Performance Modeling under Resource Constraints Using Deep Transfer Learning

Aniruddha Marathe
Lawrence Livermore National Laboratory
marathe1@llnl.gov

Rushil Anirudh
Lawrence Livermore National Laboratory
anirudh1@llnl.gov

Nikhil Jain
Lawrence Livermore National Laboratory
nikhil@llnl.gov

Abhinav Bhatele
Lawrence Livermore National Laboratory
bhatele@llnl.gov

Jayaraman Thiagarajan
Lawrence Livermore National Laboratory
jayaramanthi1@llnl.gov

Bhavya Kailkhura
Lawrence Livermore National Laboratory
kailkhura1@llnl.gov

Jae-Seung Yeom
Lawrence Livermore National Laboratory
yeom2@llnl.gov

Barry Rountree
Lawrence Livermore National Laboratory
rountree4@llnl.gov

Todd Gamblin
Lawrence Livermore National Laboratory
tgamblin@llnl.gov

ABSTRACT

Tuning application parameters for optimal performance is a challenging combinatorial problem. Hence, techniques for modeling the functional relationships between various input features in the parameter space and application performance are important. We show that simple statistical inference techniques are inadequate to capture these relationships. Even with more complex ensembles of models, the minimum coverage of the parameter space required via experimental observations is still quite large. We propose a deep learning based approach that can combine information from exhaustive observations collected at a smaller scale with limited observations collected at a larger target scale. The proposed approach is able to accurately predict performance in the regimes of interest to performance analysts while outperforming many traditional techniques. In particular, our approach can identify the best performing configurations even when trained using as few as 1% of observations at the target scale.

CCS CONCEPTS

• **General and reference → Performance**; • **Computing methodologies → Transfer learning**; *Model development and analysis*; • **Hardware →** *Power estimation and optimization*;

KEYWORDS

performance prediction, parameter selection, transfer learning, deep learning

SC17, Denver, CO, USA

DOI: 10.1145/3126908.3126969

ACM Reference format:
Aniruddha Marathe, Rushil Anirudh, Nikhil Jain, Abhinav Bhatele, Jayaraman Thiagarajan, Bhavya Kailkhura, Jae-Seung Yeom, Barry Rountree, and Todd Gamblin. 2017. Performance Modeling under Resource Constraints Using Deep Transfer Learning. In *Proceedings of SC17, Denver, CO, USA, November 12–17, 2017,* 12 pages.
DOI: 10.1145/3126908.3126969

1 INTRODUCTION

Most HPC applications and libraries are developed with the goal of providing performance portability on a variety of hardware platforms. In order to give end users the flexibility to try different performance optimizations on different platforms, application developers typically create runtime options or knobs for selecting among alternative options, e.g. different solvers, loop orderings, etc. based on the specific requirements. As application characteristics and target platforms become more sophisticated, there is a growing trend towards providing users the flexibility to fine-tune a variety of such application-level and platform-level parameters in order to efficiently solve large-scale scientific problems [16, 21]. This has resulted in a significant increase in the number of runtime options or knobs that affect performance.

Application-level parameters can be classified into two types: a) algorithmic alternatives that affect the methods used in the execution, e.g. type of linear solver, and b) sub-algorithmic parameters that affect the execution within a method, e.g. the data layout and threshold for solution refinement. Platform-level parameters include settings that affect system power usage, number of processes, cache management, etc. and must be tuned along with application parameters for best results. Understanding the combined impact of various application-level and platform-level parameters on the eventual metrics of interest and finding the optimal configuration is challenging. Further, analytical methods are inadequate in capturing the collective impact of application parameters because these parameters are often categorical, and their impact on performance and power usage is opaque to such methods.

Supercomputing centers around the world are increasingly subjected to various operating limits. For example, exascale supercomputers will be expected to operate under a 30 MW power envelop specified in the U.S. Department of Energy Exascale Initiative requirements [13]. The resource scheduler can translate such system-wide power constraints into job-level power constraints. Subsequently, we can use a power-aware runtime system to steer the allocation of power to various hardware components and to execute the application with the configuration defined in the job specification [14, 22]. This application configuration is typically chosen by the end-user who is unaware of the decisions made by the power-steering runtime system. This often leaves performance on the table. Therefore, the runtime system must intelligently select an application configuration that optimizes performance on the behalf of the user, under the resource constraints defined by the resource scheduler. This motivates us to study the effects of the interaction between the application parameters and system constraints on dependent parameters such as performance and power usage. We employ machine learning (ML) techniques to model these interactions and relationships.

Identifying the set of input parameters that impact a dependent variable in order to reduce the parameter space that needs to be observed is extremely useful in practice. We find that effectively employing an off-the-shelf deep learning solution requires observing a significantly large portion of the parameter space, which is not practically viable due to resource constraints (e.g. limit on the number of jobs we can run on different node counts). To overcome this challenge, we propose a novel deep neural network-based solution that uses transfer learning to minimize the number of observations required in the parameter space.

First, we leverage our insight that we can transfer relationships between independent parameters and dependent variables from one parameter space to another (e.g. small input problem to a large problem or small node count to a large node count). The proposed deep network learns to transfer this knowledge by training on a combination of a fraction of the parameter space from the target problem (one that is resource-constrained) and an exhaustively observed parameter space of a related problem (one that can be observed exhaustively under our resource constraints). We then evaluate the effectiveness of our deep transfer learning technique in selecting a performance-optimizing configuration when working with a subset of the parameter space. Specifically, this paper makes the following contributions:

- We introduce a novel deep learning approach that overcomes the limitations of traditional approaches in capturing the relationship between application parameters and dependent metrics such as performance, especially when limited resources are available for collecting data.
- We show that our approach results in a robust prediction model that can reliably predict the best performing configurations using data at only 1% of the parameter space for the target problem.
- We present a user-relevant metric, *recall performance*, which, unlike traditional metrics, focuses on the accuracy of prediction models for high-performing configurations.
- We show that our deep transfer learning technique performs significantly better than traditional methods when the parameter space is subjected to some constraints.

2 MOTIVATION

A typical scientific application or library provides several configuration options to select the algorithm, intermediate data processing methods, and the desired level of accuracy for an input problem specification [16]. As compute components become increasingly sophisticated, platform designers expose many system-level knobs to a variety of users including administrators, system programmers, application developers and application users [20]. Many parameters exposed by the applications and the platforms are categorical in nature, and thus, a set of choices may not result in a particular trend in application performance or other observed metrics of the application. Consequently, the complex interaction between application-level options and platform-level knobs and their combined effect on application performance is difficult to predict as highlighted in the following example.

Large-scale supercomputing installations around the world typically operate under practical resource constraints such as operating cost, total power consumption and I/O bandwidth. Figure 1 shows the execution time and aggregate power usage of several configurations of Kripke, a proxy application for 3D Sn deterministic particle transport [21], for an example scenario with an operating processor power limit of 65 W. Each grey line connects one unique combination of configuration options in the plot. Different configurations represent ways of running the same science on fixed compute resources. The choice of algorithm, 'option 1' and 'option 2' impact the code path and the ordering of computation. Two axes (with thick lines) show two observed metrics of interest: time and application power usage for each configuration. The red curve represents the configuration a naïve user would believe to be the best at the 65 W power limit (e.g., running on all compute cores with an intent to maximize power usage). The blue curve represents the configuration an expert user would normally use in the absence of a constraint on processor power. The green curve shows the empirically-discovered best-performing configuration subject to the 65 W power limit.

