Transferring Performance Prediction Models Across Different Hardware Platforms

Pavel Valov
University of Waterloo
200 University Avenue West
Waterloo, ON, Canada
pvalov@uwaterloo.ca

Jean-Christophe Petkovich
University of Waterloo
200 University Avenue West
Waterloo, ON, Canada
j2petkovich@uwaterloo.ca

Jianmei Guo*
East China University of Science and Technology
130 Meilong Road
Shanghai, China
gjm@ecust.edu.cn

Sebastian Fischmeister
University of Waterloo
200 University Avenue West
Waterloo, ON, Canada
sfischme@uwaterloo.ca

Krzysztof Czarnecki*
University of Waterloo
200 University Avenue West
Waterloo, ON, Canada
kczarnec@gsd.uwaterloo.ca

ABSTRACT

Many software systems provide configuration options relevant to users, which are often called features. Features influence functional properties of software systems as well as non-functional ones, such as performance and memory consumption. Researchers have successfully demonstrated the correlation between feature selection and performance. However, the generality of these performance models across different hardware platforms has not yet been evaluated.

We propose a technique for enhancing generality of performance models across different hardware environments using linear transformation. Empirical studies on three real-world software systems show that our approach is computationally efficient and can achieve high accuracy (less than 10% mean relative error) when predicting system performance across 23 different hardware platforms. Moreover, we investigate why the approach works by comparing performance distributions of systems and structure of performance models across different platforms.

CCS Concepts

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

Keywords

Performance Modelling, Regression Trees, Model Transfer, Linear Transformation

*Corresponding authors

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICPE'17, April 22-26, 2017, L'Aquila, Italy

© 2017 ACM. ISBN 978-1-4503-4404-3/17/04…$15.00

DOI: http://dx.doi.org/10.1145/3030207.3030216

1. INTRODUCTION

Many software systems provide *configuration options*. These configuration options usually have a direct influence on the functional behaviour of target software systems. Some configuration options may impact systems' non-functional properties, such as response time, memory consumption and throughput. Configuration options that are relevant to users are usually called *features* [5], and a particular selection of features defines a system *configuration*.

Performance prediction of configurable software systems is a highly-researched topic [5, 6, 7, 8, 17, 18, 20, 21]. For example, Guo et al. [5] predicted a system configuration's performance by using regression trees based on small random samples of measured configurations. However, none of the previous work studied whether or not it is possible to transfer performance prediction models for configurable software systems across different hardware platforms.

The need for transferring performance prediction models occurs in many application scenarios. For example, a user of a software system performs a thorough performance benchmarking of the system and builds a performance prediction model for it. However, the prediction model is built only for the particular benchmarked machine. The performance prediction process on a different machine may not be able to directly reuse previous benchmarking results and prediction models. Modern Software as a Service (SaaS), Platform as a Service (PaaS) and other cloud-based industries face similar challenges. Based on a historical performance data collected for their software on one cluster, users want to know how to tune the performance of their software systems for a new cluster with a different hardware, or how to select the best hardware platform with which to build their cluster.

We investigate the problem of performance prediction model transfer. We make the following contributions:

- We propose an approach for transferring performance models of configurable software systems across platforms with different hardware settings. This approach (1) builds a performance prediction model based on a small random sample of configurations measured on one hardware platform and (2) transfers this model using linear transformation to other hardware platforms.

- We implement the proposed approach and demonstrate its generality using three real-world configurable software systems. Our empirical results show that for majority of model transfers our approach achieves a high prediction accuracy (less than 10% mean relative error). We also observe a decreasing trend of mean relative error with the increase of the training data for the performance prediction model or for the linear transformation model.
- We carry out a thorough exploratory analysis to understand why our approach works. We compare performance distributions and structure of performance prediction models across different hardware platforms and show that the more similar distributions and prediction models across different platforms, the better transfer results are. Moreover, we carry out a comparative analysis of our methodology for different configurable systems and assess time costs of our method.

Source code and data to reproduce our experiments are available online at https://bitbucket.org/valovp/icpe2017.

2. EXAMPLE AND NOTATION

Our objective is to enable the transfer of performance prediction results from one hardware platform to another. Consider purchasing a new hardware platform to encode large amounts of video using x264, which is a configurable application for encoding video streams in the H.264/MPEG-4 AVC compression format. Media encoding programs such as x264 usually have a large number of configurable features; tuning them has a significant impact on the quality of the video output and on the time necessary to encode it. In our example, we have measurements for 11 different configuration features, each with 2 individual settings. Obtaining these performance measurements with a video that takes a modest 15 minutes to encode requires 1536 hours of execution time. Rather than exhaustively measuring same configurations on a new unstudied platform, it would be better to reuse performance data from previous tuning experiments to predict the performance of configurations on this new platform.

To formalize the problem of performance prediction, we represent features of a configurable software system as a set of binary decision variables $F = \{f_1, f_2, \dots f_{N_f}\}$, where f_i represents a particular variable and N_f represents a total number of features of the configurable software system. Each configuration $\mathbf{c}$ of the system is a set of value assignments to N_f variables f_i. We denote the set of all valid configurations of the system by $\mathbf{C}$. Table 1 represents a sample of 10 configurations along with their measured performance values. Each row of the Table 1 represents a particular configuration of x264, while each column represents a particular feature of the system.

We define performance of a system as a total time required to execute a particular system benchmark. Performance of each configuration is expected to differ in a heterogeneous collection of machines that we denote by $M = \{m_1, m_2, \dots, m_{N_m}\}$, where m_i represents a particular machine or hardware platform and N_m represents the total number of machines in the collection. Each valid configuration, $\mathbf{c}$, of the software system has an actual performance value, $a_{\mathbf{c},m_i}$, on a machine m_i, a set of configurations, $\mathbf{C}$, has a set of actual performance values, $A_{\mathbf{C},m_i}$, on machine m_i. We define *training machine*, m_{trn}, as a machine that is used to build performance prediction models for a given configurable software system (e.g., x264). In a practical setting, training machine is one on which a particular software system is already well-studied and historical performance data for the system is acquired. We define *target machine*, m_{tgt}, as a machine to which performance prediction models must be transferred.

For example, we acquire a small random sample of configurations, $\mathbf{C}_S \subset \mathbf{C}$, along with their actual performance values, $A_{S,m_{trn}} \subset A_{\mathbf{C},m_{trn}}$, together forming sample, $S_{m_{trn}}$, on our training machine m_{trn}. Our goal is to predict the performance of all other configurations, $\mathbf{C} \setminus \mathbf{C}_S$, on machine m_{trn} based on this small random sample $S_{m_{trn}}$, and subsequently to predict the performance of the whole set of valid configurations $\mathbf{C}$ on all other machines in the collection M.

Table 1: Sample of 11 randomly-selected configurations of x264 system, along with their actual performance measurements on Machine №75

Conf.	Features											Perf. (s)
$\mathbf{c}_i$	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	$a_{\mathbf{c}_i,m_{75}}$
$\mathbf{c}_1$	0	0	1	1	1	0	1	0	0	0	1	52.01
$\mathbf{c}_2$	0	1	0	1	1	1	0	0	1	0	0	24.09
$\mathbf{c}_3$	1	0	0	0	1	1	0	0	0	0	1	58.13
$\mathbf{c}_4$	1	1	0	1	1	0	0	1	0	1	0	37.49
$\mathbf{c}_5$	0	0	1	0	1	1	0	0	0	0	1	75.89
$\mathbf{c}_6$	1	1	0	1	0	0	0	1	0	0	1	51.05
$\mathbf{c}_7$	1	1	1	0	0	1	0	0	0	0	1	82.15
$\mathbf{c}_8$	1	0	0	1	1	1	0	0	0	0	1	41.40
$\mathbf{c}_9$	1	0	0	1	0	0	0	1	1	0	0	23.16
$\mathbf{c}_{10}$	0	0	0	0	1	0	1	0	1	0	0	23.20
$\mathbf{c}_{11}$	0	0	1	0	1	0	1	0	1	0	0	28.95

3. TRANSFERRING PERFORMANCE PREDICTION MODELS

The overall process of transferring performance prediction models across different hardware platforms is sequential and can be separated into the following main steps: (1) training performance prediction model, (2) training linear transfer model, and (3) transferring prediction results.

3.1 Training The Performance Prediction Model

We used regression trees for building models of the performance effects of software features. We selected regression trees as our method of model construction as they have been extensively used for performance prediction of configurable software systems and demonstrated good results [5, 17, 18, 20]. The resulting prediction models can be graphically represented and readily understood by end users. Regression trees proved effective for a thorough exploratory analysis for our model transferring problem (see Section 4.5 for more detail).

We need to choose a method for sampling training data to build regression trees. Previous studies of feature performance modelling, Guo et al. [5] and Valov et al. [18], used small random samples of measured configurations to build prediction models. The use of random sampling in those works was motivated by the idea that in practice, available measured configurations of a system might not follow any particular feature-coverage criteria and would be essentially random. In our study we also use small random samples for prediction model building.

For the purpose of experiment reproducibility we provide datasets of measured configurations for each studied software system (for more details see Section 4.1.1). However, measuring the entire configuration space $\mathbf{C}$ of each software system under test was prohibitively expensive and couldn't be done in the time budget available (for more details see Section 4.1.2). Therefore, for each system we measured only a subset of valid configurations $\mathbf{C}_{exp} \subset \mathbf{C}$.

