Transferring Pareto Frontiers across Heterogeneous Hardware Environments

Pavel Valov
pvalov@uwaterloo.ca
University of Waterloo
Waterloo, Ontario, Canada

Jianmei Guo
guojianmei@gmail.com
Alibaba Group
Shanghai, China

Krzysztof Czarnecki
kczarnec@gsd.uwaterloo.ca
University of Waterloo
Waterloo, Ontario, Canada

ABSTRACT

Software systems provide user-relevant configuration options called features. Features affect functional and non-functional system properties, whereas selections of features represent system configurations. A subset of configuration space forms a Pareto frontier of optimal configurations in terms of multiple properties, from which a user can choose the best configuration for a particular scenario. However, when a well-studied system is redeployed on a different hardware, information about property value and the Pareto frontier might not apply. We investigate whether it is possible to transfer this information across heterogeneous hardware environments.

We propose a methodology for approximating and transferring Pareto frontiers of configurable systems across different hardware environments. We approximate a Pareto frontier by training an individual predictor model for each system property, and by aggregating predictions of each property into an approximated frontier. We transfer the approximated frontier across hardware by training a transfer model for each property, by applying it to a respective predictor, and by combining transferred properties into a frontier.

We evaluate our approach by modeling Pareto frontiers as binary classifiers that separate all system configurations into optimal and non-optimal ones. Thus we can assess quality of approximated and transferred frontiers using common statistical measures like sensitivity and specificity. We test our approach using five real-world software systems from the compression domain, while paying special attention to their performance. Evaluation results demonstrate that accuracy of approximated frontiers depends linearly on predictors' training sample sizes, whereas transferring introduces only minor additional error to a frontier even for small training sizes.

CCS CONCEPTS

• **Software and its engineering** → **Software performance**; • **Computing methodologies** → *Supervised learning by regression*; • **Mathematics of computing** → *Combinatorial optimization.*

KEYWORDS

Configurable software, Performance prediction, Pareto frontier, Regression trees, Linear regression, Pareto frontier transferring

ICPE '20, April 20–24, 2020, Edmonton, AB, Canada

ACM ISBN 978-1-4503-6991-6/20/04…$15.00
https://doi.org/10.1145/3358960.3379127

ACM Reference Format:
Pavel Valov, Jianmei Guo, and Krzysztof Czarnecki. 2020. Transferring Pareto Frontiers across Heterogeneous Hardware Environments. In *Proceedings of the 2020 ACM/SPEC International Conference on Performance Engineering (ICPE '20), April 20–24, 2020, Edmonton, AB, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3358960.3379127

1 INTRODUCTION

Software systems provide *configuration options* for end users to provide flexibility in meeting their requirements. Apart from having a direct influence on a system's functional behavior, configuration options usually influence non-functional properties, like runtime performance, memory consumption, and overall computational cost. System's configuration options that are available for tuning to end users of the system are called *features* [7]. Specific choice of feature values determines a system *configuration.* Thus for each system configuration a user can acquire a set of measured properties values. We regard a configuration as *Pareto optimal* one, if no other configuration improves one or more properties of the Pareto optimal configuration without degrading at least one other property. All Pareto optimal configurations of a system define a *Pareto frontier* of this system. In other words, a Pareto frontier is a set of system configurations each of which is optimal in its own specific way.

Identifying the Pareto frontier for a configurable system is challenging. First of all, building the exact Pareto frontier of a system's configuration space requires complete knowledge of properties' values for each configuration. This might be infeasible since: (1) a configuration space usually grows exponentially with the number of features, (2) benchmarking time for a single configuration might be high, (3) while a total benchmarking budget might be relatively low. Secondly, if a user has to deploy a system across heterogeneous hardware, then benchmarking results previously acquired on one hardware might be irrelevant for another hardware and new measurements of a configuration space might be required.

The first problem of incomplete benchmarking information can be solved by approximation of properties' values for a particular hardware environment. This topic has been thoroughly investigated for various use cases [7, 9, 15, 16, 31–33, 37, 38]. For example, some researchers investigate runtime performance prediction of configurable software systems based on small random samples of measured configurations, or in another data-efficient way [7, 8, 33].

The second problem of heterogeneous hardware environments can be solved by transferring gained knowledge about system's properties across hardware platforms. This topic has also gained attention in research community [2, 3, 9, 17, 18, 31, 32, 34]. For example, researchers employ machine learning techniques to transfer knowledge about system performance across a heterogeneous computational cluster or for a simulated robotic system [17, 18, 34].

In this work we propose a novel approach for transferring Pareto frontiers of optimal configurations of highly configurable software systems across heterogeneous hardware environments. The main goal of our research is to develop a pragmatic methodology that could be applied in real-world scenarios. This goal resulted in several major constraints that guided a development of our approach.

First of all, we do not assume that a user has any control over the sampling process of a system's configuration space and might be restricted to historically benchmarked data only. Because of that, to explore systems' configuration spaces in our experiments we employ *pseudo-random sampling* in order to imitate this worst-case scenario of impossibility to make any sampling choices by a user.

Secondly, we assume that a practitioner might have a limited information about actual system properties values, and it might not cover a configuration space entirely. Therefore, we analyze different sampling sizes in our experiments, in order to provide an assessment of how accurate predictors and transferrers might be.

Thirdly, we assume that a user can be working with a closed-source software and might not have any understanding about internal workings of a system, and won't be able to use this knowledge to improve predictors or transferrers. Consequently, our methodology regards analyzed software systems as 'black-boxes' that output particular properties' values, given a configuration and a workload.

Finally, our goal to make the approach practical imposes some restrictions on predictors and transferrers themselves. Users should be able to: (1) train these models using minimal amount of training data, (2) build and validate the models in a completely automatic fashion, (3) visualize the models in an intuitive way, in order to get additional insights from training data and to verify that the models really work. All these requirements forced us to investigate basic machine learning methods as candidate models for our methodology, such as linear regression and regression tree models.

Taking into the account all previously described requirements, we propose a practical black-box approach that uses machine learning methods for approximation and transferring of Pareto frontiers of configurations across heterogeneous hardware environments. This approach (1) builds a predictor model for each system property of interest, based on a sample of configurations measured on a source hardware platform, (2) combines predicted properties values of all configurations into an approximated Pareto frontier, (3) builds a transfer model for each system property of interest, based on a sample of configurations measured on both source and destination hardware, and (4) applies the transfer models to the approximated Pareto frontier to transfer it to the destination hardware.

To sum up, in this work we make the following contributions:

- We proposed an approach for approximation and transferring of Pareto frontiers of optimal configurations across heterogeneous hardware environments, described previously.
- We comprehensively benchmarked five different configurable software systems across a heterogeneous collection of 34 hardware environments based on Microsoft Azure cloud infrastructure, to acquire necessary data for our experiments.
- We implement the proposed approach and demonstrate its generality by evaluating it using the benchmarked software systems. We regard approximated and transferred Pareto frontiers as binary classifiers that categorize all configurations into Pareto optimal and non-optimal ones on a specified hardware. Thus we can assess quality of these frontiers by using classification evaluation measures (e.g. sensitivity, specificity, and Matthew's correlation coefficient) and by analyzing measures' trends with changes in predictors and transfer models. Our empirical results demonstrate that it is possible to achieve high accuracy of transferred Pareto frontiers, according to the classification measures and trends.

Source code and data to reproduce our experiments are available online at https://bitbucket.org/valovp/icpe2020.

2 EXAMPLE AND NOTATION

To formalize the problem of Pareto frontier approximation and transferring, we need to introduce necessary definitions and notations. *Configurable software system* is a system that provides *configuration options*, e.g. compression utilities, video codecs, compilers, etc. Configuration options influence *functional properties* (e.g. compression or encoding algorithm, compilation heuristic, etc.) and *non-functional* (e.g. performance, memory consumption, scalability, etc.) of the respective configurable software system. *Feature* is a configuration option that is pertinent to system consumers, e.g. developers, system administrators, expert users, etc. We denote a particular system feature by a binary variable $f_i \in \mathbb{B}$, where $\mathbb{B} = \{0, 1\}$, and all system's features by a set of variables $\mathbb{F} = \{f_1, f_2, \dots f_{N_f}\}$, where $N_f \in \mathbb{N}$ is a total number of features of the system. *Configuration* is a unique set of actually assigned values to all N_f features. We denote a configuration by $\mathbf{c}_i \in \mathbb{B}^{N_f}$, and all configurations by a set $\mathbb{C} = \{\mathbf{c}_1, \mathbf{c}_2, \dots \mathbf{c}_{N_\mathbf{c}}\}$, where $N_\mathbf{c} \in \mathbb{N}$ is a total number of valid configurations. Each system has a set of functional or non-functional properties that we denote by $\mathbb{P} = \{p_1, p_2, \dots p_{N_p}\}$, where $N_p \in \mathbb{N}$ is a total number of such properties.

We perform our study on various *hardware environments* that we denote by $h_k \in \mathbb{N}$ which together form a heterogeneous *hardware cluster* $\mathbb{H} = \{h_1, h_2, \dots, h_{N_h}\}$, where $N_h \in \mathbb{N}$ is a total number of hardware environments. Each property p_j from the set $\mathbb{P}$ is expected to vary when measured for the same configuration $\mathbf{c}_i$ across the cluster $\mathbb{H}$. Thus each configuration $\mathbf{c}_i$ has an actual measured property value $y_{\mathbf{c}_i, p_j, h_k}$ for each property p_j on each hardware h_k. We view properties as functions that map hardware environments and configurations to actual measured values:

$$p_j : \mathbb{B}^{N_f} \times \mathbb{H} \rightarrow \mathbb{R}$$
$$p_j(\mathbf{c}_i, h_k) = y_{\mathbf{c}_i, p_j, h_k} \quad (1)$$

All actual properties' values of a configuration $\mathbf{c}_i$ on a hardware h_k form a vector that we denote by $\mathbf{y}_{\mathbf{c}_i, \mathbb{P}, h_k}$:

$$\mathbf{y}_{\mathbf{c}_i, \mathbb{P}, h_k} = [\, y_{\mathbf{c}_i, p_1, h_k}, y_{\mathbf{c}_i, p_2, h_k}, \dots, y_{\mathbf{c}_i, p_{N_p}, h_k}] \quad (2)$$

Actual properties' values of all configurations $\mathbf{c}_i$ of a sample $\mathbb{C}_S \subset \mathbb{C}$ and of the whole population $\mathbb{C}$ on a hardware h_k form the corresponding sets $\mathbb{Y}_{\mathbb{C}_S, \mathbb{P}, h_k}$ and $\mathbb{Y}_{\mathbb{C}, \mathbb{P}, h_k}$:

$$\mathbb{Y}_{\mathbb{C}_S, \mathbb{P}, h_k} = \bigcup_{\mathbf{c}_i \in \mathbb{C}_S} \{\mathbf{y}_{\mathbf{c}_i, \mathbb{P}, h_k}\} \quad (3)$$

$$\mathbb{Y}_{\mathbb{C}, \mathbb{P}, h_k} = \bigcup_{\mathbf{c}_i \in \mathbb{C}} \{\mathbf{y}_{\mathbf{c}_i, \mathbb{P}, h_k}\} \quad (4)$$

Since our primary goal is to transfer a Pareto frontier across hardware environments, we need to be able to distinguish between them. We call an environment a *source hardware environment* if we use it to measure actual properties values of configurations, to train predictors for each system property, and to approximate a Pareto frontier. We call an environment a *destination hardware environment* if we use it to train transferrers for each system property, and if we transfer the approximated frontier to this environment. We denote source and destination environments by h_{src} and h_{dst} respectively.

