithm settings (non-functional) to use. While they set fixed values for functional options, non-functional options can be chosen optimally depending on the execution environment and the choice of functional options. However, it is non-trivial to determine how a single configuration option influences performance [10, 22]. Generally, developers and users do not know how configuration options interact and which combination of options will yield the best performance [3].

To understand the influence of configuration options on the runtime of a software system, users can use automatic performance modeling techniques to create performance models [2, 22]. However, building empirical performance models for highly-configurable HPC applications is an *expensive* process. Two factors dictate the construction cost [4]: First, the number of required experiments required to measure the system's performance. It increases with every option added to the model, known as the curse of dimensionality. Second, the costs of running an experiment on an HPC system. Optimistic estimates of the operating expenses are in the millions of euros per year [20]. During the execution of performance experiments, the infrastructure is occupied, preventing other applications from running.

Users have to select a small subset of options to create performance models [4], or decide to model all options, resulting in a hard-to-quantify trade-off between model quality and number of experiments [6, 14, 17]. In most cases, however, only a subset of the options strongly impacts application performance [13, 14]. Therefore, it is possible to remove the performance-irrelevant ones from the experiments to be executed without affecting prediction accuracy. This speeds up model construction by reducing the number of required performance experiments that must be conducted on HPC computing systems. However, domain scientists and even developers of an application often do not understand the performance influences and interactions among configuration options [3, 22]. Tools can extract performance influences of options based on static and dynamic analysis automatically [4, 23] but cannot quantify their influence. Pruning of experiments is only possible if options have a linear performance impact or no impact at all [20]. In other cases, it is necessary to rely on heuristics to reduce the experiments [6, 17], or to use an expensive full-factorial experiment design – taking 368 core hours for an application with only three options already [20].

We contribute a novel approach to ease the modeling process by introducing a pre-processing step that automatically determines performance-irrelevant configuration options and removes them from the remaining modeling process[1]. With this approach, we contribute to enable easy utilization and understanding of performance modeling tools for domain scientists, thus bridging the gap between domain expertise and performance modeling expertise.

PERMAVOST '24, June 3–4, 2024, Pisa, Italy

ACM ISBN 979-8-4007-0645-5/24/06.
https://doi.org/10.1145/3660317.3660322

[1] Supplementary material: doi.org/10.5281/zenodo.10979156

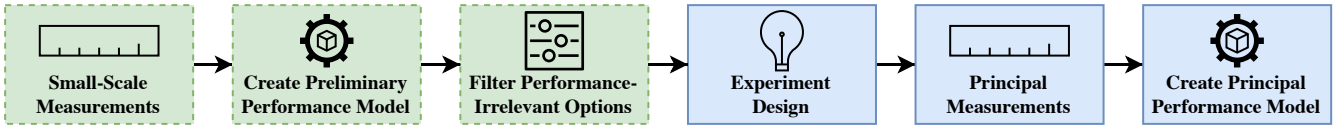

Figure 1: Our pre-processing (green, dashed) removes performance-irrelevant configuration options from the modeling process (blue, solid).

Our research aims to address these questions:

RQ1 Can we identify performance-irrelevant configuration options by reusing a performance modeling method with fewer samples?

RQ2 Can we reduce the cost of performance modeling by introducing our identification process of performance-irrelevant configuration options as a pre-processing step?

We address these questions by first evaluating if our approach can accurately classify configuration options as performance-irrelevant. Second, we evaluate if applying our approach can save costs during creation of the principal performance model, despite introducing the additional pre-processing step.

In the following, Section 2 gives an overview of the state-of-the-art of performance prediction of configurable systems. Section 3 covers our concept. Section 4 introduces our case study and how we apply our approach to it. We present our evaluation in Section 5. Section 6 shortly discusses related work, before Section 7 concludes the paper and touches upon future work.

2 STATE-OF-THE-ART

Traditionally, constructing empirical performance prediction models automatically involves three phases: designing the experiment, executing the application under consideration with selected configurations, and creating an empirical model based on the measured samples. Figure 1 shows the current process to build a performance prediction model as the last three steps (blue, solid). It starts with the experiment design phase, where the user selects the considered options and a strategy for sampling from all resulting possible configurations. Sampling strategies can, for example, rely on achieving a specific coverage, mathematical criteria, or sample configurations randomly [6]. In the next step, the application is executed with the configurations derived from the experiment design phase to collect the required sample measurements. This step is the most expensive as the user needs to conduct the performance experiments on the system they want to have a performance model for, typically a high-performance system. In the last step, the acquired measurement samples are supplied to the performance modeling tool that creates the empirical performance prediction model using a machine learning approach [6], such as Classification and Regression Trees or Multiple Linear Regression.

