Noise-Resilient Empirical Performance Modeling with Deep Neural Networks

Marcus Ritter*, Alexander Geiß*, Johannes Wehrstein*
Alexandru Calotoiu†, Thorsten Reimann*, Torsten Hoefler†, Felix Wolf*

*Technical University of Darmstadt, Department of Computer Science, Germany
†ETH Zürich, Department of Computer Science, Switzerland
{marcus.ritter,alexander.geiss1,felix.wolf}@tu-darmstadt.de
{acalotoiu,htor}@inf.ethz.ch, thorsten.reimann@sc.tu-darmstadt.de
johannes.wehrstein@stud.tu-darmstadt.de

***Abstract*—Empirical performance modeling is a proven instrument to analyze the scaling behavior of HPC applications. Using a set of smaller-scale experiments, it can provide important insights into application behavior at larger scales. Extra-P is an empirical modeling tool that applies linear regression to automatically generate human-readable performance models. Similar to other regression-based modeling techniques, the accuracy of the models created by Extra-P decreases as the amount of noise in the underlying data increases. This is why the performance variability observed in many contemporary systems can become a serious challenge. In this paper, we introduce a novel adaptive modeling approach that makes Extra-P more noise resilient, exploiting the ability of deep neural networks to discover the effects of numerical parameters, such as the number of processes or the problem size, on performance when dealing with noisy measurements. Using synthetic analysis and data from three different case studies, we demonstrate that our solution improves the model accuracy at high noise levels by up to 25% while increasing their predictive power by about 15%.**

***Index Terms*—Performance analysis, performance modeling, deep learning, artificial neural networks, high performance computing, parallel processing**

I. Introduction

The field of high-performance computing (HPC) holds great potential for answering critical questions of enormous societal impact such as drug discovery or the prediction of climate change. Consequently, there is an ever-growing demand for computing power and bigger systems. However, with the increasing size of these systems, it becomes more complicated to exploit their full potential. At the same time, application complexity is expanding in a similar fashion, requiring a continuous focus on performance to productively use existing and future large-scale machines. A powerful way of analyzing the performance of an HPC application is the use of performance models. In general, a performance model provides an analytical expression of an application's behavior at different scales, providing important insights, for instance, related to the presence of scalability bugs or the adequacy of a certain computer system for a given code [1].

Extra-P [2] is an empirical modeling tool that uses linear regression to automatically generate human-readable performance models from performance data, freeing its user from the burden of laboriously deriving performance models by reasoning. To create the performance model of an application, Extra-P requires a set of small-scale experiments using different combinations of the application's execution parameters. Similar to other regression-based empirical modeling techniques, the accuracy of the models created by Extra-P decreases with an increasing amount of noise affecting the performance measurements. This is because it becomes increasingly difficult to distinguish signal from noise, especially when dealing with multiple application parameters [3].

Several studies have already highlighted the effects of noise on performance [4]–[7]. Chunduri et al. for example report that execution times of applications on Theta, a Xeon-Phi-based Cray XC system at Argonne National Laboratory, can deviate by up to 70% [8]. Consequently, such a high variance on the measurements drastically affects the accuracy and predictive power of the created performance models, rendering it very difficult to analyze the scalability of an application. Even though this is the case, at the moment there are only a few options to reduce the amount of noise, one being to take repeated measurements of an application with the same combination of execution parameters. Another frequently used approach is the limitation of the range of discoverable functions, e.g., to polynomials and/or logarithms [9], [10]. However, these measures are usually not enough when dealing with high run-to-run variations, especially when modeling multiple application parameters [3].

In this work, we propose a novel adaptive performance-modeling technique that combines the advantages of traditional regression-based modeling with deep learning to substantially improve noise resilience. First, we apply a heuristic to estimate the amount of noise in the performance data, distinguishing between noisy and calm data. If we discover high degrees of noise, we use domain adaptation to optimize the noise resilience of a pretrained deep neural network (DNN), utilizing

This work has been funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) — WO 1589/8-1 and WO 1589/10-1, and the Spark Grant CRSK-2_190359/1. Calculations were performed on the Lichtenberg Computing Cluster at TU Darmstadt, the SuperMUC Petascale System at Leibniz Supercomputing Centre, and the Vulcan Supercomputer at Lawrence Livermore National Laboratory.

the gathered noise information, to create accurate performance predictions for a specific modeling task. Otherwise, we continue with our purely regression-based method. Using this approach we are able to discover and analyze the effects of numerical parameters, such as the number of processes or the problem size, on performance even when dealing with noisy measurements. The major contributions of our work are:

- A heuristic noise classification technique that allows the estimation of the amount of noise in performance measurements.
- A DNN based empirical modeling technique that can better cope with noisy measurements, improving Extra-P's model accuracy at high noise levels by up to 25% while increasing predictive power by about 15%.
- Three application case studies that demonstrate the advantages and inherent limitations of our new approach.

The remainder of this paper is organized as follows. After reviewing related work in Section II, we provide background on automated empirical performance modeling in Section III. We then outline our new modeling approach in Section IV, followed by an extensive evaluation based on synthetic data in Section V, quantifying its accuracy and predictive power. In Section VI, we present three application case studies, demonstrating the advantages and limitations of our new approach. Finally, we summarize our results and provide a conclusion in Section VII.

II. Related Work

Performance models are a very powerful and insightful instrument to describe and understand the performance behavior of an application. In general, there are two ways to derive such models: analytical and empirical performance modeling.

The main obstacle to analytical approaches is the required expertise and the amount of effort needed in order to analyze an entire application [4]. Therefore, several approaches have attempted to automate the performance modeling process. PALM [11] and ASPEN [12] analyze application performance based on source code annotations. Vuduc et al. [13] generate performance models automatically but require the user to manually suggest the hypotheses. Hoefler et al. generate multi-parameter performance models online, though the nature of their approach limits the size of the search space, adversely affecting accuracy [14]. In contrast, our method automatically generates empirical performance models based on a set of small-scale experiments [3].

In recent years machine-learning methods such as artificial neural networks have gained increasing popularity in the area of performance modeling [15]–[21]. Didona et al., for example, use artificial neural networks to further improve the robustness of their predictions [15]. Duplyakin et al. on the other hand use them in order to refine their regression-based performance models [22]. In contrast, we use a deep neural network to create a completely new empirical modeling approach. We then adaptively switch between our old and the new DNN based modeling technique depending on the measurement data. Other approaches focus on generating models for a very specific purpose, such as learning and predicting the performance of applications based on their input parameters [23], [24], rather than providing a generic solution.

Performance variability, the difference between execution times across repeated runs of an application in the same execution environment, is a challenge that most researches have to face when analyzing the performance of applications on HPC systems [4]–[8], [25], [26]. In general noise imposes a major challenge for regression-based multi-parameter modeling, as it simultaneously affects all target variables, masking their actual impact on performance [27]. Siegmund et al., for example, use stepwise linear regression to learn a performance-influence model function from a sample set of measured application configurations, providing accurate performance predictions in scenarios with no or only very little noise on the measurements [28]. While their method has a smaller computational overhead than ours, it depends on the assumption that performance behavior is deterministic. Consequently, different runs of the same application configuration that lead to largely different performance behavior will negatively affect model accuracy. Our approach, on the other hand, automatically adjusts to the modeling task at hand, ensuring accurate results also for noisy measurements. Carrington et al. is another example for a function fitting approach that provides accurate performance predictions for scientific applications, also requiring less computational overhead than our approach [10]. However, both methods do not consider large amounts of system noise as a factor in their evaluation. In general the comparison to other function fitting approaches is non-trivial, as they are often fundamentally different. Siegmund et al., for example, create one performance model per application, whereas we create one model per application kernel. Therefore, directly comparing cost and accuracy of the two methods is not possible.