Before we can define what Pareto frontier is, we need to introduce notions of *preference* and *domination* between different configurations. We assume that each property p_j has a *preferred direction of values* that can be inferred from a system domain, e.g. a compression software has a property 'compression rate' and higher values of this property are preferred to lower ones. For the sake of the example, let's pretend that for all properties in $\mathbb{P}$ higher values correspond to more preferred values. Then properties in $\mathbb{P}$ are in fact *utility functions* that describe a level of preference of a particular configuration. Thus we can denote that a property value $y_{\mathbf{c},p_j,h_k}$ *is preferred to* or *improves* a property value $y_{\mathbf{c}',p_j,h_k}$ simply by: $y_{\mathbf{c},p_j,h_k} > y_{\mathbf{c}',p_j,h_k}$. We say that a configuration $\mathbf{c}$ *dominates* a configuration $\mathbf{c}'$ on a hardware h_k, or $\mathbf{c} \succ^{h_k} \mathbf{c}'$, if $y_{\mathbf{c},p_j,h_k} > y_{\mathbf{c}',p_j,h_k}$ for some $p_j \in \mathbb{P}$ and $y_{\mathbf{c},p_j,h_k} \geq y_{\mathbf{c}',p_j,h_k}$ for all $p_j \in \mathbb{P}$.

Finally, we can introduce notions of *Pareto optimality* and *Pareto frontier*. We call a configuration $\mathbf{c}$ a *Pareto optimal* configuration on a hardware h_k, if there is no other configuration $\mathbf{c}'$ that improves one or more properties values of $\mathbf{c}$ without worsening at least one other property value, i.e. $\mathbf{c}$ is not dominated by any other configuration $\mathbf{c}'$. More formally, $\mathbf{c}$ is a *Pareto optimal* configuration on a hardware h_k, if there is no other $\mathbf{c}'$ such that $y_{\mathbf{c}',p_j,h_k} > y_{\mathbf{c},p_j,h_k}$ for some $p_j \in \mathbb{P}$ and $y_{\mathbf{c}',p_j,h_k} \geq y_{\mathbf{c},p_j,h_k}$ for all $p_j \in \mathbb{P}$. *Pareto frontier* of a system on a given hardware h_k is a set of all Pareto optimal configurations on this hardware that we denote by $\mathbb{C}^{PF}_{h_k}$. In other words, Pareto frontier is a set of configurations that are not strictly dominated by any other configuration, or more formally:

$$\mathbb{C}^{PF}_{h_k} = \{\mathbf{c} \in \mathbb{C} : \{\mathbf{c}' \in \mathbb{C} : \mathbf{c}' \succ^{h_k} \mathbf{c}, \mathbf{c}' \neq \mathbf{c}\} = \emptyset\} \tag{5}$$

We have introduced all necessary definitions and notations, thus we can summarize the problem statement in proper terms. We deploy a configurable software system (e.g. XZ) on a hardware platform h_{src}. We select a random training sample of configurations $\mathbb{C}_{trn} \subset \mathbb{C}$ and for each configuration $\mathbf{c}_i \in \mathbb{C}_{trn}$ we acquire an actual value $y_{\mathbf{c}_i,p_j,h_{src}}$ of each property $p_j \in \mathbb{P}$ thus forming a set of measured values $\mathbb{Y}_{\mathbb{C}_{trn},\mathbb{P},h_{src}}$. Sets $\mathbb{C}_{trn}$ and $\mathbb{Y}_{\mathbb{C}_{trn},\mathbb{P},h_{src}}$ together form a sample of measured configurations $\mathbb{S}_{\mathbb{C}_{trn},\mathbb{P},h_{src}}$ on a hardware platform h_{src}. Our goal is to build an approximated Pareto frontier $\widehat{\mathbb{C}}^{PF}_{h_{src}}$ based on the sample $\mathbb{S}_{\mathbb{C}_{trn},\mathbb{P},h_{src}}$ and to transfer this frontier to all other hardware $\mathbb{H} \setminus \{h_{src}\}$. Thus we can use the Pareto frontier as a binary classifier that separates all configurations into optimal and non-optimal ones, and based on a small random sample of measured configurations $\mathbb{S}_{\mathbb{C}_{trn},\mathbb{P},h_{src}}$ classify all configurations $\mathbb{C}$ on all hardware platforms $\mathbb{H}$.

3 TRANSFERRING PROCESS

The process of transferring Pareto frontiers can be divided into several main steps: (1) training a predictor for each studied system property (2) building an approximated Pareto frontier using trained predictors (3) training a transferrer for each studied system property (4) transferring the approximated Pareto frontier across hardware platforms using transferrers.

3.1 Training Property Prediction Models

The process of training a property prediction model can be separated into several steps: (1) selecting a data sampling method (2) sampling training data (3) selecting a property predictor model (4) training a property predictor (5) selecting an evaluation metric and a validation strategy for the trained predictor.

First of all, we need to select a method for sampling training data that will be used for training properties' predictors. There are many different ways in which one can generate a sample of training configurations $\mathbb{C}_{trn}$ from a system's configuration space: pseudo-random sampling, quasi-random sampling, experimental design techniques, and custom sampling heuristics. In the current work we use pseudo-random sampling of configurations in order to mimic the worst-case scenario, when a practitioner does not have any control over selection of training data and available measured configurations might appear completely random. The same paradigm was used in the previous work on performance prediction of configurable software systems by Guo et al. [7] and Valov et al. [33]. However, in a practical scenario where a practitioner has control over a sampling process, we would advise to use more sophisticated sampling methods like quasi-random sampling and experimental design techniques, since they provide a much more even coverage of the configuration space and should improve quality of a trained predictor. We leave a comparison of different sampling methods for property prediction models' training data for a future work.

Secondly, we need to perform actual data sampling. In the previous work [7, 33] researchers used sampling sizes that are multiples of the number of features N_f of the respective system. During evaluation of our approach we've tried all possible sampling sizes in range $[\,2, N_{\mathbb{C}}-1\,]$ in order to provide smoother trends for presenting regression and classification measures (see Section 4 for details).

Thirdly, we need to choose which model to use as property predictors. During preliminary evaluation of our approach we tried two different regression models: regression trees and random forest. We selected these models as candidates for our solution since they have already showed good results for prediction of configurable systems properties [7, 31–33, 37]. Our preliminary experiments demonstrated that unpruned regression tree models provided better prediction results than random forest models. We came to a conclusion that this happens because we work with relatively small training sample sizes. Although random forest model also generates unpruned regression trees to average upon, it trains them using observations resampled with replacement from an original sample provided to the random forest model itself. This approach helps to avoid overfitting when training samples are relatively large, but when training samples are tiny, each individual tree ends up loosing a lot of information. Therefore, in our case a single unpruned regression tree (but trained on a full training sample) will outperform an ensemble of trees that were trained on resampled data.

Fourthly, we have to train property predictors. The only thing left to do, is to select a parameter tuning strategy. We work with

unpruned regression trees, i.e. we 'grow' our tree models to a maximal possible size, when each tree has only one observation in each leaf and all available features are used in construction of the tree. We can regard regression tree predictor as a function RT, that is generated by a *fitting function fitRT*. Fitting function, given a small random training sample of configurations $\mathbb{C}_{trn}$ and their measured values $\mathbf{Y}_{\mathbb{C}_{trn},p_j,h_k}$ of a property p_j on a hardware h_k, produces corresponding predictors for the property p_j on the hardware h_k:

$$fitRT(\mathbb{C}_{trn}, \mathbf{Y}_{\mathbb{C}_{trn},p_j,h_k}, \mathbf{c}_i) = RT_{\mathbb{C}_{trn},p_j,h_k} \tag{6}$$

Predictor $RT_{\mathbb{C}_{trn},p_j,h_k}$, given a sample of configurations $\mathbb{C}_S$, can predict their values for the property p_j on the hardware h_k:

$$RT_{\mathbb{C}_{trn},p_j,h_k}(\mathbb{C}_S) = \widehat{\mathbf{Y}}_{\mathbb{C}_S,p_j,h_k} \tag{7}$$

Finally, we need to select an evaluation metric and a validation strategy for trained predictors. We have selected *mean absolute percentage error (MAPE)* as a metric for evaluation of RT models since it provides a simple and intuitive measure that is robust to outliers. *MAPE* can be thought of as a mean of multiple *absolute percentage errors (APE)*. If *APE* can be expressed as a function that consumes actual $y_{\mathbf{c}_i,p_j,h_k}$ and predicted $\hat{y}_{\mathbf{c}_i,p_j,h_k}$ property values:

$$APE(y_{\mathbf{c}_i,p_j,h_k}, \hat{y}_{\mathbf{c}_i,p_j,h_k}) = \frac{|y_{\mathbf{c}_i,p_j,h_k} - \hat{y}_{\mathbf{c}_i,p_j,h_k}|}{y_{\mathbf{c}_i,p_j,h_k}} \times 100\% \tag{8}$$

then *MAPE* can be expressed as a function that consumes sets of actual $\mathbf{Y}_{\mathbb{C}_S,p_j,h_k}$ and predicted $\widehat{\mathbf{Y}}_{\mathbb{C}_S,p_j,h_k}$ property values:

$$MAPE(\mathbf{Y}_{\mathbb{C}_S,p_j,h_k}, \widehat{\mathbf{Y}}_{\mathbb{C}_S,p_j,h_k}) = \frac{\sum_{i=1}^{N_{\mathbb{C}_S}} APE(y_{\mathbf{c}_i,p_j,h_k}, \hat{y}_{\mathbf{c}_i,p_j,h_k})}{N_{\mathbb{C}_S}} \tag{9}$$

where $N_{\mathbb{C}_S}$ is a number of configurations in the set $\mathbb{C}_S$.