To choose which configurations should be included in $\mathbf{C}_{exp}$ we used experimental design techniques. Experimental design is an efficient procedure for obtaining experimental data that can be analysed to produce valid results [1]. Experimental design techniques can maximize information obtained by a practitioner for a given experimentation budget. Selection of a concrete design for a particular experiment depends on the goal of the experiment and the number of variables involved.

In our study we use "screening" experimental designs, where the goal is to "screen out" or select the main effects that influence the response variable (in our case system performance), such as *full factorial* and *fractional factorial* experimental designs. This is achieved by selecting the most "informative" configurations for a given system.

Full factorial design generates all possible combinations of all input variables, i.e., in our case this design generates all possible configurations of a given software system. This design is suitable only for systems with a small number of features, since it generates 2^{N_f} configurations for N_f binary features.

Fractional factorial design is more suitable for systems with a large number of features N_f. This design selects only a fraction of configurations generated by a full factorial design, thus saving experimentation effort. However, when using this design some information is inevitably lost, which causes *confounding* or inability to capture some higher-order feature interactions. *Resolution* of a fractional factorial design defines the level to which main effects or lower-order interactions are confounded with higher-order interactions, i.e., how well we can assess or model main effects and lower-order interactions. For example, fractional factorial design of resolution VI provides enough information to estimate main effects and two-factor feature interactions unconfounded by four-factor (or less) and three-factor (or less) interactions respectively, what is a very precise assessment.

Another question is how many configurations should we sample to build a precise performance prediction models? Valov et al. [18] used samples of sizes $T \times N_f$ to evaluate different performance prediction models, where T is a training coefficient which can take values in $\{1, \ldots, 5\}$, and N_f is the number of features available in the configurable software system under test. It was demonstrated [18] that measuring $3 \times N_f$ random configurations permitted construction of models with high performance prediction accuracy using regression trees for majority of studied systems. We use the same heuristic $T \times N_f$, where $T = \{3, 4, 5\}$. We found that this provides sufficient coverage of the feature space for construction of accurate performance prediction models.

We denote a regression tree model by a function RT trained using a small random sample of configurations $\mathbf{C}_S$ of size $T \times N_f$ and their actual performance values $A_{S,m_{trn}}$, measured on a training machine m_{trn}, which predicts the performance value $p_{\mathbf{c},m_{trn}}$ on the machine m_{trn} for a specified configuration $\mathbf{c}$:

$$RT(\mathbf{C}_S, A_{S,m_{trn}}, \mathbf{c}) = p_{\mathbf{c},m_{trn}} \tag{1}$$

Next, we must select a metric for assessing the prediction accuracy of the trained prediction model RT, and a validation method to prevent overfitting. We use *mean relative error (MRE)* as a metric for evaluating prediction accuracy. *Relative error (RE)* is the relative difference between an actual performance value $a_{\mathbf{c}}$ and a predicted performance value $p_{\mathbf{c}}$ for a particular configuration $\mathbf{c}$:

$$RE(\mathbf{c}) = \frac{a_{\mathbf{c}} - p_{\mathbf{c}}}{a_{\mathbf{c}}} \times 100\% \tag{2}$$

Mean relative error is the average of the relative errors calculated for each individual configuration $\mathbf{c}_i$ of a particular sample of configurations $\mathbf{C}_S$,

$$MRE(\mathbf{C}_S) = \frac{\sum_{i=1}^{N_c} RE(\mathbf{c}_i)}{N_c} \tag{3}$$

where N_c is a total number of configurations in the sample $\mathbf{C}_S$.

Finally, we must select a validation method. As mentioned previously, we use random samples of configurations $\mathbf{C}_S$ of a fixed size $T \times N_f$. These configurations are sampled from the set of measured configurations $\mathbf{C}_{exp}$. Since the size of the training sample $\mathbf{C}_S$ is fixed, a natural model validation strategy is *holdout validation*. This method separates all available data, $\mathbf{C}_{exp}$, into a training set, $\mathbf{C}_S$, and a testing set, $\mathbf{C}_{exp} \setminus \mathbf{C}_S$. We train a performance prediction model RT using the training set $\mathbf{C}_S$, and assess prediction accuracy of the model using MRE over the testing set: $MRE(\mathbf{C}_{exp} \setminus \mathbf{C}_S)$.

It is worth mentioning that in an industrial setting a different validation method might be required, i.e., when the cost of performance measuring is extremely high and it is not desirable to measure all $T \times N_f$ configurations upfront or simply to have an extra set of measured configurations available. Practitioner could start with a very small training sample $\mathbf{C}_S$ and progressively train their models until they are satisfied with its accuracy. An effective validation method for these low sample sizes is *leave-one-out cross-validation (LOOCV)*. *LOOCV* separates all available data $\mathbf{C}_S$ into two sets: a testing set, consisting of only one configuration $\mathbf{c}_i$, and a training set, consisting of all other configurations $\mathbf{C}_S \setminus \mathbf{c}_i$. A prediction model RT is then trained using $\mathbf{C}_S \setminus \mathbf{c}_i$ and assessed using relative error over $re_i = RE(\mathbf{c}_i)$. This process is repeated for all possible combinations of training sets and testing sets. The overall accuracy of the prediction model RT for the sample $\mathbf{C}_S$ can be assessed by averaging all individual relative errors: $\sum_{i=1}^{N_c} re_i / N_c$.

3.2 Training The Transfer Model

To reuse the previously generated performance prediction model built for m_{trn} for performance prediction on a target machine m_{tgt} a practitioner must train a transfer model. We use linear regression models as our transfer models since we found that they provide good approximations of the transfer function (see Section 4.5.3 for more details).

The samples we use for linear models training should contain configurations that are measured on both the m_{trn} and m_{tgt} hardware platforms. From the steps described in Section 3.1, we have a training sample, $\mathbf{C}_S$, of configurations

measured on machine m_{trn} of size $T \times N_f$. Instead of measuring a completely new sample of configurations on both m_{trn} and m_{tgt} machines for training the linear model, we can measure the same configurations from $\mathbf{C}_S$ on the target machine m_{tgt}. In this way, we acquire a training sample $\mathbf{C}_S$ of size $T \times N_f$ measured on both m_{trn} and m_{tgt}.

However, measuring all $T \times N_f$ configurations on the target machine m_{tgt} may be prohibitively expensive. Instead, we measure only a subset of $\mathbf{C}_S$ on both machines $\mathbf{C}_{both} \subset \mathbf{C}_S$. We populate $\mathbf{C}_{both}$ by selecting at least five configurations from $\mathbf{C}_S$ using Sobol sampling [14] (see Section 4.6 for more details).

Using the sample $\mathbf{C}_{both}$ we can build a model to transfer performance prediction results from m_{trn} to m_{tgt}. We use simple linear regression model as a transfer model since it provides good approximation of transfer functions between different machines in our case study (see Section 4.5.3 for more details). This linear model L, given a performance value $p_{\mathbf{c},m_{trn}}$ for a configuration $\mathbf{c}$ on the machine m_{trn}, can predict performance value $p_{\mathbf{c},m_{tgt}}$ of $\mathbf{c}$ on the machine m_{tgt}:

$$L(p_{\mathbf{c},m_{trn}}) = \alpha + \beta \times p_{\mathbf{c},m_{trn}} = p_{\mathbf{c},m_{tgt}} \quad (4)$$

3.3 Transferring Prediction Results

In the previous steps we selected training and target machines (m_{trn} and m_{tgt}), built a performance prediction model RT based on a small sample $\mathbf{C}_S$ of configurations measured on m_{trn}, and built a linear transfer model L based on a small subsample $\mathbf{C}_{both} \subset \mathbf{C}_S$ of configurations measured on both m_{trn} and m_{tgt} machines. To transfer the prediction model RT to m_{tgt} we just need to transform the predictions of RT using the linear transfer model L. For example, we have a configuration $\mathbf{c}$ that is not measured neither on m_{trn} nor on m_{tgt} machines. To compute $p_{\mathbf{c},m_{tgt}}$ we can use the following equations:

$$p_{\mathbf{c},m_{trn}} = RT(\mathbf{C}_S, A_{S,m_{trn}}, \mathbf{c}) \quad (5)$$

Then we can use L to assess performance of $\mathbf{c}$ on m_{tgt}:

$$p_{\mathbf{c},m_{tgt}} = L(p_{\mathbf{c},m_{trn}}) \quad (6)$$

4. EVALUATION

In order to evaluate our approach, we address the following research questions through a set of experiments:

RQ1 How accurate are the transferred performance models created using the process described in Section 3? (Section 4.2)

RQ2 How does model accuracy vary between different configurable software systems? (Section 4.3)

RQ3 How fast is the process of transferring performance prediction models? (Section 4.4)

RQ4 Why does the proposed approach work or are the results accidental? (Section 4.5)

RQ5 What is an optimal way of building the linear transfer model? (Section 4.6)

Table 2: Summary of hardware platforms on which configurable software systems were measured; MID – Machine ID in DataMill cluster; NC – Number of CPUs; IS – Instruction set; CCR – CPU clock rate (MHz); RAM – RAM memory size (MB)