A commonly used option to address noisy measurements in empirical performance modeling is the limitation of the discoverable functions to polynomials and/or logarithms [1], [9], [10]. Other frequently applied methods are the repetition of performance measurements with the same combination of execution parameters, and the use of a more representative value for modeling such as the median or minimum. However, when dealing with high run-to-run variations in combination with multiple application parameters these measures are often not enough [3]. Another option to analyze application performance in spite of noise is an estimation based on system peak-performance metrics, as suggested by the roofline methodology [29]. Using the peak floating-point performance they are able to analyze the performance of any given arithmetic kernel on any given processor architecture. Though, in comparison to our method it does not allow an in-depth analysis of application performance at different scales based on the configuration parameters. Duplyakin et al. [22] use Gaussian process regression (GPR) [30] to increase the noise resilience of their modeling approach, while sacrificing some of their predictive power. In contrast, our novel adaptive modeling technique offers increased noise resistance without sacrificing predictive power or model accuracy.

III. Background

Our approach builds upon Extra-P [2], a tool that automatically creates performance models from a set of small-scale experiments. Each experiment represents a different configuration of an application determined by its execution parameters. A performance model is a function that describes how the performance of a program, expressed in terms of a metric such as execution time, changes, as execution parameters, such as the input problem size or the number of processes, change. The performance measurements in the set of experiments reflect this change, such that Extra-P can discover and return the underlying function. A central concept of Extra-P is the performance model normal form (PMNF), which is shown in Equation 1. The PMNF rests on the observation that the complexity of algorithms used in parallel applications is usually a combination of polynomial and logarithmic expressions. While other behaviors are theoretically possible, they are rarely found in practice.

$$f_m(x_1, \ldots, x_m) = \sum_{k=1}^{h} c_k \cdot \prod_{l=1}^{m} x_l^{i_{kl}} \cdot log_2^{j_{kl}}(x_l) \tag{1}$$

With the help of the PMNF, Extra-P expresses the effect of a number of parameters $x_1, \ldots, x_m$ on performance as a sum of terms consisting of products of polynomial and logarithmic expressions [31]. However, we set a limit of one term per parameter to find the simplest explanation for the underlying performance behavior, similar to the bias–variance trade-off in machine learning. To generate a performance model, a search space of possible model hypotheses is generated by instantiating Equation 1 with different exponents (i, j) chosen from the set E, which is shown in Equation 2. Just like the function terms, the exponents (i, j) are determined by the complexity classes that are found in real-life algorithms and applications [1]. Terms with exponents such as x^8 rarely exist in practice and are therefore not included in the standard set E. We then calculate the coefficients c_k of the hypothesis using linear regression. Finally, the best model is identified using cross-validation, choosing the hypothesis with the smallest symmetric mean absolute percentage error (SMAPE) [3]. The PMNF can be interpreted as a prior introduced into the modeling process to counter the effects of imperfect measurements, forcing Extra-P to disregard unlikely outcomes.

$$\begin{gathered} E = \left(\left\{0, \frac{1}{4}, \frac{1}{3}, \frac{1}{2}, \frac{2}{3}, \frac{3}{4}, 1, \frac{3}{2}, 2, \frac{5}{2}\right\} \times \{0,1,2\}\right) \\ \cup \left(\left\{\frac{5}{4}, \frac{4}{3}, 3\right\} \times \{0,1\}\right) \cup \left(\left\{\frac{4}{5}, \frac{5}{3}, \frac{7}{4}, \frac{9}{4}, \frac{7}{3}, \frac{8}{3}, \frac{11}{4}\right\} \times \{0\}\right) \end{gathered} \tag{2}$$

For modeling, Extra-P requires at least five experiments per considered execution parameter, reflecting different values of this parameter, while the values of all other parameters remain fixed [3]. Henceforth, we call this special combination of the different execution parameter values of an experiment measurement points. Furthermore, Extra-P requires at least one additional experiment with a measurement point outside these sequences, to decide whether the effect of the parameters is additive (independent) or multiplicative (compounded).

Depending on the target system, it can be difficult to distinguish between signal, the influence of the different parameters on performance, and noise. Noise or performance variability, describes the variation of an execution metric, such as runtime, across repeated runs of an application in the same execution environment using an identical configuration. Typical causes of performance variability are node-level effects, such as OS noise, dynamic frequency scaling, or system-level effects, such as network and file-system congestion in the presence of concurrently running jobs [7]. To counter the perturbation caused by noise, each experiment (performance measurement) must be repeated several times, five repetitions being sufficient in most scenarios [3]. A higher number of repetitions can help to increase noise resilience, though only to a certain degree. Nevertheless, with each additional model parameter, the effect of noise becomes more pronounced. If the noise is high enough, our current countermeasures, the PMNF-based prior and the use of the median of the measurement repetitions, no longer suffice.

A possible solution to this problem is the use of deep neural networks. One can think of the problem as a standard classification task, with the input of the DNN being the set of performance measurements and the output being a function hypothesis. To recreate the function search space, each class represents a possible combination of terms and exponents that build a function hypothesis. Deep neural networks in particular have shown to be more robust to noise, and adapt to many different circumstances when trained appropriately [32], [33]. Therefore, a DNN should be an ideal choice for building a noise-resilient performance modeler.

IV. Approach

Below, we introduce the idea of adaptive performance modeling. After looking at the process as a whole, which is depicted in Figure 1, we first explain the noise classification module, which estimates the amount of noise present in our performance measurements and subsequently decides which approach (DNN vs. regression) should be used for modeling. Second, we review the necessary preprocessing steps of the measurement data that are required by the DNN modeler. Third, we show how the DNN uses this data to create performance models. Finally, we explain how we make use of transfer learning to help our DNN approach adjust to different types of application parameters, measurement configurations, and noise levels.

A. Adaptive performance modeling

One can imagine the adaptive modeler as a black box, with the performance measurements as the input and an automatically generated performance model as the output. In more detail, the adaptive modeler consists of five different components: the noise estimation module, the preprocessing module, the DNN performance modeler, the transfer learning

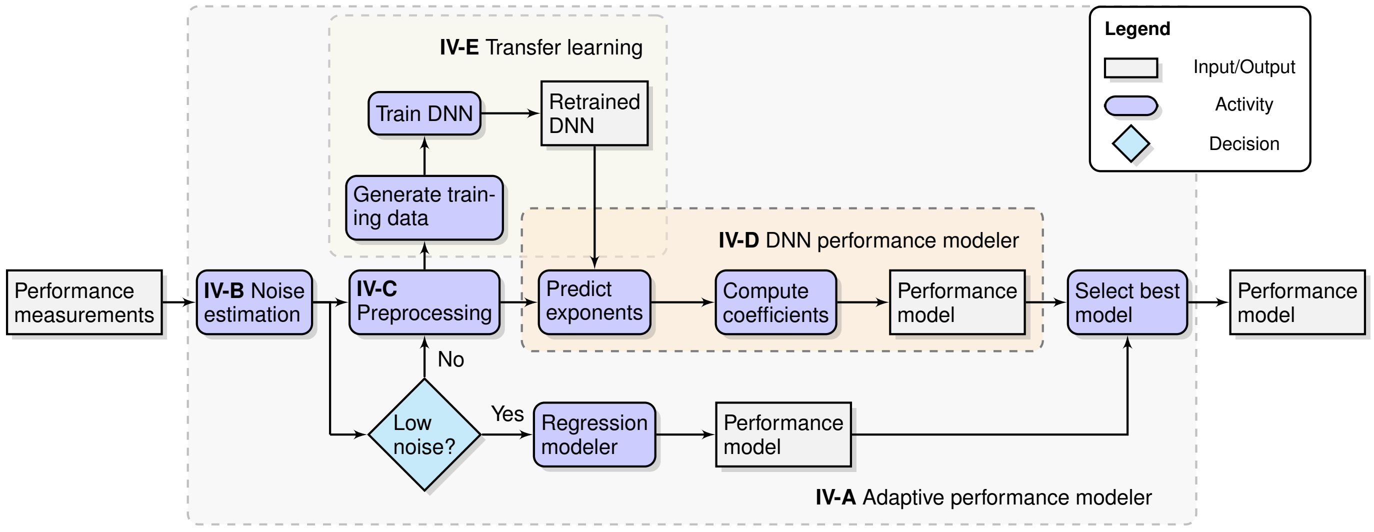


Fig. 1: Flowchart outlining the basic adaptive performance modeling process. The letters (IV-A) reference the paragraphs of the approach where the associated components of the adaptive modeler are described in detail.

module, and our old regression-based performance modeler, henceforth simply called regression modeler.