We have selected *leave-one-out cross-validation (LOOCV)* as a validation strategy for predictors, since it is especially suitable in a scenario, when the cost of measuring properties for a single configuration is very high and a practitioner wants to minimize the measurement effort. Imagine that a practitioner acquired a sample of configurations $\mathbb{C}_S$ of size $N_{\mathbb{C}_S}$ along with their measured values $\mathbf{Y}_{\mathbb{C}_S,p_j,h_k}$ of a property p_j on a hardware h_k. *LOOCV* is going to generate out of $\mathbb{C}_S$ sample: $N_{\mathbb{C}_S}$ testing samples $\mathbb{C}^i_{tst} = \{\mathbf{c}_i\}$ that consist of a single configuration and $N_{\mathbb{C}_S}$ training samples $\mathbb{C}^i_{trn} = \mathbb{C}_S \setminus \{\mathbf{c}_i\}$ that consist of all other configurations. Thus *LOOCV* generates $N_{\mathbb{C}_S}$ pairs of training and testing samples $\{(\mathbb{C}^1_{trn}, \mathbb{C}^1_{tst}), \ldots, (\mathbb{C}^{N_{\mathbb{C}_S}}_{trn}, \mathbb{C}^{N_{\mathbb{C}_S}}_{tst})\}$. Then by training a predictor, e.g. a regression tree RT, on each training sample $\mathbb{C}^i_{trn}$ and by testing the predictor on the corresponding testing sample $\mathbb{C}^i_{tst}$, we can acquire a column vector of predicted property values:

$$\widehat{\mathbf{Y}}_{\mathbb{C}_S,p_j,h_k} = \begin{bmatrix} RT_{\mathbb{C}^1_{trn},p_j,h_k}(\mathbb{C}^1_{tst}) \\ RT_{\mathbb{C}^2_{trn},p_j,h_k}(\mathbb{C}^2_{tst}) \\ \vdots \\ RT_{\mathbb{C}^{N_{\mathbb{C}_S}}_{trn},p_j,h_k}(\mathbb{C}^{N_{\mathbb{C}_S}}_{tst}) \end{bmatrix} \tag{10}$$

We can use $\mathbf{Y}_{\mathbb{C}_S,p_j,h_k}$ and $\widehat{\mathbf{Y}}_{\mathbb{C}_S,p_j,h_k}$ to calculate *MAPE* in order to assess the predictor quality.

3.2 Building an Approximated Frontier

The process of building an approximated Pareto frontier consists of the following main steps: (1) approximating all system properties $\mathbb{P}$ for all configurations $\mathbb{C}$ using trained predictors from Section 3.1, (2) calculating a Pareto frontier $\widehat{\mathbb{C}}^{PF}_{h_{src}}$ based on the approximated configurations' properties.

First of all, we need to acquire all properties' values $\mathbb{P}$ for all configurations $\mathbb{C}$, i.e. $\widehat{\mathbf{Y}}_{\mathbb{C},\mathbb{P},h_{src}}$, to assess which configurations are in fact Pareto optimal on a hardware h_{src}, and thus to generate a Pareto frontier. However, we know properties' values only for a small training sample of configurations $\mathbb{C}_{trn}$, i.e. $\mathbf{Y}_{\mathbb{C}_{trn},\mathbb{P},h_{src}}$. Therefore, we need to approximate the properties' values for remaining unmeasured configurations $\mathbb{C}_{rem} = \mathbb{C} \setminus \mathbb{C}_{trn}$, i.e. acquire $\widehat{\mathbf{Y}}_{\mathbb{C}_{rem},\mathbb{P},h_{src}}$. We can obtain $\widehat{\mathbf{Y}}_{\mathbb{C}_{rem},\mathbb{P},h_{src}}$ by systematically predicting all properties in $\mathbb{P}$ for $\mathbb{C}_{rem}$ using corresponding predictors and combining resulting column vectors into a matrix:

$$\widehat{\mathbf{Y}}_{\mathbb{C}_{rem},\mathbb{P},h_{src}} = \left[RT_{\mathbb{C}_{trn},p_1,h_{src}}(\mathbb{C}_{rem}), \cdots, RT_{\mathbb{C}_{trn},p_{Np},h_{src}}(\mathbb{C}_{rem})\right] \tag{11}$$

We can obtain $\widehat{\mathbf{Y}}_{\mathbb{C},\mathbb{P},h_{src}}$ by combining matrices of measured and approximated properties' values:

$$\widehat{\mathbf{Y}}_{\mathbb{C},\mathbb{P},h_{src}} = \begin{bmatrix} \mathbf{Y}_{\mathbb{C}_{trn},\mathbb{P},h_{src}} \\ \widehat{\mathbf{Y}}_{\mathbb{C}_{rem},\mathbb{P},h_{src}} \end{bmatrix} \tag{12}$$

Finally, based on the resulting matrix of all properties' values $\widehat{\mathbf{Y}}_{\mathbb{C},\mathbb{P},h_{src}}$ we can build the approximated Pareto frontier $\widehat{\mathbb{C}}^{PF}_{h_{src}}$ using any of the known algorithms for exact frontier construction:

$$\widehat{\mathbb{C}}^{PF}_{h_{src}} = PF(\mathbb{C}, \widehat{\mathbf{Y}}_{\mathbb{C},\mathbb{P},h_{src}}) \tag{13}$$

3.3 Training Property Transferring Models

The process of training a property transferring model can be separated into the following steps: (1) sampling training data, (2) selecting a property transferrer model, (3) training a property transferrer, (4) selecting an evaluation metric and validation strategy for the trained transferrer.

First of all, to train a property transferrer we need a training sample of configurations $\mathbb{C}_{both}$ that are measured on both source h_{src} and destination h_{dst} hardware. In Section 3.1 we've already defined a procedure for acquiring a training sample $\mathbb{C}_{trn}$ measured on h_{src}. Naturally, we can measure configuration from $\mathbb{C}_{trn}$ on h_{dst} as well, thus forming the necessary sample $\mathbb{C}_{both} \subseteq \mathbb{C}_{trn}$.

Secondly, we have to select a model to be used as a property transferrer. During our preliminary evaluation, we have tested multiple machine learning models as property transferrers, and two of them provided the best results overall: simple linear regression model (*SLR*) and unpruned regression tree model (*RT*), discussed previously. We used *SLR*, since it has been already studied in the previous work [18, 34] and has demonstrated good results. However, during preliminary evaluation we noticed that *SLR* underperforms for some studied systems, and especially multithreaded ones. On the contrary, *RT* demonstrated better performance overall and significantly better performance for parallel systems. Therefore, unlike previous work [18, 34] we recommend using *RT* as transferrers, especially when working with multithreaded software.

Thirdly, we have to train the selected transferring models using the measured training sample $\mathbb{C}_{both}$. *SLR* model is a line that is fit to the training data using classical *ordinary least squares (OLS)* methodology. OLS performs fitting by minimizing the sum of squared differences between configurations in $\mathbb{C}_{both}$ and the linear model. Thus, for each system property p_j we can acquire a corresponding linear transferrer that given property values on a source hardware will produce corresponding property values on a destination hardware:

$$\begin{aligned} SLR_{p_j}(\widehat{\mathbf{Y}}_{\mathbb{C},p_j,h_{src}}) &= \alpha + \beta \times \widehat{\mathbf{Y}}_{\mathbb{C},p_j,h_{src}} \\ &= \widehat{\mathbf{Y}}_{\mathbb{C},p_j,h_{dst}} \end{aligned} \tag{14}$$

SLR training process is completely automatic and doesn't require any parameter tuning. Training process for unpruned regression trees was discussed previously (see Section 3.1).

Finally, we need to select an evaluation metric and a validation strategy for trained property transferring models. Since transferrers might be built using even smaller samples than property prediction models, the most practical validation strategy would still be leave-one-out cross-validation (*LOOCV*). As for an evaluation metric, we again recommend using mean absolute relative error (*MAPE*).

3.4 Transferring a Pareto Frontier

During the previous steps of the process (see Section 3.1 – Section 3.3) we have: (1) selected a source h_{src} and a destination h_{dst} hardware environments, (2) sampled training configurations $\mathbb{C}_{trn}$ and measured all their properties' values $\mathbf{Y}_{\mathbb{C}_{trn},\mathbb{P},h_{src}}$ on h_{src}, (3) trained predictors RT_{p_j} for each property $p_j \in \mathbb{P}$ on h_{src}, (4) predicted all properties' values on h_{src} and acquired $\widehat{\mathbf{Y}}_{\mathbb{C},\mathbb{P},h_{src}}$, (5) built an approximated Pareto frontier $\widehat{\mathbb{C}}^{PF}_{h_{src}}$ based on $\widehat{\mathbf{Y}}_{\mathbb{C},\mathbb{P},h_{src}}$, (6) sampled configurations $\mathbb{C}_{both}$ and measured all their properties' values on both h_{src} and h_{dst}, (7) trained transferrers for each property $p_j \in \mathbb{P}$ to transfer properties' values from h_{src} to h_{dst}.

To transfer the Pareto frontier from h_{src} to h_{dst} we need to perform two steps: (1) approximate all properties $\mathbb{P}$ for all configurations $\mathbb{C}$ on h_{dst}, i.e. acquire $\widehat{\mathbf{Y}}_{\mathbb{C},\mathbb{P},h_{dst}}$, and (2) calculate the transferred Pareto frontier $\widehat{\mathbb{C}}^{PF}_{h_{dst}}$ on h_{dst} based on $\widehat{\mathbf{Y}}_{\mathbb{C},\mathbb{P},h_{dst}}$.