Systems			Machines				
XZ	x264	SQLite	MID	NC	IS	CCR	RAM
✓			73	2	i686	1733	1771
✓	✓	✓	75	2	i686	3200	977
✓			77	2	i686	2992	2024
✓			78	1	i686	1495	755
✓			79	4	x86_64	3291	7961
✓			80	8	x86_64	3401	7907
✓	✓		81	16	x86_64	2411	32193
	✓		87	1	i686	1595	249
	✓		88	1	i686	1700	978
		✓	90	2	i686	3200	977
	✓		91	1	i686	2400	1009
	✓	✓	97	2	i686	2992	873
	✓	✓	98	2	i686	2992	873
		✓	99	2	i686	2793	880
	✓		103	2	i686	3200	881
	✓		104	1	i686	1800	502
	✓	✓	105	2	i686	3200	881
	✓		106	2	i686	3192	494
		✓	125	4	x86_64	3301	7960
	✓		128	2	i686	2993	2024
		✓	130	2	i686	3198	880
		✓	146	2	i686	2998	872
		✓	157	36	x86_64	2301	15954

Table 3: Summary of measured systems; $\mathbf{N_f}$ – Number of features; NM – Number of machines on which systems were measured; NMC – Number of measured configurations

System	N_f	NM	NMC
XZ	7	7	154
x264	7	11	165
SQLite	5	10	32

4.1 Experimental Setup

4.1.1 Subject Systems

We measure the performance impact of several different features of 3 different software systems, XZ[16], x264[19], and SQLite[15]. These systems represent several common tasks performed by applications: compression of data, transformation of media, and interaction with a database. XZ is a compression utility for UNIX-like operating systems which uses LZMA2 compression. x264 is a library and utility for encoding video streams into the H.264/MPEG-4 AVC compression format. SQLite is a library and application that implements a file-oriented SQL database and is a popular choice for application file formats due to its flexibility.

Each feature that we varied was chosen either based on previous experiments in feature performance regression [5], or system documentation and preliminary experiments. For XZ, we measure performance effects from features such as varying the "extreme" parameter, varying the "sparse out-

put file" option and applying constraints on memory usage. For x264, we measure performance effects from turning on and off different assembly optimizations, varying the "frame-lookahead", and varying partition search types. For SQLite, we measure performance effects from varying the "synchronous" option, varying the journalling strategy, and varying the amount of space available to mmap.

4.1.2 Subject Hardware Platforms

We carried out our system performance measurements on DataMill[11], a distributed heterogeneous performance evaluation platform. Each machine of DataMill was setup with identical software and executes Gentoo Linux (Kernel version 3.8.13). Only the DataMill worker software, a kernel, boot manager, and logging daemon were installed on each machine on top of the base set of Gentoo packages, resulting in a minimal set of software. Table 2 summarizes hardware configurations of DataMill machines used for our experiments.

Although we did have an exclusive access to DataMill machines, we only had a limited time to use DataMill itself since it is used by many research groups. Therefore we weren't able to measure each configurable system on the whole DataMill cluster, but only on a subset of machines. Table 2 shows which machines were used for measuring performance of different software systems.

Due to constraints on experiment bandwidth on DataMill, and differing support for features on certain platforms, some feature-configuration/hardware-platform combinations were not measured. Moreover, not all experiment trials terminated correctly; thus only a subset of desired configurations were measured across all machines. Table 3 provides a summary of available machines and measured configurations for each configurable system. The scripts used to perform the experiment trials are available on the DataMill website for the purposes of experiment reproduction.

4.1.3 Measurements and Sampling

For each software system, we analyse the effect of choosing training and target machine pairs and the effect of varying the size of the performance training sample $\mathbf{C}_S$ and the transfer training sample $\mathbf{C}_{both}$ on model accuracy. The sample $\mathbf{C}_S$ varies in $\{3 \times N_f, 4 \times N_f, 5 \times N_f\}$, and the sample $\mathbf{C}_{both}$ varies in $\{5, 10, 15\}$.

As mentioned in Section 4.1.2, we measured each studied configurable system on multiple platforms as shown in Table 2 and Table 3. However, due to space constraints for each system we present results only for a subset of four different training and target machines. The data we obtained from these machines is summarized in Tables 4, 8, and 6.

We examine each combination of the training machine m_{trn}, target machine m_{tgt}, and sampling sizes for both $|\mathbf{C}_S|$ and $|\mathbf{C}_{both}|$, in a full-factorial experiment design [2, 10]. For each configurable system we have 4 training machines, 4 target machines, 3 sizes of $\mathbf{C}_S$, and 3 sizes of $\mathbf{C}_{both}$, which produces a total of $4 \times 4 \times 3 \times 3 = 144$ different test cases. To acquire mean prediction relative error for each test case, we follow the model transferring process described in Section 3 for 100 different randomly sampled $\mathbf{C}_S$ and $\mathbf{C}_{both}$.

4.2 Experiment on Prediction Accuracy

To answer RQ1, we present the results of our transferred performance prediction models across different hardware plat-

Table 4: Mean Relative Error (%) of transferred performance models of XZ system, built using different sampling sizes on training and target machines

Target Machines	Sampling Sizes	Training Machines: Machine №75			Machine №78			Machine №80			Machine №81		
		Sampling Sizes			Sampling Sizes			Sampling Sizes			Sampling Sizes		
		$3N_f$	$4N_f$	$5N_f$	$3N_f$	$4N_f$	$5N_f$	$3N_f$	$4N_f$	$5N_f$	$3N_f$	$4N_f$	$5N_f$
75	5	5.8	2.0	1.4	9.5	5.4	5.0	8.0	5.2	4.5	6.8	4.1	3.4
75	10	5.2	2.1	1.1	7.2	4.6	3.8	8.1	5.0	4.7	6.9	3.8	3.7
75	15	5.7	2.3	1.7	6.8	4.3	3.3	8.8	5.4	4.4	7.3	4.0	3.1
78	5	8.5	5.7	4.9	6.6	3.0	1.6	9.9	7.3	6.7	8.7	6.9	6.1
78	10	6.6	5.0	3.8	6.5	3.0	1.5	8.1	5.4	4.6	7.7	4.5	3.6
78	15	7.3	3.6	3.6	8.4	3.3	1.8	7.9	5.4	4.6	7.2	4.2	3.9
80	5	10.2	6.9	5.8	11.9	9.6	9.4	9.5	4.8	1.6	10.6	3.8	3.4
80	10	8.2	6.3	5.5	10.6	6.8	5.6	8.2	2.0	1.9	7.1	3.4	2.7
80	15	10.4	6.5	5.1	11.0	6.4	5.6	8.5	4.3	2.3	7.2	3.8	2.6
81	5	9.0	5.9	4.3	11.5	8.2	7.1	9.7	3.6	3.1	7.9	4.1	1.6
81	10	8.9	5.1	4.2	8.7	5.2	4.5	8.8	3.4	2.8	6.5	4.2	1.9
81	15	8.6	5.0	4.0	10.1	5.1	4.5	9.6	3.2	2.6	10.0	2.9	1.5

Table 5: Mean Relative Error (%) of XZ system added by the transferring process

Target Machines	Sampling Sizes	Training Machines: Machine №75			Machine №78			Machine №80			Machine №81		
		Sampling Sizes			Sampling Sizes			Sampling Sizes			Sampling Sizes		
		$3N_f$	$4N_f$	$5N_f$	$3N_f$	$4N_f$	$5N_f$	$3N_f$	$4N_f$	$5N_f$	$3N_f$	$4N_f$	$5N_f$
75	5	0.1	0.0	0.0	5.0	4.5	4.3	5.1	4.0	4.1	4.1	3.0	2.9
75	10	0.1	0.0	0.0	4.3	3.5	3.4	4.5	4.2	4.1	3.5	3.0	3.0
75	15	0.1	0.0	0.0	3.8	3.0	2.9	4.7	3.9	3.7	3.8	3.2	2.7
78	5	5.0	4.3	4.1	0.1	0.0	0.0	6.6	6.1	6.2	5.7	5.4	5.3
78	10	4.1	3.3	2.9	0.0	0.0	0.0	4.4	4.2	3.8	3.7	3.2	2.9
78	15	3.7	2.7	2.6	0.1	0.0	0.0	4.5	3.9	3.8	3.7	3.0	2.9
80	5	5.6	4.6	4.7	8.4	8.1	8.0	0.0	0.0	0.0	3.0	2.6	2.3
80	10	5.2	4.8	4.6	7.2	4.6	4.5	0.1	0.0	0.0	2.8	2.2	2.0
80	15	4.9	4.4	4.4	5.6	4.5	4.6	0.1	0.0	0.0	3.9	2.0	1.9
81	5	5.1	3.6	3.6	7.2	7.5	7.0	3.6	2.7	2.6	0.1	0.0	0.0
81	10	5.1	4.3	3.8	4.6	3.8	3.8	3.3	2.3	2.2	0.1	0.0	0.0
81	15	4.6	3.3	3.4	4.4	3.9	3.8	3.1	2.4	2.1	0.1	0.0	0.0

Table 6: Mean Relative Error (%) of transferred performance models of SQLite system, built using different sampling sizes on training and target machines