The first step of the modeling process is the noise estimation. Therefore, the noise module analyzes the performance measurements using our heuristic range-of-relative-deviation strategy and estimates the noise level.

Second, we need to decide when to use which modeling technique. While the DNN modeler produces much better results at high noise levels, the regression modeler achieves better results at low noise levels. Therefore, we combine both solutions, creating an adaptive approach that adjusts to the properties of the modeling task at hand. If the estimated noise level is smaller than a certain threshold, the regression modeler is used in addition to the DNN modeler. Since the threshold for selecting the modelers depends on several factors, such as the number of parameters and the modeling task, we conducted an in-depth analysis of the relationship between noise and model accuracy of both approaches. Using the results of this analysis we identified the intersections between their accuracy functions to determine the correct thresholds for switching off the regression modeler.

In the third step of the modeling process, the input data is preprocessed. Since performance measurements differ for each modeling task, the input data must be normalized and converted into a format that can be understood by the neural network.

Fourth, depending on the estimated amount of noise either only the DNN or both of the approaches are used for modeling. In scenarios with high levels of noise, it is necessary to switch off the regression-based modeler to further improve predictive power. Whereas the DNN attempts to improve extrapolation, the regression modeler focuses on optimally fitting the model to the measurement data, leading to inaccurate predictions outside of the measuring range. In each case, a new training data set is generated using the previously gathered noise and measurement information. We then use domain adaptation to optimize the pretrained generic neural network for the specific properties of the modeling task at hand. Subsequently, the retrained network is employed to predict the exponents of the performance function, followed by the determination of the coefficients using linear regression and the available measurement data.

Finally, we select the best performance model by evaluating the results of both modeling approaches against each other. Using cross-validation and the SMAPE metric we identify the model that fits the data best, thus receiving the final outcome of the adaptive performance modeling process. In case only the DNN modeler was used this step is obsolete, as its output already represents the best model that could be found.

B. Noise estimation

In general, performance measurements can be regarded as a statistical process, however, identifying the specific distribution of performance variations/noise is quite challenging, as it depends on the execution environment and the choice of execution parameters. Since we repeat each measurement at most five times [3], we cannot determine the distribution in practice. Therefore, we follow the principle of indifference and rely on the simplifying assumption that the noise distribution is uniform, a good enough approximation of any non-uniform distribution, considering the small number of samples we have.

To determine the amount of noise, i.e., the noise level, we view the measurements as a sequence of M independent uniformly distributed random variables V_P for $1 \leq P \leq M$, one per experiment. For each measurement point P we repeat the measurement rep times, resulting in the values $v_{P,s}$ for $1 \leq s \leq rep$, which lie within the noise-level around the actual value. Furthermore, we define $\bar{v}_P$ as the mean of all samples $v_{P,s}$ from V_P. We use the relative deviation (Equation 3) of the variables $v_{P,s}$ to measure the level of noise. We group all relative deviations into a set D_V and derive the range of relative deviation heuristic shown in Equation 4.

$$\mathrm{rd}(v_{P,s}) = \frac{v_{P,s} - \bar{v}_P}{\bar{v}_P} \tag{3}$$

$$\mathrm{rrd}(D_V) = \max(D_V) - \min(D_V) \tag{4}$$

The idea of the heuristic, which is directly based on the relative deviation, is to counteract the common problem that the relative deviations of the measurements for a specific measurement point does not span across the entire noise range. Since the mean value often does not correspond to the actual value, this leads to an off-center shift of the relative deviation compared to the actual value. Furthermore, not all relative deviations are shifted to the same side or by the same amount. Consequently, the combined relative deviation is much closer to the actual noise level than any relative deviation of the corresponding measurement point alone. Using the range of relative deviation we can estimate the level of noise in a set of performance measurements with an average prediction error of only 4.93%.

C. Preprocessing

Before we can use the performance measurements as an input to the DNN modeler, we first have to preprocess the raw measurement data, consisting of the tuples (P, v), each composed of one measurement point $P(x_1, x_2, \ldots, x_m)$, which is defined by the set of application execution parameters considered for the performance analysis, plus the measured value v, a performance metric such as runtime. First, we have to solve the problem of varying measurement points. Even though most applications' execution parameters are fairly similar, usually describing the number of processes and the size of the simulated domain, their values can vary greatly. In an exemplary scenario, an application could be run with a different number of processes x_1, where $\mathbf{x}_1 = (32, 64, 128, 512, 1024)$ represents a set of possible parameter values for x_1, resulting in the measurement points $P(\mathbf{x}_1)$ which are: $P(32)$, $P(64)$, $P(128)$, $P(512)$, and $P(1024)$. However, another application might require the number of processes to be scaled cubically, as it is the case for Kripke [34], e.g., $\mathbf{x}_1 = (8, 64, 512, 4096, 32\,768)$, resulting in the measurement points $P(\mathbf{x}_1)$: $P(8)$, $P(64)$, $P(512)$, $P(4096)$, and $P(32\,768)$. The same applies, of course, to measurement points with several parameters $P(x_1, \ldots, x_m)$ such as $P(\mathbf{x}_1, \mathbf{x}_2)$, where $\mathbf{x}_1 = (8, 16)$ and $\mathbf{x}_2 = (10, 20)$, resulting in the measurement points: $P(8, 10)$, $P(8, 20)$, $P(16, 10)$, and $P(16, 20)$. To solve the problem of varying measurement points, we enrich the values v for each tuple (P, v) with an implicit position information by dividing them by the elements of their respective measurement point P, resulting in the tuples $(P, \hat{\mathbf{v}})$, $\hat{\mathbf{v}} = v \oslash P = (v/x_1, \ldots, v/x_m)$. This information helps the network to adjust to application-specific characteristics.

Second, we have to solve the problem of a variable number of measurement points. System features such as memory size or the core count often restrict the choice of an applications' execution parameters, also affecting the number of experiments that can be conducted per parameter. This leads to a varying number of measurement points that are available for modeling. To solve this problem, we simply limit the number of inputs for the neural network to the interval $[5, 11] \cap \mathbb{N}$, masking unused inputs with zero. As previously described in Section III, we need at least five different values per execution parameter where all other parameter values are constant. Furthermore, we define the maximum number of values per parameter to be eleven. In practice, this is more than enough, as often seven values are already too expensive to be measured. For Kripke [34], e.g., one would require a system with more than $2\,097\,152$ processes to conduct measurements with seven different parameter values.

Finally, we have to map the tuple $(P, \hat{\mathbf{v}})$ of measured values for each measurement point to the input layer of the neural network. Since the number of possible measurement points is unbounded, they cannot be mapped one-to-one to the network's input layer. Instead, we dynamically group the tuples $(P, \hat{\mathbf{v}})$ by their positions $P(x_l)$ and map the value $\hat{v}_l$ of each group to one input neuron, such that one neuron represents an entire range of possible measurement points, rather than only one specific position. To group the input values, we first normalize the positions $P(x_l)$ of each measurement to the fixed interval $[0, 1]$, so that the position information becomes independent of the range and scale of the measurement sequence. Next, we sample the measurements at the following normalized positions $(\frac{1}{64}, \frac{1}{32}, \frac{1}{16}, \frac{1}{8}, \frac{2}{8}, \frac{3}{8}, \frac{4}{8}, \frac{5}{8}, \frac{6}{8}, \frac{7}{8}, 1)$ using a process similar to nearest-neighbor interpolation, however, each value can be sampled only once. Each sampling position maps directly to one input neuron that represents one group and gets the measurement value $\hat{v}_l$ of that group as input.