First of all, we obtain $\widehat{\mathbf{Y}}_{\mathbb{C},\mathbb{P},h_{dst}}$ by systematically transferring column vectors of properties' values from h_{src} to h_{dst}, using previously trained property transferrers:

$$\begin{aligned} &\widehat{\mathbf{Y}}_{\mathbb{C},\mathbb{P},h_{dst}} = \\ &\left[RT_{p_1}(\widehat{\mathbf{Y}}_{\mathbb{C},p_1,h_{dst}}), \cdots, RT_{p_{N_p}}(\widehat{\mathbf{Y}}_{\mathbb{C},p_{N_p},h_{dst}})\right] \end{aligned} \tag{15}$$

Then, we can calculate the approximated Pareto frontier $\widehat{\mathbb{C}}^{PF}_{h_{dst}}$ using any known algorithm for exact Pareto frontier construction:

$$\widehat{\mathbb{C}}^{PF}_{h_{dst}} = PF(\mathbb{C}, \widehat{\mathbf{Y}}_{\mathbb{C},\mathbb{P},h_{dst}}) \tag{16}$$

4 PROCESS EVALUATION

To comprehensively evaluate the proposed process of Pareto frontier approximation and transferring, we have formulated the following research questions:

- RQ1 How accurate are properties' prediction models? (Section 4.2.1)
- RQ2 How accurate are properties' transferring models? (Section 4.2.1)
- RQ3 How accurate are approximated Pareto frontiers $\widehat{\mathbb{C}}^{PF}_{h_{src}}$ compared to actual Pareto frontiers $\mathbb{C}^{PF}_{h_{src}}$ on h_{src}? (Section 4.2.3)
- RQ4 How accurate are transferred Pareto frontiers $\widehat{\mathbb{C}}^{PF}_{h_{dst}}$ compared to actual Pareto frontiers $\mathbb{C}^{PF}_{h_{dst}}$ on h_{dst}? (Section 4.2.4)

4.1 Experimental Setup

4.1.1 Subject Hardware Environments. To enhance external validity of our work, we had to perform our experiments on a wide variety of hardware environments. Moreover, we wanted to run our experiments on real-world production hardware environments that could be used by other research or development teams, what would make our research even more applicable to other practitioners. We came to a conclusion that the best hardware choice for our experiments would be a public enterprise-level cloud computing solution that provides server infrastructure as a service (IaaS). Because of that, we acquired access to Microsoft Azure cloud computing service.

Microsoft Azure is a cloud computing service that provides infrastructure as a service (IaaS), platform as a service (PaaS), and software as a service (SaaS). Microsoft provided Azure Sponsorship for our team, thus allowing us to use Azure infrastructure to run our experiments. Azure provides cloud infrastructure through a set of dedicated international data centers. After performing a thorough analysis of all virtual machines on all data centres that were available for our sponsorship, we selected 34 virtual machines that had different CPU model, RAM size, etc. We provide a summary of all selected virtual machines in Table 1 and a detailed summary of unique CPUs available on these machines in Table 2. All virtual machines ran Linux Ubuntu Xenial 16.04 LTS operating system.

4.1.2 Subject Software Systems. We build, approximate, and transfer Pareto frontiers for five different software systems: BZIP2 [26], GZIP [5], XZ [29], FLAC [19], x264 [35]. BZIP2 is a general-purpose data compression program which utilizes the Burrows-Wheeler algorithm. GZIP is a general-purpose data compression utility which employs the DEFLATE lossless compression algorithm. XZ is a multi-threaded general-purpose data compression software which uses the LZMA2 lossless compression algorithm. FLAC (Free Lossless Audio Codec) is an audio compression software that uses homonymous lossless audio coding format. x264 is a video encoding software that uses lossy H.264/MPEG-4 AVC format. We focus on systems from compression and multimedia transformation domains, since these systems are intuitive, have large userbase, and provide a variety of features. Moreover, these systems share common properties, what allows straightforward comparison of how well our approach works for different use cases.

We performed a comprehensive benchmarking of each selected software system. For each software system we selected a benchmark in order to test the generated configurations upon. We benchmarked BZIP2, GZIP, and XZ using *large text compression benchmark* [22], which represents first 10^9 bytes of Wikipedia XML archive. We benchmarked FLAC using *Ghosts I-IV* [23] music album of a band 'Nine Inch Nails', that contains 36 tracks of improvisation music, released under Creative Commons license. We benchmarked x264 using a trailer of *Sintel* [1] open-content film created and released under Creative Commons license by Blender Foundation.

Table 1: Summary of all Azure-based virtual machines; CRS – number of CPU cores that are specifically allocated to the virtual machine, RAM – amount of RAM allocated to the VM (GiB), RPC – amount of RAM allocated per CPU core of the VM (GiB), STR – amount of storage allocated to the VM (GiB), STP – storage type allocated to the VM

General Server Info			Server Architecture						Benchmarked Systems				
Name	**Azure Type**	**Deployment Region**	**CPU Model Name**	**CRS**	**RAM**	**RPC**	**STR**	**STP**	**BZIP2**	**GZIP**	**XZ**	**FLAC**	**X264**
BscA0-4171HE	BasicA0	South Brazil	AMD Opteron 4171 HE	1	0.75	0.75	20	HDD		✓			✓
BscA0-2660	BasicA0	East Japan	Intel Xeon E5-2660	1	0.75	0.75	20	HDD	✓	✓		✓	✓
BscA0-2673v3	BasicA0	West US	Intel Xeon E5-2673 v3	1	0.75	0.75	20	HDD	✓	✓		✓	✓
BscA1-4171HE	BasicA1	South Brazil	AMD Opteron 4171 HE	1	1.75	1.75	40	HDD	✓	✓		✓	✓
BscA1-2660	BasicA1	East Japan	Intel Xeon E5-2660	1	1.75	1.75	40	HDD	✓	✓	✓	✓	✓
BscA1-2673v3	BasicA1	West US	Intel Xeon E5-2673 v3	1	1.75	1.75	40	HDD	✓	✓	✓	✓	✓
BscA2-4171HE	BasicA2	South Brazil	AMD Opteron 4171 HE	2	3.5	1.75	60	HDD	✓	✓		✓	✓
BscA2-2660	BasicA2	East Japan	Intel Xeon E5-2660	2	3.5	1.75	60	HDD	✓	✓	✓	✓	✓
BscA2-2673v3	BasicA2	West US	Intel Xeon E5-2673 v3	2	3.5	1.75	60	HDD	✓	✓	✓	✓	✓
StdA0-2673v3	StandardA0	Central Canada	Intel Xeon E5-2673 v3	1	0.75	0.75	20	SSD	✓	✓		✓	✓
StdA0-2660	StandardA0	East Japan	Intel Xeon E5-2660	1	0.75	0.75	20	SSD	✓	✓		✓	✓
StdA1-2673v3	StandardA1	Central Canada	Intel Xeon E5-2673 v3	1	1.75	1.75	70	SSD	✓	✓	✓	✓	✓
StdA1-2660	StandardA1	East Japan	Intel Xeon E5-2660	1	1.75	1.75	70	SSD	✓	✓		✓	✓
StdA1v2-2660	StandardA1 v2	South Central US	Intel Xeon E5-2660	1	2	2	10	SSD	✓		✓	✓	✓
StdA1v2-2673v3	StandardA1 v2	West Central US	Intel Xeon E5-2673 v3	1	2	2	10	SSD	✓	✓	✓	✓	✓
StdA2-2673v3	StandardA2	Central Canada	Intel Xeon E5-2673 v3	2	3.5	1.75	135	SSD	✓	✓	✓	✓	✓
StdA2-2660	StandardA2	East Japan	Intel Xeon E5-2660	2	3.5	1.75	135	SSD	✓	✓	✓	✓	✓
StdA2v2-2660	StandardA2 v2	South Central US	Intel Xeon E5-2673 v3	2	4	2	20	SSD	✓		✓	✓	✓
StdA2v2-2673v3	StandardA2 v2	West Central US	Intel Xeon E5-2673 v3	2	4	2	20	SSD	✓	✓	✓	✓	✓
StdD1-2660	StandardD1	South East Australia	Intel Xeon E5-2660	1	3.5	3.5	50	SSD	✓	✓	✓	✓	✓
StdD1v2-2673v3	StandardD1 v2	Central US	Intel Xeon E5-2673 v3	1	3.5	3.5	50	SSD	✓	✓	✓	✓	✓
StdD1v2-2673v4	StandardD1 v2	South India	Intel Xeon E5-2673 v4	1	3.5	3.5	50	SSD	✓				
StdD2-2660	StandardD2	South East Australia	Intel Xeon E5-2660	2	7	3.5	100	SSD	✓	✓	✓	✓	✓
StdD2v2-2673v3	StandardD2 v2	Central US	Intel Xeon E5-2673 v3	2	7	3.5	100	SSD	✓	✓	✓	✓	✓
StdD2v2-2673v4	StandardD2 v2	South India	Intel Xeon E5-2673 v4	2	7	3.5	100	SSD	✓				
StdD2v3-2673v4	StandardD2 v3	South East Australia	Intel Xeon E5-2673 v4	2	8	4	50	SSD	✓	✓	✓	✓	✓
StdD2v3-2673v3	StandardD2 v3	South East Asia	Intel Xeon E5-2673 v3	2	8	4	50	SSD	✓	✓	✓	✓	✓
StdE2v3-2673v4	StandardE2 v3	West Europe	Intel Xeon E5-2673 v4	2	16	8	50	SSD	✓	✓	✓	✓	✓
StdF1-2673v3	StandardF1	East US	Intel Xeon E5-2673 v3	1	2	2	16	SSD	✓	✓	✓	✓	✓
StdF1-2673v4	StandardF1	South India	Intel Xeon E5-2673 v4	1	2	2	16	SSD	✓				
StdF2-2673v3	StandardF2	East US	Intel Xeon E5-2673 v3	2	4	2	32	SSD	✓	✓	✓	✓	✓
StdF2-2673v4	StandardF2	South India	Intel Xeon E5-2673 v4	2	4	2	32	SSD	✓				
StdF2sv2-8168	StandardF2 v2	West US 2	Intel Xeon Platinum 8168	2	4	2	16	SSD	✓	✓	✓	✓	✓
StdG1-2698Bv3	StandardG1	East US 2	Intel Xeon E5-2698B v3	2	28	14	384	SSD	✓	✓	✓	✓	✓