Different approaches can be applied to implement these three steps. For instance, the performance modeling tool DECART [8], building on CART [7], uses random sampling for the experiment design and Classification and Regression Trees as a learning technique to create a model of the correlation between option selections and performance measurements. Figure 2 shows an overview of the modeling process. DECART employs automated resampling

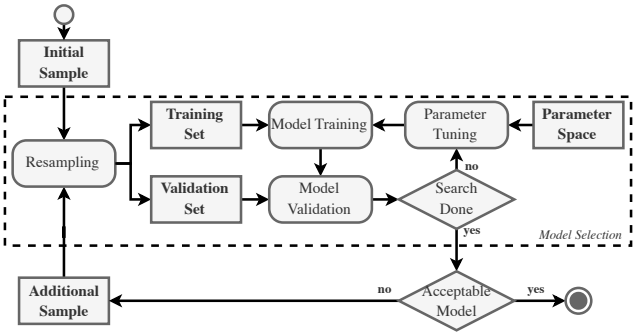

Figure 2: Overview of DECART, as depicted in Guo et al. [8].

and parameter tuning to reuse the available measurement data efficiently. It uses resampling to partition the samples into a training set for learning the performance prediction model and a validation set to evaluate the produced results. This allows integrated model validation without requiring additional validation measurement sets. Parameter tuning is used to systematically and automatically search through the parameter space of CART in order to find the parameter values that produce the performance prediction model with the highest prediction accuracy.

While there are many different tools (e.g., DECART [8], SPLConqueror [22], Extra-P [2]) that implement the general process, they can generally be classified into black-box and white-box approaches. Black-box approaches treat the application as black-box, only learning the performance models from the correlation between measurement data and configurations executed. White-Box approaches leverage knowledge about the application and how the configuration options impact performance for performance modeling. Tools such as Perf-Taint [4] and Comprex [23] extract possible performance impacts of options automatically based on static and dynamic analysis but cannot quantify their impact. An option influencing only small parts of the execution could have a critical impact, while it could also just change minor things. Take the code snippet in Listing 1 as an example: While the variable `global_offset` does impact the number of iterations via the `local_offset` variable in the loop, it only changes the iteration count by one. However, `x` directly impacts the number of iterations. Given a high value for `x`, we can assume that `global_offset` does not significantly impact the performance. Therefore, we do not have to model the impact of the `global_offset` configuration options on performance. Nevertheless, state-of-the-art tooling does not provide a way to gain and utilize this knowledge. As the tooling does not provide a strategy to guide the user in selecting the parameters that significantly

impact performance, users have to collect samples that consider all configuration options, resulting in high costs for performance modeling.

Listing 1: Example computation using configuration options global_offset and x.

```
void calculateAll(int global_offset, int x) {
  int localOffset =
        globalOffset % 2 == 0 ? 0 : 1;
  for(int i = localOffset; i < x; i++) {
    calculate();
  }
}
```

3 CONCEPT

With our approach, we improve the parameter selection for the experiment design phase. Figure 1 shows the envisioned performance modeling process: Instead of directly starting the experiment design process (blue, solid), we first employ our optimization process (green, dashed) that identifies and removes performance-irrelevant configuration options. After that, the usual experiment design phase can be conducted with a reduced parameter set. Our optimization process involves three additional steps: Considering all available configuration options, we first collect samples in a cheap way by conducting small-scale experiments (see Section 3.1). We then build a preliminary performance prediction model from the collected samples using a preexisting performance modeling method (see Section 3.2). Based on this performance model, we classify all configuration options as either *performance-relevant* or *performance-irrelevant* (see Section 3.3). Thus, it is possible to exclude performance-irrelevant options from the further modeling process.

Our approach allows users to select options from a reduced set of only performance-relevant configuration options without losing predictive power in the resulting performance model. We identify two key benefits: First, knowledge about the internal structure of the examined application or expert knowledge about performance engineering is not required anymore, as our approach classifies the options into performance-relevant and -irrelevant. Second, in addition to saving a significant amount of time by modeling fewer parameters during the actual performance modeling, we can execute the parameter identification experiments on cheaper compute infrastructures, such as consumer desktop computers or workstations, even if the final performance model utilizes high-performance systems for measurement acquisition. We hereby build on results of previous studies [13] that have shown that if a configuration option or interaction between configuration options is measured to have an influence on performance on one hardware, this property is typically preserved across differing environments.

