Learning to Sample: Exploiting Similarities Across Environments to Learn Performance Models for Configurable Systems

Pooyan Jamshidi
University of South Carolina
USA

Miguel Velez, Christian Kästner
Carnegie Mellon University
USA

Norbert Siegmund
Bauhaus-University Weimar
Germany

ABSTRACT

Most software systems provide options that allow users to tailor the system in terms of functionality and qualities. The increased flexibility raises challenges for understanding the configuration space and the effects of options and their interactions on performance and other non-functional properties. To identify how options and interactions affect the performance of a system, several sampling and learning strategies have been recently proposed. However, existing approaches usually assume a fixed environment (hardware, workload, software release) such that learning has to be repeated once the environment changes. Repeating learning and measurement for each environment is expensive and often practically infeasible. Instead, we pursue a strategy that transfers knowledge across environments but sidesteps heavyweight and expensive transfer-learning strategies. Based on empirical insights about common relationships regarding (i) influential options, (ii) their interactions, and (iii) their performance distributions, our approach, L2S (Learning to Sample), selects better samples in the target environment based on information from the source environment. It progressively shrinks and adaptively concentrates on interesting regions of the configuration space. With both synthetic benchmarks and several real systems, we demonstrate that L2S outperforms state of the art performance learning and transfer-learning approaches in terms of measurement effort and learning accuracy.

CCS CONCEPTS

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

KEYWORDS

Software performance, configurable systems, transfer learning

ACM Reference Format:
Pooyan Jamshidi, Miguel Velez, Christian Kästner, and Norbert Siegmund. 2018. Learning to Sample: Exploiting Similarities Across Environments to Learn Performance Models for Configurable Systems. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3236024.3236074

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.
ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3236074

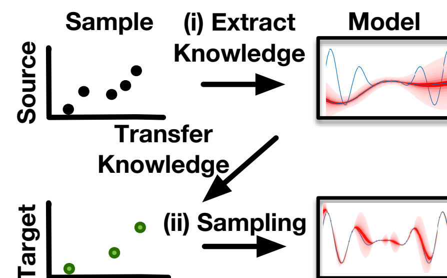

Figure 1: L2S performs guided sampling employing the knowledge extracted from a source environment.

1 INTRODUCTION

Software systems are increasingly becoming more configurable [69]. Highly-configurable software systems, such as Web servers, robotics software, and big data engines, are used in dynamic and uncertain environments. Users of such systems are interested in knowing the consequences of changing the configuration options that are available to them (e.g., performance vs. accuracy). Similarly, administrators are interested in identifying the most energy-efficient configuration of the system under a specific workload. To answer such questions, performance-influence models can characterize how options and their interactions affect performance in these systems [14, 19, 28, 40, 54, 57, 58, 71, 72]. In large configuration spaces, such approaches can be expensive to build, but they often yield accurate models. However, existing approaches usually focus on a *single environment* (fixed hardware, fixed workload, fixed software release) and may need to relearn models from scratch when environments change, which is expensive and slow. We pursue a strategy to efficiently learn models in changed environments, reusing information gained previously or in environments where measurements are cheaper to obtain (e.g., in simulations).

Consider the following two scenarios, in which we would like to *reuse* performance models across environments:

- *Scenario 1: Hardware change*: The developers of a software system perform a thorough performance benchmarking of the system in its staging environment and built a performance model, for instance, in cloud [7, 10]. Therefore, the model may not be able to provide accurate predictions for the performance of the system in the actual production environment (*e.g.*, due to the instability of measurements in the cloud [37, 48]).
- *Scenario 2: Workload change*: The developers of a database system build a performance model for the system using a read-heavy workload, however, the model may not be able to provide accurate predictions once the workload changes to a write-heavy one. The reason is that if the workload changes, different functionalities of the software might get activated more often and so the non-functional behavior changes, too.

Fortunately, performance models typically exhibit *similarities* across environments, even environments that differ substantially in terms of hardware, workload, or version [8, 30, 31, 63]. The challenge is to (i) *identify* similarities and (ii) *make use* of them to ease learning of performance models.

We design L2S, a sampling strategy that exploits common similarities across environments found in a recent empirical study [30] of 36 environment changes: Not all configuration options and their interactions are influential from the performance perspective, but those which are influential are similar across environments. For example, the configuration option that determines the indexing mechanism of a database management system is influential for both read-heavy and write-heavy workloads, though with different effects. Such common relationships allow us to design a sampling strategy that, based on information from a source environment, focuses performance exploration on a rather small subspace of the whole configuration space in the target environment (here called "interesting region"), whereby we are able to learn the performance model more efficiently. That is, L2S extracts transferable knowledge from the source to drive the selection of more informative samples in the target environment (cf. Figure 1). A standard learner (e.g., CART [16]) then uses the samples generated by L2S and produces a performance-prediction model for the target environment. More specifically, L2S works as follows: Based on identifying interesting regions from the performance model of the source environment, it iteratively generates and selects configurations in the target environment. The approach considers three characteristics from the source models: (i) influential options, (ii) option interactions, and (iii) diversity of samples. We determine influential options and their interactions from regression models [5] of measurements in the source and establish diversity by taking into account the performance distribution fitted to the measurements in the source environment using kernel density estimation [5, 16, 59].

We evaluate L2S based on over 100 synthetic models and *4 real-world configurable software systems*, with a various number of configuration options, in multiple environmental changes. We observed:

(1) Building *custom models* for every environment is unrealistically expensive and using a single model to predict the performance of all possible environments is not good enough, often leading to a high prediction errors [45, 63].
(2) *Simple transfer learning* approaches that assume changes in the environments are homogeneous (e.g., model shift [63]) work only for simple environmental changes, but lead to a high prediction error when the assumption of a homogeneous change does not hold, which is common in practice [30].
(3) There exist more *sophisticated transfer learning* that reuse source data with the hope to capture correlation, as a similarity measure, between environments using learners such as Gaussian Processes (GP) [31]. This approach works for severe environment changes with a low number of options, but, due to the restriction of the underlying learning mechanism, it is expensive to use and does not scale for larger numbers of configuration options. Moreover, the quality of its prediction accuracy depends on the quality of the selected data points from the source and if the source is too dissimilar to the target, it may even lead to more inaccurate performance models [31, 51].
(4) *Our approach*, L2S, extracts knowledge from the source to inform the sampling of the target. L2S exploits more variety of similarities (e.g., options and their interactions) and performs better even for severe environmental changes. Combined with an off-the-shelf learner, L2S outperforms state-of-the-art transfer learning [31, 63] as well as traditional performance learning [19] in prediction accuracy.

Overall, our contributions are the following:

- L2S, a sampling approach that selects samples in interesting regions by exploiting prior knowledge from cheap sources.
- An approach for generating representative environmental changes on synthetically generated performance models to enable a large-scale evaluation of our approach.
- A thorough evaluation of L2S for learning performance models of four real-world configurable systems as well as over hundreds synthetically generated models, including a comparison to state-of-the-art performance modeling approaches.

2 THE BIG PICTURE

2.1 State of the art of performance analysis

For understanding the performance behavior of a software system, performance models that predict the performance of the system in different configurations using a form of mathematical performance model can help. Historically, white-box models [13] are developed manually using domain-specific knowledge about the system's internal structure. White-box models are typically built early in the life-cycle, by studying the underlying design and architecture of the software system and are parameterized with the configuration options [13]. They are used for identifying performance bottlenecks so that developers can redesign the system. Queuing networks, Petri Nets, and Stochastic Process Algebras are commonly used [20].

More recently, mainly due to the abundance of data-driven approaches, the emphasis has been on black-box models, which focus on configuration options that are readily available to adjust the behavior of a system. These black-box models do not make any assumption on the design and architecture but are learned from observations: running the system in different sampled configurations and extrapolating predictions for other configurations [45, 57]. A common strategy to use machine-learning techniques to generalize a model that characterizes system performance [19, 57, 63]. The learned performance model can be used for (i) performance debugging [18, 57], (ii) performance tuning [21, 24, 25, 39, 40, 46, 61, 63, 67] by feeding the learned model to an optimizer to find good performing configurations, (iii) detect configuration related bugs, and (iv) capacity planning [2, 12–14, 23, 29, 31, 32].