Table 2: Summary of all CPUs used in the experiment; TCH – technology node (nm), FRQ – CPU core frequency (MHz), FBS – front-side bus frequency (MHz), CM – clock multiplier, CRS – number of CPU cores, TRD – number of threads, L1i – Level 1 instruction cache size (KB), L1d – Level 1 data cache size (KB), L2 – Level 2 cache size, L3 – Level 3 cache size, PMC – PassMark CPU benchmark score (higher is better), PMT – PassMark single thread benchmark score (higher is better), PMR – PassMark overall CPU rank in PassMark database (lower is better)

General		Architecture							Caches per CPU				Caches per core			PassMark Scores		
CPU Model Name	**First Seen**	**Type**	**TCH**	**FRQ**	**FBS**	**CM**	**CRS**	**TRD**	**L1i**	**L1d**	**L2**	**L3**	**L1i**	**L1d**	**L2**	**PMC**	**PMT**	**PMR**
AMD Opteron 4171 HE	Q4 2010	K10	45	2100	3200		6	6	384	384	3072	6144	64	64	512	**3664**[1]	**732**[1]	**1147**[1]
Intel Xeon E5-2660	Q2 2012	Sandy Bridge	32	2200	4000	22	8	16	256	256	2048	20480	32	32	256	**11048**	**1387**	**265**
Intel Xeon E5-2673 v3	Q2 2015	Haswell	22	2400	4800	24	12	24	384	384	3072	30720	32	32	256	**16383**	**1666**	**116**
Intel Xeon E5-2698B v3	Q2 2014	Haswell	22	2000	4800	20	16	32	512	512	4096	40960	32	32	256	**21042**[2]	**1846**[2]	**51**[2]
Intel Xeon E5-2673 v4	Q4 2016	Broadwell	14	2300	4800	23	20	40	640	640	5120	51200	32	32	256	**21474**	**1792**	**46**
Intel Xeon Platinum 8168	Q4 2017	Skylake	14	2700	5200	27	24	48	768	768	24576	33792	32	32	1024	**29131**	**2073**	**3**

[1] PassMark Scores are provided for AMD Opteron 4170 HE
[2] PassMark Scores are provided for Intel Xeon E5-2698 v3

Figure 1: *Compression time* metric distributions for each software system. Each line represents the metric's distribution on a particular hardware.

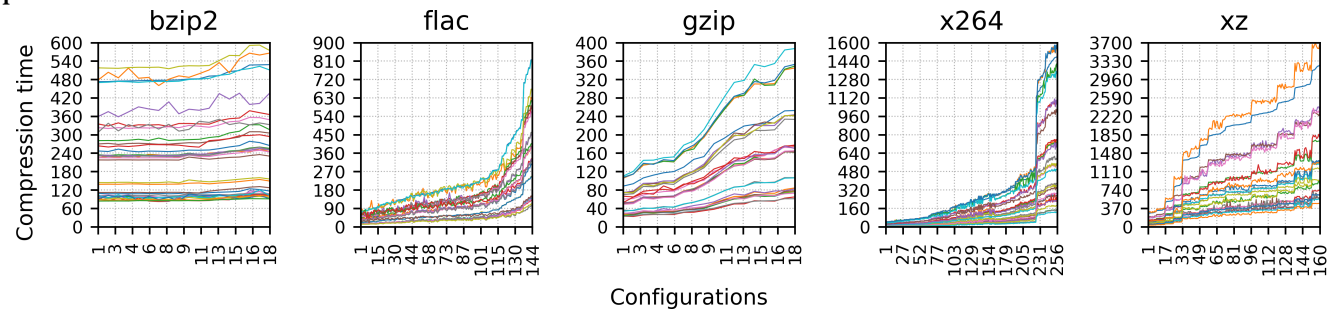

Figure 2: MAPE error distributions when predicting *compression time* metric using *regression trees* as predictors. Each line represents the error's distribution on a particular hardware.

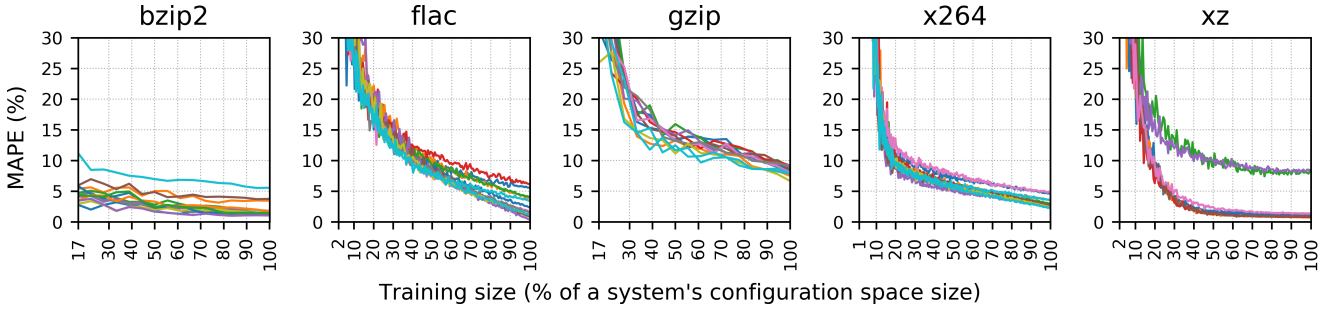

Figure 3: MAPE error distributions when transferring *compression time* metric using *using linear models* as transferrers. Each line represents the error's distribution on a particular hardware.

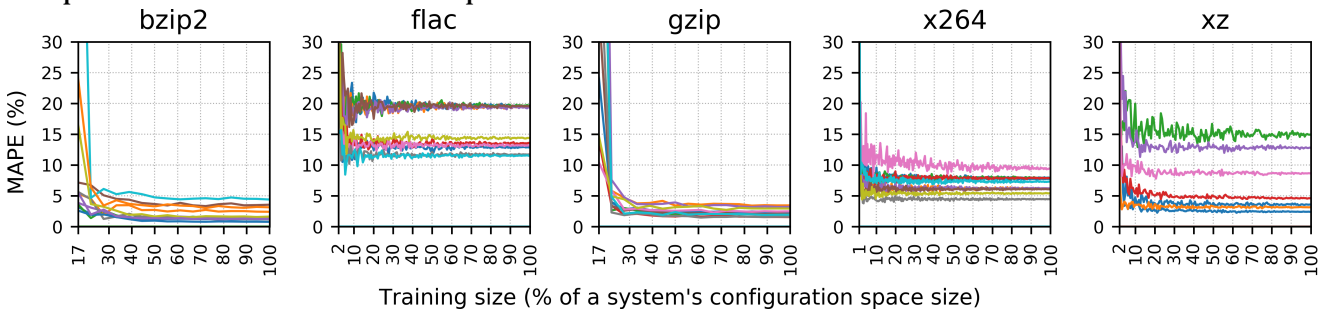

Figure 4: MAPE error distributions when transferring *compression time* metric using *regression trees* as transferrers. Each line represents the error's distribution on a particular hardware.

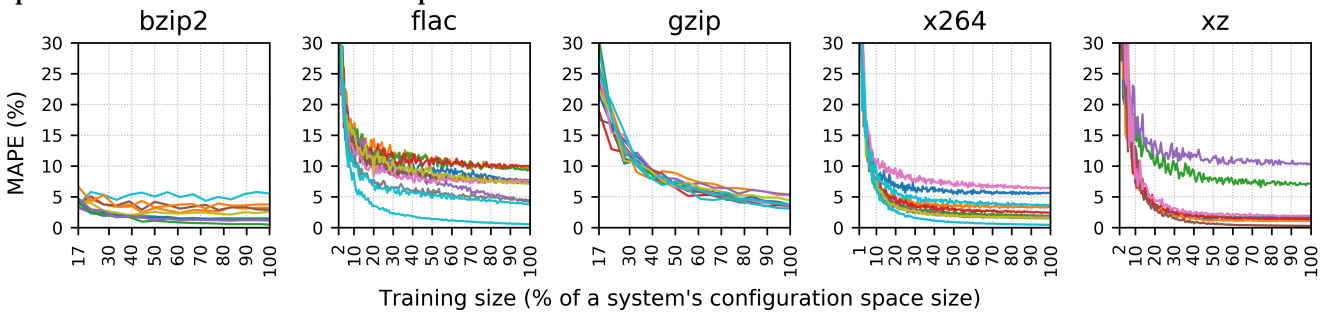

Figure 5: Classification measures' distributions, applied to evaluation of Pareto frontiers, *approximated using regression trees*. Each line represents a particular measure: TPR (green), TNR (red), PPV (blue), NPV (orange), MCC (violet).

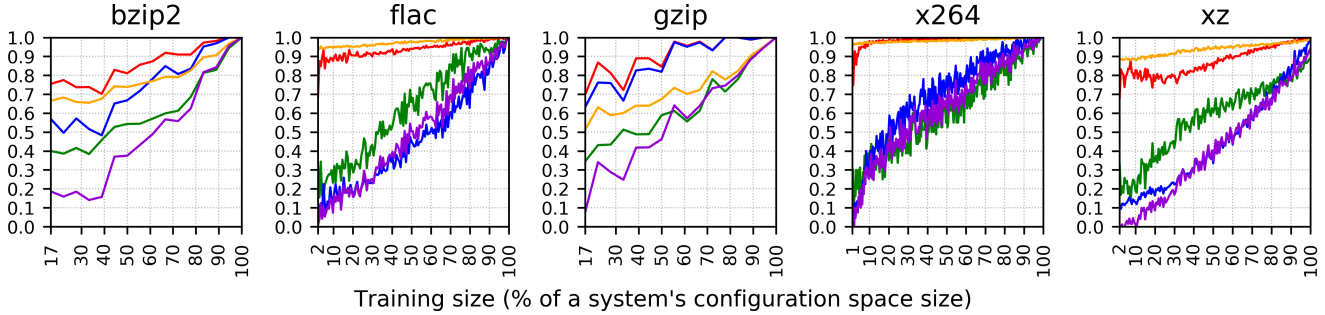

Figure 6: Classification measures' distributions, applied to evaluation of Pareto frontiers, *transferred using linear models*. Each line represents a particular measure: TPR (green), TNR (red), PPV (blue), NPV (orange), MCC (violet).