Figure 1 shows practical limitations of a simple power-aware runtime system. First, a configuration that is believed by an expert user to yield the best performance may not be the best performing one subject to an operating constraint. We found that applying a simple power-optimizing runtime system [14] did not improve the performance of the expert's configuration of Kripke.

Second, the combined impact of the choice of number of MPI processes and OpenMP thread count is non-trivial as shown in Figure 1. The power usage, active cores, operating frequency and the processor thermal headroom are impacted by the distribution of MPI processes and OpenMP threads on processor cores, which in turn affects application performance. Also, as the number of cores on the processors increases and as the power management features of modern processors become more complex, the relationship between the power and thermal characteristics of the processor for an application becomes combinatorially complex [10]. Thus, picking the optimal distribution of MPI processes and OpenMP threads in

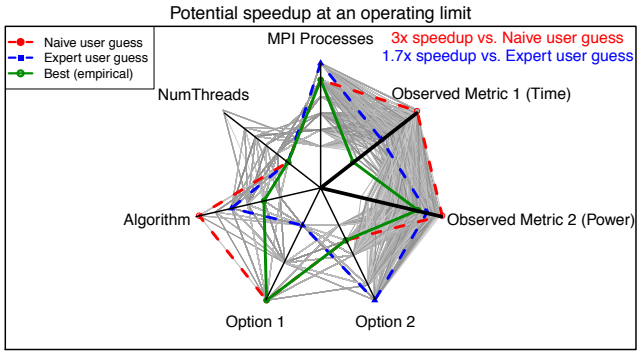


Figure 1: The parameter space and dependent metrics of Kripke at a 65 W operating limit on processor power. The configuration connected with the green curve shows the best configuration under the power limit. The red and blue lines show the naïve and expert user guesses respectively.

the presence of other application-level options and platform-level knobs to extract the maximum performance becomes important.

Finally, the naïve user's configuration that consumes the most processor power may not render the best performance as shown by the red curve. In fact, based on the empirical information from our test applications, this observation holds for several combinations of applications and inputs, regardless of the operating constraint (data not shown in Figure 1 due to space limitations). In general, for a certain operating constraint and input problem, selecting the best-performing configuration with limited prior knowledge of the application characteristics is a non-trivial problem because of the potentially complex interaction between the parameter space and the resulting compute intensiveness. Hence, we need sophisticated machine-learning techniques to assist an intelligent power-optimizing runtime system.

3 EXPERIMENTAL SETUP

This section describes the applications used in our case studies, their configurable parameters, clusters used for the experiments, their platform-specific parameters, and the data-collection and post-processing operations.

3.1 Applications

Kripke is a proxy application for ARDRA, a production transport code for particle physics [21]. `new_ij` is distributed as a test driver for the hypre library [16], which is used by many large-scale scientific applications. For this work, we evaluate the 27-point Laplacian problem implemented by `new_ij` as a representative test problem and refer to the application as hypre. For both these applications, a set of fixed options define the input problem which is being solved. Additionally, there are parameters that configure the specific methods used and impact the performance obtained without affecting the science and the correctness of the solution obtained.

Table 1 shows the set of application-level parameters available in Kripke. Six different orderings are supported for executing compute kernels in Kripke. This option, referred to as Nesting Order

Table 1: Kripke parameters

Nesting Order	DGZ, DZG, ZDG, ZGD, GZD, GDZ
Energy Group Set	1, 2, 4, 8, 16, 32, 64
Direction Set	8, 16, 32
Parallel Method	Sweep, Block Jacobi
Fixed options	–groups, –quads –zones, –legendre –niter

Table 2: hypre solver parameters for `new_ij`

Solver	**Smoother**
AMG	Hybrid Gauss-Seidel
AMG-PCG	Hybrid backward Gauss-Seidel
DS-PCG	Forward L1-Gauss-Seidel
AMG-GMRES	Chebyshev
DS-GMRES	
AMG-CGNR	**Coarsening options**
DS-CGNR	hmis
PILUT-GMRES	pmis
ParaSails-PCG	
AMG-BiCGSTAB	**Pmx**
DS-BiCGSTAB	2
GSMG	4
GSMG-PCG	6
GSMG-GMRES	
ParaSails-GMRES	**Fixed options**
DS-LGMRES	-intertype 6
AMG-LGMRES	-tol 1e-8
DS-FlexGMRES	-agg_nl 1
AMG-FlexGMRES	-CF 0

in Table 1, is represented using three letters: Energy Groups (G), Directions (D) and Spatial Zones (Z). Additionally, energy group and direction sets can be configured by the user through the 'Gset' and 'Dset' parameters. Parallel solver method decides the solver being used for the computation over a given nesting order and has two options: Sweep (default) or Block Jacobi. All these options can be configured by the user without affecting the correctness of the solution computed by Kripke.

Fixed options that define the test problem in Kripke include 'groups', 'quads' and 'zones', which define the total number of zones assigned to each compute core. The Legendre expansion order defines the number of moments for the specified order, while 'niter' decides the number of solver iterations. Since the physics represented by the application changes significantly over different Legendre orders, we choose two values for it (0 and 9) to define two input problems, denoted as Kripke-L0 and Kripke-L9, respectively in our results.

`new_ij` is a test program that allows evaluation of different hypre solver parameters, such as solver type, smoother type, coarsening strategy, and interpolation scheme on a number of different test problems. In our work, we vary the solver options summarized in Table 2 for the 27-point Laplacian test problem. 27-point Laplacian is a 3D Laplace problem discretized using a 27-point finite difference stencil on a cube.

Table 3: MPI+OpenMP configuration on 64 nodes (up to 1536 cores)

No. of MPI processes	No. of OpenMP threads/process
128	1, 2, 4, 6, 8, 10, 12
256	1, 2, 4, 6
384	1, 2, 4
512	1, 3
768	1, 2
1536	1

The `new_ij` options in Table 2 are allowed to vary over four different areas: solver, smoother, coarsening scheme, and interpolation operator. Additionally, there are four options that are kept fixed as listed in Table 2. The solvers considered are stand-alone algebraic multigrid (AMG), along with a number of different preconditioned Krylov subspace methods. In the case of AMG or solvers preconditioned with it (AMG-PCG, AMG-GMRES, AMG-CGNR, AMG-BiCGSTAB, AMG-LGMRES, and AMG-FlexGMRES), the implementation used is hypre's BoomerAMG solver [19].

Krylov solvers used are: preconditioned conjugate gradient (PCG), GMRES, CGNR, BiCGSTAB, LGMRES (the accelerated GMRES method of Baker, et. al. [2]), and FlexGMRES (the inner-outer preconditioned GMRES method of Saad [31]). Other preconditioners used besides AMG are diagonal scaling (DS), PILUT [16], GSMG [9], and ParaSails [8]. The smoothers used are all described in [1].

The coarsening options are two independent-set based coarsening algorithms (HMIS and PMIS) which were designed with low-complexity in mind, to enable good performance on large problems on massively parallel machines [11]. Most modern classical AMG methods use one of these two coarsening schemes. The `-Pmx` option controls the interpolation operator, bounding the number of entries per row at the given number (2, 4, and 6 in our experiments). This option is used to further reduce operator complexity and improve parallel performance.

3.2 Cluster used and its configuration

We ran our experiments on a 324-node Ivy Bridge cluster with InfiniBand QDR interconnect. Each node in the cluster has two 12-core Intel Xeon E5-2695 v2 processors running at 2.4 GHz frequency and 128 GB of DRAM. We used Intel RAPL over *msr-safe* to set the processor power limit between 50W and 100W in steps of 5W for each combination of parameters in the rest of the parameter space. Intel Turbo was enabled so that the applications could extract maximum performance under the power limit [20]. We excluded nodes that were pre-characterized in the minimum and maximum power efficiency spectrum to avoid the effects of manufacturing variability on the distribution of performance and power usage [17]. We compiled both applications with $-O2$ flag. Table 3 lists the MPI+OpenMP configurations used on 64 nodes.