D. DNN performance modeler

In order to use a DNN to predict the performance behavior of a parallel application, we formulate the problem of identifying a performance function for an application kernel, based on empirical measurement data, as a standard classification task from the domain of machine learning. For each parameter that needs to be modeled the resulting function has exactly one term $c_k \cdot x_l^{i_{kl}} \cdot log_2^{j_{kl}}(x_l)$. To create a performance model, we need to identify the correct exponents and coefficients for each of these terms. Therefore, we use a neural network to predict the exponents (all at once) of each term using classification. Based on the function search space, which is defined by the PMNF and the set of exponents E (see Equation 1 and 2), there is only a limited number of possible exponent combinations, limiting the number of classes the DNN has to predict to 43.

To create a performance model, first, the exponents for each function term are classified. Subsequently, the network's top three classification results are used to construct a set of performance hypotheses based on the PMNF, which is straightforward as the output of the network is the probability distribution of the classes. Next, the function's coefficients are determined using regression. Finally, we choose the best hypothesis using cross-validation and the SMAPE metric.

In order to develop multi-parameter models, we first model the effects of each parameter on performance separately. We then create a new multi-parameter search space by combining the single-parameter hypotheses with each other, testing all possible additive and multiplicative combinations. Finally, we choose the best model, again using cross-validation and the SMAPE metric.

One may wonder why we do not use a machine learning-based regression approach instead of the classification. This is because a regression approach would need to predict several dependent target variables at the same time. Furthermore, this would lead to a tremendous increase of the exponent search space, since they can no longer simply be selected from a predefined set. Thus, solving such a multi-target regression problem would be much more complicated than the classification problem favored by us. For the classification, we use a feed-forward deep neural network, with an input layer (11 neurons, one per available measurement points), five hidden layers (2×1500, 750, 2×250 neurons), and one output layer, all dense. The hidden layers use the hyperbolic tangent as an activation function, while the output layer has 43 neurons (as many as target classes) and uses a softmax function to predict the 43 classes, i.e., possible exponent combinations.

We train the neural network (using AdaMax) with synthetic data by instantiating the PMNF from Equation 1 with random exponents from Equation 2 and coefficients uniformly sampled from the interval $[0.001, 1000]$. Additionally, we add a term ν that represents a random noise level on the measurements masking performance behavior. The amount of noise n is generated in the interval $n \in [0, 100\%]$ using the uniform distribution, where $n = 10\%$ equals a deviation of $\pm5\%$ from the actual value. After creating a set of synthetic performance functions of the form $f_1(x_1) = c_0 + c_1 \cdot x_1^i \log_2^j(x_1) + \nu$, we generate a random set of measurement points for each of them. To imitate realistic application scenarios and parameters, we generate random sequences of measurement points that are either linear, small linear, small exponential, or uniformly distributed. Possible parameter-value sets for the measurement points are for example: $\mathbf{x}_1 = (4, 8, 16, 32, 64)$; $\mathbf{x}_1 = (10, 20, 30, 40, 50)$; or $\mathbf{x}_1 = (8, 64, 512, 4096, 32\,768)$. Finally, we simulate the repetition of experiments, a commonly used method to reduce the amount of noise in the conducted performance measurements by sampling up to five values per measurement point and compute their median value.

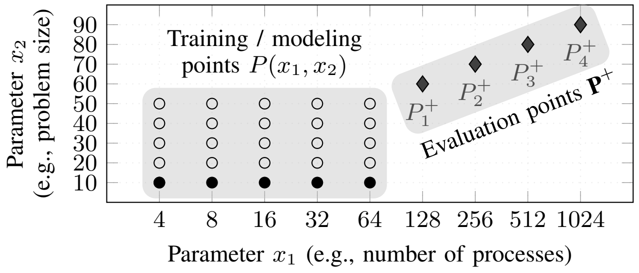
E. Transfer learning

As previously described, we train our DNN with synthetic data, imitating realistic application scenarios. However, due to the complexity of the classification task, which is determined by the number of classes, the varying measurement points, their variable number, and the noise level, it is still difficult to achieve optimal results for specific modeling tasks. Ideally, one would train a network designed to tackle one specific modeling problem. However, this would not only be very complicated, but also costly and time-consuming. Therefore, we apply domain adaptation [35], a specialized form of transfer learning, to optimize a pretrained neural network for the modeling task at hand, enabling our DNN to create good performance models even in the presence of high amounts of noise without requiring any user input. Transfer learning describes the idea of using knowledge learned from solving one problem for solving a different but related problem [36]. For example knowledge that was acquired from modeling the performance of an application could be useful for learning how to model another application. Domain adaptation, is a more specialized form of transfer learning, describing the ability to apply the same algorithm trained in one or more source domains to a different but related domain [35]. These domains always have the same feature space but different distributions. For our modeling example, this means that every modeling task has parameters, parameter values, measurement points, etc., but their values are usually always different, i.e., the learning task itself is not changing, though the networks input values are. Domain adaptation is especially useful to us, as we have more training data available for the first training task, pretraining the DNN for a variety of modeling scenarios. Thus, a few examples are enough to quickly learn a generalizing representation for the second training task, retraining the DNN for the modeling problem at hand.

To apply the domain adaptation to our approach, we generate a new synthetic training data set based on the properties of the modeling task at hand. This includes among others, the amount of noise in the performance measurements, the number of parameters, the measurement points, and their quantity. Then we simply retrain our DNN using the created data set to learn a generalizing representation for this specific modeling task. Thus, our approach automatically learns to adapt to all kinds of modeling scenarios. In order to achieve the best possible results we always use domain adaptation before modeling. Concretely, this means that we retrain our DNN for each modeling task, the downside being an overall increased modeling time caused by the time required for retraining the DNN. Although the increase in modeling time is quite significant, it is negligible compared to the gain in accuracy and predictive power. A detailed discussion of the computational overhead of the adaptive modeling technique, as well as a detailed explanation of how we use domain adaptation to optimize the model quality for a real-life application can be found in the application case studies in Section VI.

V. Synthetic Evaluation

To evaluate our novel adaptive modeling technique in comparison with the regression modeler, we conduct an extensive synthetic data analysis. Therefore, we analyze both modeling techniques in terms of their two key aspects: model accuracy and predictive power. We define model accuracy as the percentage of correct performance models for the test data set. It can be easily calculated by dividing the number of correct models by the number of total modeling tasks in the test set. To determine whether a model is correct, we compare it with its synthetic baseline and examine if the distance d between the lead exponents (the exponents with the biggest



Training / modeling points $P(x_1, x_2)$
Evaluation points $\mathbf{P}^+$
P_1^+
P_2^+
P_3^+
P_4^+
Parameter x_2 (e.g., problem size)
Parameter x_1 (e.g., number of processes)
90
80
70
60
50
40
30
20
10
4
8
16
32
64
128
256
512
1024

Fig. 2: Performance experiments required for one and two-parameter analysis. The solid black circles represent a list of measurement points $P(x_1)$ using the parameter-value set $\mathbf{x}_1$ that would be sufficient to generate a single-parameter model for parameter x_1. All circles represent the measurement points required for a two-parameter model describing how x_1 and x_2 affect performance. The solid diamonds are the measurement points reserved for evaluation.

overall impact on performance) is smaller as or equal to $\frac{1}{4}$, $\frac{1}{3}$, or $\frac{1}{2}$. This means for example that if the lead exponent distance of a model is smaller or equal to $\frac{1}{4}$ we account it as correct for this accuracy bucket. In general, the smaller the lead exponent distance of a model, the more accurate it describes the performance behavior found in the measurement data. We define the predictive power of a model as the extrapolation accuracy for a measurement point that lies outside the range used for modeling (see Figure 2). To determine the extrapolation accuracy we compare the prediction result of the created model with its synthetic baseline and calculate the percentage error. We then calculate the median percentage error over all models created for the synthetic test data set using four different evaluation points $\mathbf{P}^+$ that have not been used for modeling. The smaller the median percentage error, the higher is the extrapolation accuracy of the created models. Thus, we can determine and compare the predictive power of both modeling approaches.