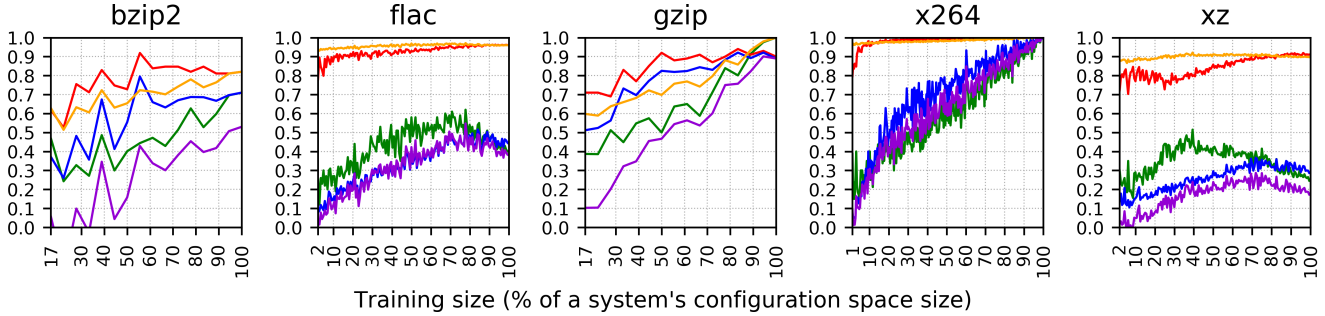

Figure 7: Classification measures' distributions, applied to evaluation of Pareto frontiers, *transferred using regression trees*. Each line represents a particular measure: TPR (green), TNR (red), PPV (blue), NPV (orange), MCC (violet).

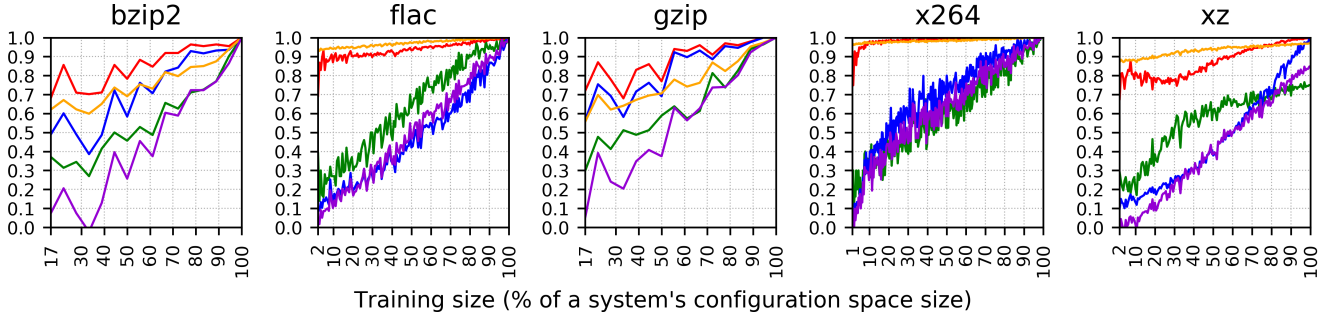

Figure 8: Distributions of deltas between classification measures of approximated and transferred (using regression trees) Pareto frontiers. Each line represents a particular measure: TPR (green), TNR (red), PPV (blue), NPV (orange), MCC (violet).

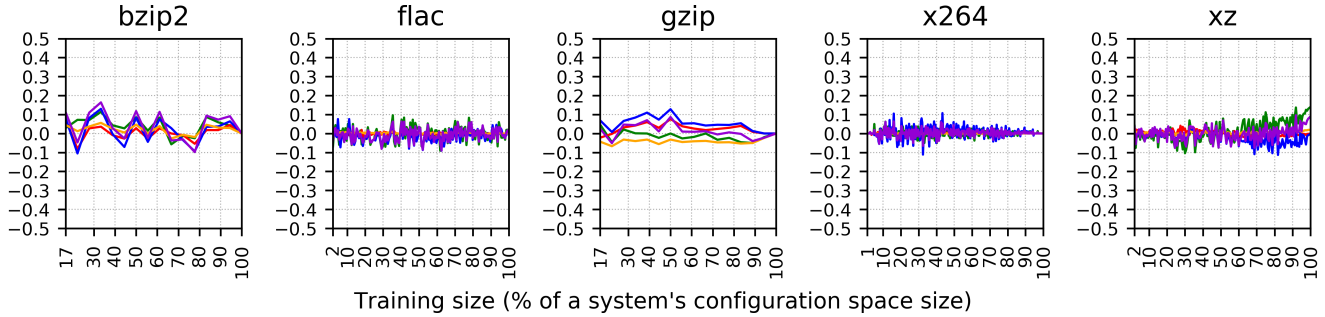

Table 3: Summary of benchmarked configurable software systems; N_f – Number of varied system features; NC – Number of benchmarked system configurations; NS – Number of servers on which systems were benchmarked.

Name	N_f	NC	NS
BZIP2	2	18	33
GZIP	3	36	28
XZ	4	160	27
FLAC	5	144	29
x264	8	256	30

For each software system we selected a set of configurable features that had demonstrated the strongest influence on studied systems' metrics during our preliminary experiments. We vary two different BZIP2 features: *n* and *small* (see [27] for description). We vary three different GZIP features: *n*, *rsyncable*, and *synchronous* [6]. We vary four different XZ features: *n*, *no-sparse*, *extreme*, *check* [30]. We vary five different FLAC features: *n*, *verify*, *lax*, *replay-gain*, and *p* [20]. We analyze eight different X264 features: *b-adapt*, *me*, *no-mbtree*, *no-scenecut*, *rc-lookahead*, *ref*, *subme*, and *trellis* [36].

Using the selected features, we generated a set of all possible valid configurations for each system. To improve internal validity, we measured each configuration on all hardware environments 10 times, what is the largest number that our Azure budget allowed.

For each configuration we measured two properties: *compression time* and *compressed size*. We selected these properties because they are intuitive, easy to measure, and are universal to all studied systems. It is worth to notice that *compressed size* property doesn't change it's value for a single configuration across hardware. Nevertheless, since our approach works using small samples of measured configurations, it has to approximate this property for the whole configuration space. Therefore, *compressed size* property still has a strong influence on approximated and transferred Pareto frontiers. We leave analysis of multiple varying properties for a future work.

Comprehensive benchmarking allowed us to perform and analyze the overall Pareto transferring process for each software system across all possible pairs of source and destination environments. Moreover, we exhaustively analyzed all possible training sample $\mathbb{C}_S$ sizes $[\,2, N_{\mathbb{C}} - 1\,]$ (see Section 3.1) and transferring sample $\mathbb{C}_{both}$ sizes $[\,2, N_{\mathbb{C}_{both}} - 1\,]$ (see Section 3.3). However, since internal Azure infrastructure is being constantly updated, not all hardware environments were available for benchmarking of all software systems. Table 1 highlights which hardware environments we used for benchmarking of each particular software system. Table 2 shows specifications for each unique CPU that appeared in different hardware environments. Finally, Table 3 provides general information about system benchmarking process.

4.2 Experimental Results

4.2.1 Predictors and Transferrers Accuracy. In order to answer RQ1, we performed a comprehensive evaluation of properties' predictors for all configurable software systems and hardware environments. As mentioned in Section 3.1, after preliminary evaluation we selected regression trees *RT* as our property prediction models, and we assessed quality of predictors using *MAPE*, acquired using *LOOCV* validation. Figure 2 presents evaluation results by displaying distributions of property predictors' *MAPE* with increase of predictors' training sample size. For all software and hardware we observe a strong decreasing trend of *MAPE* with increase of training sample size. For all software and hardware regression trees could achieve *MAPE* less than 10%, and for majority of combinations less than 5%. This observation agrees with previous research and demonstrates that regression trees are well fit for predicting properties of configurable software systems.

In order to answer RQ2, we performed a comprehensive evaluation of properties' transferrers for all configurable software systems and hardware environments. As we mentioned previously in Section 3.3, although after preliminary experiments we again selected regression trees *RT* as our property transferring models, we also performed a comprehensive evaluation of simple linear regression models *SLR*, since linear regression displayed high transferring performance in previous research [17, 34]. We assessed quality of transferrers using *MAPE* and *LOOCV* validation.

Evaluation shows that performance of linear transferrers highly varies from one system to another. Figure 3 presents evaluation results by displaying distributions of *linear-based transferrers' MAPE* with increase of transferrers' training sample size. While for some systems, e.g. BZIP2 and GZIP, linear transferrers achieve *MAPE* smaller than 5%, for FLAC linear transferrers can barely achieve 20% for majority of hardware platforms. Linear transferrers also exhibit poor performance for XZ parallel compression software.

On the contrary, tree-based transferrers displayed much more stable performance. Figure 4 presents evaluation results by displaying distributions of *tree-based transferrers' MAPE* with increase of transferrers' training sample size. Regression trees provide significantly better results for transferring FLAC, x264, and XZ software systems, and comparable results for BZIP2. Although regression trees provide slightly worse results for GZIP than linear models when assessing them using *LOOCV* on a training sample, regression trees still outperform linear models when assessing actually transferred Pareto frontiers for GZIP systems.

4.2.2 Assessing Pareto Frontiers Accuracy. Before we can compare actual, approximated, and transferred Pareto frontiers, we have to select a way of assessing their prediction quality. We regard a Pareto frontier as a binary classifier that separates all system's configurations into optimal and non-optimal ones on a given hardware. Therefore, to assess quality of a Pareto frontier we can use any of classical statistical measures for binary classifiers.

Researchers use different statistical measures when assessing binary classifiers in their works. We decided to include several basic measures in order to provide a comprehensive and intuitive description of how well do approximated and transferred Pareto frontiers classify configurations. Table 4 presents all statistical measures used in our work in a form of a *confusion matrix*. We selected matrix representation since it shows all measures in a concise and structured way. To make statistical measures more intuitive, we indicate their preferred values using special symbols. ($\uparrow$) indicates that higher values of a respective measure are preferred to lower ones, while ($\downarrow$) indicates that lower values are preferred instead.