3.3 Data collection

We used a light-weight sampling mechanism to record application performance, and processor and memory power usage at 200 milliseconds interval with less than 0.1% performance overhead [23]. We calculated the average processor and memory power draw over the entire application run based on the instantaneous power samples. We also recorded the execution time based on the end-to-end wall clock time of the application. We ran all configurations for each application three times and used the samples closest to the median performance for our study. The total number of valid configurations for Kripke and hypre are 17k and 50k, respectively.

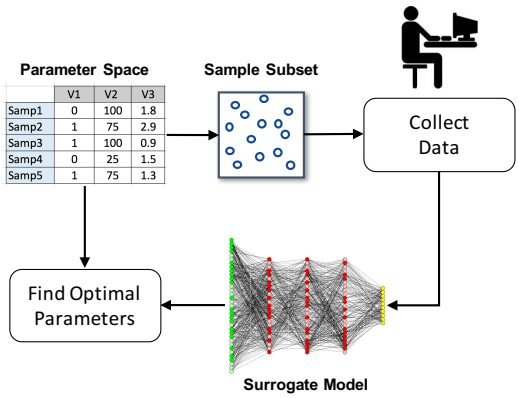


Figure 2: Overview of our approach to find a performance-optimizing configuration in the parameter space.

4 APPROACH

Determining an optimal configuration of application-level and platform-level parameters requires a qualitative understanding of the parameter space. The inherent high-dimensionality of the parameter space and the sensitivities of the dependent variables/metrics with respect to different parameters make this problem extremely challenging. More specifically, we need a reasonable understanding of the geometry of the response variable in the high-dimensional parameter space, so that we can identify the configurations close to the global optimum. However, this requires a combinatorially large number of samples in high dimensions, which has a prohibitively high cost in terms of data collection, making it practically impossible.

A natural approach to tackle this problem is to build a surrogate model (see Figure 2) that can potentially replace the actual data collection step by a mathematical model to predict the performance metric based on a practically observable sample space. An attractive feature of this approach is that the surrogate (e.g. regressor) can be constructed using a small fraction of the samples required to build an actual geometric description and the user can evaluate the performance of all parameter choices without actually running the application with all configurations. However, a drawback of this approach is that the approximate model can introduce significant uncertainties into the process and hence produce inaccurate predictions. The level of uncertainty depends on a number of factors: (a) sample size, (b) complexity of the model, and (c) geometry of the function that the surrogate model is approximating.

In this section, we describe the formulation for constructing the surrogate models and study the impact of sample size and model choice on recovering the underlying function that relates the parameter space to the response variable of application performance. To

optimize the surrogate model, we present a novel transfer-learning technique based on deep neural networks.

4.1 Surrogate Model Design

4.1.1 Formulation. Supervised machine learning methods are typically well-suited to exploit the dependencies between independent variables in empirical application data, while modeling the relationship between them and a response variable (e.g. execution time or processor power usage).

Denoting the parameter space by $\mathcal{P}$ and its cardinality by d, each run $\mathbf{x} \in \mathcal{P}$ can be represented as a d-dimensional vector. Let y indicate the corresponding response variable. The goal of surrogate modeling is to approximate the function $f : \mathcal{P} \mapsto y$. The problem of selecting the subset of samples used to train the surrogate model, referred to as *experiment design*, has been extensively studied in the statistics literature. In this paper, we employ uniform random sampling to construct the training set.

4.1.2 Pipeline. We now describe the analysis pipeline used for optimal parameter search:

a) Sample design: Let $\mathcal{L}$ be the empirical data set available for an application. We construct a uniformly random sample $\mathcal{L}_s \subset \mathcal{L}$ to build the surrogate model. Note that, $\mathcal{L}_s$ corresponds to the set of tuples $\{\mathbf{x}_i, y_i\}$, where y_i denotes the actual performance metric obtained by running the application.

b) Preprocessing: An important characteristic of $\mathcal{P}$ is that several of the parameters are discrete-valued in nature, while the rest are continuous. In order to handle them together, we transform each discrete variable into multiple binary variables corresponding to each of the discrete states. In addition, we perform centering and scaling of the parameters independently, to be zero mean and unit variance. Standardization of a dataset is a common requirement for many machine learning estimators, since they might behave badly if the individual features are not close to being normally distributed.

c) Model learning: For a given training sample size, the complexity of the machine learning model directly controls the amount of uncertainty in the prediction. Taking this behavior into account, we choose an appropriate machine learning algorithm and build the surrogate model f. Though execution time is a natural response variable in these studies, we observed that the surrogate models, in the quest to reduce the average modeling error were biased towards runs with high execution times. To avoid this behavior, we use the inverse of execution time as the response variable and we refer to this as the *efficiency measure*. As expected, this measure is higher for runs with lower execution time.

d) Evaluation: Using the surrogate model, we predict the performance metric for each of the samples in the set $\mathcal{L}_t = \mathcal{L} - \mathcal{L}_s$, i.e., difference between the two sets. The quality of the surrogate model f can be evaluated using a variety of metrics from the statistics literature. Common examples include the mean squared error, mean absolute error (MAE), mean absolute percentage error (MAPE) and the R^2 statistic. We use MAPE for comparing the different surrogate models:

$$MAPE = \frac{100}{n} \sum_{i=1}^{n} \left| \frac{P_{\text{actual}}(i) - P_{\text{predicted}}(i)}{P_{\text{actual}}(i)} \right|$$

where $P_{\text{actual}}(i)$ is the actual performance and $P_{\text{predicted}}(i)$ is the predicted performance of i^{th} sample. The goal of this evaluation is to estimate the expected performance of the surrogate model in describing the characteristics of the parameter space, thus enabling the user to determine an optimal parameter setting for the application.

4.1.3 Analysis. As described earlier, both the training sample size and the choice of the machine learning algorithm are critical in determining the level of uncertainty in the resulting surrogate model, which in turn controls the probability of choosing a parameter setting close to the optimum. Consequently, we analyze the prediction performance by varying the size of $\mathcal{L}_s$ as $\frac{f \times L}{100}$, where $f \in \{1, 2, ..., 90\}$ and L is the size of the parameter space. In each of the cases, we use the remainder $\mathcal{L} - \mathcal{L}_s$ to evaluate the prediction accuracy. Further, to obtain statistically meaningful results, we repeat the random sampling process for 50 independent trials and collect the summary statistics.

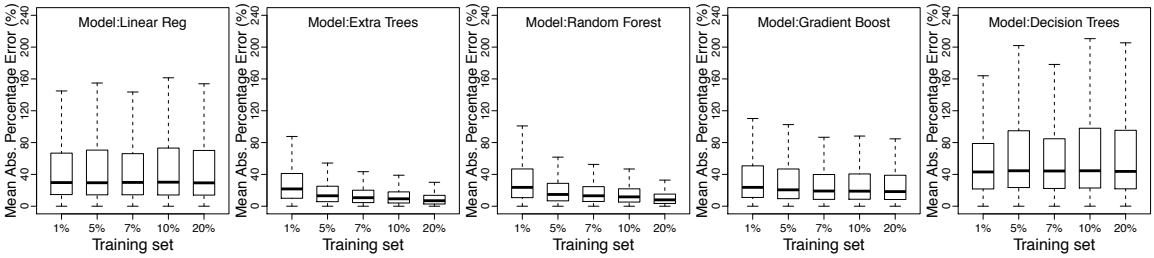
For the choice of the machine learning algorithm, we use a suite of models from the `scikit-learn` package [27], characterized by various degrees of complexity – linear regression, ridge (ℓ_2–regularized linear) regression, decision trees, and a set of ensemble methods including random forests, extremely randomized trees and gradient boosted machines. The results for these initial experiments are shown in Figure 3 for Kripke-L0 and hypre. We observe a few trends that are generally consistent across all three applications (including Kripke-L9) – the prediction accuracy improves and MAPE variance reduces as we incorporate more training data. This is in line with what we expect – as we observe more of the space, the surrogate models tend to get better at predicting the performance.