2.2 Challenges

Our goal is to decrease the cost of learning a black-box performance model by selecting a small but still representative set of samples from which we can derive an accurate model (i.e., accurately captures the regularities in the samples, but also generalizes well to unseen data). Unfortunately, for highly-configurable systems, we face the following problems that were acknowledged by previous research [19, 28, 45, 54, 57]:

- *Curse of dimensionality.* As the dimensionality of the configuration space increases, the number of samples required to maintain the accuracy of the learned model can increase exponentially [16]. Also, most learners cannot scale to high-dimensional spaces [49].
- *Cost of sample acquisition.* Acquiring a sample may have a non-negligible cost. For example, to obtain a sample for a configurable database system, we can spend half an hour for loading the gigabytes of data and run a benchmark query against the database

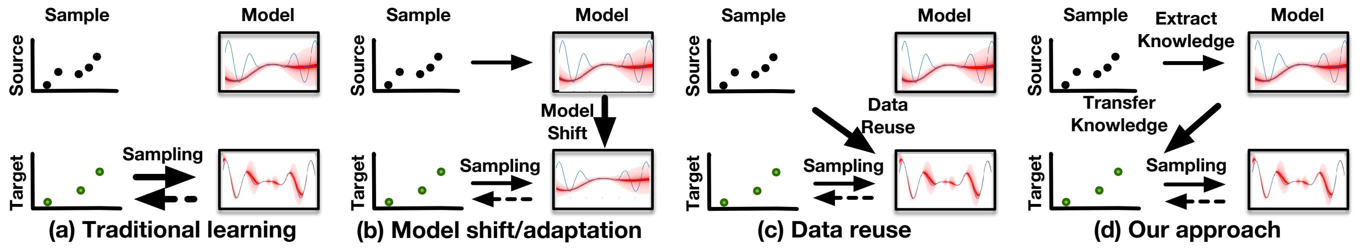

Figure 2: The big picture of performance model learning. (a) Traditional learning using only data sampled from the target environment; (b, c, d) Different variations of performance model learning employing a form of transfer learning.

[28, 64]. This cost could be even worse, e.g., for long-running batch systems, limiting the rate at which samples can be acquired. For instance, acquiring samples corresponding to 1% of a configuration space with 50 binary options and average measurement time of 10 minutes takes around 2,000,000,00 years!

- *Fixed environment and relearning from scratch.* A key assumption in most prior work is measuring performance for a *fixed environment, e.g.*, fixed workload [19, 41, 42, 54, 57, 58, 72]. This means that the efforts we spent on learning the performance model cannot be reused when changing the environment, but we must learn a new model from scratch.

2.3 Transfer learning

Recently, the use of transfer learning has been suggested to decrease the cost of learning by transferring performance predictive models [63], measurement data [31], or configuration constraints [8] across environments. Similar to humans that learn from previous experience and transfer the learning to accomplish new tasks easier, here, knowledge about performance behavior gained in one environment can be reused effectively to learn models for changed environments with a lower cost.

Figure 2 summarizes prior work in performance-model learning with a contrast to our approach: Figure 2(a) shows traditional performance model learning, where the sampling takes place only for the target environment, the source data and model, if present, do not contribute in the learning process in the target environment. Figure 2(b) shows transfer learning by *shifting* the model that has been learned in the source to predict the performance of the system in the target environment [63]. Figure 2(c) shows transfer learning by *data reuse* that exploits the source samples to learn a model in the target environment capturing the similarities between the source and target samples using a correlation measure [31].

2.4 The intuition behind our approach

Our approach (Figure 2(d)) is based on extracting characteristics that likely remain stable across environments. For example, a configuration option c_x that does not affect any considerable change in the performance of the system in one environment is also less likely to have an influence in a different environment; hence, we do not prioritize generating samples with varying values of c_x, because such samples will likely not improve our understanding of the performance characteristics of the system. But, if an option has a significant influence in one environment there is a chance that it has a significant influence (albeit with different strength) in a different environment. We extract information from the source model that guides the sampling in the target environment, to gather fewer but more informative samples, and therefore, to gain a better understanding of the system performance with less cost.

We propose the following list of information useful for sampling, based on empirical observations across many environment changes in a prior study [30]:

(1) *Active subspace*: First, we detect the interesting region of the target environment from which we will sample. We perform this by determining statistically significant options that contribute to the performance of the system in the source environment.
(2) *Option interactions*: Second, we prioritize our sampling on certain option interactions that are statistically influential in the source environment. The main reason is that covering all possible combinations of the options in the reduced subspace may still be prohibitively large. Instead, by concentrating on the more informative samples, that cover only important interactions, we would gain more information earlier.
(3) *Diversity and coverage*: Third, it has been observed in several studies that performance distribution of configurable systems are multi-modal [28, 30, 45, 59]. For learning accurate performance models, we require samples that proportionally cover the true empirical performance distribution. We, therefore, use pseudo-random sampling that iteratively selects the samples that make the distribution of the target more similar to the distribution of the source.

3 L2S: LEARNING TO SAMPLE

Our approach consists of three phases (see Figure 3):

(1) *Knowledge extraction* (Section 3.1): L2S gathers relevant characteristics (i.e., influential options, influential interactions, and performance distribution) from the source environment.
(2) *Active sampling* (Section 3.2): L2S performs a guided sampling that takes advantage of the pieces of knowledge extracted from the source to take informative samples in the target.
(3) *Learning*: A learner uses the samples that are generated by L2S iteratively to build the performance model for the configurable system in the target environment. This third step is standard and we reuse off-the-shelf learners.

3.1 Knowledge extraction

We extract knowledge that is used in L2S for space reduction and sample prioritization. We first describe the concepts of influential options and interactions and then the machinery to extract them from the source environment.

3.1.1 What we extract: influential options, interactions, and performance distribution. *Influential options* are configuration options that have a statistically significant influence on performance. That is, when comparing the pairs of configurations in which this option is enabled and disabled respectively, an influential option has a consistent effect to speed up or slow down the program, beyond

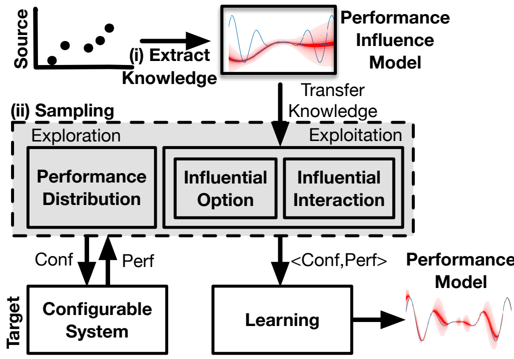

Figure 3: An overview of L2S.

random chance. *Option interactions* are considered as nonlinear effects on performance, where the influence of two options combined is different from the sum of their individual influences [57, 58].

Performance distributions defines a probability distribution over performance measures. In other words, it can be thought of as providing the probabilities of occurrence of different possible performance measures for a system in a specific environment. Previous research observed that performance distributions, in configurable systems, are multi-modal [28, 45]. This observation is a result of discrete decisions about options, *i.e.*, some options are highly-influential in performance, while others have little or no impact on the performance metric. For instance, in a database, page size may interact with the hard drive of the host machine and result in a performance measure that is far away from most of the other measurements, therefore, a new peak may appear in the performance distribution of the system.

3.1.2 *The representation of the extraction.*

For determining the influential options and their interactions, we need a machinery that *automatically* performs the statistical analysis for determining whether an option or an interaction is influential.

For capturing the influence of options and interactions on performance, we create a model using *stepwise linear regression* [22] from the source data. We used stepwise regression for two reasons. First, it is an efficient method to capture statistically significant options and interactions, in an iterative manner, that *scales* to a high-dimensional space because of its approximated nature by ignoring insignificant terms and thus learning a model with only significant options [50]. Second, it also provides a self-expressive model as an output that we can check whether the model can provide the information that we require for sampling in the target. If the source data is not representative enough to learn a credible model, we could measure more configurations to add to the source data. Note that we assume the measured samples in the source environment are readily available or cheap [31].