In the following, we elaborate on our approach in detail. Section 3.1 explains how we keep the sampling process for our small-scale experiments cheap yet extensive enough to collect meaningful samples. Section 3.2 details our requirements for a performance modeling method used to create the preliminary performance models. Finally, Section 3.3 explains how we identify irrelevant options from the preliminary performance models.

3.1 Small-Scale Measurements

Our identification process adds three pre-processing steps to the performance modeling pipeline. Therefore, we must ensure that it decreases cost in the later stages of performance modeling, outweighing the additional cost incurred. To reduce the number of measurements for the optimization step, we only measure two different values for numeric configuration options because, for the identification of relevant options, we only need to detect a leap in the runtime when changing option values. For non-binary and non-numeric configuration options, such as selection options, however, we have to analyze every possible configuration value as we cannot assume a (partial) order. In addition to the number of samples, their cost is also relevant. To keep the cost of the individual samples low, we use small, yet realistic problem sizes and value ranges of configuration options. It is the responsibility of the user to select these values carefully using domain knowledge. Note that we do not require the user of our approach to have knowledge about the internals of the application or expertise in performance engineering. However, we do assume them to be familiar with the domain of the application. That means that they can configure the application according to the problem they want to compute using functional options.

3.2 Create Preliminary Performance Model

Our proposed process can leverage any performance modeling method. The sole prerequisite is that it derives an empirical performance model derived from runtime measurements. Ideally, it should swiftly produce preliminary models that offer a realistic representation of application performance based on small-scale measurements. As our process is a pre-processing step, we prefer black-box models as white-box models come with significant instrumentation overhead. We select DECART [8] (see Section 2) as the exemplary performance modeling method to build the preliminary performance models, as it promises to build performance models from a few random samples that have a decent prediction accuracy in the 90% range [8] and take only a few seconds to learn.

3.3 Filter Performance-Irrelevant Options

In this last step, we examine the created performance model regarding the options it uses for creating its performance prediction. In the case of DECART, the models contain a list of configuration options used within the model. Inherently, only the configuration options integrated into the models can impact its performance prediction. Therefore, we classify every option in that list as performance-relevant and all other options, which do not appear in the model, as performance-irrelevant.

4 CASE STUDY: PACE3D

We illustrate our concept presented in Section 3 with a real-world case study based on Pace3D (Parallel Algorithms for Crystal Evolution) [12], a multi-physics framework for digital material research. Pace3D is highly configurable, offering more than 170 tools for pre-

and post-processing of computations alone. Consequently, the flexibility offered by the software system introduces many configuration options. This makes it challenging for the domain scientists using the software to understand the performance impacts of the many options and consequently choose a configuration that will lead to good performance. However, the immense number of configuration options and their interactions make building performance models for the whole application with current approaches impractical.

For our case study, we set fixed values for functional options and consider only non-functional options that do not change the final result of the simulation. We consider 34 configuration options for the chosen computation scenario, which consist of 9 binary options, two binary vector options with two elements each, three selection options, 19 numeric options, and one numeric vector with three elements. The reduced set includes, among others, the simulation volume, number of preprocessing steps, simulation coefficients, time steps, random generator settings, numeric scaling factors, and the number of MPI processes.

4.1 Small-scale Measurements

As DECART only supports binary options, we map every non-binary option to a binary representation. We employ two predefined values (low/high) for numeric configuration options to reduce the number of required measurements. Thus, the 34 numeric options are represented as 66 binary options. DECART uses a feature-size heuristic to prescribe the number of required samples. Therefore, our simple model will have $N = 66$ options. We generate samples randomly.

As illustrated in Figure 2, the DECART modeling process is iterative, repeating the sampling and modeling process until the learned model has a validation error below 10%. As we do not know how many samples and resulting model prediction accuracy we need to identify performance-irrelevant options, we repeat the sampling process ten times. With these measurements, we can create models with an increasing number of samples from N to $10N$ that we can evaluate separately.

Performing Measurements. We run our experiments on an on-premise cluster on nodes with an AMD Opteron 2378 8-Core processor @ 2.4 GHz and 16 GB memory. This cluster is regularly used for simulation runs of Pace3D, thus a realistic execution environment for such a simulation. We repeat each measurement five times, observing a mean coefficient of variation of 4.84%.

