arXiv:1911.01817v1 [cs.SE] 1 Nov 2019

Whence to Learn? Transferring Knowledge in Configurable Systems using BEETLE

Rahul Krishna, Vivek Nair, Pooyan Jamshidi, Tim Menzies, *IEEE Fellow*

Abstract—As software systems grow in complexity and the space of possible configurations increases exponentially, finding the near-optimal configuration of a software system becomes challenging. Recent approaches address this challenge by learning performance models based on a sample set of configurations. However, collecting enough sample configurations can be very expensive since each such sample requires configuring, compiling, and executing the entire system using a complex test suite. When learning on new data is too expensive, it is possible to use *Transfer Learning* to "transfer" old lessons to the new context. Traditional transfer learning has a number of challenges, specifically, (a) learning from excessive data takes excessive time, and (b) the performance of the models built via transfer can deteriorate as a result of learning from a poor source. To resolve these problems, we propose a novel transfer learning framework called BEETLE, which is a "bellwether"-based transfer learner that focuses on identifying and learning from the most relevant source from amongst the old data. This paper evaluates BEETLE with 57 different software configuration problems based on five software systems (a video encoder, an SAT solver, a SQL database, a high-performance C-compiler, and a streaming data analytics tool). In each of these cases, BEETLE found configurations that are as good as or better than those found by other state-of-the-art transfer learners while requiring only a fraction ($\frac{1}{7}$th) of the measurements needed by those other methods. Based on these results, we say that BEETLE is a new high-water mark in optimally configuring software.

Index Terms—Performance Optimization, SBSE, Transfer Learning, Bellwether.

1 Introduction

A problem of increasing difficulty in modern software is finding the right set of *configurations* that can achieve the best performance. As more functionality is added to the code, it becomes increasingly difficult for users to understand all the options a software offers [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12]. It is hard to overstate the importance of good configuration choices and the impact poor choices can have. For example, it has been reported that for Apache Storm, the throughput achieved using the worst configuration was *480 times slower* than that achieved by the best configuration [3].

Recent research has attempted to address this problem usually by creating accurate performance models that predict performance characteristics. While this approach is certainly cheaper and more effective than manual configuration it still incurs the expense of extensive data collection. This is undesirable, since the data collection must be repeated if the software is updated or the workload of the system changes.

Rather than learning new configurations afresh, in this paper, we ask if we can learn from existing configurations. Formally, this is called "transfer learning"; i.e., the transfer of information from selected "*source*" software configurations running on one environment to learn a model for predicting the performance of some "*target*" configurations in a different environment. Transfer learning has been extensively explored in other areas of software analytics [13], [14], [15], [16], [17], [18], [19], [20]. This is a practical possibility since often when a software is being deployed in a new environment[1], there are examples of the system already executing under a different environment. Note that, to the best of our knowledge, this paper is among the earliest studies to apply transfer learning for performance optimization. Our proposed transfer learning method is significantly faster than any currently available transfer learning methods in identifying near-optimum configurations for a software system.

Transfer learning can only be useful in cases where the source environment is similar to the target environment. If the source and the target are not similar, knowledge should not be transferred. In such situations, transfer learning can be unsuccessful and can lead to a *negative transfer*. Prior work on transfer learning focused on "*What to transfer*" and "*How to transfer*", by implicitly assuming that the source and target are related to each other. However, those work failed to address "*From where (whence) to transfer*" [22]. Jamshidi et al. [21] alluded to this and explained when transfer learning works but, did not provide a method which can help in selecting a suitable source. In this paper, we focus on solving that problem.

The issue of identifying a suitable source is a common problem in transfer learning. To address this, some researchers [23], [24], [25], [20] have recently proposed the use of the *bellwether* effect, which states that:

> "When analyzing a community of software data, there is at least one exemplary source data, called bellwether(s), which best defines predictors for all the remaining datasets . . . "

Inspired by the success of bellwethers in other areas, this paper defines and evaluates a new transfer learner for software configuration called Bellwether Transfer Learner

• *V. Nair, R. Krishna, and T. Menzies are with the Department of Computer Science, North Carolina State University, Raleigh, USA.*
E-mail: {vivekaxl, i.m.ralk and tim.menzies}@gmail.com
• *P. Jamshidi is with the Department of Computer Science and Engineering, Univesity of South Carolina, USA.*
E-mail: pooyan.jamshidi@gmail.com

Manuscript received December XX, 2018.