For each evaluation we generate 100 000 test functions by instantiating the PMNF from Equation 1 with uniform independent and identically distributed random coefficients $c_k \in [0.001, 1000]$ and random exponents i_{kl} and j_{kl}, selected from Equation 2. We then evaluate both modeling techniques for each function in the resulting test-set for 5^m measurement points, where $m = 1, 2, 3$ is the number of parameters. Since in practice most applications do not have more than three parameters influencing performance, for this analysis we focus on scenarios where $m \leq 3$. Additionally, we simulate the repetition of experiments by sampling five values v per measurement point P and compute their median value. The parameter-value sets $\mathbf{x}_1, \ldots, \mathbf{x}_m$ for the measurement points $P(x_1, \ldots, x_m)$ for each parameter x_l are drawn from a variety of different series (e.g., $\mathbf{x}_1 = (4, 8, 16, 32, 64)$ or $\mathbf{x}_2 = (10, 20, 30, 40, 50)$) to simulate different types of application parameters. Furthermore, we generate four additional measurement points $\mathbf{P}^+ = (P_1^+, \ldots, P_4^+)$, which are not included in this training data set (see Figure 2), to evaluate the predictive power of the created models. The coordinates for these are selected by continuing each sequence $\mathbf{x}_l$ (e.g., $\mathbf{x}_1^+ = (128, 256, 512, 1024)$). In order to analyze the noise resilience of our new approach, we apply a wide range of different noise levels $n = 2\%, 5\%, 10\%, 20\%, 50\%, 75\%, 100\%$ using the uniform distribution, where $n = 2\%$ is equal to up to $\pm 1\%$ of divergence from the actual value.

A. Model accuracy

One parameter: Figure 3(a) shows that both modeling approaches reach a very high accuracy for low noise levels $n \leq 10\%$, with more than 95% of the models being correct. With an increasing amount of noise $n \geq 10\%$ the adaptive modeler begins to outperform the regression-based approach, improving the accuracy by up to 22% at 100% of noise for $d \leq \frac{1}{4}$. Even though it becomes more and more difficult to identify the correct lead exponents, the adaptive modeler significantly improves the number of correct models for all accuracy buckets.

Two parameters: Again both modeling approaches reach a very high accuracy for low levels of noise $n \leq 10\%$, with more than 90% of the models being correct. As expected, when dealing with one additional parameter, the results are overall slightly less accurate than before, which also leads to an increase in the differences between the accuracy buckets. Nevertheless, as shown by Figure 3(b), the accuracy improvement achieved by the adaptive modeler has increased to 25% at 100% of noise for $d \leq \frac{1}{4}$.

Three parameters: With the addition of another parameter the modeling accuracy is decreasing again. As shown in Figure 3(c) it becomes much harder to predict the lead exponents for high levels of noise $n \geq 10\%$, especially within $d \leq \frac{1}{4}$. Overall the results of the adaptive modeler are much more consistent and trustworthy, as it predicts over 65% of the models correctly within $d \leq \frac{1}{2}$ over the entire range of noise.

The 99% confidence intervals of all percentages of correct model data deviate at most 2% (in absolute terms) from the reported model accuracy values.

B. Predictive power

One parameter: The results of the single-parameter analysis in Figure 3(d) show that for low noise levels $n \leq 10\%$, the prediction error of both approaches is very low, ranging between 0.17% and 1.31%. Consequently, the performance predictions for the evaluation points $\mathbf{P}^+$ are all very accurate. For high levels of noise $n \geq 20\%$, the adaptive modeler clearly outperforms the regression-based approach at the evaluation points P_2^+, P_3^+ and P_4^+ that are farther away from the training data set. For 50% of noise, the adaptive modeler reduces the prediction error from 18.06% to only 7.2% for P_4^+ by about a factor of two. This trend continues for the following noise levels and evaluation points. The 99% confidence intervals for the median relative errors for $m = 1$ are within 5% of relative deviation from the reported predictive power.

Two parameters: The results of the evaluation in Figure 3(e) show that the prediction error of both modeling approaches is

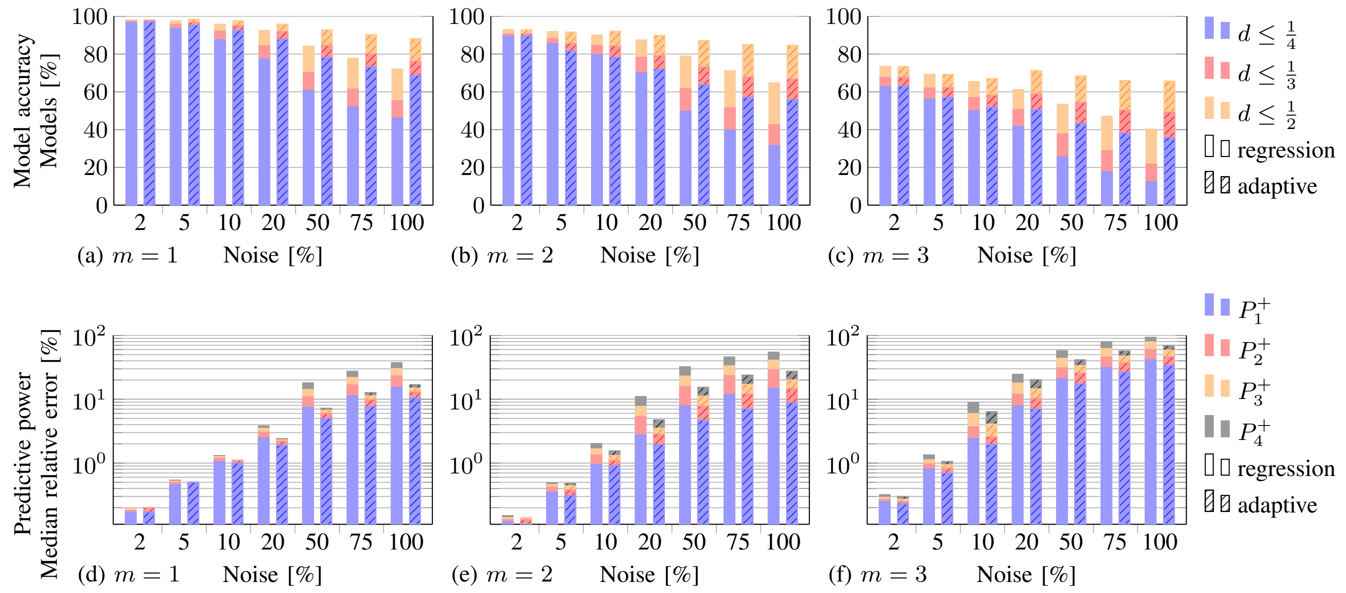


Fig. 3: Comparison of the model accuracy and predictive power of the regression and adaptive modeler for different numbers of parameters $m = 1, 2, 3$ and noise levels $n = 2\%, 5\%, 10\%, 20\%, 50\%, 75\%, 100\%$. The Figures (a), (b), and (c) outline the percentage of models where the distance of the lead exponents d is smaller as or equal to $\frac{1}{4}$, $\frac{1}{3}$, or $\frac{1}{2}$ when compared to the synthetic baseline. The smaller the distance, the more accurate the performance model. The Figures (d), (e), and (f) show the median relative prediction error in percent for the four evaluation points $\mathbf{P}^+$ when compared to the synthetic baseline. The higher the error the lower the predictive power of the models.

still very low for small amounts of noise, ranging between 1.55% and 2.02% for P_4^+. For noise levels $n \geq 10\%$ the adaptive modeler performs increasingly better than the regression-based approach reducing the prediction error for the evaluation point P_4^+ from 32.2% to 15.13% for $n = 50\%$, from 45.9% to 24.43% for $n = 75\%$, and from 54.6% to 28.1% for $n = 100\%$. Again this corresponds to an overall reduction by about a factor of two. The 99% confidence intervals of the reported median relative errors for $m = 2$ deviate at most 12.5%, for P_1^+ and P_2^+ they are within 7.4% of relative deviation from the reported predictive power.