4.2 Create Preliminary Performance Model

Using the small-scale measurements, we create ten inputs for DECART, each with 66 samples more than the one before. We supply each measurement individually to DECART, meaning that the first input file contains $66 * 5 = 330$ measurements, as we repeated each measurement five times. We choose 10-fold-cross-validation as resampling and grid search as a parameter optimization algorithm as these values proved best [8]. DECART generates multiple performance models for every input with an increasing number of samples used in the training set, requiring the user to review and select the acceptable models. Guo et al. [8] recommend selecting a model with a validation error below 10%. If there is no such model, the sample size should be increased. However, as our identification process is only a preprocessing step, we will choose the next best model even if it has a higher prediction error when no model is available with a validation or generalization error below 10%. Table 1 shows an overview of our selected models. It lists the number of measurements used for creating each model, the time for creating it, and validation and generalization errors. While validation and generalization errors are below 10% for nine out of ten models, with only the generalization error of model ten being slightly above 10%, their explanatory power is inherently limited to the number of experiments conducted for the respective model.

Model No.	No. of Experiments	Validation Error (%)	Generalization Error (%)	Model Time (s)
1	330	6.17	6.43	5.26
2	660	7.21	6.67	7.13
3	990	6.34	8.05	8.74
4	1320	8.06	8.93	10.10
5	1650	8.31	7.80	13.62
6	1980	8.23	9.53	12.14
7	2310	9.09	9.87	12.43
8	2640	8.63	9.35	15.24
9	2970	9.11	9.95	14.83
10	3300	7.82	10.51	13.86

Table 1: Our preliminary performance models for PACE3D, each with an increasing number of experiments used.

5 EVALUATION

We evaluate our approach based on the case study from Section 4. To evaluate our approach with the previously detailed case study, we first assess if our approach accurately classifies options as performance-irrelevant. Second, we evaluate if our approach can save costs for creating an exhaustive performance model despite the additional pre-processing steps introduced.

5.1 RQ1: Accurate Identification

We create ten performance prediction models with increasing sampling sizes (see Section 4.2) and analyze the models by inspecting which configuration options they use. Table 2 shows the results: Our model analysis classifies six to sixteen options out of 34 as irrelevant to the performance. We evaluate if this classification is correct two-fold: First, we measure the relative runtime deviation when changing the considered option. Second, we interview the lead developer of Pace3D to state his assessment on the performance relevance of all considered options.

5.1.1 Measuring Performance Impact. To measure the performance impact of options identified as irrelevant, we measure the relative runtime deviation when changing the considered option. All other options are activated or set to a default value. We repeat each measurement five times. Table 2 shows our results, indicating that the identified options are indeed performance-irrelevant. We show the deviation of runtime in percent when changing the respective option. As we use the same configurations for measuring the performance impact of a specific option across models, the percentage

Option Name	Samples Measured During Pre-Processing										Expert
	66	132	198	264	330	396	462	528	594	660	Assessment
Φ dynamic memory	0.46 ×	- ✓	- ✓	- ✓	- ✓	- ✓	- ✓	- ✓	- ✓	- ✓	Major
Φ block data	0.47 ×	- ✓	- ✓	- ✓	- ✓	- ✓	- ✓	- ✓	- ✓	- ✓	Minor
Φ avg RDTIC	- ✓	2.39 ×	- ✓	- ✓	- ✓	- ✓	- ✓	- ✓	- ✓	- ✓	Minor
Φ driving force nI.	- ✓	- ✓	- ✓	- ✓	- ✓	2.33 ×	- ✓	- ✓	- ✓	- ✓	Minor
Φ # Active Phases	0.60 ×	- ✓	- ✓	- ✓	- ✓	- ✓	- ✓	- ✓	- ✓	- ✓	Minor
Φ phi index	0.60 ×	- ✓	- ✓	- ✓	- ✓	- ✓	- ✓	- ✓	- ✓	- ✓	Minor
Φ avg driving force	3.71 ×	- ✓	- ✓	- ✓	3.71 ×	- ✓	- ✓	- ✓	- ✓	- ✓	Minor
Φ # LROPB locksize	- ✓	2.03 ×	2.03 ×	- ✓	- ✓	- ✓	- ✓	- ✓	- ✓	- ✓	Minor
prec. smear iterations	4.88 ×	- ✓	- ✓	- ✓	- ✓	- ✓	- ✓	- ✓	- ✓	- ✓	Minor
IO buffer scale factor	0.12 ×	- ✓	- ✓	- ✓	- ✓	- ✓	- ✓	- ✓	- ✓	- ✓	Minor
RNG manual seed	2.29 ✓	- ×	- ×	- ×	- ×	- ×	- ×	- ✓	- ×	- ×	No
SI scal. factor ampere	0.48 ✓	0.48 ✓	0.48 ✓	0.48 ✓	0.48 ✓	0.48 ✓	0.48 ✓	0.48 ✓	0.48 ✓	0.48 ✓	No
SI scal. factor candela	0.02 ✓	0.02 ✓	0.02 ✓	0.02 ✓	0.02 ✓	0.02 ✓	0.02 ✓	0.02 ✓	0.02 ✓	0.02 ✓	No
SI scal. factor kelvin	0.21 ✓	0.21 ✓	0.21 ✓	0.21 ✓	0.21 ✓	0.21 ✓	0.21 ✓	0.21 ✓	0.21 ✓	0.21 ✓	No
SI scal. factor kg	0.08 ✓	0.08 ✓	0.08 ✓	0.08 ✓	0.08 ✓	0.08 ✓	0.08 ✓	0.08 ✓	0.08 ✓	0.08 ✓	No
SI scal. factor meter	0.22 ✓	0.22 ✓	0.22 ✓	0.22 ✓	0.22 ✓	0.22 ✓	0.22 ✓	0.22 ✓	0.22 ✓	0.22 ✓	No
SI scal. factor mol	0.89 ✓	- ×	0.89 ✓	- ×	- ×	- ×	0.89 ✓	- ×	0.89 ✓	- ×	No
SI scal. factor second	1.74 ✓	1.74 ✓	1.74 ✓	1.74 ✓	1.74 ✓	1.74 ✓	1.74 ✓	1.74 ✓	1.74 ✓	1.74 ✓	No
control output	1.89 ×	- ✓	- ✓	1.89 ×	- ✓	- ✓	- ✓	1.89 ×	- ✓	- ✓	Major/Minor