1. We use this term ***environment*** as per its precise technical definition from the transfer learning literature. According to Jamshidi et al. [21], an "environment" refers to the external factors influencing the performance of the system such as workload, hardware, version of the software. For more technical details on this term, see §3.

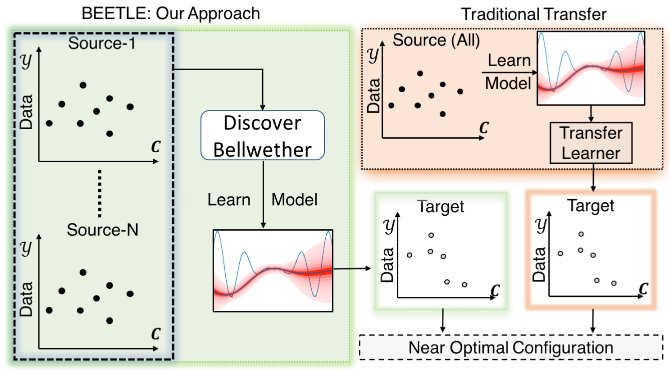


Fig. 1: Traditional Transfer Learning compared with using bellwethers to discover near optimal configurations.

(henceforth referred to as **BEETLE**). BEETLE can perform knowledge transfer using just a few samples from a carefully identified source environment(s).

For evaluation, we explore five real-world software systems from different domains– a video encoder, a SAT solver, a SQL database, a high-performance C-compiler, and a streaming data analytics tool (measured under 57 eviroments overall). In each case, we discovered that BEETLE found configurations as good as or better than those found by other state-of-the-art transfer learners while requiring only $\frac{1}{7}$-th of the measurements needed by those other methods. Reducing the number of measurements is an important consideration since collecting data in this domain can be computationally and monetarily expensive.

This article is structured as follows: The rest of this section highlights our key contributions (§1.1) and presents the research questions (§1.2) answered in this paper. §2 presents some motivation for this work. §3 describes the problem formulation and explains the concept of Bellwethers. §4 describes BEETLE followed by a quick overview of the prior work in transfer learning in performance configuration optimization in Section §5. In §6, we present experimental setup and followed by answers to research questions in §7. In §8, we discuss our findings further and answer some additional questions pertaining to our results. §9 discusses some threats to validity, related work and conclusion are presented in §10 and §11 respectively.

1.1 Contributions

Overall, this work makes the following contributions:

- *Source selection*: We show that the *bellwether effect* exists in performance optimization and that we can use this to discover suitable sources (called bellwether environments) to perform transfer learning. (§7)
- *Cheap source selection:* BEETLE, using bellwethers, evaluates at most $\approx 10\%$ of the configuration space (§4).
- *Simple Transfer learning using Bellwethers:* We develop a novel transfer learning algorithm using bellwether called BEETLE that exploits the bellwether environment to construct a simple transfer learner (§4).
- *More effective than non-transfer learning:* We show that using the BEETLE is *better* than non-transfer learning approaches. It is also lot more economical. (§7).
- *More effective than state-of-the-art methods:* Configurations discovered using the bellwether environment are better than the state-of-the-art methods [26], [27] (§7).
- *Reproduction Package:* To assist other researchers, a reproduction package with all our scripts and data are available online (see https://git.io/fjsky).

1.2 Research questions

RQ1: Does there exist a Bellwether Environment? In the first research question, we ask if there exist bellwether environments to train transfer learners for performance optimization. We hypothesize that, if these bellwether environments exist, we can improve the efficacy of transfer learning.