Three parameters: For three parameters the prediction error is, as expected, generally higher than before. Furthermore, the results in Figure 3(f) show that with each additional parameter it becomes increasingly difficult to predict the performance for each evaluation point. While analyzing the results one has to consider that the measurement points chosen for evaluation are scaled multiple times over three dimensions (depending on the number of parameters), and are therefore far away from the training data set. Consequently, as shown in Figure 2, predicting the performance for P_4^+ is a much more difficult task than for P_1^+. Having said this, for low levels of noise $n = 10\%$ the prediction error is still quite small, reaching between 6.36% and 8.93%. For high levels of noise the prediction error of both techniques rises quickly, though compared to the regression-based approach the adaptive modeler reduces the prediction error from 57.46% to 41.39% for $n = 50\%$, from 79.31% to 57.18% for $n = 75\%$, and from 93.18% to 69.46% for $n = 100\%$ of noise for P_4^+. The 99% confidence intervals of the reported median relative errors for $m = 3$ are within 5% of relative deviation from the reported predictive power.

VI. Application Case Studies

In the following section, we present three application case studies, Kripke, FASTEST, and RELeARN, that highlight the advantages and disadvantages of our solution. Similar to the synthetic evaluation, we analyze both modeling approaches in terms of their model accuracy and predictive power. Though, this time we define accuracy as a grade of closeness for specific models when comparing to a theoretical baseline. For the predictive power, we still apply the same definition as in Section V, however, using only one evaluation point per application. Furthermore, we use our heuristic strategy and analyze the amount of noise in the performance measurements to better understand how the noise level affects the predictive power of the models. Finally, we examine the computational overhead of the adaptive modeler by comparing the amount of time both approaches need to model the main kernels of each application. Since the application or kernel runtime is particularly affected by noise, for the entire analysis we focus on modeling this performance metric.

However, before we analyze the performance of our novel modeling technique, we first provide a detailed explanation of how we use domain adaptation to improve model quality in a real-life scenario using Kripke as an example.

As previously introduced, the following notation $P(x_1, \ldots, x_m)$ for a measurement point is subsequently used to describe the points that are either used for the model creation or evaluation of the application case studies. Here, each x_l describes a parameter of the respective case study and $\mathbf{x}_l$ represents a set of values for a specific parameter x_l, that is, the parameter values used for the experiments of a case study. Furthermore, the use of $\mathbf{x}_l$ in combination with a measurement point, for example $P(\mathbf{x}_1, x_2)$, indicates that the parameter values from the set $\mathbf{x}_1$ of parameter x_1 are used to represent several measurement points at once.

Kripke is an open-source 3D Sn deterministic particle transport code, designed to explore how different data layouts, programming paradigms, and architectures affect the implementation and performance of discrete-ordinates transport [34]. The performance measurements were conducted on Vulcan, an IBM BG/Q system at Lawrence Livermore National Laboratory, covering three execution parameters: the number of processes x_1, the number of direction-sets x_2, and the number of energy groups x_3, the latter both influencing the problem size. In total, we have done 750 experiments with 150 measurement points and five repetitions each. To collect these measurements, we varied the parameter values of $\mathbf{x}_1 = (8, 64, 512, 4096, 32\,768)$, $\mathbf{x}_2 = (2, 4, 6, 8, 10, 12)$, and $\mathbf{x}_3 = (32, 64, 96, 128, 160)$. For modeling, we use all experiments except for the ones with $x_2 = 12$ (i.e., 625 in total), while for evaluation we use the measurement point $P^+(x_1 = 32\,768, x_2 = 12, x_3 = 160)$.

FASTEST is a tool for the simulation of flows in complex three dimensional configurations [37]. We measured its performance on SuperMUC, a petascale system at Leibniz Supercomputing Centre, covering two execution parameters: the number of processes x_1 and the problem size per process x_2. We measured its performance by varying $\mathbf{x}_1 = (16, 32, 64, 128, 256, 512, 1024, 2048)$ and $\mathbf{x}_2 = (8192, 16\,384, 32\,768, 65\,536, 131\,072)$, repeating each measurement five times. To create the models, we use two lines of five measurement points for each parameter where the other parameter is constant. To model the effects of x_1 on performance, we use five measurement points $P(\mathbf{x}_1, x_2)$, where $\mathbf{x}_1 = (16, 32, 64, 128, 256)$ and $x_2 = 131\,072$. To model the effects of x_2 on performance, we use another five points $P(x_1, \mathbf{x}_2)$, where $x_1 = 256$ and $\mathbf{x}_2 = (8192, 16\,384, 32\,768, 65\,536, 131\,072)$. As these two lines of points overlap each other at $P(x_1 = 256, x_2 = 131\,072)$ in total we use nine measurement points for modeling. To evaluate our model we use the measurement point $P^+(x_1 = 2048, x_2 = 8192)$.

RELeARN simulates the rewiring of connections between neurons in the brain [38]. We measured its performance on Lichtenberg, a compute cluster at Technical University of Darmstadt, considering two execution parameters: the number of processes x_1 and the problem size x_2, represented by the number of neurons to be simulated. By varying $\mathbf{x}_1 = (32, 64, 128, 256, 512)$ and $\mathbf{x}_2 = (5000, 6000, 7000, 8000, 9000)$ we measured its performance for 25 different configurations with two repetitions each. For modeling, we again use two lines of five measurement points per parameter where the other parameter is constant. In total, we use nine measurement points for modeling, these are: $P(\mathbf{x}_1, x_2)$, where $\mathbf{x}_1 = (32, 64, 128, 256, 512)$ and $x_2 = 5000$ for x_1, and $P(x_1, \mathbf{x}_2)$, where $x_1 = 32$ and $\mathbf{x}_2 = (5000, 6000, 7000, 8000, 9000)$ for x_2. This time the measurement points overlap at $P(x_1 = 32, x_2 = 5000)$. Finally, to evaluate the created models, we use the measurement point $P^+(x_1 = 512, x_2 = 9000)$.

A. *Transfer learning*

As described in the approach in Section IV our DNN modeler is pretrained using a variety of different realistic application scenarios. To optimize the model accuracy and predictive power of a specific modeling task or application even further, we use domain adaptation to retrain the neural network, which is used to identify the correct function terms and their exponents. To allow a better understanding of this procedure, we provide a detailed explanation using Kripke as an example.

The first important step prior to the retraining procedure of the network using domain adaptation is the estimation of the noise level affecting the performance measurements using our range-of-relative-deviation heuristic. Then we preprocess the input data, which means we analyze the measurements to obtain all information that is important for retraining and modeling. This includes the number of parameters, the parameter-value sets, the measurement points and their measured values. In the case of Kripke we identified a mean noise level of 17.44 % on the performance measurements. Furthermore, we have three configuration parameters x_1, x_2, x_3 with the parameter-value sets $\mathbf{x}_1 = (8, 64, 512, 4096, 32\,768)$, $\mathbf{x}_2 = (2, 4, 6, 8, 10, 12)$, and $\mathbf{x}_3 = (32, 64, 96, 128, 160)$, representing 150 measurement points. Using this information we create a new data set for retraining the network based on the specific properties of Kripke, i.e., we generate a fixed amount of synthetic training samples per class that can be predicted by the DNN. Therefore, we use the same number of parameters, measurement points, and parameter-value sets as in our Kripke experiments. When generating the synthetic measurement values we additionally add a random noise level based on the range identified in the Kripke measurements, here $[3.66, 53.67]\%$. Depending on the modeling task at hand we also simulate measurement repetitions. In this case, we simulate up to five repetitions per measurement point. The resulting training data set is then used to retrain the previously trained generic network. The required retraining time depends on the number of training epochs and the size of the data set, determined by the number of samples per class. In general, the more epochs and the larger the sample size per class, the more accurate the results, but at the same time the retraining time and the computational overhead is increasing. Usually, we use one retraining epoch and a sample size of 2000 per class. Finally, the retrained network is used by the DNN modeler to predict the pair of exponents using the preprocessed input data.