Table 2: Identified irrelevant options and the runtime deviation (%) when changing them. Dashes indicate that the respective option is identified as relevant. Measurements that match the expert assessment are marked with a Checkmark (✓), while measurements that do not match are marked with a cross (×).

values are the same within rows. Cells with dashes indicate that the respective option has been identified as relevant by the model. The biggest deviation caused by one of the performance-irrelevant options in runtime is 4.88%. In contrast, options that strongly impact performance often cause a performance difference of over 100%. Notably, some options are identified as performance-irrelevant by all models, while others are flagged by only some or even only one model. Model one, which uses the smallest number of samples, identifies most options as performance-irrelevant. While this could also be due to the samples not catching the performance impact of some of the options, our evaluation shows that only one of the options, `precond.smear.iterations`, could be judged as performance-relevant with an impact on the runtime of 4.88%.

5.1.2 Developer Statement. We asked one of the main developers of Pace3D, who has been working on the software for a long time, to fill out a questionnaire, indicating if they think that a specific configuration option has no, minor, or a major impact on performance. We also gave the possibility to indicate that they were unsure about the impact of the option. Table 2 shows how our expert classified the options and what options the respective model classified as irrelevant. The expert identified eleven options to be performance-irrelevant. Six of them were classified as irrelevant by all models. Model one further identified `ManualSeed` and model one, three, seven, and nine identified `Settings.SIscalingfactor.mol` as being performance-irrelevant, while the other three were not identified as irrelevant by any model. The expert further identified nine options as having a major performance impact. Seven of them were classified as relevant by all models, with the other two being misclassified by model one (`DynamicMemory`) and model one, four, and eight (`ControlOutput`), respectively. However, he further stated that `ControlOutput` would have a major impact only for configurations that have a short runtime, which will become minor for long-running simulations. As simulations used to simulate real-world scenarios are usually longer-running, we deem this misclassification to be tolerable. Ten other options identified as irrelevant by some models are classified as having a minor impact on performance by our expert. Five of them were misclassified exclusively by model one. However, misclassifications happen up to including model eight.

Overall, model one has the most misclassifications according to the expert statement. This is to be expected as it had the fewest training data.

Few models misclassified options with a major performance impact as being performance-irrelevant. Moreover, up to including model eight, some options are being classified as irrelevant while having a minor impact on the performance according to our expert. No model identified all eleven performance-irrelevant options. While not identifying all performance-irrelevant reduces potential savings by not being able to exclude them from the principal modeling, it does not impact the quality of the principal performance model. Moreover, as only model one classified an option being relevant also in large-scale settings as irrelevant, we can conclude that our approach can save costs while not compromising model quality.

5.1.3 Discussion. The results in Section 5.1.1 imply that using model one, which is the cheapest preliminary performance model, is already sufficient for pruning performance-irrelevant configuration options. However, according to our expert statement presented in Section 5.1.2, it misclassifies some options with a minor performance impact as being irrelevant. Crucially, it misclassifies an option that has a major impact on performance according to our

expert. As model one uses only a very limited number of samples, it likely did not see enough data to capture their performance influence. Models nine and ten have the least misclassifications according to our measurements and the expert statement: While not misclassifying any performance-relevant option as irrelevant, they identify seven and six, respectively, out of eleven performance-irrelevant options. However, as we only evaluate our approach with one case study, it is too early for a general recommendation on how many samples to use based on our data. For this, further experimentation is needed.