Result: We find that bellwether environments are prevalent in performance optimization. That is, in each of the software systems, there exists at least one environment that can be used to construct superior transfer learners.

RQ2: How many performance measurements are required to discover bellwether environments? Having established that bellwether environments are prevalent, the purpose of this research question is to establish how many performance measurements are needed in each of the environments to discover these bellwether environments.

Result: We can discover a potential bellwether environment by measuring as little as 10% of the total configurations across all the software system.

RQ3: How does BEETLE compare to non-transfer learning methods? The alternative to transfer learning is just to use the target data (similar to methods proposed in prior work) to find the near-optimal configurations. In the literature are many examples of this "non-transfer" approach [28], [7], [12], [10] and for our comparisons, we used the performance optimization model proposed by Nair et al. [10] at FSE '17.

Result: Our experiments demonstrate that transfer learning using bellwethers (BEETLE) outperforms other methods that do not use transfer learning both in terms of cost and the quality of the model.

RQ4: How does BEETLE compare to state-of-the-art transfer learners? Finally, in this research question we compare BEETLE with two other state-of-the-art transfer learners used commonly in performance optimization (for details see §5). The purpose of this research question is to determine if a simple transfer learner like BEETLE with carefully selected source environments can perform as well as other complex transfer learners that do not perform any source selection.

Result: We show that a simple transfer learning using bellwether environment (BEETLE) just as good as (or better than) current state-of-the-art transfer learners.

2 Motivation

With the appearance of continuous software engineering and devops, *configurability* has become a primary concern of software engineers. System administrators today develop and use different versions software programs under running several different workloads and in numerous environments. In doing so, they try to apply software engineering methods to best configure these software systems. Despite their best efforts, the available evidence is that they need to be better assited in making all the configuration decisions. Xu et al. [1] reports that, when left to their own judgementments, developers ignore up to 80% of configuration options, which exposes them to many potential problems. For this reason, the research community is devoting a lot of effort to configuration studies, as witnessed by many recent software engineering research publications [4], [5], [6], [7], [8], [12], [9], [10], [29]. For details, see §10 for the additional related work.

Without automatic support (e.g., with systems like BEETLE), humans find it difficult to settle on their initial choice for software configurations. The available evidence [2], [3], [30] shows that system administrators frequently make poor configuration choices. Typically, off-the-shelf defaults are used, which often behave poorly. There are various examples presented in the literature which have established that choosing default configuration can lead to sub-optimal performance. For instance, Van Aken et al. report that the default MySQL configurations in 2016 assume that it will be installed on a machine that has 160MB of RAM (which, at that time, was incorrect by, at least, an order of magnitude) [2]. Also, Herodotou et al. [30] report that default settings for Hadoop results in the *worst possible* performance.

Traditional approaches to finding good configuration are very resource intensive. A typical approach uses sensitivity analysis [31], where performance models are learned by measuring the performance of the system under a limited number of sampled configurations. While this approach is cheaper and more effective than manual exploration, it still incurs the expense of extensive data collection about the software [28], [7], [4], [10], [12], [29], [8], [32], [3]. This is undesirable since this data collection has to be repeated if ever the software is updated or the environment of the system changes abruptly. While we cannot tame the pace of change in modern software systems, we can reduce the data collection effort required to react to that change. The experiments of this paper make the case that BEETLE scales to some large configuration problems, better than the prior state of the art. Further, it does so using fewer measurements than existing state-of-the-art methods.

Further, we note that BEETLE is particularly recommended in highly dynamic projects where the environments keep changing. When context changes, so to must the solutions applied by software engineers. When frequently recomputing best configurations, it becomes vitally important that computation cost is kept to a minimum. Amongst the space of known configuration tools, we most endorse BEETLE for very dynamic environments. We say this since, of all the systems surveyed here, BEETLE has the lowest CPU cost (and we conjecture that this is so since BEETLE makes the best use of old configurations).