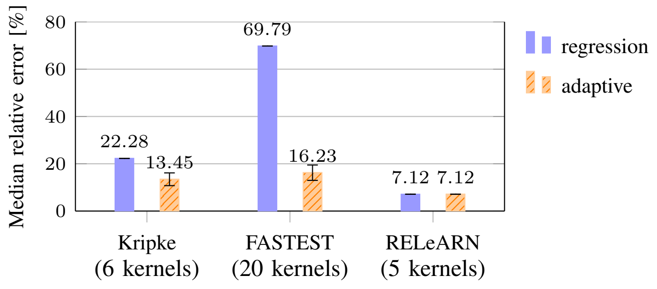


Fig. 4: Comparison of the median relative prediction error in percent between the regression and the adaptive modeler for modeling the performance-relevant kernels of the case studies. The error bars show the 99% confidence intervals of the reported median relative error.

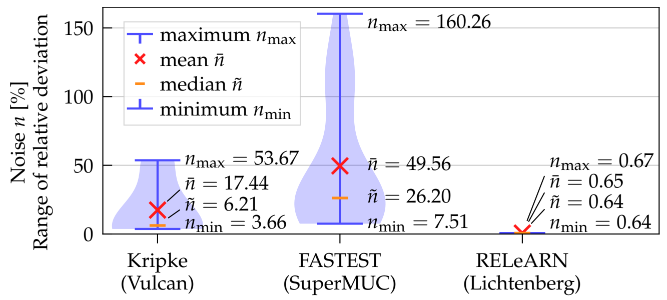


Fig. 5: Noise-level distributions of the performance measurements of the application case studies.

B. Model accuracy

For Kripke, the kernel `SweepSolver` is of special interest to us because it encapsulates the physics simulated by the application. As the name implies, it solves the simulated problem using an MPI-based parallel sweep algorithm [34]. Due to the nature of the algorithm, the performance of the Sn solver should be determined by the number of processes x_1, group sets x_3, zone sets (which we did not consider), and direction sets x_2 [34]. Therefore, we expect an approximate runtime behavior similar to $\mathcal{O}(x_2 \cdot x_3^{4/5} + x_1^{1/3} + x_3^{4/5})$ [31]. In fact, the model created by both of our approaches $8.51 + 0.11 \cdot x_1^{1/3} \cdot x_2 \cdot x_3^{4/5}$ is very similar to this theoretical expectation, the only difference being the missing additional term for $x_3^{4/5}$ and additive term combination.

For RELeARN we focus on analyzing the `connectivity update`, a function that updates the connections of the simulated neurons, dominating the asymptotic complexity of the computation. From the literature we expect an approximate runtime behavior of $\mathcal{O}(x_2 \log_2^2(x_2) + x_1)$ [38]. The result of the adaptive modeler $-2216.41 + 325.71 \cdot \log_2(x_1) + 0.01 \cdot x_2 \log_2^2(x_2)$ reflects this expectation almost one to one, the only minor inaccuracy being that it predicts $\log_2(x_1)$ instead of x_1. The result of the regression modeler $-284.32 + 5.42 \cdot \log_2^2(x_1) + 3.53 \cdot 10^{-3} \cdot x_2 \log_2(x_2)$ on the other hand is less accurate, predicting both functions terms incorrectly.

For FASTEST there are no theoretical performance expectations or analytical models, thus we can not analyze its model accuracy.

C. Predictive power

For analyzing the predictive power of both modeling approaches, we only consider the performance relevant kernels of each case study, meaning the ones that contribute more than one percent to the overall application runtime. This restriction is necessary as many small kernels show huge performance variance and therefore disproportionately negatively affect the median relative prediction error. Figure 4 shows the results of the prediction analysis. For Kripke, the regression modeler achieved an average median relative prediction error of 22.28% for all six performance relevant kernels. In comparison, the adaptive modeler's prediction error is only 13.45%, a reduction of 8.83%. In the case of FASTEST, the adaptive modeler reduces the prediction error by 53,56% from 69.79% to only 16.23% for the considered 20 kernels. Finally, for RELeARN both modelers produced the exact same result, a prediction error of 7.12%.

D. Noise analysis

Using our heuristic strategy we analyzed the amount of noise and its distribution on all performance measurements of Kripke, FASTEST, and RELeARN. We found that the noise levels for all applications are more or less uniformly distributed, but we cannot say with certainty which distribution really exists, since five measurements per-application configuration are too few samples to statistically prove a certain type of distribution. Figure 5 shows the distribution of the different noise levels on the performance measurements of the application case studies, indicating the average $\bar{n}$, median $\tilde{n}$, minimum $n_{\min}$, and maximum $n_{\max}$ noise levels in percent. For RELeARN we found only minimal amounts of noise in the measurements, ranging between $n_{\min} = 0.64\%$ and $n_{\max} = 0.67\%$. The absence of noise explains why the adaptive modeler was not able to achieve an improvement in prediction power compared to the regression modeler. In the case of Kripke, we measured an average noise level of $\bar{n} = 17.44\%$. Furthermore, the levels of noise are much more diverse, ranging between $n_{\min} = 3.66\%$ and $n_{\max} = 53.66\%$. However, high noise levels occur only rarely. For FASTEST, we found an even higher amount of noise in the measurements, ranging between $n_{\min} = 7.51\%$ and $n_{\max} = 160.27\%$. With an average noise level of $\bar{n} = 49.56\%$, FASTEST is most affected among all case studies and shows why the adaptive modeler was able to improve the predictive power of the models by 45%.

E. Computational overhead

As one might expect, the improved predictive power of the adaptive modeler does not come for free, but at the expense of an overall increased modeling time. This is owed to the retraining of the DNN whenever we apply domain adaptation to optimize the pretrained network for a specific modeling task. Figure 6 shows a comparison of the amount

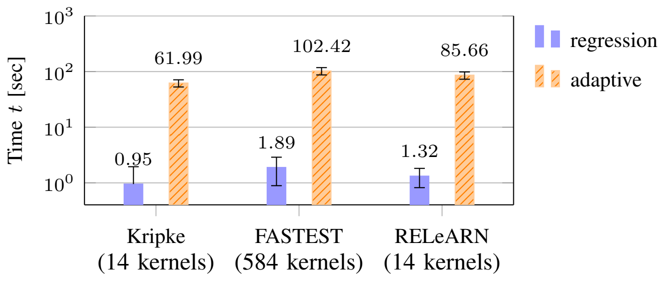


Fig. 6: Comparison of the amount of time t in seconds required by the regression and adaptive modeler for modeling the main kernels of the application case studies. The error bars show the 99% confidence intervals of the reported time.

of time t in seconds that is required by both approaches to model the main kernels of each case study. For Kripke the adaptive modeler is about 65 times slower than the regression-based approach, taking roughly 61.99 seconds to create all models. In the case of FASTEST, it performs slightly better, being only about 54 times slower. Finally, for RELeARN the adaptive modeler is again about 64 times slower, taking 85.66 seconds to model all kernels. As expected for a larger number of kernels, the overall modeling time for FASTEST is higher than for the other applications. Though, the number of modeled kernels actually has only a small influence on the total modeling time. In reality, most of the time required for modeling is spent on retraining the DNN, where the computational overhead for retraining is increasing/decreasing with the diversity of the noise levels found in the performance measurement. Therefore, the higher diversity of noise levels in the FASTEST measurements (see Figure 5) explains the overall increased modeling time. Considering that conducting the performance measurements required for modeling usually takes at least days, the shown overall increase in modeling time is negligible compared to the improved predictive power.

VII. Conclusion

Our adaptive approach can effectively reduce the impact of noise on the creation of empirical performance models that describe the effects of numerical execution parameters, such as the number of processes or the problem size, improving both their accuracy as well as their predictive power. While linear regression performs extremely well on data with low noise, the DNN shows significantly better results at high noise levels. With our heuristic noise estimator as the decision-maker, the adaptive modeler combines the best of both worlds, applying each method when its advantages are used best. The synthetic evaluation shows that our solution surpasses the regression-based approach's model accuracy by up to 25% while improving the predictive power by about 15%. In a real-life scenario, our solution performed even better, lowering the average extrapolation error for the CFD code FASTEST from more than 62% down to only 16%, thus increasing the predictive power of the created performance models by 46%.