Target Machines	Sampling Sizes	Training Machines: Machine №75			Machine №99			Machine №125			Machine №157		
		Sampling Sizes			Sampling Sizes			Sampling Sizes			Sampling Sizes		
		$3N_f$	$4N_f$	$5N_f$	$3N_f$	$4N_f$	$5N_f$	$3N_f$	$4N_f$	$5N_f$	$3N_f$	$4N_f$	$5N_f$
75	5	0.7	0.5	0.4	0.7	0.6	0.6	1.4	1.3	1.3	2.8	2.6	2.6
75	10	0.8	0.5	0.4	0.7	0.6	0.6	1.3	1.2	1.2	2.6	2.6	2.4
75	15	0.7	0.5	0.4	0.7	0.6	0.6	1.3	1.2	1.1	2.6	2.5	2.4
99	5	0.8	0.7	0.6	0.8	0.6	0.6	1.6	1.4	1.3	2.6	2.6	2.6
99	10	0.8	0.6	0.6	0.9	0.6	0.6	1.5	1.3	1.2	2.6	2.4	2.4
99	15	0.7	0.6	0.6	0.8	0.6	0.6	1.4	1.3	1.2	2.6	2.4	2.4
125	5	1.5	1.4	1.3	1.6	1.5	1.4	1.9	1.5	1.1	2.6	2.4	2.3
125	10	1.4	1.3	1.3	1.5	1.4	1.3	1.9	1.5	1.1	2.7	2.3	2.2
125	15	1.4	1.3	1.2	1.5	1.4	1.3	1.8	1.5	1.1	2.4	2.3	2.1
157	5	3.7	3.5	3.5	3.5	3.4	3.3	3.2	2.9	2.7	4.4	4.1	3.6
157	10	3.4	3.2	3.2	3.2	3.1	3.0	3.0	2.7	2.5	4.2	4.0	3.7
157	15	3.3	3.2	3.2	3.1	3.0	2.9	3.0	2.7	2.5	4.3	3.9	3.5

Table 7: Mean Relative Error (%) of SQLite system added by the transferring process

Target Machines	Sampling Sizes	Training Machines: Machine №75			Machine №99			Machine №125			Machine №157		
		Sampling Sizes			Sampling Sizes			Sampling Sizes			Sampling Sizes		
		$3N_f$	$4N_f$	$5N_f$	$3N_f$	$4N_f$	$5N_f$	$3N_f$	$4N_f$	$5N_f$	$3N_f$	$4N_f$	$5N_f$
75	5	0.0	0.0	0.0	0.6	0.5	0.4	1.1	1.1	1.0	2.6	2.5	2.5
75	10	0.0	0.0	0.0	0.5	0.4	0.4	1.2	1.1	1.0	2.4	2.4	2.4
75	15	0.0	0.0	0.0	0.5	0.5	0.4	1.1	1.0	1.0	2.4	2.3	2.3
99	5	0.6	0.6	0.5	0.0	0.0	0.0	1.3	1.2	1.1	2.6	2.5	2.4
99	10	0.6	0.5	0.5	0.1	0.0	0.0	1.1	1.1	1.0	2.4	2.3	2.2
99	15	0.6	0.5	0.5	0.0	0.0	0.0	1.2	1.0	0.9	2.2	2.1	2.1
125	5	1.3	1.1	1.1	1.4	1.4	1.3	0.0	0.0	0.0	2.3	2.1	2.0
125	10	1.2	1.2	1.1	1.3	1.3	1.3	0.0	0.0	0.0	2.1	2.0	1.9
125	15	1.3	1.2	1.1	1.4	1.2	1.2	0.0	0.0	0.0	2.2	2.1	2.0
157	5	2.9	2.7	2.7	2.8	3.0	2.8	2.7	2.7	2.4	0.0	0.0	0.0
157	10	2.8	2.6	2.5	2.8	2.6	2.6	2.4	2.4	2.3	0.0	0.0	0.0
157	15	2.8	2.7	2.5	2.8	2.6	2.5	2.6	2.3	2.2	0.0	0.0	0.0

Table 8: Mean ± Standard Deviation [Mean Confidence Interval] of the relative error (%) of transferred performance models of x264 system, built using different sampling sizes on training and target machines

Target Machines	Sampling Sizes	Training Machines: Machine №75			Machine №81			Machine №88			Machine №103		
		Sampling Sizes			Sampling Sizes			Sampling Sizes			Sampling Sizes		
		$3N_f$	$4N_f$	$5N_f$	$3N_f$	$4N_f$	$5N_f$	$3N_f$	$4N_f$	$5N_f$	$3N_f$	$4N_f$	$5N_f$
75	5	5.3±9.3 [5.2, 5.5]	3.2±7.0 [3.1, 3.4]	2.0±5.1 [1.9, 2.1]	13.5±10.2 [13.3, 13.6]	13.1±9.7 [12.9, 13.2]	13.0±9.3 [12.8, 13.1]	5.4±9.4 [5.3, 5.5]	3.5±6.3 [3.4, 3.6]	2.4±4.6 [2.3, 2.4]	5.2±8.4 [5.1, 5.4]	3.5±6.3 [3.4, 3.6]	2.4±4.1 [2.3, 2.5]
75	10	5.5±9.8 [5.3, 5.6]	3.2±7.1 [3.1, 3.3]	1.7±4.6 [1.7, 1.8]	12.8± 9.3 [12.6, 12.9]	12.3±8.8 [12.2, 12.5]	12.1±8.8 [12.0, 12.3]	5.0±8.5 [4.9, 5.1]	3.2±6.0 [3.1, 3.3]	2.3±4.8 [2.2, 2.4]	4.7±8.2 [4.6, 4.9]	3.1±5.8 [3.0, 3.2]	2.1±4.0 [2.1, 2.2]
75	15	5.4±9.5 [5.2, 5.6]	3.4±7.2 [3.2, 3.5]	2.0±5.2 [1.9, 2.1]	12.5± 9.1 [12.4, 12.7]	12.1±8.5 [12.0, 12.2]	12.0±8.4 [11.8, 12.1]	5.2±8.8 [5.0, 5.3]	3.3±6.4 [3.2, 3.4]	2.3±4.6 [2.3, 2.4]	4.8±8.3 [4.7, 5.0]	3.0±5.7 [3.0, 3.1]	2.1±4.0 [2.0, 2.1]
81	5	14.3±12.9 [14.1, 14.5]	13.2±10.6 [13.0, 13.4]	12.8±9.8 [12.7, 13.0]	6.0±7.5 [5.9, 6.1]	4.7±6.0 [4.6, 4.8]	3.7±5.3 [3.6, 3.8]	13.7±11.7 [13.5, 13.8]	12.9±10.1 [12.7, 13.0]	12.7±10.0 [12.6, 12.9]	13.6±11.2 [13.4, 13.8]	12.7±9.9 [12.6, 12.9]	12.3±9.2 [12.2, 12.5]
81	10	12.3±10.1 [12.2, 12.5]	11.4± 8.8 [11.3, 11.5]	11.0±8.4 [10.9, 11.2]	5.8±7.1 [5.7, 5.9]	4.7±6.2 [4.6, 4.8]	3.9±5.4 [3.8, 4.0]	12.1± 9.6 [11.9, 12.2]	11.4± 8.7 [11.2, 11.5]	11.0± 8.3 [10.9, 11.2]	11.9± 9.8 [11.7, 12.0]	11.2±8.8 [11.0, 11.3]	10.7±8.2 [10.6, 10.9]
81	15	11.7± 9.2 [11.6, 11.9]	11.0± 8.9 [10.9, 11.1]	10.6±8.1 [10.5, 10.7]	6.3±7.7 [6.1, 6.4]	4.7±6.1 [4.6, 4.8]	3.9±5.5 [3.8, 4.0]	12.2±10.1 [12.0, 12.3]	11.1± 8.5 [10.9, 11.2]	10.8± 7.9 [10.6, 10.9]	11.6± 9.1 [11.5, 11.7]	10.8±8.0 [10.6, 10.9]	10.6±8.0 [10.4, 10.7]
88	5	4.7±7.9 [4.6, 4.8]	3.3±5.8 [3.2, 3.4]	2.3±4.2 [2.2, 2.4]	13.7±10.1 [13.6, 13.9]	12.8±9.8 [12.6, 12.9]	12.4±8.9 [12.2, 12.5]	5.3± 9.2 [5.1, 5.4]	3.8±8.6 [3.6, 3.9]	2.2±5.6 [2.1, 2.3]	5.8±8.2 [5.6, 5.9]	3.9±5.8 [3.8, 4.0]	3.0±3.9 [2.9, 3.1]
88	10	5.0±9.0 [4.9, 5.2]	3.0±5.7 [2.9, 3.1]	2.2±4.2 [2.1, 2.2]	12.3± 9.0 [12.2, 12.5]	12.1±9.1 [11.9, 12.2]	11.8±8.6 [11.7, 12.0]	5.8±10.1 [5.6, 6.0]	3.5±7.3 [3.4, 3.7]	2.0±5.2 [1.9, 2.1]	5.2±8.1 [5.0, 5.3]	3.7±5.8 [3.6, 3.7]	2.8±4.2 [2.7, 2.9]
88	15	5.0±8.6 [4.8, 5.1]	3.3±6.3 [3.2, 3.4]	2.2±4.4 [2.1, 2.3]	12.2± 8.9 [12.1, 12.4]	11.7±8.6 [11.6, 11.9]	11.7±8.4 [11.6, 11.8]	5.4± 9.9 [5.3, 5.6]	3.5±7.2 [3.3, 3.6]	2.1±5.3 [2.0, 2.2]	5.1±8.1 [5.0, 5.2]	3.4±5.3 [3.3, 3.5]	2.7±4.0 [2.7, 2.8]
103	5	5.6±9.2 [5.5, 5.7]	3.6±6.2 [3.5, 3.7]	2.6±4.4 [2.5, 2.7]	14.4±11.0 [14.3, 14.6]	13.4±9.8 [13.3, 13.6]	13.2±9.4 [13.1, 13.4]	6.0±8.5 [5.9, 6.2]	4.5±6.8 [4.4, 4.6]	3.1±4.1 [3.1, 3.2]	5.9±10.1 [5.7, 6.1]	3.7±7.4 [3.6, 3.8]	2.1±5.0 [2.1, 2.2]
103	10	5.3±9.1 [5.1, 5.4]	3.3±6.3 [3.2, 3.4]	2.2±4.1 [2.2, 2.3]	13.2± 9.6 [13.1, 13.4]	12.6±9.0 [12.5, 12.8]	12.8±9.2 [12.7, 13.0]	5.9±9.2 [5.8, 6.1]	3.8±5.9 [3.7, 3.9]	3.1±4.5 [3.0, 3.1]	6.2±10.6 [6.0, 6.4]	3.2±6.6 [3.1, 3.3]	2.1±4.9 [2.0, 2.1]
103	15	5.3±8.9 [5.1, 5.4]	3.3±6.0 [3.2, 3.4]	2.3±4.3 [2.3, 2.4]	13.4±11.3 [13.2, 13.5]	12.8±9.2 [12.6, 12.9]	12.4±8.8 [12.3, 12.6]	5.6±9.0 [5.5, 5.8]	4.0±6.5 [3.9, 4.1]	2.9±4.8 [2.9, 3.0]	5.8± 9.6 [5.6, 6.0]	3.2±6.9 [3.1, 3.4]	2.4±5.7 [2.3, 2.5]