As a more concerete example, consider an organization that runs, say, N heavy Apache Spark workloads on the cloud. To optimize the performance of Apache Spark on the given workloads, the DevOps Team need to find the optimal solutions for each of these workloads, i.e., conduct performance optimization N times. This setup has two major shortcomings–

Hardware Change: Even though the DevOps engineer of a software system performs a performance optimization for a specific workload in its staging environment, as soon as the software is moved to the production environment the optimal configuration found previously may be inaccurate. This problem is further accentuated if the production environment changes due to the ever-expanding cloud portfolios. It has been reported that cloud providers expand their cloud portfolio more than 20 times in a year [33].

Workload Change: The developers of a database system can optimize the system for a read-heavy workload, however, the optimal configuration may change once the workload changes to, say, a write-heavy counterpart. The reason is that if the workload changes, different functionalities of the software might get activated more often and so the nonfunctional behavior changes too. This means that as soon as a new workload is introduced (new feature in the organization's product) or if the workload changes, the process of performance optimization needs to be repeated.

Given the fragility of traditional performance optimization, it is imperative that we develop a method to learn from our *previous experiences* and hence reduce the burden of having to find optimum configurations ad nauseam.

3 Definitions and Problem Statement

Configuration: A software system, $\mathcal{S}$, may offer a number of configuration options that can be changed. We denote the total number of configuration options of a software system $\mathcal{S}$ as N. A configuration option of the software system can either be a (1) continuous numeric value or a (2) categorical value. This distinction is very important since it impacts the choice of machine learning algorithms. The configuration options in all software systems studied in this paper are a combination of both *categorical* and *continuous* in nature. The learning algorithm used in this paper namely, *Regression Trees*, are particularly well suited to handle such a combination of continuous and categorical data.

A configuration is represented by c_i, where i represents the i^{th} configuration of a system. A set of all configurations is called the *configuration space*, denoted as $\mathcal{C}$. Formally, $\mathcal{C}$ is a Cartesian product of all possible options $\mathcal{C}$ = Dom(c_1) × Dom(c_2) × ... × Dom(c_N), where Dom(c_i) is either $\mathbb{R}$ (Real Numbers) or $\mathbb{B}$ (Catergorical/Boolean value) and N is the number of configuration options.

	`ATOMIC`	`USE_LFS`	`SECURE`	LATENCY (μs)
c_1	0	0	0	100
c_2	0	0	1	150
⋮	⋮	⋮	⋮	⋮
c_N	1	1	1	400

Fig. 2: Some configuration options for `SQLite`.

As a simple example, consider a subset of configuration options from `SQLite`, i.e., $\mathcal{S} \equiv$ `SQLite`. This is shown in Figure 2. The subset of `SQLite` offers three configuration options namely, `ATOMIC` (atomic delete), `USE_LFS` (use large file storage), and `SECURE` (secure delete), ie., $N = 3$. The last column contains the *latency* in μs when various combinations of these options are chosen.

Environment: As defined by Jamshidi et al. [21], the different ways a software system is deployed and used is called its *environment* (e). The environment is usually defined in terms of: (1) ***workload*** (w): the input which the system operates upon; (2) ***hardware*** (h): the hardware on which the system is running; and (3) ***version*** (v): the state of the software.

Note that, other environmental changes might be possible (e.g., JVM version used, etc.). For example, consider software system Apache Storm, here we must ensure that an appropriate JVM is installed in an environment before it can be deployed in that environment. Indeed, the selection of one version of a JVM over another can have a profound performance impact. However, the perceived improvement in the performance is due to the optimizations in JVM, not the original software system being studied. Therefore, in this paper, we do not alter these other factors which do not have a direct impact on the performance of the software system. The following criteria is used to define an environment:

1) Environmental factors of the software systems that we can vary in the deployment stack of the system. This prevents us from varying factors such as the JVM version, CPU frequency, system software, etc., which define the deployment stack and not the software system.
2) Common changes developers choose to alter in the software system. In practice, it is these factors that affect the performance of systems the most [21], [27], [26], [34].
3) Factors that are most amenable for transfer learning. Preliminary studies have shown that factors such as workload, hardware, and software version lend themselves very well to transfer learning [21], [27].