References

[1] A. Calotoiu, T. Hoefler, M. Poke, and F. Wolf, "Using automated performance modeling to find scalability bugs in complex codes," in *SC '13: Proc. of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, paper 45.

[2] "Extra-P – automated performance-modeling tool," www.scalasca.org/software/extra-p.

[3] M. Ritter, A. Calotoiu, S. Rinke, T. Reimann, T. Hoefler, and F. Wolf, "Learning cost-effective sampling strategies for empirical performance modeling," in *Proc. of the 2020 IEEE 34th International Parallel and Distributed Processing Symposium*. IEEE, 2020, pp. 884–895.

[4] F. Petrini, D. J. Kerbyson, and S. Pakin, "The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q," in *SC '03: Proc. of the 2003 ACM/IEEE conference on Supercomputing*. ACM, 2003, paper 55.

[5] T. Hoefler, T. Schneider, and A. Lumsdaine, "Characterizing the influence of system noise on large-scale applications by simulation," in *SC '10: Proc. of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2010.

[6] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz, "Runtime measurements in the cloud: Observing, analyzing, and reducing variance," *Proc. of the VLDB Endowment*, vol. 3, pp. 460–471, 2010.

[7] T. Patki, J. J. Thiagarajan, A. Ayala, and T. Z. Islam, "Performance optimality or reproducibility: That is the question," in *SC '19: Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2019, paper 77.

[8] S. Chunduri, K. Harms, S. Parker, V. Morozov, S. Oshin, N. Cherukuri, and K. Kumaran, "Run-to-run variability on Xeon Phi based Cray XC systems," in *SC '17: Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2017, paper 52.

[9] S. F. Goldsmith, A. S. Aiken, and D. S. Wilkerson, "Measuring empirical computational complexity," in *Proc. of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. ACM, 2007, pp. 395–404.

[10] L. Carrington, A. Snavely, and N. Wolter, "A performance prediction framework for scientific applications," *Future Generation Computer Systems*, vol. 22, no. 3, pp. 336–346, February 2006.

[11] N. R. Tallent and A. Hoisie, "Palm: Easing the burden of analytical performance modeling," in *ICS '14: Proc. of the 28th ACM international conference on Supercomputing*. ACM, 2014, pp. 221–230.

[12] K. L. Spafford and J. S. Vetter, "Aspen: A domain specific language for performance modeling," in *SC '12: Proc. of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012, paper 84.

[13] R. Vuduc, J. W. Demmel, and J. A. Bilmes, "Statistical models for empirical search-based performance tuning," *International Journal of High Performance Computing Applications*, vol. 18, no. 1, pp. 65–94, February 2004.

[14] A. Bhattacharyya, G. Kwasniewski, and T. Hoefler, "Using compiler techniques to improve automatic performance modeling," in *PACT '15: Proc. of the 2015 International Conference on Parallel Architecture and Compilation*. ACM, 2015, pp. 468–479.

[15] D. Didona, F. Quaglia, P. Romano, and E. Torre, "Enhancing performance prediction robustness by combining analytical modeling and machine learning," in *ICPE '15: Proc. of the 6th ACM/SPEC International Conference on Performance Engineering*. ACM, 2015, pp. 145–156.

[16] B. C. Lee, D. M. Brooks, B. R. de Supinski, M. Schulz, K. Singh, and S. A. McKee, "Methods of inference and learning for performance modeling of parallel applications," in *PPoPP '07: Proc. of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2007, pp. 249–258.

[17] F. Nadeem, D. Alghazzawi, A. Mashat, K. Fakeeh, A. Almalaise, and H. Hagras, "Modeling and predicting execution time of scientific workflows in the grid using radial basis function neural network," *Cluster Computing*, vol. 20, pp. 2805–2819, 2017.

[18] R. Neill, A. Drebes, and A. Pop, "Automated analysis of task-parallel execution behavior via artificial neural networks," in *Proc. of the 2018 IEEE International Parallel and Distributed Processing Symposium Workshops*. IEEE, 2018, pp. 647–656.

[19] J. J. Thiagarajan, R. Anirudh, B. Kailkhura, N. Jain, T. Islam, A. Bhatele, J.-S. Yeom, and T. Gamblin, "PADDLE: Performance analysis using a data-driven learning environment," in *Proc. of the 2018 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2018, pp. 784–793.

[20] P. Neumann, "Sparse grid regression for performance prediction using high-dimensional run time data," in *Euro-Par 2019: Parallel Processing Workshops*, ser. Lecture Notes in Computer Science. Springer, 2020, pp. 601–612.

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

[22] D. Duplyakin, J. Brown, and R. Ricci, "Active learning in performance analysis," in *Proc. of the 2016 IEEE International Conference on Cluster Computing*. IEEE, 2016, pp. 182–191.

[23] E. Ipek, B. R. De Supinski, M. Schulz, and S. A. McKee, "An approach to performance prediction for parallel applications," in *European Conference on Parallel Processing*. Springer, 2005, pp. 196–205.

[24] B. C. Lee, D. M. Brooks, B. R. de Supinski, M. Schulz, K. Singh, and S. A. McKee, "Methods of inference and learning for performance modeling of parallel applications," in *PPoPP '07: Proc. of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2007, pp. 249–258.

[25] K. B. Ferreira, P. Bridges, and R. Brightwell, "Characterizing application sensitivity to OS interference using kernel-level noise injection," in *SC'08: Proc. of the 2008 ACM/IEEE Conference on Supercomputing*. IEEE, 2008.

[26] A. Maricq, D. Duplyakin, I. Jimenez, C. Maltzahn, R. Stutsman, and R. Ricci, "Taming performance variability," in *OSDI'18: Proc. of the 13th USENIX conference on Operating Systems Design and Implementation*. USENIX Association, 2018, pp. 409–425.

[27] J. Gillberg, P. Marttinen, M. Pirinen, A. J. Kangas, P. Soininen, M. Ali, A. S. Havulinna, M.-R. Järvelin, M. Ala-Korpela, and S. Kaski, "Multiple output regression with latent noise," *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 4170–4204, 2016.

[28] N. Siegmund, A. Grebhahn, S. Apel, and C. Kästner, "Performance-influence models for highly configurable systems," in *ESEC/FSE 2015: Proc. of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 284–294.

[29] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[30] A. McHutchon and C. E. Rasmussen, "Gaussian process training with input noise," in *NIPS'11: Proc. of the 24th International Conference on Neural Information Processing Systems*. Red Hook, NY, USA: Curran Associates Inc., 2011, pp. 1341–1349.

[31] A. Calotoiu, D. Beckinsale, C. W. Earl, T. Hoefler, I. Karlin, M. Schulz, and F. Wolf, "Fast multi-parameter performance modeling," in *Proc. of the 2016 IEEE International Conference on Cluster Computing*. IEEE, 2016, pp. 172–181.

[32] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[33] M. L. Seltzer, D. Yu, and Y. Wang, "An investigation of deep neural networks for noise robust speech recognition," in *Proc. of the 2013 IEEE international conference on acoustics, speech and signal processing*. IEEE, 2013, pp. 7398–7402.

[34] A. J. Kunen, T. S. Bailey, and P. N. Brown, "Kripke-a massively parallel transport mini-app," Lawrence Livermore National Lab (LLNL), Livermore, CA (United States), Tech. Rep., 2015.

[35] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*, ser. Adaptive computation and machine learning. MIT Press, 2016.

[36] L. Torrey and J. Shavlik, "Transfer learning," in *Handbook of research on machine learning applications and trends: algorithms, methods, and techniques*. IGI global, 2010, pp. 242–264.

[37] M. Kornhaas, M. Schäfer, and D. Sternel, "Efficient numerical simulation of aeroacoustics for low mach number flows interacting with structures," *Computational Mechanics*, vol. 55, pp. 1143–1154, 2015.

[38] S. Rinke, M. Butz-Ostendorf, M.-A. Hermanns, M. Naveau, and F. Wolf, "A scalable algorithm for simulating the structural plasticity of the brain," *Journal of Parallel and Distributed Computing*, vol. 120, pp. 251–266, 2018.