A regression model comprises several polynomial terms that determine the performance of the system under a configuration. Each term may refer to one or more options ($o_i \in O$), describing the influence of that option or an interaction [57]:

$$f(o_1, \cdots, o_d) = \beta_0 + \sum_{o_i \in O} \beta_i o_i + \sum_{o_i, o_j \in O} \beta_{i..j}(o_i..o_j), \qquad (1)$$

where $\beta \in \mathbb{R}$ represents the coefficients of the model, $\beta_i o_i$ represents the performance impact of individual options, and $\beta_{i..j}(o_i..o_j)$ represents the performance impact for interactions among multiple options (comprising not only quadratic terms, but also higher order terms up to the number of individual options). In this work, we assume configuration options are binary, so if an option appears in the performance model, the option is influential. Note that numeric and categorical options can be transformed into binary options by selecting two specific (typically extreme) values of the options corresponding to zero and one, in this transformation though we sacrifice preciseness. Since the appearance of a term in the model is based on a statistical analysis, as we will describe, the structure of the model gives us a direct means to identify individual options and interactions with strong influences (e.g., based on p-value).

As an example, consider a configurable database system with options o_1 (encryption), o_2 (compression), o_3 (statistics), o_4 (page-size) and o_5 (DBsize) and a corresponding model:

$$f(\cdot) = 2 + 3o_1 + o_2 + 9o_4 + 8o_5 - 7o_1o_2 + 0.5o_1o_4, \qquad (2)$$

Option o_3 (statistics) does not appear in the performance model, because it does not have a significant influence on the performance of the system based on the model. In contrast, we observe an influential interaction between o_1 (encryption) and o_2 (compression), which enhances the performance of the system when combined.

3.1.3 *The details of the extraction process.*

We use both *forward selection* and *backward elimination* of the options or their interactions to learn the regression model. More specifically, we use the p-value of an F-statistic (similar to T-test; determines if a group of variables are jointly significant [16]) to decide whether to add a term to the model or remove one. If a term is not currently in the model, the null hypothesis is that the term would have a zero coefficient if added to the model (i.e., $\beta_i = 0$). If there is sufficient evidence (e.g., p-value < 0.05) to reject the null hypothesis, the term will be added to the model. Conversely, if a term is currently in the model, the null hypothesis is that the term has a zero coefficient. If there is insufficient evidence to reject the null hypothesis (e.g., p-value $>$ 0.05), the term is removed from the model. More specifically:

(1) We fit an *initial model* to the data, and then compare the explanatory power of incrementally larger and smaller models.
(2) *Forward selection*: If any terms (options, e.g., o_1, or their interactions, e.g., o_1o_2) not in the model have p-values less than an entrance threshold (we set the threshold to 0.05), add the one with the smallest p-value and repeat this step, otherwise proceed to the next step.
(3) *Backward elimination*: If any model terms have p-values larger than an exit threshold (we set the threshold to 0.05), remove the one with the largest p-value and go to the previous step.
(4) Learning *terminates* when neither (2) nor (3) improve the model.

The output of the knowledge extraction phase is a performance-influence model specific to the source environment. Our code for learning and extraction process including some tutorials are made available at https://github.com/cmu-mars/model-learner/tree/tutorial. Note that we also extract the performance distribution of the source. To build a performance distribution for a system, we fit a probability distribution to the measured performance of the source using kernel density estimation [59].

3.2 Active sampling

L2S does iterative sampling that is guided by a predetermined exploitation-exploration rate, e.g., $eer = 0.1$. The exploitation-exploration rate determines the chance by which we take a sample from the list of configurations derived from the performance-influence model in Section 3.2.1 (*exploitation*) or takes a sample from the whole configuration space (*exploration*).

3.2.1 Exploitation. Based on the regression model from the source, we generate and prioritize a list of sample configurations as follows: (i) L2S sorts the terms in the regression model from highest absolute coefficients to the lowest absolute coefficient and store the sorted index in a list. (ii) L2S then selects, from the list, the term with the highest priority. (iii) L2S transforms each term into a configuration as follows: the options that appear in the selected term are enabled and the options that do not appear are disabled. For instance, the term $7o_1o_2$ in (2) will be transformed to a configuration in which o_1 and o_2 are enabled and all other options are disabled. The output of this step is a prioritized list of configurations that will be used in the iterative sampling phase.

3.2.2 Exploration. L2S takes samples according to the performance distribution of the source that was extracted prior to sampling. The intuition behind this is to make the performance distribution of the target "similar" to the source. The modes in the performance distributions are typically unproportioned. Therefore, in the sampling process, the configurations that are associated with the less populated modes may be missed and this results in learning a model that does not cover the response surface properly. This may lead to a performance model that yields inaccurate predictions. We use Kullback-Leibler (KL) divergence [11] to compare the similarity between the performance distributions. L2S randomly samples the configuration space and selects a configuration from this randomly set that make the target distribution more similar to the source. More specifically, it measures the KL divergence of the distributions in the source and target after adding a selected configuration to the target data using the corresponding performance measurement from the source environment as an approximation. L2S finally selects a configuration that makes the largest decrease to the KL divergence between the posterior distributions.

4 EXPERIMENTAL RESULTS

We design and run experiments to compare our L2S sampling approach with respect to state-of-the-art performance modeling and learning approaches in terms of *effectiveness* and *efficiency* of samples, *scalability* to high-dimensional spaces, *sensitivity* to the degree of similarity among source and target environments, and practical relevance to real configurable systems. Specifically, we will explore the following research questions:

RQ1*: Are the samples selected by* L2S *effective and efficient for learning accurate performance models in the target environment?* Answering this question gives evidence whether the selected samples are informative enough to efficiently learn a reliable and accurate performance model for the target environment. The rational behind this question is based on prior evidence that different samples have different information value for the learning process [31]. As part of RQ1, we are especially interested in the robustness of our approach with respect to size and difficulty of the models and severity of the environment change.

RQ2*: How does* L2S *perform for sampling real-world configurable software systems?* The answer to this question will shed some light whether our approach works in practice for real systems with moderately large number of configuration options and realistic scenarios for environmental changes that happen in practice.

To answer these questions, we designed and ran experiments using both synthetic datasets (to increase internal validity, explore scalability, and evaluate different environmental characteristics), as well as real-world configurable systems (to ensure external validity with regard to practical systems). This way, we address both internal and external validity [56]. We use three specific metrics for our evaluations: (i) prediction accuracy (for measuring effectiveness), (ii) number of measurements (for measuring efficiency), and (iii) learning time (for measuring scalability).

4.1 Experimental data

We evaluate our approach both using synthetic models and real-world systems. With synthetic models, we can repeat evaluations on many different models while controlling the size and complexity of the models and the degree of similarity between a source and a target model. That is, instead of using only a small number of real-world systems (which are expensive to measure to establish ground truth), we can explore a large number of models and explore the sensitivity of our solution and other state of the art approaches to various characteristics of models and environment changes. Since we can cheaply lookup performance results in the models, we can also explore models of large size that would be prohibitively expensive to sample in real-world systems (we could still build models with few samples, but testing accuracy in an evaluation requires many additional measurements that are expensive to acquire). In addition, adding a small number of real-world systems demonstrates that our results are applicable under realistic conditions.

4.1.1 Synthetic models. We carefully generated pairs of synthetic models (source and target) to simulate the performance behavior of configurable systems across environmental changes. Since performance modeling approaches and transfer strategies exploit common characteristics of real-world systems and environmental changes, we do not generate models entirely randomly, but generate models pseudo-randomly, such that they still follow typical characteristics identified in empirical studies of performance models and environmental changes [30, 59], for example, such that not all options interact in a system and such that influential options are often preserved across environments. This way, we can generate many models that differ in many aspects and in which we can explicitly control characteristics such as the relation between influential and non-influential options, but in which many other factors are randomized. To generate pairs of synthetic models, we proceed in two steps:

(i) **Generating a source model**. To generate source models that follow realistic distributions of performance values and interactions similar to real-world software systems, we use the *Thor* generator [59]. Thor uses measurement data from dozens of real systems (e.g., Apache Web server) and combines them with kernel density estimation to rescale the data to the options, interactions, and configurations of a given configuration space. Although the output performance model is synthetically generated, it follows interactions and option influences on the performance of real-world software systems. In our evaluation, we will specifically vary the number of options and the rate of relevant options in the generated models and let Thor generate all other aspects of the models in the described pseudo-random way. Such a model acts as a ground truth from which we can derive samples by emulated benchmark runs.