When simple statistical inference models viz. linear regression and ridge regression are used to build the surrogate model, the prediction performance is highly unsatisfactory, indicating the need for more sophisticated models. In particular, we observe that linear models tend to overcompensate for larger values of the response variable and are highly inadequate in modeling discrete valued variables, resulting in large errors. In contrast, when sophisticated models are used, we observe that the prediction accuracy improves considerably in terms of MAPE. In particular, we notice that Extremely Randomized Trees (ExtraTrees) and Random Forests (RandomForests) are the best performing models. Thus, in the rest of the paper, we compare our results with ExtraTrees and RandomForests.

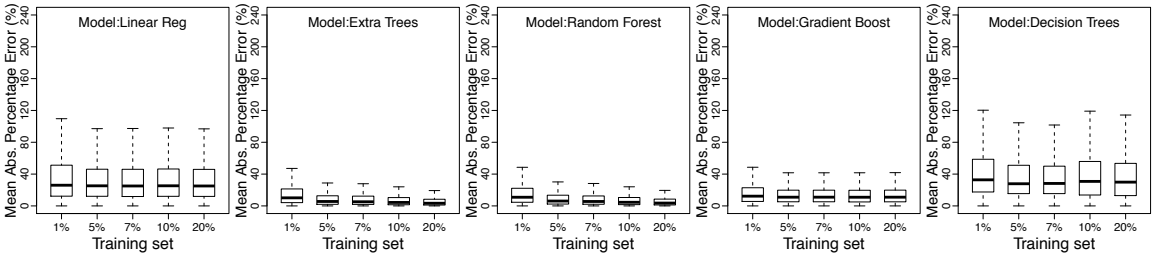
4.2 Limitations of Traditional Models

The pipeline discussed previously focused on modeling the performance as a function of the input parameters, utilizing only a subset of samples, $\mathcal{L}_s$. Subsequently, insights from the surrogate were used to generalize the predictions to the unobserved cases in $\mathcal{L}_t$. While this approach produces high-quality predictions in terms of MAPE over the entire training size, it has some key limitations.

First, a performance-oriented user is only interested in the model which is accurate for the best performing configurations. Figure 4 shows the MAPE metric for the top 20% performing configurations of Kripke-L0 and hypre using ExtraTrees and RandomForests.



(a) Kripke-L0 comparison



(b) hypre comparison

Figure 3: Comparison of Mean Absolute Percentage Error (MAPE) scores of different models for Kripke-L0 and hypre parameter space.

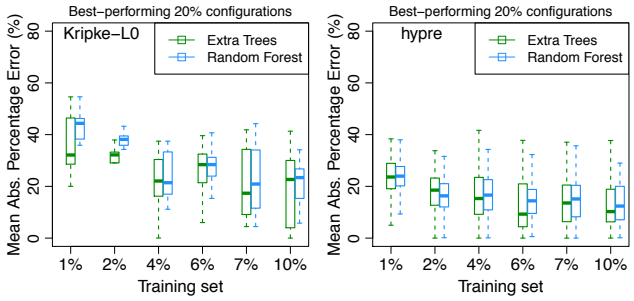


Figure 4: Performance of ExtraTrees and RandomForests for Kripke-L0 and hypre in terms of MAPE.

Compared to Figure 3, the performance of both ExtraTrees and RandomForests is consistently worse for the top 20% configurations than over the entire parameter space. Both ExtraTrees and RandomForests perform worse for the top 20% configurations because both models optimize for the common-case average-performing configurations and, in turn, introduce significant error towards the high-performing region of the parameter space.

Second, the performance of both models at small sample sizes is not particularly satisfactory. From a user's point of view, a large cardinality of the set $\mathcal{L}_s$ implies that a large portion of the application parameter space must be actually tested before selecting the optimal configuration. This necessitates the design of machine learning techniques that can potentially produce accurate surrogate models with fewer initial samples.

4.3 Transfer Learning to Improve Surrogate Models

For applications in HPC, our past experience suggests that executions using fewer resources (potentially for smaller input problems) share runtime characteristics with executions using more resources (potentially for large problems). Thus, data from low-resource experiments can be applied towards the modeling process of the target resource-constrained scenario to improve the fidelity of a surrogate model. This approach is commonly referred to as *transfer learning* in the machine-learning community.

Transfer learning imports hidden relationships between variables of a previously seen search space (*source domain*) to another search space (*target domain*) with slightly different domain characteristics. Specifically in our context, we aim to transfer our knowledge about the surrogate model for data from a specific application execution scenario ("source") to another execution scenario ("target"). For example, in the context of MPI applications, we define the source domain at fewer MPI processes, and the target domain at more MPI processes in a weak-scaling fashion. Since this technique depends on the parameters unique to an application, we apply it to each application separately.

4.3.1 PerfNet Architecture. We propose to use a neural network approach, referred to as *PerfNet*, to transfer learning. Let us denote the parameter space for source and target domains as $\mathcal{L}^{(Src)}$ and $\mathcal{L}^{(Trgt)}$ respectively. Using the same notation as earlier, $\mathcal{L}_s^{(Src)}$ and $\mathcal{L}_s^{(Trgt)}$ correspond to the sample subsets for building the surrogate in the two domains. Note that, $|\mathcal{L}_s^{(Trgt)}| << |\mathcal{L}_s^{(Src)}|$, i.e., the size of the target parameter set used for training is much smaller. Finally, $\mathcal{L}_t^{(Trgt)}$ and $\mathcal{L}_t^{(Src)}$ are the corresponding sets for

remaining samples in the parameter space, which our modeling approach does not have access to during the training stage.

The PerfNet model consists of three fully-connected deep neural networks, N_{source}, N_{target}, and N_{final}. First, model N_{source} is trained on the source dataset, $\mathcal{L}_s^{(Src)}$, which is a surrogate model for the source parameter space, much like the regression models discussed earlier. Similarly, we train the second neural network N_{target} as a surrogate for the target dataset. However, since we have a very small number of samples in the target parameter space, a neural network cannot be trained to work effectively. Instead, we perform the first stage of transfer learning, where we take a copy of the trained N_{source}, to serve as the *initialization* for the target network. Next, we train this network with $\mathcal{L}_s^{(Trgt)}$, which converges must faster to give a strong surrogate model for the target set.

In our implementation, the fully connected networks are created using the standard methods in Keras[1]. In all the networks, we use fully connected layers with a rectified linear unit (ReLu) activation, followed by batch-normalization in between each layer.

Networks N_{source} and N_{target} can be used independently as surrogates for their respective parameter spaces. However, we go one step further in order to build a single model N_{final} that can act as a unified surrogate across both the parameter spaces. We achieve this by transforming $\mathcal{L}_s^{(Src)}$ and $\mathcal{L}_s^{(Trgt)}$ to a latent space, using the networks N_{source} and N_{target}. This transformation is achieved by the output of the respective neural networks, when only the first layer is used.