For a more detailed description of the factors that were changed and those that were left unchanged, see Table 1.

Formally, we say an environment is $\mathbf{e} = \{w, h, v\}$ where $w \subseteq W$, $h \subseteq H$, and $v \subseteq V$. Here, W, H, V are the space of all possible hardware changes H; all possible software versions V, and all possible workload changes W. With this, the environment space is defined as $\mathcal{E} \subset \{W \times H \times V\}$, i.e., a subset of environmental conditions $\mathbf{e}$ for various workloads, hardware, and environments.

Performance: For each enviroment $\mathbf{e}$, the instances in our data are of the form $\{(c_1, y_1), ..., (c_N,\ y_N)\}$, where c_i is a vector of configurations of the i-th example and it has a corresponding performance measure $y_i \in Y_{S,c,e}$ associated with it. We denote the performance measure associated with a given configuration (c_i) by $y = f(c^i)$. We consider the problem of finding the near-optimal configurations (c^*) such that $f(c^*)$ is better than other configurations in $C_{A,e}$, i.e.,

$$\begin{array}{ll} f(c^*) \leq f(c)\ \forall c \in C_{A,h,w,v} \setminus c^* & \text{for min objective} \\ f(c^*) \geq f(c)\ \forall c \in C_{A,h,w,v} \setminus c^* & \text{for max objective} \end{array}$$

Bellwethers: In the context of performance optimization, the bellwether effect states that: *For a configurable system, when performance measurements are made under different environments, then among those environments there exists one exemplary environment, called the bellwether, which can be used determine near optimum configuration for other environments for that system.* We show that, when performing transfer learning, there are exemplar source environments called the bellwether environment(s) ($\mathcal{B} = e_{s1}, e_{s2}, ..., e_{sn} \subset E$), which are the best source environment(s) to find near-optimal configuration for the rest of the environments ($\forall e \in E \setminus \mathcal{B}$).

Problem Statement: The problem statement of this paper:

Find a near-optimal configuration for a target environment (S_{e_t}), by learning from the measurements ($\langle c, y \rangle$) for the same system operating in different source environments (S_{e_s}).

In other words, we aim to reuse the measurements from a system operating in an environment to optimize the same system operating in the different environment thereby reducing the number of measurements required to find the near-optimal configuration.

4 BEETLE: BEllwethER Transfer LEarner

This section describes BEETLE, a bellwether based approach that finds the near-optimal configuration using the knowledge in the "bellwether" environment. BEETLE can be separated into two main steps: (i) *Discovery:* finding the bellwether environment, and (ii) *Transfer:* using the bellwether environment to find the near-optimal configuration for target environments. These steps will are explained in greater detail in §4.1 and §4.2. We outline it below,

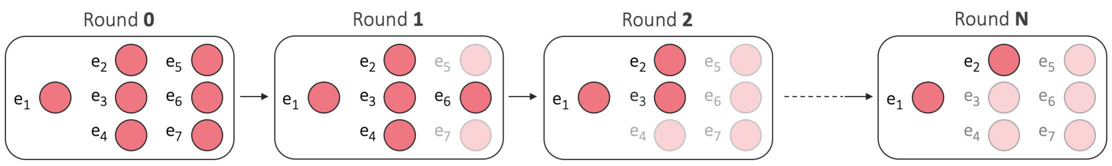
1) *Discovery:* Leverages the existence of the bellwether effect to *discover* which of the available environments are best suited to be a *source enviroment* (known as the *bellwether environment*). To do this, BEETLE uses a **racing algorithm** to sequentially evaluate candidate environments [35]. In short,
   a) A fraction (about 10%) of all available data is sampled. A prediction model is built with these sampled datasets.
   b) Each enviroment is used as a *source* to build a prediction model and all the others are used as *targets* in a round-robin fashion.
   c) Performance of all the enviroments are measured and are statistically ranked from the best source environemnt to the worst. Environments with a *poor* performance (i.e., those ranked last) are eliminated.
   d) For the remaining enviroments, another 10% of the samples are added and the steps (a)–(c) are repeated.
   e) When the ranking order doesn't change for a fixed number of repeats, we terminate the process and nominate the best ranked enviroment(s) as the bellwether.
2) *Transfer:* Next, to perform transfer learning, we just use these bellwether environments to train a performance prediction model with **Regression Trees**.