5.2 RQ2: Cost of Performance Modeling

We estimate the costs saved for creating principal performance models using black-box and white-box approaches separately.

Black-Box Modeling. Black-box performance modeling techniques rely on heuristics to estimate the required number of samples. This results in a hard-to-quantify trade-off between number of samples and model accuracy [6, 14]. Some techniques iteratively use more samples, as proposed also by DECART [8]: In each step, a model is trained with part of the data and validated with the rest. If the error exceeds a given threshold, more samples need to be added to the training data. Thus, it is hard to quantify the costs saved by our approach. However, it will still prove helpful to prune options from the configuration space, as we expect the model to converge faster due to less meaningful samples being collected. In other words, the same number of samples will result in better model quality.

White-Box Modeling. Automatic white-box approaches that do not employ heuristics for the sampling process and thus promise a good model quality need a full-factorial experiment design [4] with 5 measured values per configuration option. They can only prune configurations to measure if options do not interact at all or if they linearly affect system runtime [20]. However, they cannot quantify the impact of a configuration option, requiring the user to either model all configuration options or exclude them solely based on the number of functions they are influencing, not their actual impact. Moreover, these approaches do not support binary configuration options. In order to capture the influence of a binary configuration option, we would thus need to create two separate performance models and compare them to assess the performance impact of the binary option. Capturing performance impacts of all binary options would thus mean creating performance models for any combination of binary configuration options. For each performance model, *we save 78125 experiments by pruning the six numerical configuration options identified by model 10, which translates to 30482 core hours*, assuming an average runtime of 0.39 core hours we observed during our small-scale sampling. This estimate does not even consider that users would likely opt for measuring larger problem sizes and value ranges during the principal measurements. Gathering all of our samples took 1288 core hours.

5.3 Threats to Validity

After discussing our findings, we discuss threats to validity. We follow the guidelines of Wohlin et al. [24] and Runeson and Höst [18].

Internal Validity. Our evaluation indicates that the identified configuration options in later models indeed do not have a performance impact. However, the potential impact of filtering them on the prediction accuracy of resulting performance models remains uncertain. Additionally, the influence of these options in unmeasured scenarios outside the expert's awareness remains to be seen. Nevertheless, the consistency observed in the measured scenarios as well as the expert assessment reinforces our confidence in the overall findings.

External Validity. Our evaluation is currently limited to a single case study application. While our selected application is a real-world HPC application, an assessment of the generalizability of the approach requires further research. However, we are confident in the generalizability of our approach, as we designed it to be applicable to any HPC application.

6 RELATED WORK

In the following, we discuss related research in the areas of performance modeling, configurable systems, and experiment pruning.

Performance Modeling of Configurable Software. Various approaches exist for creating performance models of configurable software [1, 14, 16, 22]. Ha and Zhang [9] create models using deep neural networks. Shu et al. [21] present PERF-AL that uses neural networks with adversarial learning. Han et al. [11] focus on finding performance bugs by ranking configuration options regarding their performance impact. All require sampling across the whole configuration space. Our approach is orthogonal and can be used as pre-processing to any performance modeling approach for creating the principal performance model.

Pruning of Experiments. Sarkar et al. [19] propose heuristic strategies for the cost-effective sampling of configurations but do not consider reducing the configuration space by pruning options. Velez et al. [23] use program analysis to identify which configuration option influences control-flow statements in a code region. They use this knowledge to select option values for configurations that allow the exploration of all paths. However, this considers only binary configuration options. Nair et al. [15] use dimensionality reduction to reduce the configuration space and the number of required performance measurements. This approach assumes that all the options are equally important and only works for numeric options. Schmid et al. [20] reduce the number of required experiments by using parameter interaction knowledge gained through a taint analysis. They only consider numeric configuration options. In contrast, our approach works for any option type. Dominguez-Trujillo et al. [5] aim at reducing the amount of data needed to analyze performance variation in HPC applications (as caused by, e.g., operating system management activites or inconsistent system cooling patterns) by focusing on maxima distributions.

7 CONCLUSION AND FUTURE WORK

In this paper, we present an approach to identify performance-irrelevant configuration options. By excluding these options from the performance modeling process, costs for constructing performance models can be reduced. Our approach is easily applicable by domain scientists, as users are neither required to have knowledge about the internal structure of the application nor performance engineering expertise.

Our evaluation shows that we can identify performance-irrelevant options while not incorrectly filtering performance-relevant options. The cost saved by filtering the performance-irrelevant options from the principal modeling is expected to be much higher than the cost incurred by the additional small-scale measurements needed for our approach.