Use of output of the first layer works well in practice because the hidden representations from each network are transformed in such a way that they optimally predict the performance for the respective domains. The hidden representations are also tuned to capture many invariances in the original raw data. Finally, in some cases, the same input settings (features) can give dramatically different results based on the parameter space under consideration. As a result, the latent features learn a mapping from the input parameter space to a latent space where the latent features can be easily distinguished. We then train the final neural network, N_{final}, on the latent features and treat them as a single large dataset. The training procedure is depicted in Figure 5.

For a test case, we first map the test sample to the latent space using the appropriate network, N_{source} or N_{target} depending on whether it is from the source or target parameter space. The latent space representation is then passed through the prediction network, N_{final}, to get an estimate of the performance.

4.4 Using the Proposed Methodology

This section lists the steps to practically deploy our proposed technique for selecting an optimal configuration of an HPC application with a large performance-centric input parameter space. Typical approaches for such an exploration involve many resource-consuming runs at large scales, denoted by $\mathcal{L}^{(Trgt)}$. Instead, we reduce the total resource consumption by conducting most of the experiments at a smaller scale as following:

[1]https://github.com/fchollet/keras

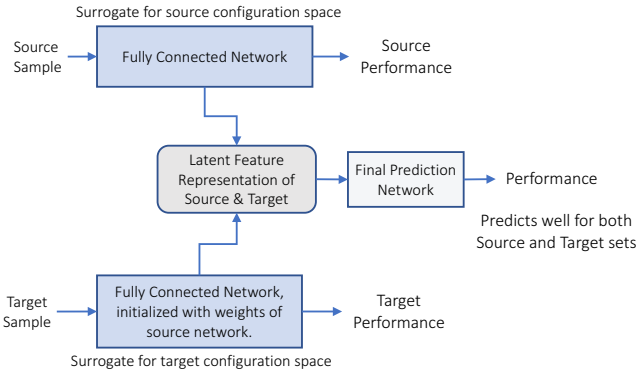


Figure 5: PerfNet consists of 3 neural networks internally, such that a single model can make reliable predictions on two parameter spaces.

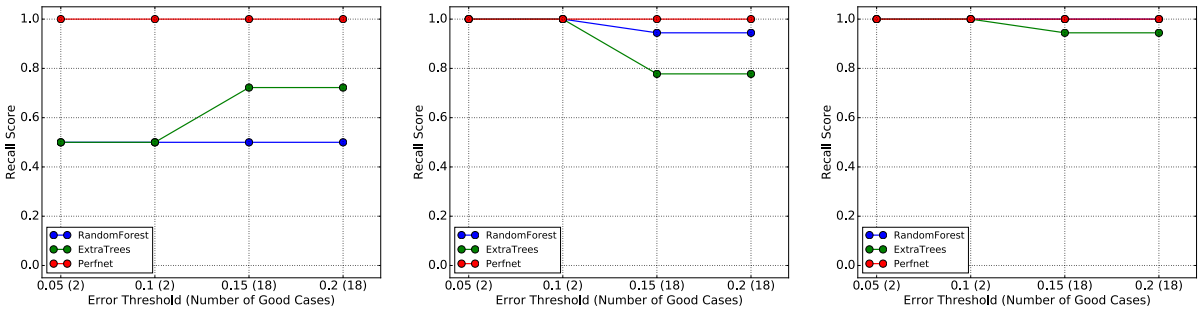
(1) Identify a small-scale scenario, $\mathcal{L}^{(Src)}$, that is related to the large-scale scenario, $\mathcal{L}^{(Trgt)}$, and run experiments to collect performance/power data for the exhaustive parameter space for $\mathcal{L}^{(Src)}$. For example, in the context of MPI applications, $\mathcal{L}^{(Src)}$ can be execution at fewer MPI processes when $\mathcal{L}^{(Trgt)}$ signifies execution at more MPI processes. This should require significantly fewer resources than running at $\mathcal{L}^{(Trgt)}$.
(2) Collect performance/power data at $\mathcal{L}^{(Trgt)}$ for a small set of $\mathcal{S}_{init}$ samples selected uniformly randomly (e.g. 1% of total space).
(3) Use the proposed transfer learning technique to build a surrogate model for predicting performance/power on $\mathcal{L}^{(Trgt)}$ using data collected for $\mathcal{L}^{(Trgt)}$ ($\mathcal{S}_{init}$ samples) and $\mathcal{L}^{(Src)}$ (entire parameter space).
(4) Use the surrogate model to predict performance/power for the entire configuration space and identify the best performing $\mathcal{S}_{hyp}$ (e.g. 100) configurations.
(5) Run the $\mathcal{S}_{hyp}$ configurations to obtain performance/power for $\mathcal{L}^{(Trgt)}$ and select the best performing configuration.
(6) Obtain the importance of individual parameters from PerfNet to make tuning decisions.

Section 5 shows that the proposed methodology is able to identify the best configuration with high probability using very small values of $\mathcal{S}_{init}$ and $\mathcal{S}_{hyp}$, and this makes finding a good configuration significantly less expensive.

5 EVALUATION

This section compares the performance of PerfNet with existing techniques using two case studies. In the first case study, we use PerfNet to find the best performing configurations for Kripke and hypre (described in Section 3). The second case study shows that our approach is also effective under an operational limit defined in terms of the maximum processor power usage.

In the results presented in Section 4.1, we compared different ML techniques using a commonly used scoring metric, namely MAPE. However, we also showed that MAPE computed over the entire



(a) Kripke-L0 with 1% samples in $\mathcal{L}_s^{(Trgt)}$ (b) Kripke-L0 with 5% samples in $\mathcal{L}_s^{(Trgt)}$ (c) Kripke-L0 with 10% samples in $\mathcal{L}_s^{(Trgt)}$

Figure 6: Comparison of recall metric scores of different models for Kripke-L0. PerfNet consistently shows better score than other ML methods.

parameter space does not accurately characterize the prediction accuracy for configurations with high performance (top 20%). Furthermore, we find that even when MAPE is computed over the top 20% configurations, it does not indicate the model's effectiveness in recovering the best performing configurations. This motivates the need to develop a new scoring function that satisfies this need.

Note that the prediction accuracy of PerfNet in terms of MAPE over the entire parameter space is similar to both ExtraTrees and RandomForests. Moreover, for the top 20% configurations, PerfNet typically leads to a better median value for MAPE than ExtraTrees and RandomForests if a small training set is used. With larger training sets, the worst-case value of MAPE for PerfNet is significantly lower than ExtraTrees and RandomForests.

5.1 *Recall*: A New Metric to Evaluate Surrogate Models

Broadly speaking, the two primary considerations for a performance analyst are to: (a) understand the usefulness of a surrogate model in identifying the optimal configurations for an application, and (b) use a small number of samples to train the surrogate, since the analyst has to actually run the application to generate the training set. Thus, the usefulness of a surrogate model can be quantified based on the effective number of test runs that the analyst has to run, S_{eff}, before arriving at a final solution in the parameter space. In our pipeline, the analyst first generates the training set by running S_{init} cases. Using the surrogate model, we predict the execution time for the rest of the parameter space $\mathcal{L}_t^{(Trgt)}$. Next, we choose S_{hyp} hypothesis samples from $\mathcal{L}_t^{(Trgt)}$ that have the best predicted value for the dependent metric. Following this, we obtain empirical results for the hypothesis samples by running the application for those samples and pick the best configuration, thus making $S_{eff} = S_{init} + S_{hyp}$.