We conjecture that once a *bellwether source environment* is identified, it is possible to build a simple transfer model without any complex methods and still be able to discover near-optimal configurations in a target environment.

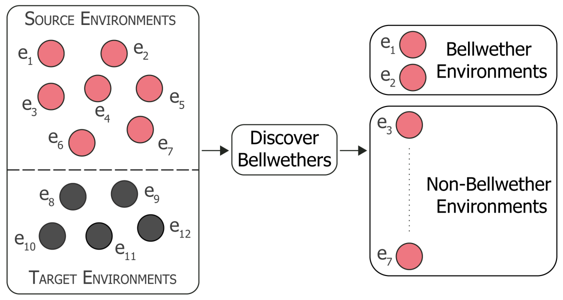
4.1 *Discovery*: Finding Bellwether Environments

In the previous work on bellwethers [20], the discovery process involved a round-robin experimentation comprised of the following steps:

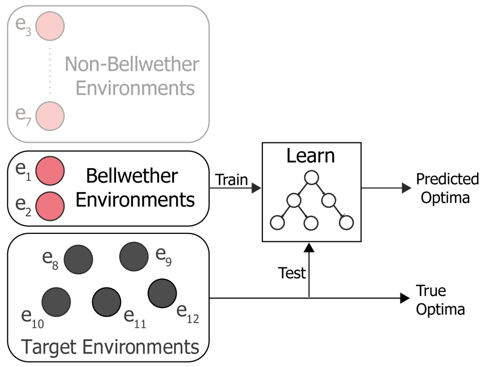
1) Pick an enviroment $\mathbf{e}_i$ from the space of all available enviroments, i.e., $\mathbf{e}_i \in \mathcal{E}$.
2) Use $\mathbf{e}_j$ as a *source* to build a prediction model.



(a) Elimination of non-bellwether environments



(b) An example of *Discovery*



(c) An example of *Transfer*

```
def FindBellwether(sources, frac, budget, thres, lives):
    while lives or cost > budget:
        "Sample configurations"
        for i, source in enumerate(sources):
            sampled_source[i].add_sample(source, step_size)
        "Get cost"
        cost += get_cost(sampled_source)
        "Evaluate pair−wise performances"
        cur_ranks = get_perf_rankings(sampled)
        "Loose life if no sources are removed"
        if prev_ranks == cur_ranks:
            lives −= 1
            prev_ranks = cur_ranks
            continue
        else:
            "Remove non−bellwether environments"
            sampled_sources = remove_non_bellwethers(cur_ranks)
    return sample_sources[cur_rank[0]]
```

(d) Pseudocode for *Discovery*

```
def BEETLE(sources, target, budget):
    "Find the bellwether environments"
    bellwether = FindBellwether(sources, frac, budget, thres, lives)
    "If there are more than one bellwethers, pick one at random"
    if len(bellwether) > 1:
        bellwether = random.choice(bellwether)
    "Train a prediction model with the bellwether"
    prediction_model = regTree.train(bellwether)
    "Predict the performance various configurations in target"
    best_config = prediction_model.predict_best(target)
    "Return the configuration with the best performance"
    return best_config
```

(e) Pseudocode for *Transfer*

Fig. 3: BEETLE framework and Pseudocode.

3) Using all the *other* enviroments $\mathbf{e}_j \in \mathcal{E}$ and $\mathbf{e}_j \neq \mathbf{e}_i$ as the *target*, determine the prediction performance of $\mathbf{e}_i$.
4) Next, repeat the steps by choosing a different $\mathbf{e}_i \in \mathcal{E}$
5) Finally, rank the performances of all the enviroments and pick the best ranked enviroment(s) as bellwether(s).