(ii) **Generating a target model**. To generate a target model, we change the source model, in a way that it maintains some (controlled level of) similarity to the source. Since no such generator exists, we developed GenPerf[1], that synthetically generates mutations of performance models: We generate and apply genetic mutations to the source model, while encoding the intended similarity characteristics as a fitness function. Again, the similarity characteristics are based on insights from empirical observations [30]. Our fitness function can optimize three similarity characteristics:

- *Correlation of source and target response:* By correlating the performance of all configurations across source and target, we can aim for very similar or very dissimilar models based on Pearson metric. For example, simple linear shifts (e.g., the target model is always 50 % slower than the source) have high correlations and are easy to exploit by some transfer learning strategies [63], whereas in practice both high and low correlations are observed [30].
- *Stable influential options and their interactions:* Determining which percentage of influential options and interactions remain stable between the source and target allow us to further control similarity. In practice, it is common that a large percentage of options and interactions remain influential, even across severe changes [30].
- *Similarity of performance distributions:* Finally, we control to what degree performance distributions of source and target are similar, measured with the KL divergence. Most environment changes maintain similar distributions despite larger changes [30].

A fitness function optimizes for all three similarity characteristics and pursues both 'easy' environment changes with strong similarities across all characteristics as well as 'hard' environment changes with low correlation but moderately high degrees of stability of influential options and performance distributions. Technically, we use a weighted average of the three similarity characteristics, in which weights control their relative importance.

To modify the model, we define four mutation operators: (i) **Add or remove option** to vary the number of influential options, (ii) **add or remove interaction** to varies the number of influential interactions, (iii) **change a coefficient** of a model term to vary the influence of options or interactions on performance, and (iv) **switch sign** of a model term's coefficient to drastically vary the influence of an option or interaction. We apply these mutation operators in an iterative way according to their probability for 1000 generations.

In our experiments, we systematically vary the difficulty of the environment change and distinguish three levels based on empirical observations: *easy:* $correlation > 0.9$, *moderate*:: $0.4 < correlation < 0.6$, and *hard:* $correlation < 0.1$. We kept the stable influence options proportional to the size of the model, equal to 1/5 of the number of options. Similarity we motivate decreasing of KL divergence across generations by formulating the fitness proportional to the inverse of divergence.

Example. Using Thor, we generate a model with 11 options based on the performance characteristics of the LLVM compiler. The resulting model has 5 influential options and no interactions:

$$source_model(\cdot) = 207.69o_1 + 16.06o_2 + 16.88o_3 + 12.03o_4 + 14.82o_5$$

Using the fitness function for a 'moderate' environment change, we find the following model with fitness equals 1 after genetic evolution with 1000 generations:

$$target_model(\cdot) = 10.95o_2 + 12.82o_3 + 9.44o_4 + 72.67o_5 + 43.01o_6 + 39.47o_3o_6 + 6.95o_1o_2o_4 - 12o_2o_3o_5$$

[1] https://github.com/pooyanjamshidi/GenPerf

Table 1: Overview of the real-world subject systems.

System	d	$\lvert C\rvert$	Environment changes
DNN	12	4 096	easy: azure/tf→aws-micro/tf hard: azure/tf→aws-micro/theano
XGBoost	11	2 048	easy: NUC4/covtype→azure/covtype hard: covtype→CNAE
Storm	11	2 048	easy: WordCount→RollingCount hard: SOL→RollingCount
SaC	50	71 267	easy: srad→hotspot hard: kmeans→nw

d: number of configuration options; C: configurations

Note how source and target are different, but still share certain characteristics, such as similar influential options and similar coefficient, but also some differences such as an extra option o_6 and extra interactions.

4.1.2 Real-world systems. In addition to our synthetic models, we extensively measured performance of four different configurable software systems, described in Table 1. These systems come from different domains, are written in different programming languages, and have a different number of options. All are from domains in which parameter tuning is important. For each system, we select two environment changes, one that we expect to be rather easy and one that we expect to be more difficult.

DNN is a set of algorithms for deep neural networks. We selected 6 hyper-parameters of the optimization algorithm [33] and 6 architecture-related parameters that determine the depth of the network. We used a time series dataset from the UCR Archive [9] as workload and measured the inference time as the response. As environment change, we varied hardware on which the network is deployed (easy) and varied hardware together with the specific deep learning framework (hard), see Table 1 for details. The measurement code, as well as experimental data, can be found here: hhttps://github.com/pooyanjamshidi/deeparch-xplorer.

XGBoost implements gradient boosting algorithms for supervised learning problems [17]. We selected 11 configuration options including the booster parameters and learning task parameters. We used two standard datasets as workload to train models and measure training time as the response variable. We varied both hardware (expected easy environment change) and workload (hard). The measurement code, as well as experimental data, can be found here: https://github.com/pooyanjamshidi/xgboost-xplorer.

Apache Storm [1] is a distributed real-time stream processing system. We selected 11 configuration options and measured throughput as response on standard benchmarks (SOL, WordCount, and RollingCount). As environment changes, we varied between similar workloads (easy) and dissimilar workloads (hard). The measurement code, as well as experimental data, can be found here: https://github.com/pooyanjamshidi/storm-xplorer.

SaC is a compiler for high-performance applications [53]. We reused measurement data from a prior study [30], in which 50 options were selected that control optional optimizer passes. The response is the execution time of a benchmark program. Again, we varied between similar benchmark programs (easy) and dissimilar ones (hard) as environment changes. The measurement data can be found here: https://github.com/pooyanjamshidi/ase17.

4.2 Experimental setup

In this section, we define the independent (IV) and dependent variables (DV) for the evaluation.

***IV-1**: Learning approaches.* We compare L2S (with GP as a learner) against the following state of the art approaches, which represent different strategies illustrated in Figure 2:

(1) Model-shift: A model-shift transfer learning approach [63].
(2) DataReuseTL: A data-reuse transfer learning approach, using random sampling in source and target environment [31].
(3) L2S+DataReuseTL: An integration of DataReuseTL with the L2S sampler; that is, we use L2S instead of random sampling in the target environment, to investigate whether the L2S sampling can be beneficial to data reuse approaches.
(4) Random+CART: Random sampling in the target environment and CART as a learner [19], without any transfer learning.

***IV-2**: Samples from target environment.* To identify how quickly each approach can learn with few samples in the target environment, we systematically vary the number of samples from the target environment between 2 and 70 samples.

***IV-3**: Size of the configuration space:* Using Thor (Section 4.1.1), we generate models with a different numbers of options (10, 20, 30, 50, 100) that determine the dimensionality; default 20.

***IV-4**: Model complexity:* Using Thor (Section 4.1.1), we generate models with different number of possible interactions that determine the complexity of the generated model. By default, 50% of all options will be influential and a model will have half (low), the same (medium), or twice (high) as many influential interactions as options; default high.

***IV-5**: Severity of environmental changes.* Using our genetic evolution approach (Section 4.1.1), we simulate environment changes at three different levels of severity (easy, moderate, and hard), as discussed in Section 4.1.1, with a default of hard.

***IV-6**: Samples from source environment.* Transfer learning approaches may perform differently depending on how many samples have been taken from the source environment. By default, we take 100 random samples, but consider also 1,000, and 10,000.

***IV-7**: Exploitation-exploration.* As final independent variable, we varied the exploitation-exploration rate of the learning process (Section 3.2) between 0 to 1; default 0.1.

***DV-1**: Prediction accuracy.* To assess effectiveness, we measure the prediction accuracy of the learned model as follows: Given the learned model and the ground truth from the synthetic model (or systematic measurements of the target system), we compare the predicted performance with the actual performance of the system on a large number of configurations (evaluation set). For small configuration spaces, we use all configurations that have not been sampled for learning as evaluation set, while for large configuration spaces, we evaluate 10,000 random configurations.