The quality of a surrogate model depends directly on its fidelity to include the globally optimal and near optimal cases in the effective sample set $\mathcal{L}_{eff}^{(Trgt)}$. In order to quantify this, we compute the ratio of the number of "good" configurations that are included in $\mathcal{L}_{eff}^{(Trgt)}$ and the actual number of good configurations in the entire parameter space. Here, good samples refer to those whose performance is within α% of the absolute best performance (global maximum). The metric can be computed as:

$$R(\alpha) = \frac{|\{i \mid i \in L_{eff}^{(Tgrt)}, P(i) \geq (1-\alpha)P_{\text{best}}\}|}{|\{i \mid \forall i, P(i) \geq (1-\alpha)P_{\text{best}}\}|}$$

where, $|.|$ represents the cardinality of a set, $L_{eff}^{(Tgrt)}$ is the set of configurations for which the user has empirical results (includes the training set and the predicted best S_{hyp} configurations), α is the performance threshold, $P(i)$ is the actual performance of configuration i, and P_{best} is the actual best performance.

This metric is similar to the *Recall* score used in detection, and retrieval applications. The best model is the one that consistently has a Recall score close to 1.0, implying that it is able to consistently identify nearly all of the best performing configurations. In our experiments we fixed values of $S_{hyp} = 100$ and $\alpha = 5, 10, 15, 20\%$ respectively. It is important to note that this metric is stricter than MAPE since it is only sensitive to the regime of the parametric space where the performance is obtained. This regime is of most interest to the performance analyst as their ultimate goal is to pick the best performing configuration.

5.2 Case Study I: Finding the Best Performing Configuration

5.2.1 Kripke. Figure 6 shows comparisons of PerfNet, ExtraTrees and RandomForests for Kripke-L0 using the recall metric. Figures (a), (b), and (c) correspond to Kripke-L0 for different sizes of training sets – 1%, 5% and 10%, respectively. The X-axis shows the error tolerance (α) that determines what qualifies as a good sample. Along with the error, we also show the number of actual good samples in parentheses, according to the current threshold. This number indirectly indicates the complexity of the inherent function.

It is immediately evident that even though ExtraTrees and RandomForests are nearly as good as PerfNet in terms of MAPE, the models show a large disparity in terms of recall score. In particular, PerfNet is able to identify the best performing configurations accurately even when S_{init} is just 1% of the total samples in the parameter space. Specifically, PerfNet is better than ExtraTrees and

RandomForests by up to 0.5 recall score across different training sizes. For the same training set size, the scores of ExtraTrees and RandomForests change over increasing threshold as more samples qualify as good samples, but the number of samples we observe is fixed at S_{hyp}. We find that the scores of ExtraTrees and RandomForests decrease with increasing error threshold whereas PerfNet's score remains constant at 1.0. We observe similar performance of PerfNet vs. ExtraTrees for Kripke-L9.

5.2.2 hypre. Figure 7 shows the comparison of PerfNet, ExtraTrees and RandomForests for hypre using the recall metric. Similar to Figure 6, PerfNet performs consistently better than ExtraTrees and RandomForests across different error thresholds and training sizes. Specifically, PerfNet performs better than ExtraTrees and RandomForests by up to 1.0 recall score. The scores of all three models drop consistently with increasing thresholds at all training sizes due to the reason noted previously, i.e. more samples qualify as good samples (higher denominator in the computation of recall metric) but S_{hyp} is fixed at 100. This sensitivity to threshold for hypre is unusual and indicates a highly complex relationship between its parameter space and performance than Kripke-L0 and Kripke-L9. Nonetheless, PerfNet is able to identify several good configurations, and the recall score is lower only because a large number of configurations (relative to S_{hyp}) are good configurations with higher threshold.

5.3 Case Study II: Best Configuration Under Operational Power Limit

This section evaluates the effectiveness of PerfNet in enabling selection of power-optimizing configurations of Kripke and hypre under a hardware-enforced processor power limit. Figure 8 compares the recall metric of PerfNet, ExtraTrees and RandomForests with 1% training samples at 70W, 80W and 90W processor power limits for Kripke-L0. In all three plots, PerfNet is able to predict the best configurations (5% threshold) and consistently remains above a strong 0.75 recall score even as the number of samples increases with increasing power limit. ExtraTrees improves in terms of recall score with the smallest error threshold at 70W but consistently underperforms toward larger deviation from the best performance. RandomForests performs equally poorly with all three power limits.

Figure 9 compares recall score for PerfNet with ExtraTrees and RandomForests at processor power limits of 70W, 80W and 90W with 4% training samples in $\mathcal{L}^{(Trgt)}$ for hypre. Similar to Figure 8, PerfNet shows a strong recall score of above 0.8 for up to 10% of performance deviation threshold. The other two models perform worse than PerfNet at 70W power limit. With increase in power limit from 70W to 90W, and therefore at a larger training set, the rate of degradation in recall metric of PerfNet at higher threshold is much slower than other models. This observation highlights that PerfNet's response to the increase in the size of training set is much better than the other models.

In summary, the results for both Kripke-L0 and hypre show that the proposed surrogate modeling approach enables accurate configuration search in high-dimensional parameter spaces, and in particular PerfNet demonstrates higher tolerance to samples with large deviations from the best performing configuration, when compared to other surrogate models.

6 RELATED WORK

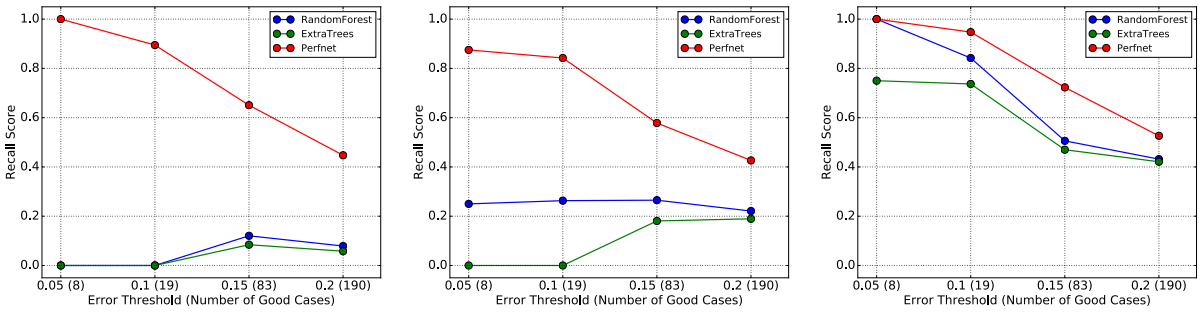
A large body of literature exists on the broad topic of tuning the application and platform parameters for performance optimization. We categorize the existing work in several ways and show the novelty of our proposed technique in each category.

Active learning-based approaches: Online tuning of application parameters with active-learning methods has been a widely researched topic. Balaprakash et al. present an iterative parallel algorithm that builds surrogate performance models for scientific applications on several architectures [3]. Ogilvie et al. present a low-cost, online predictive algorithm to select training samples by building heuristics and reduce training overhead [25]. Chen et al. present a similar online approach to improve the selection of online samples in the parameter space for non-HPC applications [7]. Bergstra et al. [5] build *regression models* that map tuning parameters to run times to aid a search. While these are potentially useful approaches for a class of HPC applications, the parameter space available for online tuning is typically limited for common HPC applications (e.g., MPI processes, data pre-processing step, etc.), and therefore cannot be applied easily.