Table 9: Mean ± Standard Deviation of the time cost (ms) of building performance prediction and transferring models of x264 system, using different sampling sizes on training and target machines

Target Machines	Sampling Sizes	Training Machines: Machine №75			Machine №81			Machine №88			Machine №103		
		Sampling Sizes			Sampling Sizes			Sampling Sizes			Sampling Sizes		
		$3N_f$	$4N_f$	$5N_f$	$3N_f$	$4N_f$	$5N_f$	$3N_f$	$4N_f$	$5N_f$	$3N_f$	$4N_f$	$5N_f$
75	5	5.7±0.9	5.4±0.7	5.9±0.7	5.5±0.5	6.1±0.5	6.3±0.8	5.6±0.5	5.7±0.6	6.1±0.5	6.5±0.5	6.4±1.1	6.7±0.6
75	10	6.0±0.6	6.0±0.6	6.0±0.4	5.3±0.9	5.7±0.5	6.2±0.4	5.5±0.5	5.8±0.4	5.9±0.7	5.8±0.6	6.4±0.8	6.3±0.8
75	15	5.3±0.5	5.5±0.5	6.1±0.7	5.7±0.6	5.8±0.6	6.3±0.6	5.5±0.7	5.8±0.6	5.5±0.5	5.8±0.9	6.5±1.1	6.2±0.6
81	5	5.8±0.9	5.6±0.5	6.0±0.4	5.5±0.5	5.9±0.8	6.0±0.6	5.5±0.7	6.1±0.3	5.9±0.7	5.7±0.6	6.0±1.0	6.2±0.9
81	10	5.5±1.0	5.5±0.5	5.9±0.7	5.3±0.6	5.7±0.5	6.1±0.7	5.9±0.7	6.0±0.8	5.8±0.4	5.9±0.5	7.0±0.8	6.1±0.5
81	15	6.2±0.7	6.1±0.8	6.1±0.7	5.4±0.7	5.8±0.4	6.0±0.9	5.6±0.5	5.9±0.3	5.9±0.5	5.2±0.9	6.3±0.6	6.2±0.7
88	5	6.6±0.5	5.8±0.6	5.8±0.6	5.9±0.5	5.8±0.6	5.8±0.4	5.5±0.8	5.7±0.8	6.4±0.9	5.3±0.5	6.0±0.4	6.2±0.6
88	10	5.9±0.8	6.2±0.7	6.2±0.4	5.4±0.7	5.5±0.5	6.0±0.4	5.5±0.5	6.0±0.6	6.1±0.9	5.1±0.7	6.2±0.9	6.4±0.5
88	15	6.0±0.4	5.9±0.5	6.2±1.0	5.4±0.8	5.9±0.3	6.2±0.6	5.7±1.0	5.8±0.6	5.8±0.6	5.2±0.7	5.9±0.7	6.7±1.4
103	5	5.7±0.6	5.9±0.7	6.0±0.6	5.5±0.5	6.0±0.4	5.5±0.7	5.8±0.4	5.7±0.5	6.0±0.4	5.4±0.5	5.5±0.7	6.3±0.8
103	10	5.7±0.8	5.6±0.7	6.3±0.6	5.7±0.8	6.0±0.0	6.2±1.0	5.5±0.5	5.8±0.6	6.3±0.8	5.2±0.9	5.8±0.7	6.5±0.9
103	15	5.6±0.7	5.9±0.5	6.2±0.6	5.8±0.7	5.7±0.5	6.4±0.9	5.7±0.8	6.0±0.6	6.4±0.9	5.6±0.8	6.0±0.8	6.2±0.7

Table 10: Mean Relative Error (%) of x264 system added by the transferring process

Target Machines	Sampling Sizes	Training Machines: Machine №75			Machine №81			Machine №88			Machine №103		
		Sampling Sizes			Sampling Sizes			Sampling Sizes			Sampling Sizes		
		$3N_f$	$4N_f$	$5N_f$	$3N_f$	$4N_f$	$5N_f$	$3N_f$	$4N_f$	$5N_f$	$3N_f$	$4N_f$	$5N_f$
75	5	0.3	0.2	0.1	12.4	11.9	12.0	2.3	2.0	1.6	2.5	2.1	1.6
75	10	0.2	0.2	0.1	11.7	11.5	11.6	1.9	1.9	1.6	2.6	1.9	1.5
75	15	0.1	0.2	0.1	11.8	11.4	11.7	2.3	1.8	1.6	2.9	1.8	1.5
81	5	11.8	11.0	11.0	0.2	0.1	0.1	11.5	11.0	11.1	11.5	10.8	10.7
81	10	10.1	9.4	9.7	0.2	0.1	0.1	10.1	9.5	9.5	10.2	9.3	9.2
81	15	9.9	9.3	9.3	0.2	0.2	0.1	9.5	9.6	9.3	9.9	9.4	9.1
88	5	2.3	2.1	1.6	12.3	12.0	11.7	0.2	0.1	0.1	3.8	2.8	2.5
88	10	2.6	1.9	1.8	11.5	11.4	11.5	0.4	0.1	0.1	3.0	2.5	2.2
88	15	2.1	1.7	1.5	11.1	11.4	11.3	0.3	0.2	0.1	3.3	2.7	2.2
103	5	2.6	1.9	1.8	12.9	12.3	12.3	3.8	2.9	2.3	0.3	0.1	0.1
103	10	2.6	1.8	1.4	12.1	11.7	11.8	3.3	2.5	2.2	0.2	0.2	0.1
103	15	2.6	1.7	1.4	11.8	11.8	11.9	3.1	2.3	2.1	0.1	0.2	0.1

forms (Table 4, Table 6, Table 8). As we can see from the results, the majority of training and target machine pairs have strong monotonically decreasing trends in their mean relative error with the increase in training sample size from $3 \times N_f$ to $5 \times N_f$. This follows the intuition that more training data results in more accurate performance prediction models.

We can also observe sharp decrease in the mean relative error when the sampling size increases from 5 to 15 configurations. This is expected, as again, more training data generally leads to better model accuracy. However, this trend is not always monotonic and in some special cases doesn't hold at all. These observations agree with our analysis of learning curves of linear transformation models performed in Section 4.6. From Figure 4 and Figure 5 we can see that even samples as small as 5 configurations can provide very good approximations of the linear transfer model due to the simplicity of linear models. However, when training with small sample sizes (in our experience, on the interval $[5, 20]$) the linear transfer model may get stuck in a local cost-minimum where the generated model may be biased. This is the reason for the non-monotonically decreasing error we observe in Tables 4, 6, and 8.

Table 8 shows not only mean values of the relative errors from our models, but also standard deviations and confidence intervals at the 95% confidence level. From Table 8 we can see that although our mean relative error is often small, the standard deviations we measure are relatively large and can exceed the mean in absolute value. However, confidence intervals for the mean value, calculated using bootstrapping [2, 10], are narrow and are almost always less than or equal to 0.5% of the mean relative error.

The data obtained from our experiments $\mathbf{C}_{exp}$, described in Section 3.1, is sufficient to capture feature interactions up to order three. However, we built prediction models RT using only small samples $\mathbf{C}_S \subset \mathbf{C}_{exp}$. Therefore, although

sample $\mathbf{C}_S$ may permit making a good approximation of its corresponding performance distribution, its possible that our performance models, $RT_1, RT_2, \ldots, RT_n$, simply cannot capture all feature interactions that are captured by $\mathbf{C}_{exp}$. This creates a situation where RT produces precise performance predictions for the majority of tested configurations $\mathbf{C}_{exp} \setminus \mathbf{C}_S$ (less that 1% relative error), but for some configurations, which contain uncaptured feature interactions, RT can produce very inaccurate predictions (with more than 50% relative error). This is why we see low mean relative errors, high standard deviations (because of the small set of very large outliers), and very narrow confidence intervals (since it is hard for bootstrapping to capture these outliers).