As specific metric, we use the *mean absolute percentage error (MAPE)* between the response predicted from the learned model ($\hat{f}(\boldsymbol{c})$) with the actual performance ($f(\boldsymbol{c})$) across all configurations in the evaluation set. The absolute percentage error for a single configuration c is defined as $ape(\hat{f}, f) = |\hat{f}(\boldsymbol{c}) - f(\boldsymbol{c})|/f(\boldsymbol{c}) \times 100$. Although we could use other metrics such as root mean squared percentage error, MAPE is scale independent and we can compare the errors across different data sets and response measures [27].

To compare approaches across different sample sizes (IV-2), we report the cumulative MAPE (mean average percentage error) that averages the errors for different sample sizes (reflecting the area under the accuracy curve): $c\text{-}mape = \Sigma_{i=1}^{T} ape(\hat{f}_i, f)/T$, where T is the number of iterations and $\hat{f}_i$ is the learned model at iteration i.

***DV-2**: Performance.* To assess efficiency, we additionally measure the time it takes to train a performance model for a given sample of measurement data. The time is measured on a MacBook Pro with 3.1 GHz Intel Core i7 CPU and 16GB of Memory.

4.3 Results (RQ1): Effectiveness and Efficiency

First, we plot accuracy (DV-1) and performance (DV-2) of different learning approaches (IV-1) for different sample sizes in the target environment (IV-2) in Figure 4, using a single synthetic model in which we used default values for all other independent parameters. For this model, after around 20 iterations (i.e., taking 20 samples from the target environment), L2S achieves near perfect predictions, while other approaches cannot reach the same accuracy even after exhausting the experimental budget at iteration 70. Without any transfer learning, even 500 iterations of pure random sampling (Random+CART does not reach similar accuracy as 20 targeted samples selected with L2S (see diamond shape annotation with 500 in Figure 4). Model-shift is generally ineffective and does not improve much with additional samples, due to negative transfer (with default hard difficulty, i.e., low correlation); DataReuseTL and L2S+DataReuseTL are reasonably effective but have higher learning costs, that increase quickly with additional samples taken.

While a single synthetic model cannot answer the research question, our experimental setting allows us to perform comparisons across many models. In the following, we will further systematically explore how the other independent variables influence the accuracy of the different learning approaches (IV-1). We, therefore, vary each independent variable (IV-3 to IV-7, one at a time) while keeping other variables at their defaults. We report distributions of our cumulative accuracy measure using box-plots from repeated observations with 5 synthetic models for each setting.

Model Complexity. With increased model complexity (IV-4), it becomes more difficult to capture the hidden structures in dimensional spaces. Our results, in Figure 5, show that L2S's effectiveness decreases slightly with increased complexity, but it still outperforms all other approaches.

Sensitivity to Change Severity. With increased change severity (IV-5), as expected, our results in Figure 8 show that L2S's effectiveness decreases. Nonetheless, it still outperforms all other approaches, some of which struggle even more with severe changes: The sharpest drop in effectiveness can be observed for Model-shift, because it highly relies on the correlation across environments—for hard changes with weak correlations, the models that are learned by Model-shift on the source are not representative for the target environment. Also, DataReuseTL performs better than the Model-shift approach, which can be attributed to the fact that DataReuseTL relies on source data which are much richer than a model that has been learned on them. More specifically, DataReuseTL uses a distance metric to find the appropriate regions of the source for predicting the target response. Therefore, even if part of the target response is related to the target, DataReuseTL can find the appropriate regions in the configuration space that are correlated and learn a more accurate model for the target environment [31].

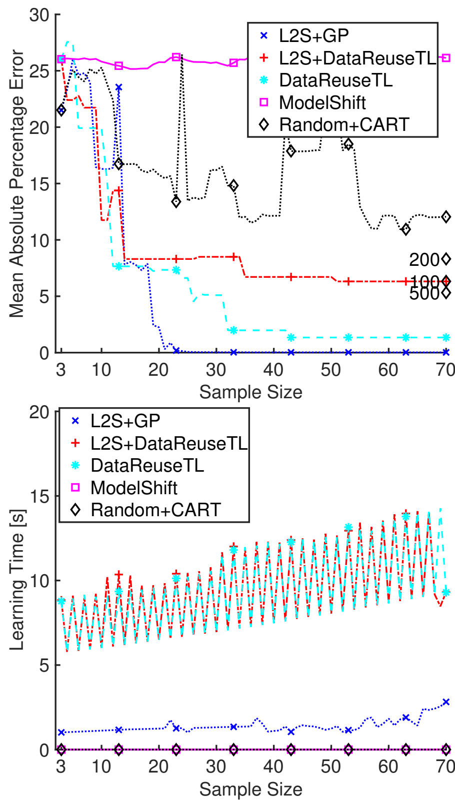

Figure 4: Comparing accuracy and performance of learning approaches for different sample sizes for a synthetic model with default parameters.

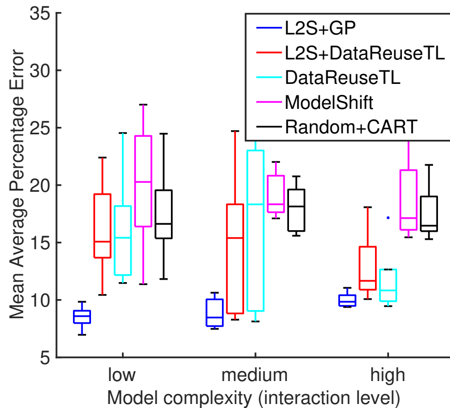

Figure 5: Sensitivity analysis with model complexity.

Sensitivity to source samples. Transfer learning approaches can usually benefit from more source samples because they can get more accurate source models from which they transfer knowledge. When varying the number of source samples (IV-6), we find that L2S accuracy has slightly been improved, but as expected the gain of accuracy for DataReuseTL was higher (plot omitted due to space restrictions). More importantly though, we observe, as shown in Figure 6 that learning times exponentially increase with more source samples, whereas L2S and Model-shift scale well with a reasonable overhead. The main reason is that L2S distill the data to knowledge

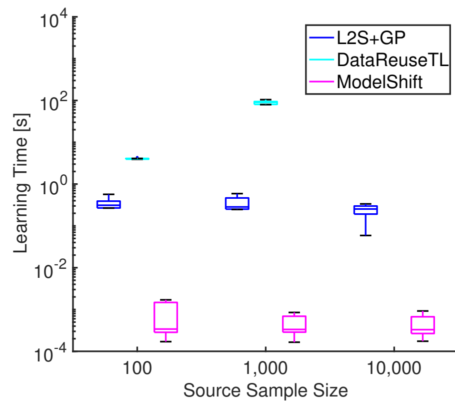

Figure 6: Runtime overhead of transfer learning approaches with respect to source sample size. Note that DataReuseTL did not finished within 5 hours for 10 000 source samples.

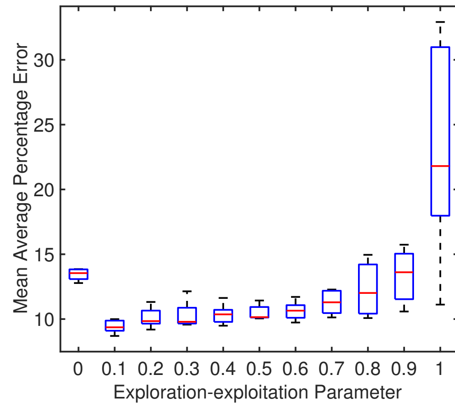

Figure 7: Sensitivity with the exploration-exploitation rate. A value of zero for the parameter means full exploitations, while one means full explorations.

that will be used for sampling, similarly Model-shift distill the data to a model for predicting the target, whereas, DataReuseTL reuses the data from the source in addition to the data that was taken from the target to learn a predictive model (cf. Figure 2). Therefore, the number of training samples that are used in the learning process of DataReuseTL are substantially higher than our approach and this contributes to the excessive runtime overhead of DataReuseTL.