Analytical models for parameter tuning: Analytical models have been proposed to predict application performance based on online and offline measurements of intrinsic application characteristics. Previous work on platform tuning for performance optimization includes fine-tuning the operating system and programmable platform parameters subject to power constraint. Towards the power-aware performance optimization problem, Curtis-Mowry et al. present a prediction-based approach on modeling application performance based on hardware counters, concurrency throttling and DVFS [10]. Zhang et al. present a hybrid hardware/software system with better reaction times by maximizing the utilization of certain platform features such as hyperthreading, clock modulation and memory controllers [33]. Such approaches are limited to a single task or at processor level, and their scope does not take into account the application context and algorithm-level options beyond these levels.

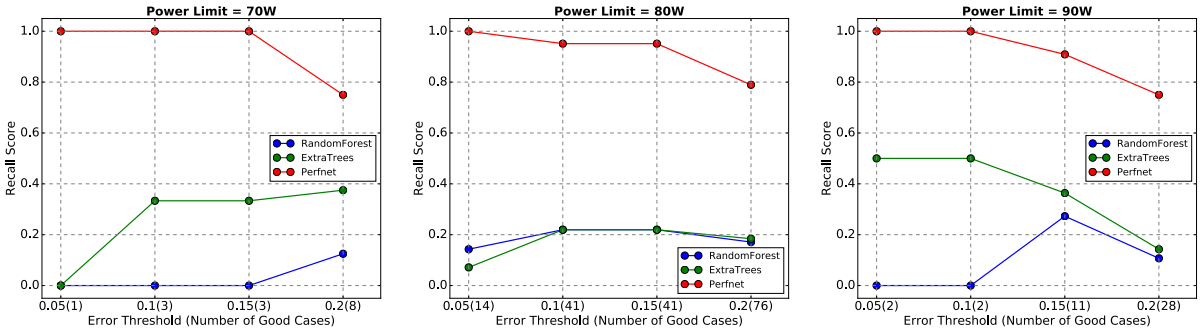
ML techniques for performance tuning on target paradigms: Developing models on one application or platform and applying them to target application or platforms has been a highly researched topic in the domain of performance optimization. Roy et al.[30] present techniques for auto-tuning search algorithms by exploiting performance models on one architecture and applying them on other architectures. Price et al.[28] present a hybrid approach to auto-tuning that combines empirical sampling and a predictive performance model with the goal of optimizing configuration space search on progressively larger sets of target platforms. Falch et al.[15] use machine learning to fine-tune the parallelizing runtime system for target applications on GPU platforms. Muralidharan et al. [24] and Ding et al. [12] use machine learning models for code-variant tuning. Although these approaches have some overlap with certain aspects of our work, they differ in their objectives and application of the ML techniques significantly from our approach.

ML techniques for application parameter tuning: Previous work most closely related to our approach is as follows. Grebhahn et al. [18] use performance-influence model to combine heuristics



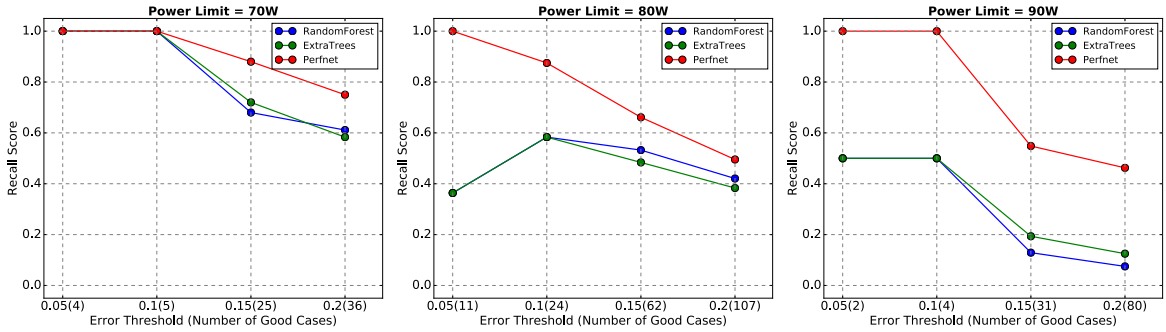
(a) hypre with 1% samples in $\mathcal{L}_s^{(Trgt)}$ (b) hypre with 5% samples in $\mathcal{L}_s^{(Trgt)}$ (c) hypre with 10% samples in $\mathcal{L}_s^{(Trgt)}$

Figure 7: Comparison of recall metric of different models. PerfNet consistently shows better score than other ML methods for hypre.



(a) Kripke-L0 with 1% samples in $\mathcal{L}_s^{(Trgt)}$ (b) Kripke-L0 with 1% samples in $\mathcal{L}_s^{(Trgt)}$ (c) Kripke-L0 with 1% samples in $\mathcal{L}_s^{(Trgt)}$

Figure 8: Comparison of recall metric scores of PerfNet with ExtraTrees and RandomForests for Kripke-L0 at power limits of 70W, 80W and 90W. The plots compare the recall prediction accuracy of the models with 1% of sample space for training (within the power limits). The X-axis corresponds to a user-relevant threshold of tolerable deviation from the absolute best configuration



(a) hypre with 4% samples in $\mathcal{L}_s^{(Trgt)}$ (b) hypre with 4% samples in $\mathcal{L}_s^{(Trgt)}$ (c) hypre with 4% samples in $\mathcal{L}_s^{(Trgt)}$

Figure 9: Comparison of recall metric scores of PerfNet with ExtraTrees and RandomForests for hypre at power limits of 70W, 80W and 90W with 4% of sample space for training (within the power limits).

for sample selection with incorporating domain knowledge specifically for Algebraic Multi-Grid solver configurations. Balaprakash et al. [4] present an automatic multi-objective modeling approach on predicting application performance and power usage based on hardware performance counters using traditional methods. Bergstra et al. [6] apply machine learning techniques for optimal parameter selection specifically for applications in the computer vision domain. We have shown that our approach of incorporating domain knowledge with deep neural networks is superior to traditional models for predicting the best performing configurations.

Power-constrained optimization: Previous work on power-aware and power-constrained performance optimization focuses on platform-centric power management features and scaling application configurations around the platform-centric features. Existing runtime systems use empirical and on-line knowledge of platform configuration space and its correlation with the performance and power usage of the application. Conductor [22] by Marathe et al. and the Global Extensible Open Power Manager (GEOPM) [14] by Eastep et al. deploy forms of power reallocation strategy presented by Rountree et al. [29] to slow down non-critical parts of the application and reuse the excess power to speed up the critical parts of the application under a power budget. Both these approaches are limited in the scope of the parameter space than our framework and are therefore not directly comparable. Patki et al. [26] and Sarood et al. [32] explore the performance characteristics of a power-constrained job on a cluster with hardware over-provisioning from the perspective of resource scheduler by including the number of compute resources in the configuration space. Such (and other) approaches at the level of the resource scheduler are beyond the scope of our proposed approach which operates within the context of a job with fixed scheduler-defined resources constraints.

7 CONCLUSION

In this work, we have presented an effective method to identify high-performing application configurations when limited resources are available for collecting training performance data. Building upon the intuition that in HPC, executions at a smaller scale (in input problem size or node count) can be used to characterize executions at a larger scale, we demonstrated the general effectiveness of PerfNet for parameter space tuning. We showed that PerfNet, a deep learning technique augmented with domain transfer learning, is capable of predicting complex relationships between application-level and platform-level parameters, and dependent metrics such as execution time.