We can assess accuracy of transferred performance models from a slightly different perspective. We can evaluate how much worse are transferred models compared to "native" models, generated specifically for a target hardware platform. To achieve that, for each performance model trained on m_{trn} and transferred to m_{tgt}, we generate a "native" performance model using the same set of configurations which were measured on m_{tgt}. Then we calculate mean relative error of the native model and subtract it from the mean relative error of the transferred model, thus assessing how much we lose in accuracy when transferring a model from a different platform. Results of this assessment are presented in Table 5, Table 7 and Table 10. One can notice that added mean relative error, when transferring prediction models to the same machine, has a non-zero value. This is caused by the fact that implementation of CART that we use in our study is not exactly deterministic and in some special cases it might generate slightly different prediction models from the same training data. This causes different predictions by these models and thus non-zero difference of mean relative prediction error.

In summary, prediction accuracy generally improves with increasing sampling sizes on training and target machines. However, the proposed approach relies on good sampling strategies for generating the training data $\mathbf{C}_S$ such that it captures necessary feature interactions. This may cause problems in a practical setting where the training sample $\mathbf{C}_S$ might not follow any feature-coverage criteria. We suggest using experimental design for generating samples of configurations for measurement on target machines as they will maximize the amount of information available in the training sample $\mathbf{C}_S$.

4.3 Experiment on System Comparison

To answer RQ2 we compare Tables 6 and 8. As we can see from these tables the mean relative errors for x264 are much higher than those of SQLite. The same process of performance model transfer produces different prediction accuracy for different configurable systems. This is not unexpected as we can generally expect that different systems will have varying levels of predictability in the performance effects of their features, and in the performance effects of the interactions between features. In the case of x264, many features, such as the size of the window used for a filter, or the number of passes used for encoding, have compounding effects with other features. Configurable features of x264 have many complex interactions that can geometrically increase or decrease its encoding performance. Furthermore, for special cases like video encoders, many chipsets include on-board hardware decoding support, further complicating accurate

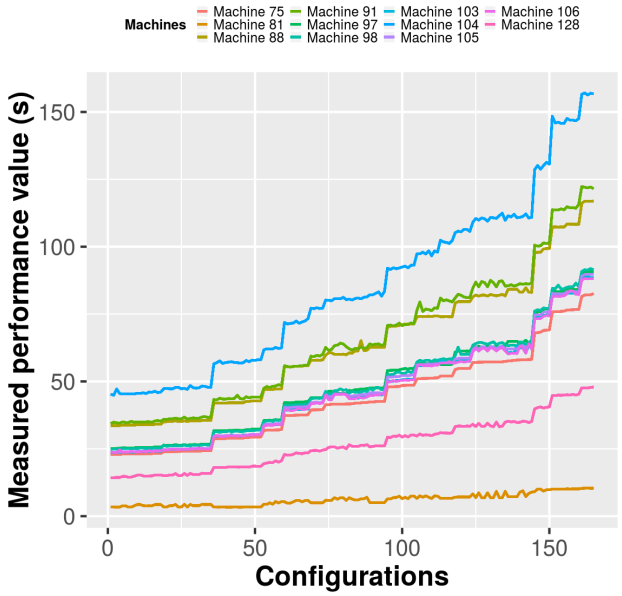

Figure 1: Performance distributions of x264 deployed on different machines

prediction of feature performance across different hardware platforms. On the other hand, simpler software systems like SQLite have far fewer features and many that do not interact significantly, it is much simpler to predict as a result.

4.4 Experiment on Time Cost

Toward answering RQ3, Table 9 shows the execution time of building both performance prediction models RT and performance transfer models L for different training sample sizes $|\mathbf{C}_S|$ and $|\mathbf{C}_{both}|$. We can see from this table that the amount of training time necessary for building both prediction and transfer models is a small fraction of the time necessary to benchmark individual configurations, let alone exhaustively exploring all feature setting combinations in $\mathbf{C}$ on even a modest sized benchmark of a given software system. For comparison, Table 1 shows examples of the amount of time necessary for benchmarking individual configurations of x264.

4.5 Exploratory Analysis

Toward answering RQ4, we conduct a thorough analysis of our methodology. We investigate the performance distributions of configurable systems deployed on multiple hardware platforms, compare the structure of performance models trained on different hardware platforms, and show that linear models are effective approximations for performance transfer models. Therefore we show that the accuracy of our results is not accidental or a result of over-fit, and provide explanations of why our approach works.

4.5.1 Analysis of Performance Distributions

To assess the feasibility of transferring performance models of systems between different hardware platforms, we analyzed the similarity of their performance distributions. Studied systems have many features, thus their feature spaces are highly multidimensional and difficult to represent in a man-

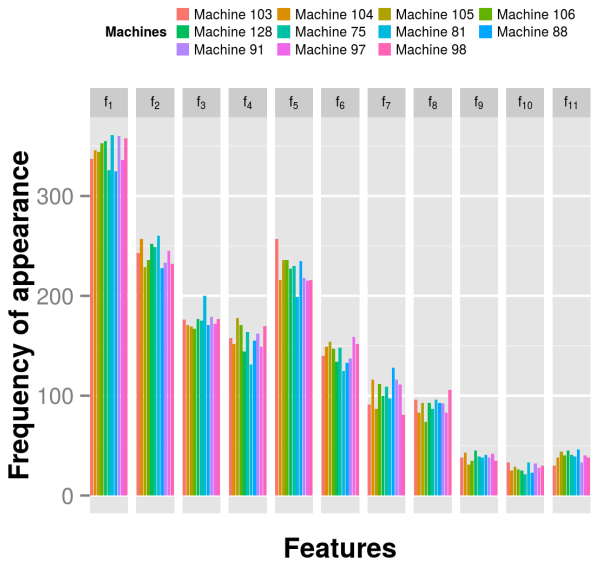

Figure 2: Feature distributions of regression trees trained for performance prediction of x264 on different machines

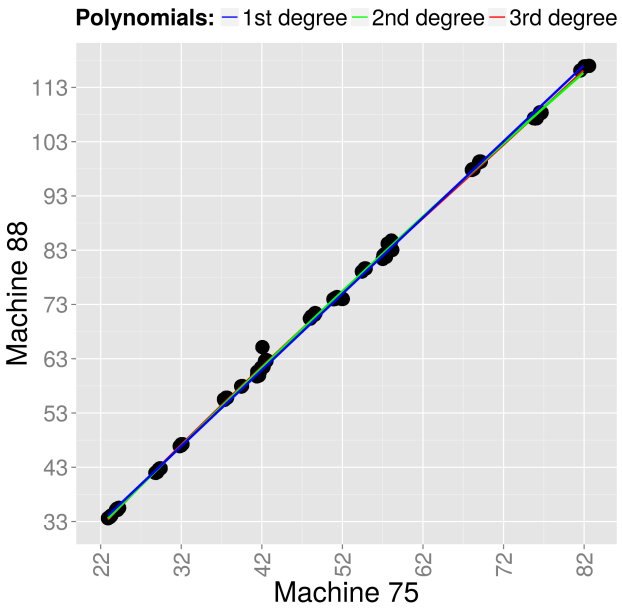

Figure 3: Transformation between performance distributions of x264 system deployed on Machine №75 and Machine №88

ner readily interpretable by the human eye. To visualise the performance distributions of our systems, we take a sample of the configurations that we measure and sort them by the performance of one of the benchmarked hardware platforms.

Figure 1 presents performance distributions of x264 deployed on different hardware platforms. We can see that almost all distributions have very similar shapes, although different in absolute values. Though it is only a cursory analysis of the similarity of the performance distributions of our systems across machines, it does give us confidence that even simple polynomial transformations between these distributions could give us good predictions between hardware platforms. There is a clear pattern indicating that configurations retain their relative performance profile across different hardware platforms, i.e., configuration with low relative performance on one platform, will have low relative performance on another platform.

4.5.2 *Comparison Analysis of Regression Trees*

Regression trees are built by recursively partitioning training dataset into subsets using dataset features. Therefore, features used for dataset partitioning play a major part in defining the structure of a regression tree. Listing all features used in the nodes of a regression tree can be used as a metric for comparing the structure of two different regression trees. Thus by using this metric we can assess the similarity of two regression trees built for the same configurable system, but deployed on different hardware platforms.

Following this logic, we built a feature distribution of regression trees trained for performance prediction of a system deployed on different platforms. From Figure 2, we can see that the distributions of features used by trees on different hardware platforms are very similar to each other. From this we can conclude that the structure of the trees themselves are similar across the different hardware platforms we use in our experiments. Therefore it should be possible for us to train a regression tree for performance prediction of a configurable system on one platform and reuse this tree, with small modifications, on another platform.

4.5.3 *Analysis of Distributions Transformations*

To select a method for transferring prediction models across different platforms we investigated *transfer models* between training and target machine distributions. We used visualizations of the relationships between training and target machine performance distributions to guide the selection of the models used for transfer. An example of the visualizations we used is shown in Figure 3. The x-axis corresponds to a configuration's performance on the training platform, while the y-axis corresponds to that same configuration's performance on the target platform.