Exploration-exploitation. Finally, we explore sensitivity to L2S's parameter that controls the combination of exploiting the knowledge from the source and exploration via pseudo-random sampling in the target (IV-7). Our results, in Figure 7, indicate that the pseudo-random sampling contributes to learning more accurate models, but that for effective learning, most samples should be drawn from exploiting source knowledge. Focusing mostly on exploitation with low values between 0.1 and 0.5 seems to be effective.

Answer to RQ1. The results confirm that L2S outperforms other approaches with respect to effectiveness and efficiency across many different models. The results are robust with regard to model size, model complexity, and severity of changes.

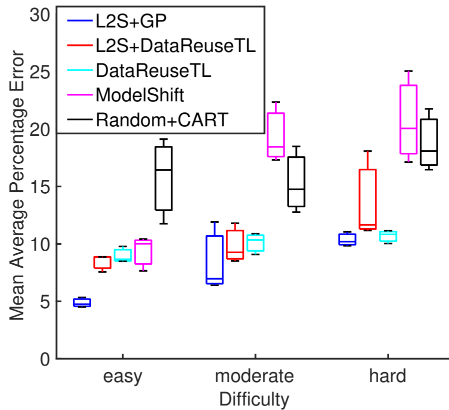

Figure 8: Sensitivity with different change difficulty.

4.4 Results (RQ2): Real Systems

To demonstrate applicability, we additionally evaluate effectiveness for learning performance models of real-configurable systems. As opposed to synthetically generated models and emulated environmental changes, we perform large-scale performance measurements of real systems in different environments. We measured the performance of the systems in Table 1 in different environments over the course of two months of 24/7 experimental time. We plot the prediction error (DV-1) curves for different learning strategies (IV-1) and different sample sizes (IV-2) for 3 of the 8 analyzed environment changes in Figure 9. The remaining plots are available at https://github.com/pooyanjamshidi/L2S.

Our results are consistent with those from synthetic models: L2S outperforms other approaches by sampling the space more efficiently. However, it is also comparable to L2S+DataReuseTL for XGBoost and SAC (both environment changes) and to Random+CART and DataReuseTL for Storm (hard change). Our results also scale to the large configuration space of SAC, where they even show a more pronounced benefit of L2S. Also as expected, L2S typically outperforms other approaches by a larger margin on the environment changes we judged to be more severe. Moreover, in large environmental changes, where DataReuseTL result in negative transfer (e.g., SaC), L2S contributes in learning a more accurate model in L2S+DataReuseTL by taking more informative samples.

Answer to RQ2. Experiments with eight environment changes of four real-world systems confirm that L2S outperforms other approaches in terms of effectiveness.

4.5 Insights

Our experimental results show that:

- Transfer learning based on *knowledge transfer* for sampling in the target environment, L2S, outperforms transfer learning based on *model-shift* and *data reuse* as well as non-transfer learning approaches, especially for severe environment changes.
- Transfer learning based on *model-shift* works well for 'easy' environment changes, but suffers from high prediction errors for more severe changes.
- Transfer learning based on *data-reuse* may result in a negative transfer for severe environmental changes leading to inaccurate models. It furthermore suffers from scalability issues because it caries over raw data, rather than abstracted knowledge that L2S and *model-shift* do.
- Transfer learning based on *data-reuse* that was enhanced by L2S sampling, i.e., L2S+DataReuseTL, often performed better than the original approach.

The key intuition behind these observations is that both classes of transfer learning (i.e., model shift and data reuse) assume that the performance measurements in the target environment are correlated with the ones in the source. However, this assumption only holds in small environmental changes [30]. On the contrary, L2S is not based on the assumption of high correlation across environments, but it is based on the pieces of knowledge that stay consistent across environments including large changes.

4.6 Threats to validity

Internal and construct validity. The use of synthetic models and a controlled environment allows us to rule out many alternative explanations for our results, such as measurement noise or the influence of model complexity. However, our results hinge on how representative our synthetic models are for real systems and environmental changes. As discussed, we increase representativeness by carefully designing approaches to generate source and target models (Section 4.1.1), using a published approach designed and validated for this purpose [59] and combining it with a custom approach for generating environmental changes designed based on the insight from a large-scale empirical study of real systems [30]. Additional confidence can be gained from the fact that the results from synthetic models align with those from eight real environmental changes. Nonetheless, our results must be interpreted within the constraints of how those models were generated.

For evaluations with real systems, measurement noise cannot be excluded and may affect the results, even though we carefully established ground truth by measuring the performance on dedicated systems and repeating the measurements several times.

We also compared the accuracy of different learning approaches. We implemented the approaches according to the description provided in the corresponding papers and we set the parameters according to the recommendations provided by the authors, but cannot exclude smaller differences due to implementation differences. To control the randomness of sampling, we repeated 3 times and averaged the results.

External validity. We used a diverse set of real-world configurable systems from different domains and a large number of purposefully selected environmental changes. Despite additional confidence from synthetic models, the reader must be careful when generalizing results beyond the studied systems. L2S only supports sampling configurable systems with binary options and the results cannot be generalized to non-binary configuration spaces.

5 RELATED WORK

Learning and optimization. Several models (e.g., support-vector machines [71], decision trees [40], Fourier sparse functions [72]), sampling strategies (e.g., active learning [57]), and optimization strategies (e.g., search-based optimization and evolutionary algorithms [21, 67]) have been used for performance prediction and tuning of configurable systems [66]. Several aspects from reducing measurement efforts, increasing prediction accuracy, and model reliability have been investigated.

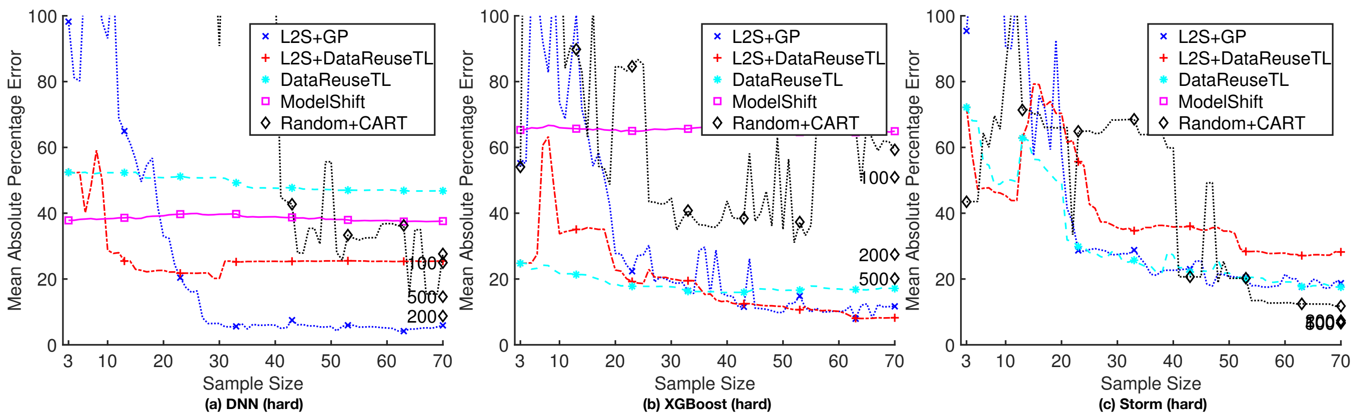

Figure 9: Prediction accuracy of approaches on real systems.

Optimization algorithms have also been applied to find best configurations using only a limited sampling budget: Recursive random sampling [70], hill climbing [68], direct search [73], optimization via guessing [47], Bayesian optimization [28], and multi-objective optimization [14]. Also, the importance of configuration options for optimization has been explored [4].

Our work is related to the performance-model learning and optimization research mentioned above: We learn performance-influence model on a source environment to extract information that can inform sampling for a target environment. Our aim is to learn a less costly but more accurate performance model for the target environment than learning from scratch. All existing learning approaches still remain valid and important when there is no source environment (i.e., previous measurements) be available.

Sampling. *Static sampling*. Although a proper sampling strategy is a key factor for decreasing the cost of learning performance models, this area is not entirely explored yet. Several experimental designs (*e.g.*, Full Factorial Designs, Fractional Factorial Designs, and Response Surface Designs) [38] have been developed to extract informative samples that guarantee certain statistical properties (e.g., option interactions). They have also been applied in the domain of configurable systems [19, 54, 57]. Several binary option sampling heuristics have been proposed [58] with the goal of selecting configurations to learn the influence of each individual binary option and their two-way interactions. These sampling strategies especially have been used to identify interactions that are related to detect bugs in combinatorial testing [36].