For future work, we would like to evaluate the approach with additional case studies. We would also like to compare different performance modeling approaches regarding their ability to identify performance-irrelevant options through small-scale samples. For example, DeepPerf [9] and PERF-AL [21] use deep neural networks that, despite having a longer training time than DECART, may require even fewer samples. Furthermore, building principal performance models using an iterative approach would allow us to quantify how much faster the model converges by disregarding irrelevant options.

ACKNOWLEDGMENTS

We would like to thank P. Uhrich for their work on this topic during their master thesis. Larissa Schmid is supported by the Ministry of Science, Research and the Arts Baden-Württemberg (Az: 7712.14-0821-2) and the pilot program Core Informatics at KIT (KiKIT) of the Helmholtz Association (HGF). This work is further supported by the research project *SofDCar* (19S21002), which is funded by the German Federal Ministry for Economic Affairs and Climate Action.

REFERENCES

[1] Alexandru Calotoiu, David Beckinsale, Christopher W. Earl, Torsten Hoefler, Ian Karlin, Martin Schulz, and Felix Wolf. 2016. Fast Multi-parameter Performance Modeling, In CLUSTER'16. *IEEE International Conference on Cluster Computing*, 172–181. https://doi.org/10.1109/CLUSTER.2016.57

[2] Alexandru Calotoiu, Torsten Hoefler, Marius Poke, and Felix Wolf. 2013. Using Automated Performance Modeling to Find Scalability Bugs in Complex Codes, In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (Denver, Colorado), William Gropp and Satoshi Matsuoka (Eds.). *2013 SC - International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Article 45, 12 pages. https://doi.org/10.1145/2503210.2503277

[3] Mikaela Cashman, Myra B. Cohen, Priya Ranjan, and Robert W. Cottingham. 2018. Navigating the maze: the impact of configurability in bioinformatics software. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE '18)*. Association for Computing Machinery, New York, NY, USA, 757–767. https://doi.org/10.1145/3238147.3240466

[4] Marcin Copik, Alexandru Calotoiu, Tobias Grosser, Nicolas Wicki, Felix Wolf, and Torsten Hoefler. 2021. Extracting Clean Performance Models from Tainted Programs, In PPoPP'21 (Korea), Jaejin Lee and Erez Petrank (Eds.). *ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, 403–417. https://doi.org/10.1145/3437801.3441613

[5] Jered Dominguez-Trujillo, Keira Haskins, Soheila Jafari Khouzani, Christopher Leap, Sahba Tashakkori, Quincy Wofford, Trilce Estrada, Patrick G. Bridges, and Patrick M. Widener. 2020. Lightweight Measurement and Analysis of HPC Performance Variability. In *2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 50–60. https://doi.org/10.1109/PMBS51919.2020.00011

[6] Alexander Grebhahn, Norbert Siegmund, and Sven Apel. 2019. Predicting Performance of Software Configurations: There is no Silver Bullet. *CoRR* abs/1911.12643 (11 2019). arXiv:1911.12643 http://arxiv.org/abs/1911.12643

[7] Jianmei Guo, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, and Andrzej Wąsowski. 2013. Variability-aware performance prediction: A statistical learning approach, In ASE'13. *International Conference on Automated Software Engineering*, 301–311. https://doi.org/10.1109/ASE.2013.6693089

[8] Jianmei Guo, Dingyu Yang, Norbert Siegmund, Sven Apel, Atrisha Sarkar, Pavel Valov, Krzysztof Czarnecki, Andrzej Wasowski, and Huiqun Yu. 2018. Data-Efficient Performance Learning for Configurable Systems. *Empirical Softw. Engg.* 23, 3 (6 2018), 1826–1867. https://doi.org/10.1007/s10664-017-9573-6

[9] Huong Ha and Hongyu Zhang. 2019. DeepPerf: Performance Prediction for Configurable Software with Deep Sparse Neural Network, In ICSE'19, Gunter Mussbacher, Joanne M. Atlee, and Tevfik Bultan (Eds.). *International Conference on Software Engineering*, 1095–1106. https://doi.org/10.1109/icse.2019.00113

[10] Xue Han and Tingting Yu. 2016. An Empirical Study on Performance Bugs for Highly Configurable Software Systems. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (Ciudad Real, Spain) *(ESEM '16)*. Association for Computing Machinery, New York, NY, USA, Article 23, 10 pages. https://doi.org/10.1145/2961111.2962602