By exploring several possible transfer models for all systems in our case study, we found that a polynomial regression model provides a good approximations of the transfer function between machines. To evaluate our hypothesis, we fitted three polynomial models to the transformation data: 1^{st}, 2^{nd} and 3^{rd} degree polynomials. For all those software systems and hardware platforms that we tested, we found that a 1^{st} degree polynomial provides an excellent fit of our transformation data, while 2^{nd} and 3^{rd} degree polynomials appear to cause overfitting and unnecessary complications of the transfer model.

4.5.4 *Summary*

We believe that our proposed process of performance model transfer achieved high prediction accuracy as a result of several main factors. Firstly, the studied configurable systems have very similar performance distributions when deployed on different hardware platforms. Secondly, the transfer function between these distributions is simple and can be easily

approximated using a linear model. Finally, the prediction models trained on the studied software systems have very similar structure when built independently on different platforms. All of these factors together allowed us to use simple and robust methods for performance prediction and model transfer, which resulted in high accuracy achieved by our proposed approach.

4.6 Building Linear Transfer Models

To answer RQ5 we performed a thorough analysis of transfer model building process and tried to answer several important questions. (1) Is it possible to measure only a subset of $\mathbf{C}_S$ on both machines $\mathbf{C}_{both} \subset \mathbf{C}_S$ and build a reliable linear transfer model? (2) Which sampling method to use for $\mathbf{C}_{both}$ to achieve acceptable results faster? (3) Is it possible to figure out a minimum amount of configurations to measure on both m_{trn} and m_{tgt} machines? (4) What is the amount of measured configurations after which additional measurements are not necessary? Toward answering these questions, we decided to analyse the learning curves of the linear transfer models.

We evaluated three different methods of sampling $\mathbf{C}_{both}$: Walker's alias sampling [13], stratified sampling [12] and Sobol sampling [14]. *Walker's alias sampling* [13] is a random sampling method which is the default sampling strategy in the R programming language. Walker's alias sampling is an example of a classical pseudo-random sampling method that generates new samples according to a specified probability distribution. *Stratified sampling* [12] is a random sampling method that exhaustively divides a sampled population into mutually exclusive subsets of observations before performing actual sampling. This allows to cover the sampled population more evenly, which in some cases significantly improves representation of the whole population by a sample. *Sobol sampling* [14] is an example of a quasi-random sampling method. Sobol sampling is similar to pseudo-random sampling, as it generates new samples with respect to a given probability distribution, however quasi-random methods are specifically designed to cover a sampled population more uniformly than pseudo random strategies.

To generate a linear transfer model between the performance distributions of machines m_{trn} and m_{tgt}, we build a training dataset using all available configurations $\mathbf{C}_{exp}$. An example of this training data and the resulting linear transfer model is shown in Figure 3.

The algorithm we used for building learning curves for linear transfer models is as follows:

1. Initialize $\mathbf{C}_{both}$ by randomly sampling a configuration from $\mathbf{C}_{exp}$.
2. Randomly sample another configuration from the set $\mathbf{C}_{exp} \setminus \mathbf{C}_{both}$ and add it to the training sample $\mathbf{C}_{both}$.
3. Build a linear model L based on the sample $\mathbf{C}_{both}$.
4. Assess how well L approximates transformation by using *mean squared error (MSE)* over the full set of configurations $\mathbf{C}_{exp}$.
5. If the set $\mathbf{C}_{exp} \setminus \mathbf{C}_{both}$ is non-empty go to the Step 2. Otherwise, build the learning curve of L by combining mean squared errors for different sizes of $\mathbf{C}_{both}$.

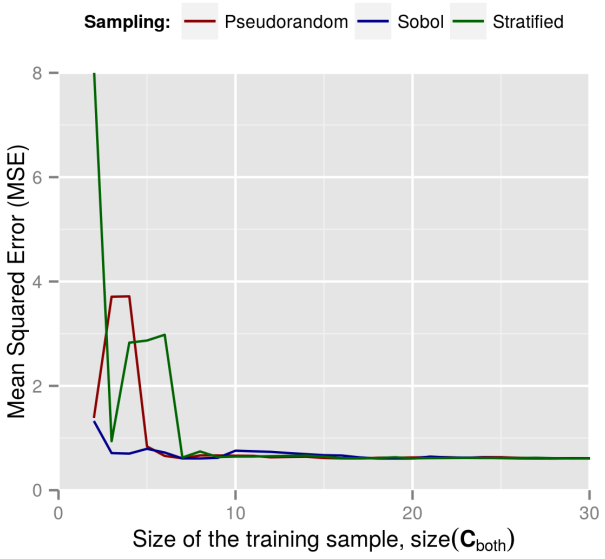

Figure 4: Learning curves of a linear transformation between performance distributions of x264 system on Machine №75 and Machine №88

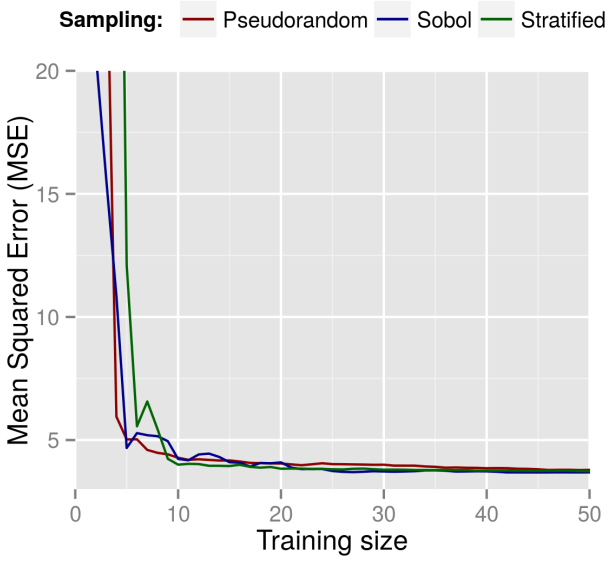

Figure 5: Average learning curve of a linear transformation between performance distributions of x264 system

Figure 4 represents the learning curve of a linear model approximating the transfer function between the performance distributions of Machine №75 and Machine №88 when running the x264 software system. Figure 5 shows the average learning curve of a linear transformation between performance distributions of individual machines running the x264 software system.

We gained several key insights through analysis of our averaged learning curves for all studied configurable software systems and learning curves for all combinations of training and target machines. Firstly, it is possible to measure only a small sample of configurations $\mathbf{C}_{both} \subset \mathbf{C}_S$ to build a reliable linear transformation between two performance distributions. Secondly, we noticed that for all systems, the biggest improvement in the performance of our linear transfer models occurs when $size(\mathbf{C}_{both}) \in [2, 10]$. However, when $size(\mathbf{C}_{both}) > 20$ practically no performance improvement from additional samples is observed. As a result, we recommend that the size interval for training linear transfer models be set to $size(\mathbf{C}_{both}) \in [10, 20]$.

4.7 Threats to validity

To enhance internal validity, we implemented automated random sampling of configurations $\mathbf{C}_S$ on training machines and $\mathbf{C}_{both}$ on target machines. As was mentioned in Section 4.1.3, $\mathbf{C}_S$ varies in $\{3 \times N_f, 4 \times N_f, 5 \times N_f\}$, and $\mathbf{C}_{both}$ varies in $\{5, 10, 15\}$. For each combination of these sampling sizes, $\mathbf{C}_S$ and $\mathbf{C}_{both}$ were independently and randomly sampled ten times. Thus resulting mean relative errors presented in Tables 4, 8, 6 are averaged over ten independent transferring experiments. This allowed us to avoid bias caused by selecting training data for prediction and transfer models.

An obvious threat to external validity is that the results are derived from experiments on a limited number of software systems and a limited range of hardware. To reduce the threat we benchmarked three configurable systems with different sizes, number of features and covering different application domains. All of the studied systems are used in real-world settings. When benchmarking subject configurable systems we measured each configuration three times. Thus actual performance values in our study are averages over three independent measurements. This allowed us to address possible measurement error in our experiment.

To further enhance external validity, we measured each system on multiple hardware platforms with different number of CPUs, instruction sets, clock rates and memory sizes (see Table 2 for more details). We performed transferring experiments for all possible pairs of machines with differing hardware configurations and presented a subset of these experiments in Tables 4, 8, 6, 9.

However, we acknowledge that our experiments investigated a very limited set of software systems and hardware platforms and current results might not extrapolate very well to other hardware and software. We suspect that our approach might not work when transferring performance prediction model of a software system that is specifically designed for a particular hardware configuration. For example, some software systems might use hardware acceleration, like GPUs, for it's tasks. If such a system is deployed on a hardware that doesn't have a dedicated GPU, it's performance distribution might appear completely distorted. Thus linear transformation might not provide a good approximation of a transfer model. This hypothesis should be investigated in future work.

5. RELATED WORK

Performance prediction of configurable software systems is a highly researched topic. Researches investigate which models are best suited for predicting system performance, which strategies can be used for tuning these models, and how to minimize the amount of measured configurations necessary for model training.

Guo et al. [5] proposed a variability-aware approach for performance prediction of configurable software systems based on small random samples of measured configurations. To reveal a correlation between a selection of configuration options and system performance, the authors use CART. They perform a case study of the proposed approach using six configurable software systems with different application domains, implementation languages, configuration spaces and sizes. This study shows that the proposed approach on average achieves prediction accuracy of 94%, when using small samples for CART prediction model training. Finally, the authors show that the approach achieves the best results when a training sample has a similar performance distribution as the whole population of configurations.