Adaptive sampling. Adaptive sampling designs for statistical experiments, also known as response-adaptive designs, are ones where the observations or the model learned on them are used to adjust the experiment as it is being run. Active sampling, as an instance of adaptive sampling, chooses subsequent samples based upon the models learned previously. For instance, Bayesian optimization [55], a sequential design strategy, have been used for performance tuning of configurable systems [28]. Since the performance model is unknown, the Bayesian strategy is to treat it as a random function and place a prior over it. The prior captures our beliefs about the performance behavior of the system. After observing the system performance for a configuration, the prior is updated to form the posterior distribution. The posterior distribution, in turn, is used to construct an acquisition function that determines what configuration should be measured next. Bayesian optimization has also been used with transfer learning to select appropriate data from the source domain [52]. Progressive and projective sampling are used for the performance prediction of configurable systems [54].

Our work is essentially categorized as a sampling approach that has been informed using external knowledge pieces that have been extracted from similar environments with a lower cost.

Transfer learning. The idea of using measurement data or extracting some types of knowledge from other environments has been used in different application areas: MapReduce applications [71], anomaly detection [60], micro-benchmarking [26], consistency-analysis of parameter dependencies [73], detection of performance regressions [15], and performance predictions based on similarity search [62].

Transfer learning has been used in the context of self-adaptive software [31], configuration dependency transfer across system version for optimization [8], co-design exploration for system software [6], model transfer across hardware [63], and configuration optimization [3]. Transfer learning has also been applied in defect predictions [35, 43, 44] and effort estimation [34].

Our work can be viewed as a type of transfer learning, but we do not shift a model [63, 65] or reuse source data [30], instead, we transfer knowledge across environments, using the insights from [30], to inform sampling (Figure 2, see Section 2).

6 CONCLUSIONS

The current approaches target a static scenario where one needs to learn an initial performance model for a specific environment. Here, we target the use case when the environment changes. We proposed L2S, a guided sampling strategy, which sits on top of any learning mechanisms and is able to exploit knowledge pieces from a similar environment and take informative samples. We have performed extensive experiments using over 100 synthetic models as well as 4 configurable systems demonstrating that L2S outperforms state of the art transfer learning approaches as well as traditional sampling and learning mechanisms for performance analysis.

ACKNOWLEDGMENT

This work has been supported in part by the National Science Foundation (awards 1318808, 1552944, and 1717022) and AFRL and DARPA (FA8750-16-2-0042). Siegmund's work is supported by the DFG under the contracts SI 2171/2 and SI 2171/3-1.

REFERENCES

[1] *Apache Storm.* 2018. http://storm.apache.org/.
[2] J. P. S. Alcocer, A. Bergel, S. Ducasse, and M. Denker. Performance evolution blueprint: Understanding the impact of software evolution on performance. In *Proc. of Working Conference on Software Visualization (VISSOFT)*, pages 1–9. IEEE, 2013.
[3] M. Artač, editor. *Deliverable 5.2: DICE delivery tools-Intermediate version.* 2017. http://www.dice-h2020.eu/.
[4] A. Biedenkapp, M. T. Lindauer, K. Eggensperger, F. Hutter, C. Fawcett, and H. H. Hoos. Efficient parameter importance analysis via ablation with surrogates. In *AAAI*, pages 773–779, 2017.
[5] C. M. Bishop. *Pattern recognition and machine learning.* Springer, New York, 2006.
[6] B. Bodin, L. Nardi, M. Z. Zia, H. Wagstaff, G. Sreekar Shenoy, M. Emani, J. Mawer, C. Kotselidis, A. Nisbet, M. Lujan, B. Franke, P. H. Kelly, and M. O'Boyle. Integrating algorithmic parameters into benchmarking and design space exploration in 3D scene understanding. In *Proceedings of the International Conference on Parallel Architectures and Compilation (PACT)*, pages 57–69. ACM, 2016.
[7] A. Brunnert, A. van Hoorn, F. Willnecker, A. Danciu, W. Hasselbring, C. Heger, N. Herbst, P. Jamshidi, R. Jung, J. von Kistowski, A. Koziolek, J. Kross, S. Spinner, C. Vögele, J. Walter, and A. Wert. Performance-oriented devops: A research agenda. *SPEC-RG-2015-01, RG DevOps Performance*, 2015.
[8] H. Chen, W. Zhang, and G. Jiang. Experience transfer for the configuration tuning in large-scale computing systems. *IEEE Trans. on Knowledge and Data Eng. (TKDE)*, 23(3):388–401, 2011.
[9] Y. Chen, E. Keogh, B. Hu, N. Begum, A. Bagnall, A. Mueen, and G. Batista. The ucr time series classification archive, July 2015. www.cs.ucr.edu/~eamonn/time_series_data/.
[10] J. Cito, P. Leitner, T. Fritz, and H. C. Gall. The making of cloud applications: An empirical study on software development for the cloud. In *Proc. Int'l Symp. Foundations of Software Engineering (FSE)*, pages 393–403. ACM, 2015.
[11] T. M. Cover and J. A. Thomas. *Elements of information theory.* John Wiley & Sons, 2012.
[12] A. Elkhodary, N. Esfahani, and S. Malek. Fusion: A framework for engineering self-tuning self-adaptive software systems. In *Proc. Int'l Symp. Foundations of Software Engineering (FSE)*, pages 7–16. ACM, 2010.
[13] N. Esfahani, A. Elkhodary, and S. Malek. A learning-based framework for engineering feature-oriented self-adaptive software systems. *IEEE Trans. Softw. Eng. (TSE)*, 39(11):1467–1493, 2013.
[14] A. Filieri, H. Hoffmann, and M. Maggio. Automated multi-objective control for self-adaptive software design. In *Proc. Int'l Symp. Foundations of Software Engineering (FSE)*, pages 13–24. ACM, 2015.
[15] K. C. Foo, Z. M. J. Jiang, B. Adams, A. E. Hassan, Y. Zou, and P. Flora. An industrial case study on the automated detection of performance regressions in heterogeneous environments. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 159–168. IEEE Press, 2015.
[16] J. Friedman, T. Hastie, and R. Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics New York, 2001.
[17] J. H. Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.
[18] A. Grebhahn, N. Siegmund, H. Köstler, and S. Apel. Performance prediction of multigrid-solver configurations. In *Software for Exascale Computing-SPPEXA 2013-2015*, pages 69–88. Springer, 2016.
[19] J. Guo, K. Czarnecki, S. Apel, N. Siegmund, and A. Wasowski. Variability-aware performance prediction: A statistical learning approach. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 301–311. IEEE, 2013.
[20] M. Harchol-Balter. *Performance modeling and design of computer systems: queueing theory in action.* Cambridge University Press, 2013.
[21] C. Henard, M. Papadakis, M. Harman, and Y. Le Traon. Combining multi-objective search and constraint solving for configuring large software product lines. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 517–528. IEEE, 2015.
[22] R. R. Hocking. A biometrics invited paper. the analysis and selection of variables in linear regression. *Biometrics*, 32(1):1–49, 1976.
[23] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. Dynamic knobs for responsive power-aware computing. In *In Proc. of Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.
[24] H. H. Hoos. Automated algorithm configuration and parameter tuning. In *Autonomous search*, pages 37–71. Springer, 2011.
[25] H. H. Hoos. Programming by optimization. *Communications of the ACM*, 55(2):70–80, 2012.
[26] K. Hoste, A. Phansalkar, L. Eeckhout, A. Georges, L. K. John, and K. De Bosschere. Performance prediction based on inherent program similarity. In *Proc. of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 114–122. ACM, 2006.
[27] R. J. Hyndman and A. B. Koehler. Another look at measures of forecast accuracy. *International journal of forecasting*, 22(4):679–688, 2006.
[28] P. Jamshidi and G. Casale. An uncertainty-aware approach to optimal configuration of stream processing systems. In *Proc. Int'l Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 39–48. IEEE, September 2016.
[29] P. Jamshidi, M. Ghafari, A. Ahmad, and C. Pahl. A framework for classifying and comparing architecture-centric software evolution research. In *Proc. of European Conference on Software Maintenance and Reengineering (CSMR)*, pages 305–314. IEEE, 2013.
[30] 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 *Proc. Int'l Conf. Automated Software Engineering (ASE)*. ACM, 2017.
[31] P. Jamshidi, M. Velez, C. Kästner, N. Siegmund, and P. Kawthekar. Transfer learning for improving model predictions in highly configurable software. In *Proc. Int'l Symp. Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 2017.
[32] P. Kawthekar and C. Kästner. Sensitivity analysis for building evolving and & adaptive robotic software. In *Proceedings of the IJCAI Workshop on Autonomous Mobile Service Robots (WSR)*, 7 2016.
[33] D. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
[34] E. Kocaguneli, T. Menzies, and E. Mendes. Transfer learning in effort estimation. *Empirical Software Engineering*, 20(3):813–843, 2015.
[35] R. Krishna, T. Menzies, and W. Fu. Too much automation? The bellwether effect and its implications for transfer learning. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 122–131. ACM, 2016.
[36] D. R. Kuhn, R. N. Kacker, and Y. Lei. *Introduction to combinatorial testing.* CRC press, 2013.
[37] P. Leitner and J. Cito. Patterns in the chaos - a study of performance variation and predictability in public IaaS clouds. *ACM Trans. on Internet Technology (TOIT)*, 16(3):15, 2016.
[38] D. C. Montgomery. *Design and analysis of experiments.* John Wiley & Sons, 2017.
[39] A. Murashkin, M. Antkiewicz, D. Rayside, and K. Czarnecki. Visualization and exploration of optimal variants in product line engineering. In *Proc. Int'l Software Product Line Conference (SPLC)*, pages 111–115. ACM, 2013.
[40] V. Nair, T. Menzies, N. Siegmund, and S. Apel. Faster discovery of faster system configurations with spectral learning. *arXiv preprint arXiv:1701.08106*, 2017.
[41] V. Nair, T. Menzies, N. Siegmund, and S. Apel. Faster discovery of faster system configurations with spectral learning. *Automated Software Engineering*, pages 1–31, 2017.
[42] V. Nair, T. Menzies, N. Siegmund, and S. Apel. Using bad learners to find good configurations. In *Proc. Int'l Symp. Foundations of Software Engineering (FSE)*, ESEC/FSE 2017, pages 257–267, New York, NY, USA, 2017. ACM.
[43] J. Nam and S. Kim. Heterogeneous defect prediction. In *Proc. Int'l Symp. Foundations of Software Engineering (FSE)*, pages 508–519. ACM, 2015.
[44] J. Nam, S. J. Pan, and S. Kim. Transfer defect learning. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 382–391. IEEE, 2013.
[45] J. Oh, D. Batory, M. Myers, and N. Siegmund. Finding product line configurations with high performance by random sampling. In *Proc. Int'l Symp. Foundations of Software Engineering (FSE)*. ACM, 2017.
[46] R. Olaechea, D. Rayside, J. Guo, and K. Czarnecki. Comparison of exact and approximate multi-objective optimization for software product lines. In *Proc. Int'l Software Product Line Conference (SPLC)*, pages 92–101. ACM, 2014.
[47] T. Osogami and S. Kato. Optimizing system configurations quickly by guessing at the performance. In *Int'l Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2007.
[48] C. Pahl, P. Jamshidi, and O. Zimmermann. Architectural principles for cloud software. *ACM Trans. on Internet Technology (TOIT)*, 2017.
[49] C. E. Rasmussen and C. K. Williams. *Gaussian processes for machine learning*, volume 1. MIT press Cambridge, 2006.
[50] E. B. Roecker. Prediction error and its estimation for subset-selected models. *Technometrics*, 33(4):459–468, 1991.
[51] M. T. Rosenstein, Z. Marx, L. P. Kaelbling, and T. G. Dietterich. To transfer or not to transfer. In *NIPS 2005 Workshop on Transfer Learning*, volume 898, 2005.
[52] S. Ruder and B. Plank. Learning to select data for transfer learning with Bayesian Optimization. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, 2017.
[53] SaC compiler. www.sac-home.org.
[54] A. Sarkar, J. Guo, N. Siegmund, S. Apel, and K. Czarnecki. Cost-efficient sampling for performance prediction of configurable systems. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 342–352. IEEE, November 2015.
[55] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. de Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2016.
[56] J. Siegmund, N. Siegmund, and S. Apel. Views on internal and external validity in empirical software engineering. In *Proc. Int'l Conf. Software Engineering (ICSE)*, volume 1, pages 9–19. IEEE, 2015.
[57] N. Siegmund, A. Grebhahn, S. Apel, and C. Kästner. Performance-influence models for highly configurable systems. In *Proc. Europ. Software Engineering Conf. Foundations of Software Engineering (ESEC/FSE)*, pages 284–294. ACM, August 2015.
[58] N. Siegmund, S. S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, and G. Saake. Predicting performance via automated feature-interaction detection. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 167–177. IEEE, 2012.
[59] N. Siegmund, S. Sobernig, and S. Apel. Attributed variability models: Outside the comfort zone. 2017.