Table 4: Statistical measures for assessing a Pareto frontier, represented as a confusion matrix

	Condition Positive: $P = TP + FN$	Condition Negative: $N = FP + TN$		
Predicted Condition Positive: $PCP = TP + FP$	True Positive: $TP\uparrow$	False Positive: $FP\downarrow$	Positive Predictive Value: $PPV\uparrow = TP/PCP$	False Discovery Rate: $FDR\downarrow = FP/PCP$
Predicted Condition Negative: $PCN = FN + TN$	False Negative: $FN\downarrow$	True Negative: $TN\uparrow$	False Omission Rate: $FOR\downarrow = FN/PCN$	Negative Predictive Value: $NPV\uparrow = TN/PCN$
	True Positive Rate: $TPR\uparrow = TP/P$	False Positive Rate: $FPR\downarrow = FP/N$	F_1 score: $F_1\uparrow = 2 \times (PPV \times TPR)/(PPV + TPR)$	
	False Negative Rate: $FNR\downarrow = FN/P$	True Negative Rate: $TNR\uparrow = TN/N$	Matthews correlation coefficient: $MCC\uparrow = \sqrt{PPV \times NPV \times TPR \times TNR} - -\sqrt{FDR \times FOR \times FPR \times FNR}$	

A core of a confusion matrix is formed by a contingency table, showing frequency distributions of optimal and non-optimal configurations. These are the most basic classification metrics, from which all other metrics can be derived. *True positive $TP\uparrow$ (negative $TN\uparrow$)* is an amount of actually optimal (non-optimal) configurations correctly classified as such by a Pareto frontier. *False positive $FP\downarrow$ (negative $FN\downarrow$)* is an amount of actually non-optimal (optimal) configurations misclassified as optimal (non-optimal) by a frontier.

The following measures allows a practitioner to answer specific questions about efficiency of a frontier in general. *True positive rate ($TPR\uparrow$)* shows a probability that a frontier contains all actually optimal configurations. *True negative rate ($TNR\uparrow$)* shows how likely will a frontier leave out all actually non-optimal configurations.

The next block of measures allows a practitioner to answer questions about classification results by a Pareto frontier. *Positive predictive value ($PPV\uparrow$)* represents a probability that a configuration, classified as optimal by a Pareto frontier, is truly optimal. *Negative predictive value ($NPV\uparrow$)* shows how likely a configuration, classified as non-optimal by a Pareto frontier, is truly non-optimal.

Although previously presented statistical measures provide information about a Pareto frontier quality, these measures are best regarded together in groups, since this way they provide a more comprehensive understanding of a frontier's performance. Therefore, we also included an 'integral' measure of a binary classification behavior. *Matthews correlation coefficient (MCC)* is considered to be one of the best measures for working with data that has strong quantitative differences between classes of observations. *MCC* takes values in $[-1, +1]$, where -1 corresponds to a completely misclassifying frontier, 0 to a random frontier, and $+1$ to a perfect frontier.

4.2.3 Approximated Frontiers Accuracy. To answer RQ3 we assess accuracy of Pareto frontiers, approximated using our methodology. We calculate an approximated Pareto frontier $\widehat{\mathbb{C}}^{PF}_{h_{src}}$ based on system properties $\mathbb{P}$ approximated for all configurations $\mathbb{C}$ using tree-based predictors that we train using samples of measured configurations $\mathbb{C}_{trn}$ (see Section 3.2). We regard approximated Pareto frontiers as binary classifiers and evaluate them using presented statistical measures (see Section 4.2.2). Figure 5 shows evaluation results of Pareto frontiers of all software systems for 'BscA1-2660' hardware environment (see Table 1), approximated using tree-based predictors that are trained using samples $\mathbb{C}_{trn}$ of different sizes.

We observe that TNR and NPV demonstrate high values almost instantly, and provide almost perfect results for FLAC, x264, and XZ systems. This means that a frontier can very efficiently and with high certainty catch non-optimal configurations. However, this happens because classification categories are highly unbalanced in our case, since a number of optimal configurations is generally much smaller than a number of non-optimal ones. Because of that, even if a frontier classify all configurations as non-optimal, TNR and NPV might demonstrate high values in some cases. This effect becomes more apparent with a larger system's configuration space. Therefore, we cannot claim that approximated frontiers exhibit high classification accuracy based on TNR and NPV values only.

TPR and PPV exhibit a gradual growth with increase of predictors' training sample sizes. This means that ability of a frontier to capture optimal configurations and certainty that an optimally-classified configuration is truly optimal, both highly depend on a training sample size. We can explain this behavior by several major factors. First of all, classes of optimal and non-optimal configurations are highly unbalanced in our case. Since our approach cannot impose any restrictions on a sampling process and has to work with randomly-sampled data, truly optimal configurations on average become significantly underrepresented in predictors' training samples. Thus, TPR and PPV gradually improve with availability of new truly optimal configurations. Secondly, inability to accurately capture optimal configurations by an approximated frontier based on small random samples of measured configurations lies in a structure of regression trees. Regression trees are limited by sampled property values and cannot output any value that is smaller than a minimal or larger than a maximal sampled value. Therefore, when regression trees are limited by a min-max interval of a random sample, they will be limited to a subinterval of possible values and consequently only to a part of an actual frontier, making accurate approximation of the whole frontier not possible.

Finally, we observe that MCC demonstrates almost linear growth approximately from 0 to 1, while avoiding negative values. This means that the presented Pareto frontier overall starts as a nearly-random classifier, but with increase of predictors' training samples improves into an almost-perfect classifier, while avoiding complete misclassification of configurations.

To sum up, we evaluated our approach for approximating frontiers that works by individually predicting system's properties using general-purpose machine-learning models trained using random samples of measured configurations. We demonstrated that in general our approach works, but its overall quality is linearly dependent on a training sample's size. In order to improve accuracy of our approach in future work, we might need to relax some initial assumptions like ability to control a sampling process.

4.2.4 Transferred Frontiers Accuracy. To answer RQ4 we assess accuracy of Pareto frontiers, transferred using our approach. We acquire a transferred Pareto frontier $\widehat{\mathbb{C}}^{PF}_{h_{dst}}$ by transferring approximated values of all properties $\mathbb{P}$ using transferrers that we train using samples of configurations $\mathbb{C}_{both}$ measured on both source and destination hardware (see Section 3.4). Figure 6 and Figure 7 present evaluation results of transferred Pareto frontiers of all software systems for linear-based and tree-based transferrers respectively. The analyzed Pareto frontiers were transferred from 'BscA1-2660' to

'StdF2sv2-8168' hardware environments (see Table 1 for comparison). We present results for these hardware environments, because out of all servers for which we could acquire full benchmarking data for all software systems, these environments differ the most. Thus, Figure 6 and Figure 7 present the most difficult available scenario for evaluation of our methodology.

Although linear regression demonstrated strong results for transferring prediction models across heterogeneous hardware previously [18, 34], comparison of Figures 6 and 7 clearly demonstrates that regression trees completely outperform linear models for majority of studied software systems. This difference is especially strong for FLAC and XZ software systems where MCC for linear-based transferring barely reaches 0.5 and 0.3 respectively.

To visually represent how much distortion to the approximated frontier does the transferring process add, we calculate difference between approximated and transferred frontiers for all statistical measures and present them on Figure 8. Thus, we can observe that the transferring process has a very limited impact on distortion of an approximated frontier even for small training samples sizes.

To sum up, we have demonstrated that transferring of approximated Pareto frontiers across heterogeneous hardware environments using unpruned regression trees is possible and doesn't significantly affect the resulting frontier's accuracy. However, a transferred frontier cannot demonstrate a high quality if an original approximated frontier doesn't show high classification results. Therefore, in future work we plan to improve our approach for minimalistic practical approximation of Pareto frontiers.

5 THREATS TO VALIDITY

To increase internal validity of our research, during evaluation of our methodology, we trained properties' predictors and transferrers using random samples of measured configurations, and for each training size we generated 10 different random samples. Thus, all statistical measures that describe predictors, transferrers, and frontiers, are averaged over 10 different instances. Therefore, we avoid bias caused by random variations in models' training samples.

To increase external validity of our research, we performed evaluation of our approach using five configurable software systems with different code bases, features, configuration space sizes, parallelization capabilities, and application domains. We benchmarked the selected software systems on 34 heterogeneous hardware platforms with different CPU models, available CPU cores, RAM sizes, and storage types. When benchmarking software systems, we measured each configuration 10 times and averaged over these measurements to get final properties' values. This allowed us to avoid bias induced by random aberrations during a benchmarking process.

We tried to address the most obvious threats to internal and external validity of our work, but we acknowledge that this might not be enough. Although we explored a variety of general-purpose software systems including parallelized ones, we suspect that there might be other configurable systems with different features, architectures, or application domains, whose properties might exhibit completely different unsystematic behavior across variable hardware. Moreover, we expect this behavior to occur when a practitioner redeploys a particular software system that is optimized for a specific hardware architecture. For example, when a system that supports GPU-acceleration is redeployed to a hardware without a dedicated graphical unit. We plan to investigate such software-hardware interaction in a future work.

6 RELATED WORK

This work is mostly related to two topics: (1) black-box model-based performance prediction of highly configurable software systems and (2) transferring of performance models across hardware environments. However, approaches presented in related work are usually different in philosophy, what leads to investigation of different software and hardware systems, and to usage of different sampling strategies, prediction and transferring models, etc.

We begin by highlighting differences of our work with the most related research on model-based performance prediction. To make our research practical and reproducible, we analyze *general-purpose open-source software*, whereas some other work investigates highly specialized systems like: AI planners [10, 24], SAT & MIP solvers [12], search algorithms [14], or custom scientific software for supercomputers [2, 3]. Moreover, some research [9] doesn't analyze configurations, what simplifies the prediction problem. We use *pseudo-random sampling* to explore configuration spaces, assuming that a practitioner might not control the sampling process. However, some researchers do not make this assumption and use a variety of sampling strategies like: n-wise sampling heuristics [28], breakdown algorithms [37], collection of microarchitecture-independent metrics across hardware [9], or expectation that a practitioner completely controls the sampling process [25]. Finally, majority of related work utilizes different models as performance predictors, such as: artificial neural networks [21], Fourier analysis [38, 39], Gaussian Processes [13, 14, 18, 24, 37], logistic regression [24], projected processes [13], regression splines [4, 21, 37], modified regression trees [24], ensembles of regression trees [12, 33], ridge regression [11], and support vector regression [33].