[11] Xue Han, Tingting Yu, and Michael Pradel. 2021. ConfProf: White-Box Performance Profiling of Configuration Options, In ICPE (France), Johann Bourcier, Zhen Ming (Jack) Jiang, Cor-Paul Bezemer, Vittorio Cortellessa, Daniele Di Pompeo, and Ana Lucia Varbanescu (Eds.). *International Conference on Performance Engineering*, 1–8. https://doi.org/10.1145/3427921.3450255

[12] J. Hötzer, A. Reiter, H. Hierl, P. Steinmetz, M. Selzer, and Britta Nestler. 2018. The parallel multi-physics phase-field framework Pace3D. *Journal of Computational Science* 26 (5 2018), 1–12. https://doi.org/10.1016/j.jocs.2018.02.011

[13] Pooyan Jamshidi, Norbert Siegmund, Miguel Velez, Christian Kästner, Akshay Patel, and Yuvraj Agarwal. 2017. Transfer learning for performance modeling of configurable systems: An exploratory analysis. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 497–508. https://doi.org/10.1109/ASE.2017.8115661

[14] Sergiy Kolesnikov, Norbert Siegmund, Christian Kästner, Alexander Grebhahn, and Sven Apel. 2019. Tradeoffs in Modeling Performance of Highly Configurable Software Systems. *SOSYM* 18, 3 (June 2019), 2265–2283. https://doi.org/10.1007/s10270-018-0662-9

[15] Vivek Nair, Tim Menzies, Norbert Siegmund, and Sven Apel. 2018. Faster discovery of faster system configurations with spectral learning. *Automated Software Engg.* 25, 2 (jun 2018), 247–277. https://doi.org/10.1007/s10515-017-0225-2

[16] Juliana Alves Pereira, Mathieu Acher, Hugo Martin, Jean-Marc Jézéquel, Goetz Botterweck, and Anthony Ventresque. 2021. Learning software configuration spaces: A systematic literature review. *Journal of Systems and Software* 182 (2021), 111044. https://doi.org/10.1016/j.jss.2021.111044

[17] Marcus Ritter, Alexandru Calotoiu, Sebastian Rinke, Thorsten Reimann, Torsten Hoefler, and Felix Wolf. 2020. Learning Cost-Effective Sampling Strategies for Empirical Performance Modeling. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 884–895. https://doi.org/10.1109/IPDPS47924.2020.00095

[18] Per Runeson and Martin Höst. 2008. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering* 14, 2 (dec 2008), 131–164. https://doi.org/10.1007/s10664-008-9102-8

[19] Atrisha Sarkar, Jianmei Guo, Norbert Siegmund, Sven Apel, and Krzysztof Czarnecki. 2015. Cost-Efficient Sampling for Performance Prediction of Configurable Systems (T), In ASE'15, Myra B. Cohen, Lars Grunske, and Michael Whalen (Eds.). *International Conference on Automated Software Engineering*, 342–352. https://doi.org/10.1109/ase.2015.45

[20] Larissa Schmid, Marcin Copik, Alexandru Calotoiu, Dominik Werle, Andreas Reiter, Michael Selzer, Anne Koziolek, and Torsten Hoefler. 2022. Performance-detective: automatic deduction of cheap and accurate performance models. In *Proceedings of the 36th ACM International Conference on Supercomputing* (Virtual Event) *(ICS '22)*. Association for Computing Machinery, New York, NY, USA, Article 3, 13 pages. https://doi.org/10.1145/3524059.3532391

[21] Yangyang Shu, Yulei Sui, Hongyu Zhang, and Guandong Xu. 2020. Perf-AL: Performance Prediction for Configurable Software through Adversarial Learning. In *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* (Bari, Italy) *(ESEM '20)*. Association for Computing Machinery, New York, NY, USA, Article 16, 11 pages. https://doi.org/10.1145/3382494.3410677

[22] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. 2015. Performance-Influence Models for Highly Configurable Systems, In ESEC/FSE'15 (Bergamo, Italy), Elisabetta Di Nitto, Mark Harman, and Patrick Heymans (Eds.). *ESEC/SIGSOFT FSE*, 284–294. https://doi.org/10.1145/2786805.2786845

[23] Miguel Velez, Pooyan Jamshidi, Norbert Siegmund, Sven Apel, and Christian Kästner. 2021. White-Box Analysis over Machine Learning: Modeling Performance of Configurable Systems, In ICSE '21 (Madrid, Spain). *International Conference on Software Engineering*, 1072–1084. https://doi.org/10.1109/icse43902.2021.00100

[24] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in Software Engineering*. Springer Berlin Heidelberg, Berlin, Heidelberg. I–XXIII, 1–236 pages. https://doi.org/10.1007/978-3-642-29044-2