[60] C. Stewart, K. Shen, A. Iyengar, and J. Yin. Entomomodel: Understanding and avoiding performance anomaly manifestations. In *Proc. Int'l Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 3–13. IEEE, 2010.

[61] J. Styles, H. H. Hoos, and M. Müller. Automatically configuring algorithms for scaling performance. In *Learning and Intelligent Optimization*, pages 205–219. Springer, 2012.

[62] E. Thereska, B. Doebel, A. X. Zheng, and P. Nobel. Practical performance models for complex, popular applications. In *SIGMETRICS Perform. Eval. Rev.*, volume 38, pages 1–12. ACM, 2010.

[63] P. Valov, J.-C. Petkovich, J. Guo, S. Fischmeister, and K. Czarnecki. Transferring performance prediction models across different hardware platforms. In *Proc. Int'l Conf. on Performance Engineering (ICPE)*, pages 39–50. ACM, 2017.

[64] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the International Conference on Management of Data*, pages 1009–1024. ACM, 2017.

[65] X. Wang, T.-K. Huang, and J. Schneider. Active transfer learning under model shift. In *International Conference on Machine Learning*, pages 1305–1313, 2014.

[66] M. Woodside, G. Franks, and D. C. Petriu. The future of software performance engineering. In *Future of Software Engineering (FOSE)*, pages 171–187. IEEE, 2007.

[67] F. Wu, W. Weimer, M. Harman, Y. Jia, and J. Krinke. Deep parameter optimisation. In *Proc. of the Annual Conference on Genetic and Evolutionary Computation*, pages 1375–1382. ACM, 2015.

[68] B. Xi, Z. Liu, M. Raghavachari, C. H. Xia, and L. Zhang. A smart hill-climbing algorithm for application server configuration. In *13th International Conference on World Wide Web (WWW)*, pages 287–296. ACM, 2004.

[69] T. Xu, L. Jin, X. Fan, Y. Zhou, S. Pasupathy, and R. Talwadker. Hey, you have given me too many knobs!: Understanding and dealing with over-designed configuration in system software. In *Proc. Int'l Symp. Foundations of Software Engineering (FSE)*, pages 307–319, New York, NY, USA, August 2015. ACM.

[70] T. Ye and S. Kalyanaraman. A recursive random search algorithm for large-scale network parameter configuration. In *Int'l Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 196–205. ACM, 2003.

[71] N. Yigitbasi, T. L. Willke, G. Liao, and D. Epema. Towards machine learning-based auto-tuning of mapreduce. In *Proc. Int'l Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 11–20. IEEE, 2013.

[72] Y. Zhang, J. Guo, E. Blais, and K. Czarnecki. Performance prediction of configurable software systems by Fourier learning. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 365–373. IEEE, 2015.

[73] W. Zheng, R. Bianchini, and T. D. Nguyen. Automatic configuration of internet services. *ACM SIGOPS Operating Systems Review*, 41(3):219–229, 2007.