We continue by highlighting differences of our research with the most related work on transferring of performance prediction models. First of all, to make our work reproducible and practical, we use *general-purpose hardware* based on AMD and Intel processors, while some researchers analyze supercomputing hardware [2, 3, 21] or robotic systems [18]. Secondly, after performing a comparison of different models, we selected regression trees as our transferrers. However, some researchers prefer using different approaches for their use cases like: Gaussian processes [17], custom transferring models that are based on extensive software instrumentation and profiling [31, 32], or upfront measurements of a collection of benchmarking software across studied hardware environments [9].

7 CONCLUSION AND FUTURE WORK

We proposed and evaluated a practical, easy-to-use, and black-box approach based on general-purpose machine-learning models for approximation and transferring of Pareto frontiers of optimal configurations. We perform approximation of a frontier by (1) building an unpruned regression tree model for each property to act as a predictor, and then (2) combining properties' predictions into an approximated frontier. Our evaluation shows a strong decreasing trend in predictors' error and a linearly increasing trend of an overall classification accuracy of a resulting approximated frontier, with

increase of predictors' training sample sizes. We perform transferring of a frontier by (1) building an unpruned regression tree model for each property to act as a transferrer, and then (2) combining transferred values of the approximated frontier. Our evaluation shows a strong decreasing trend in transferrers' error and a linearly increasing trend of a transferred frontier accuracy, with increase of transferrers' training sample sizes. Moreover, an overall accuracy of a transferred Pareto frontier mostly depends on the approximated Pareto frontier's accuracy than on the transferring process itself.

In future work we plan to make a substantial improvement to our approach for approximation and transferring of Pareto frontiers. First of all, we might relax some initial assumptions and assume that a practitioner has control over a configuration space's sampling process. This would allow us to use more sophisticated sampling approaches like quasi-random sampling or experimental design techniques, in order to cover a configuration space more evenly and improve prediction accuracy. Secondly, we plan to investigate more advanced machine learning techniques than regression trees that are not inherently limited by a min-max interval of a studied system's property and can extrapolate beyond already seen values.

REFERENCES

[1] Blender Foundation. Sintel Trailer. https://media.xiph.org/.
[2] E. A. Brewer. *Portable High-performance Supercomputing: High-level Platform-dependent Optimization*. PhD thesis, Cambridge, MA, USA, 1994.
[3] E. A. Brewer. High-level optimization via automated statistical modeling. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 80–91, New York, NY, USA, 1995. ACM.
[4] M. Courtois and M. Woodside. Using regression splines for software performance analysis. In *Proceedings of the 2nd International Workshop on Software and Performance*, WOSP '00, pages 105–114, New York, NY, USA, 2000. ACM.
[5] Gailly, J.-l. and Adler, M. GNU Gzip, free, open source, and patent free data compressor. https://www.gnu.org/software/gzip/.
[6] Gailly, J.-l. and Adler, M. GNU Gzip Manual. https://www.gnu.org/software/gzip/manual/gzip.html#Invoking-gzip.
[7] J. Guo, K. Czarnecki, S. Apel, N. Siegmund, and A. Wasowski. Variability-aware performance prediction: A statistical learning approach. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ASE'13, pages 301–311, Piscataway, NJ, USA, November 2013. IEEE Press.
[8] J. Guo, D. Yang, N. Siegmund, S. Apel, A. Sarkar, P. Valov, K. Czarnecki, A. Wasowski, and H. Yu. Data-efficient performance learning for configurable systems. *Empirical Software Engineering*, 23(3):1826–1867, June 2018.
[9] K. Hoste, A. Phansalkar, L. Eeckhout, A. Georges, L. K. John, and K. De Bosschere. Performance prediction based on inherent program similarity. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, PACT '06, pages 114–122, New York, NY, USA, 2006. ACM.
[10] A. E. Howe, E. Dahlman, C. Hansen, M. Scheetz, and A. von Mayrhauser. Exploiting competitive planner performance. In S. Biundo and M. Fox, editors, *Proceedings of the 5th European Conference on Planning: Recent Advances in AI Planning*, ECP '99, pages 62–72, Berlin, Heidelberg, 1999. Springer-Verlag.
[11] F. Hutter, Y. Hamadi, H. H. Hoos, and K. Leyton-Brown. Performance prediction and automated tuning of randomized and parametric algorithms. In F. Benhamou, editor, *Proceedings of the 12th International Conference on Principles and Practice of Constraint Programming*, CP'06, pages 213–228, Berlin, Heidelberg, 2006. Springer-Verlag.
[12] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In C. A. C. Coello, editor, *Proceedings of the 5th International Conference on Learning and Intelligent Optimization*, LION'05, pages 507–523, Berlin, Heidelberg, 2011. Springer-Verlag.
[13] F. Hutter, H. H. Hoos, K. Leyton-Brown, and K. Murphy. Time-bounded sequential parameter optimization. In C. Blum and R. Battiti, editors, *Proceedings of the 4th International Conference on Learning and Intelligent Optimization*, LION'10, pages 281–298, Berlin, Heidelberg, 2010. Springer-Verlag.
[14] F. Hutter, H. H. Hoos, K. Leyton-Brown, and K. P. Murphy. An experimental investigation of model-based parameter optimisation: SPO and beyond. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09, pages 271–278, New York, NY, USA, 2009. ACM.
[15] F. Hutter, L. Xu, H. H. Hoos, and K. Leyton-Brown. Algorithm runtime prediction: Methods & evaluation. *Artificial Intelligence*, 206:79–111, January 2014.
[16] F. Hutter, L. Xu, H. H. Hoos, and K. Leyton-Brown. Algorithm runtime prediction: Methods & evaluation (extended abstract). In Q. Yang and M. Wooldridge, editors, *Proceedings of the 24th International Joint Conference on Artificial Intelligence*, IJCAI'15, Palo Alto, California, USA, July 2015. AAAI Press.
[17] P. Jamshidi, N. Siegmund, M. Velez, C. Kästner, A. Patel, and Y. Agarwal. Transfer learning for performance modeling of configurable systems: An exploratory analysis. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 497–508, Piscataway, NJ, USA, 2017. IEEE Press.
[18] P. Jamshidi, M. Velez, C. Kästner, N. Siegmund, and P. Kawthekar. Transfer learning for improving model predictions in highly configurable software. In *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '17, pages 31–41, Piscataway, NJ, USA, 2017. IEEE Press.
[19] Josh Coalson, Xiph.Org Foundation. FLAC – Free Lossless Audio Codec. https://xiph.org/flac/.
[20] Josh Coalson, Xiph.Org Foundation. FLAC Manual. https://xiph.org/flac/documentation_tools_flac.html.
[21] 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 *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, pages 249–258, New York, NY, USA, 2007. ACM.
[22] Mahoney, M. Large Text Compression Benchmark. http://mattmahoney.net/dc/text.html.
[23] Nine Inch Nails. Ghosts I-IV. https://archive.org/details/nineinchnails_ghosts_I_IV.
[24] M. Roberts, A. Howe, and L. Flom. Learned models of performance for many planners. In *ICAPS 2007 Workshop AI Planning and Learning*, pages 36–40, 2007.
[25] A. Sarkar, J. Guo, N. Siegmund, S. Apel, and K. Czarnecki. Cost-efficient sampling for performance prediction of configurable systems (t). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE '15, pages 342–352, Washington, DC, USA, November 2015. IEEE Computer Society.
[26] Seward, J. and Mena F. Bzip2, free, open source, and patent free data compressor. https://sourceware.org/bzip2/.
[27] Seward, J. and Mena F. Bzip2 Manual. https://sourceware.org/bzip2/manual/manual.html#options.
[28] N. Siegmund, S. S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, and G. Saake. Predicting performance via automated feature-interaction detection. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 167–177, Piscataway, NJ, USA, 2012. IEEE Press.
[29] The Tukaani Project. XZ Utils. http://tukaani.org/xz/. Accessed April. 17th, 2016.
[30] The Tukaani Project. XZ Utils Manual. https://linux.die.net/man/1/xz. Accessed April. 17th, 2016.
[31] E. Thereska, B. Doebel, A. X. Zheng, and P. Nobel. Practical performance models for complex, popular applications. *ACM SIGMETRICS Performance Evaluation Review*, 38(1):1–12, June 2010.
[32] E. Thereska, B. Doebel, A. X. Zheng, and P. Nobel. Practical performance models for complex, popular applications. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '10, pages 1–12, New York, NY, USA, 2010. ACM.
[33] P. Valov, J. Guo, and K. Czarnecki. Empirical comparison of regression methods for variability-aware performance prediction. In *Proceedings of the 19th International Conference on Software Product Line*, SPLC '15, pages 186–190, New York, NY, USA, 2015. ACM.
[34] P. Valov, J.-C. Petkovich, J. Guo, S. Fischmeister, and K. Czarnecki. Transferring performance prediction models across different hardware platforms. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ICPE '17, pages 39–50, New York, NY, USA, 2017. ACM.
[35] VideoLAN Organization. x264, the best H.264/AVC encoder. http://www.videolan.org/developers/x264.html. Accessed April. 15th, 2016.
[36] VideoLAN Organization. x264, the best H.264/AVC encoder. http://www.chaneru.com/Roku/HLS/X264_Settings.htm. Accessed April. 15th, 2016.
[37] D. Westermann, J. Happe, R. Krebs, and R. Farahbod. Automated inference of goal-oriented performance prediction functions. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 190–199, New York, NY, USA, 2012. ACM.
[38] Y. Zhang, J. Guo, E. Blais, and K. Czarnecki. Performance prediction of configurable software systems by fourier learning (t). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE '15, pages 365–373, Washington, DC, USA, November 2015. IEEE Computer Society.
[39] Y. Zhang, J. Guo, E. Blais, K. Czarnecki, and H. Yu. A mathematical model of performance-relevant feature interactions. In *Proceedings of the 20th International Systems and Software Product Line Conference*, SPLC '16, pages 25–34, New York, NY, USA, 2016. ACM.