We also showed that traditional scoring metrics such as MAPE are not suitable for evaluating machine learning methods when the goal is to find the best-performing configurations. To overcome the limitations of traditional metrics, we formulated a new metric called recall that is directly related to the task of finding the best-performing configurations. Using the recall metric, we showed that PerfNet outperforms other machine learning methods such as Extremely Randomized Trees and Random Forests, especially when the amount of data available at large scale is extremely limited.

ACKNOWLEDGMENTS

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-736726).

REFERENCES

[1] Allison H. Baker, Robert D. Falgout, Tzanio V. Kolev, and Ulrike Meier Yang. 2011. Multigrid Smoothers for Ultraparallel Computing. *SIAM Journal on Scientific Computing* 33 (2011), 2864–2887. Issue 5.

[2] Allison H. Baker, Elizabeth R. Jessup, and Thomas Manteuffel. 2006. A Technique for Accelerating the Convergence of Restarted GMRES. *SIAM J. Matrix Anal. Appl.* 26 (2006), 962–984. Issue 4.

[3] Prasanna Balaprakash, Robert B Gramacy, and Stefan M Wild. 2013. Active-learning-based surrogate models for empirical performance tuning. In *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*. IEEE, 1–8.

[4] Prasanna Balaprakash, Ananta Tiwari, Stefan M Wild, Laura Carrington, and Paul D Hovland. 2016. AutoMOMML: Automatic Multi-objective Modeling with Machine Learning. In *International Conference on High Performance Computing*. Springer, 219–239.

[5] J Bergstra, N Pinto, and D Cox. 2012. Machine learning for predictive auto-tuning with boosted regression trees. In *Proceedings of Innovative Parallel Computing*. 1–9.

[6] James Bergstra, Daniel Yamins, and David Cox. 2013. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *International Conference on Machine Learning*. 115–123.

[7] Jiahong K Chen, Ray-Bing Chen, Akihiro Fujii, Reiji Suda, and Weichung Wang. 2017. Surrogate-Assisted Tuning for Computer Experiments with Qualitative and Quantitative Parameters. (2017).

[8] Edmond Chow. 2001. Parallel Implementation and Practical Use of Sparse Approximate Inverse Preconditioners with a Priori Sparsity Patterns. *International Journal of High Performance Computing Applications* 15 (2001), 56–74. Issue 1.

[9] Edmond Chow. 2003. An unstructured multigrid method based on geometric smoothness. *Numerical Linear Algebra With Applications* 10 (2003), 401–421.

[10] M. Curtis-Maury, A. Shah, F. Blagojevic, D.S. Nikolopoulos, B.R. de Supinski, and M. Schulz. 2008. Prediction models for multi-dimensional power-performance optimization on many cores. In *International Conference on Parallel Architectures and Compilation Techniques*.

[11] Hans De Sterck, Ulrike Meier Yang, and Jeffrey J. Heys. 2006. Reducing Complexity in Parallel Algebraic Multigrid Preconditioners. *SIAM J. Matrix Anal. Appl.* 27 (2006), 1019–1039. Issue 4.

[12] Yufei Ding, Jason Ansel, Kalyan Veeramachaneni, Xipeng Shen, Una-May O'Reilly, and Saman Amarasinghe. 2015. Autotuning algorithmic choice for input sensitivity. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*. 379–390.

[13] U.S. D.O.E. 2016. Exascale Initiative. http://www.exascaleinitiative.org/pathforward. (2016).

[14] Jonathan Eastep, Steve Sylvester, Christopher Cantalupo, Federico Ardanaz, Brad Geltz, Asma Al-Rawi, Fuat Keceli, and Kelly and Livingston. 2016. Global Extensible Open Power Manager: A Vehicle for HPC Community Collaboration Toward Co-Designed Energy Management Solutions. In *7th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems, 2016*. 43–53.

[15] Thomas L Falch and Anne C Elster. 2017. Machine learning-based auto-tuning for enhanced performance portability of OpenCL applications. *Concurrency and Computation: Practice and Experience* 29, 8 (2017).

[16] Robert D. Falgout and Ulrike Meier Yang. 2002. HYPRE: A Library of High Performance Preconditioners. In *Computational Science–ICCS 2002*. Springer, 632–641.

[17] Neha Gholkar, Frank Mueller, and Barry Rountree. 2016. Power Tuning HPC Jobs on Power-Constrained Systems. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT '16)*. ACM, 179–191.

[18] Alexander Grebhahn, Norbert Siegmund, Harald Köstler, and Sven Apel. 2016. Performance prediction of multigrid-solver configurations. In *Software for Exascale Computing*. Springer, 69–88.

[19] Van Emden Henson and Ulrike Meier Yang. 2002. BoomerAMG: A parallel algebraic multigrid solver and preconditioner. *Applied Numerical Mathematics* 41 (2002), 155–177. Issue 1.

[20] Intel. 2011. Intel-64 and IA-32 Architectures Software Developer's Manual, Volumes 3A and 3B: System Programming Guide. (December 2011).

[21] AJ Kunen, TS Bailey, and PN Brown. 2015. KRIPKE-A massively parallel transport mini-app. *Lawrence Livermore National Laboratory (LLNL), Livermore, CA, Tech. Rep* (2015).

[22] Aniruddha Marathe, Peter E Bailey, David K Lowenthal, Barry Rountree, Martin Schulz, and Bronis R de Supinski. 2015. A Run-Time System for Power-Constrained HPC Applications. In *International Supercomputing Conference*.

[23] Aniruddha Marathe, Hormozd Gahvari, Jae-Seung Yeom, and Abhinav Bhatele. 2016. LibPowerMon: A Lightweight Profiling Framework to Profile Program Context and System-Level Metrics. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops*. 1132–1141.

[24] Saurav Muralidharan, Manu Shantharam, Mary Hall, Michael Garland, and Bryan Catanzaro. 2014. Nitro: A Framework for Adaptive Code Variant Tuning. In *Proceedings of the IEEE International Symposium on Parallel & Distributed Processing*. 501–512.

[25] William F Ogilvie, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2014. Fast automatic heuristic construction using active learning. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 146–160.

[26] Tapasya Patki, David K Lowenthal, Barry Rountree, Martin Schulz, and Bronis R de Supinski. 2013. Exploring hardware overprovisioning in power-constrained, high performance computing. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*. ACM, 173–182.

[27] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[28] James Price and Simon McIntosh-Smith. 2015. Improving Auto-Tuning Convergence Times with Dynamically Generated Predictive Performance Models. In *Embedded Multicore/Many-core Systems-on-Chip (MCSoC), 2015 IEEE 9th International Symposium on*. IEEE, 211–218.

[29] Barry Rountree, David K. Lowenthal, Bronis de Supinski, Martin Schulz, and Vincent W. Freeh. 2009. Adagio: Making DVS Practical for Complex HPC Applications. In *International Conference on Supercomputing*. Yorktown Heights, N.Y., USA.

[30] Amit Roy, Prasanna Balaprakash, Paul D Hovland, and Stefan M Wild. 2016. Exploiting performance portability in search algorithms for autotuning. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*. IEEE, 1535–1544.

[31] Yousef Saad. 1993. A Flexible Inner-Outer Preconditioned GMRES Algorithm. *SIAM Journal on Scientific Computing* 14 (1993), 461–469. Issue 2.

[32] Osman Sarood, Akhil Langer, Abhishek Gupta, and Laxmikant Kale. 2014. Maximizing throughput of overprovisioned HPC data centers under a strict power budget. In *Supercomputing*.

[33] Huazhe Zhang and Henry Hoffmann. 2016. Maximizing Performance Under a Power Cap: A Comparison of Hardware, Software, and Hybrid Techniques. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, 545–559.