Valov et al. [18] extended the work of Guo et al. [5], by carrying out an empirical comparison of regression methods for the problem of variability-aware performance prediction. They compare prediction accuracy of four methods: CART, Bagging, Random Forest and SVM. For each method the authors generate multiple parameter settings, by using Sobol sampling, and select parameters that provided the best, the average and the worst prediction accuracy for each method. By analysing prediction accuracy of methods for combinations of different parameter settings, target configurable software systems and training sampling sizes, they assess which methods provide the best prediction most of the time, i.e. which method is the most robust one. Results showed Bagging to be the most robust technique for performance prediction, even when allowing an interval for selecting the best performance accuracy.

Westermann et al. [20] proposed an approach for automatic, measurement-based method for inferring performance prediction functions. To minimize the amount of measurements, they developed three algorithms that iteratively select new data points if necessary. To build actual performance prediction functions, the authors use four different regression and interpolation methods: MARS, CART, GP, and Kriging. To validate built prediction functions, they use three different strategies. They provide a framework for evaluation of function building methods for performance prediction and for evaluation of different combinations of function building methods and parameter tuning strategies. Finally, they evaluated the methodology for performance prediction in two industrial case studies.

Hutter et al. [7, 8] performed a comprehensive study of methods for configurable algorithms runtime prediction. Authors propose new methods for performance prediction based on random forests and approximate Gaussian processes. Moreover, authors show how methods of survival analysis can be used for improving random forest technique to better handle incomplete performance measurements. With respect to the actual domain of algorithms which performance is predicted, authors investigated satisfiability (SAT), travelling salesperson (TSP) and mixed integer programming (MIP) problems and inferred new probing and timing features for them. Finally, authors present a comprehensive evaluation of different performance prediction methods including ridge regression and it's variants, neural networks, regression trees, Gaussian processes and random forests.

Our work is similar to the aforementioned ones [5, 7, 8, 18, 20], since we also use CART for building performance prediction models. However, we do not perform a comparison of different regression methods, parameter tuning techniques or sampling strategies. Instead, we concentrate on transferring generated performance prediction models to different hardware environments.

System performance prediction across different hardware platforms is a highly researched topic as well. Researches investigate different methodologies for cross-platform performance prediction and ways for collecting necessary performance data.

Thereska et al. [17] proposed an approach for performance modelling of complex popular applications such as Microsoft's Office suite and Visual Studio. All explored applications were specifically instrumented to export their current state as well as all necessary performance relevant metrics. Moreover, all explored applications were deployed on multiple machines, thus allowing them to monitor how each particular application with different configurations behaves in various hardware platforms. To predict performance of a system on a particular hardware platform, the authors (1) select configuration options (both software and hardware) that influences a chosen performance metric the most, (2) use similarity search to select hardware platforms with similar hardware configuration, (3) return distribution of possible performance metric values from a number of similar configurations as the result.

Hoste et al. [6] proposed an approach for performance prediction of a given software application on a set of hardware platforms, to find out which platform provides the best performance for the given application. Authors propose a set of special applications, called a benchmark suite, that have their microarchitecture-independent characteristics and performance values measured. Using this measured benchmark suite, authors build a data transformation matrix that is used to transform applications performance characteristics and values into points in so-called benchmark space. Benchmark space is populated by applications from the benchmark suite and by the target application which performance needs to be predicted. Finally, performance prediction of the target application is carried out by taking weighted average of performance values of neighbouring applications in the benchmark space.

Although Thereska et al. [17] and Hoste et al. [6] propose methodologies for performance prediction of configurable software systems across different hardware platforms, they are completely different from ours. In the current work we build a prediction model for one hardware platform and transfer it to another using a linear model. On the contrary, Thereska uses similarity search on the collected performance data and Hoste uses transformation to a specially designed benchmark space and neighbourhood search to predict system performance on the target platform.

6. CONCLUSION AND FUTURE WORK

We proposed an approach for transferring performance prediction models of configurable software systems across different hardware platforms. We performed a rigorous exploratory analysis of the proposed methodology, including: performance distributions comparison, regression models structure comparison, linear transformation analysis, and comparison of different sampling strategies. We observed a high correlation between performance distributions similarity and high prediction accuracy of our method. We showed that similarity of performance distributions is correlated with structure of performance prediction models. We demonstrated that linear model provides a good approximation of transformation between performance distributions of a system deployed in different hardware environments and showed that it is possible to build a reliable linear transfer model using a small sample of measured configurations $\mathbf{C}_{both}$, where $size(\mathbf{C}_{both}) \in [5, 10]$.

We performed a thorough quantitative analysis of our methodology. We showed that our approach achieves high accuracy (less than 10% mean relative error) for majority of prediction model transfers. Moreover, we observe a decreasing tendency of prediction error with increase of the training data for prediction or linear transfer models. Finally, we demonstrated that time required for building both performance prediction and linear transfer models is negligible (less than 10 ms) compared to time budget required for acquiring configuration measurements.

In future work we plan to test our approach on more configurable systems. All systems, that we've explored so far, demonstrate very similar performance distributions across a wide range of heterogeneous hardware environments. However, if a software system is specifically developed or tuned for a particular hardware configuration or architecture, it might exhibit a dramatically different performance distribution when deployed on a completely different hardware platform. We intend to find and analyse these systems in order to improve generality of our methodology. This might require using a different performance prediction model or using a more complex transformation between performance distributions.

We plan to extend our approach by varying not only configuration options of studied systems, but also systems' workload during benchmarking. We suspect that variations in system workload might influence transferability of system's performance prediction model by distorting system's performance distribution across different hardware environments.

Another direction of future work might be enhancing our approach using queuing networks. A large body of work has been aggregated on theory and application of queueing networks to performance modelling and prediction of configurable software and hardware systems. Balsamo et al. [3] and Cortellessa et al. [4] propose approaches for automatic derivation of performance models, based on different types of queueing networks, from UML-based designs of software systems. Kowal et al. [9] propose a queueing-networks-based approach for performance modelling and prediction of families of systems (e.g. software product lines or complex configurable software systems) by approximating performance models using ordinary differential equations and symbolically analysing them. All this work can be used to completely or partially abstract studied systems as queueing networks thus improving performance prediction across different hardware environments.

7. ACKNOWLEDGEMENTS

We thank anonymous reviewers for their comments, corrections and ideas provided. This work was partially supported by Natural Sciences and Engineering Research Council of Canada, Pratt & Whitney Canada, and Shanghai Municipal Natural Science Foundation (No. 17ZR1406900).

8. REFERENCES

[1] NIST/SEMATECH e-Handbook of Statistical Methods. http://www.itl.nist.gov/div898/handbook/.

[2] J. Antony. *Design of Experiments for Engineers and Scientists*. Butterworth-Heinemann, 2003.

[3] S. Balsamo and M. Marzolla. Performance Evaluation of UML Software Architectures with Multiclass Queueing Network Models. In *Proceedings of the 5th International Workshop on Software and Performance*. ACM, 2005.

[4] V. Cortellessa and R. Mirandola. PRIMA-UML: a performance validation incremental methodology on early UML diagrams. *Science of Computer Programming*, July 2002.

[5] J. Guo, K. Czarnecki, S. Apel, N. Siegmund, and A. Wasowski. Variability-aware performance prediction: A statistical learning approach. In *Proc. ASE*. IEEE, 2013.

[6] 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.

[7] F. Hutter, L. Xu, H. H. Hoos, and K. Leyton-Brown. Algorithm runtime prediction: Methods & evaluation. *Artificial Intelligence*, 206(0):79–111, Jan. 2014.

[8] F. Hutter, L. Xu, H. H. Hoos, and K. Leyton-Brown. Algorithm runtime prediction: Methods & evaluation (extended abstract). In *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI)*, July 2015.

[9] M. Kowal, M. Tschaikowski, M. Tribastone, and I. Schaefer. Scaling Size and Parameter Spaces in Variability-Aware Software Performance Models (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015.

[10] D. Montgomery. *Design and Analysis of Experiments*. John Wiley & Sons, 2008.

[11] J. C. Petkovich, A. Oliveira, Y. Zhang, T. Reidemeister, and S. Fischmeister. DataMill: A Distributed Heterogeneous Infrastructure For Robust Experimentation. *Software: Practice and Experience*, pages n/a–n/a, 2015.

[12] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C (2nd Ed.): The Art of Scientific Computing*. Cambridge Univ. Press, 1992.

[13] B. D. Ripley. *Stochastic simulation*, volume 316. John Wiley & Sons, 2009.

[14] I. Sobol and Y. Levitan. A pseudo-random number generator for personal computers. *Computers and Mathematics with Applications*, 37(4–5):33–40, 1999.

[15] SQLite. SQLite. https://www.sqlite.org/. Accessed April. 15th, 2016.

[16] The Tukaani Project. XZ Utils. http://tukaani.org/xz/. Accessed April. 17th, 2016.

[17] E. Thereska, B. Doebel, A. X. Zheng, and P. Nobel. Practical performance models for complex, popular applications. *SIGMETRICS Perform. Eval. Rev.*, 38(1):1–12, June 2010.

[18] 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.

[19] VideoLAN Organization. x264, the best H.264/AVC encoder. http://www.videolan.org/developers/x264.html. Accessed April. 15th, 2016.

[20] 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.

[21] Y. Zhang, J. Guo, E. Blais, and K. Czarnecki. Performance prediction of configurable software systems by fourier learning. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2015.