The above methodology is a form of an exhaustive search. While it worked for the relatively small datasets in [23], [20], the amount of data in this paper is sufficiently large (see Table 1) that scoring all candidates using every sample is too costly. More formally, let us say that we have M candidate enviroments with N measurements each. The classical approach, described above, will construct M models. If we assume that the model construction time is a function of number of samples $f(N)$, then for one round in the round-robin, the computation time will $O(M \cdot f(N))$. Since this is repeated M times for each enviroment, the total computational complexity is $O(M^2 \cdot f(n))$. When M and/or N is/are extremely large, it becomes necessary to seek alternative methods. Therefore, in this paper, we use a racing algorithm to achieve computational speedups.

Instead of evaluating every available instance to determine the best source enviroment, *Racing algorithms* take the following steps:

- Sample a small fraction of instances from the original enviroments to minimize computational costs.
- Evaluate the performance of enviroments statistically.
- Discarded the enviroments with the poorest performance.
- Repeated the process with the remaining datasets with slightly larger sample size.

Figure 3a shows how BEETLE eliminates inferior environments at every iteration (thus reducing the overall number of environments evaluated). Since each iteration only uses a small sample of the available data, the model building time also reduces significantly. It has been shown that racing algorithms are extremely effective in model selection when the size of the data is arbitrarily large [35], [36].

In Figure 3b, we illustrate the discovery of the bellwether environments with an example. Here, there are two groups of environments:

(i) Group 1: Environments $e_1, e_2, ..., e_7$, for which performance measurements have been gathered. One or more these environment(s) are potentially bellwether(s).
(ii) Group 2: Environments $e_8, e_9, ..., e_{12}$, these represent the target environments, for which need to determine an optimal configuration.

In the discovery process, BEETLE's objective is to *find bellwethers* from among the environments in Group 1. And, later in the *Transfer* phase, we use the bellwether enviroments to find the near-optimal configuration for the target environments from Group 2.

Fig. 3d outlines a pseudocode for the algorithm used to find bellwethers. The key steps are listed below:

- *Lines 3–5*: Randomly sample a small subset of configurations from the source environments. The size of the subset (of configurations) is controlled by a predefined parameter *frac*, which defines the percent of configurations to be sampled in each iteration.
- *Line 6–7*: Calculate sampling cost for the configurations.
- *Line 8–9*: Use the sampled configurations from each environment as a *source* build a prediction model with regression trees. For all the remaining enviroments, this regression tree model is used to predict for optimum configuration. After using every enviroment as a *source*, the environments are ranked from best to worst using the evaluation criteria discussed in §6.2.
- *Line 10–14*: We check to see if the rankings of the enviroments have changed since the last iteration. If not, then a "life" is lost. We go back to *Line 3* and repeat the process. When all lives are expired, or we run out of the budget, the search process terminates. This acts as an early stopping criteria, we need not sample more data if those samples do not help in improving the outcome.
- *Line 15–17*: If there is some change in the rankings, then new configuration samples are informative and the environments that are *ranked last* are eliminated. These environments are not able to find near-optimal configurations for the other environments and therefore cannot be bellwethers.
- *Line 18*: Once we have exhausted all the lives or the sampling budget, we simply return the source project with the best rank. These would be the bellwether enviroments.

On *line 6–7* we measure the sampling cost. In our case, we use the number of samples as a proxy for cost. This is because each measurement consumes computational resources which in turn has a monetary cost. Therefore, it is commonplace to set a budget and sample such that the budget is honored. Our choice of using the number of measurements as a cost measure was an engineering judgment, this can be replaced by any user-defined cost function such as (1) the actual cost, or (2) the wall-clock time. The accuracy of either of the above are depenedent on the business context. If one is either constrained by runtime or there is large variance in the measurements of time per configuration, then the wallclock time might be a more reasonable measure. On the other hand, if the cost of measurements is the limiting factor, it makes sense to use the actual measurement cost. Using number of samples encompasses these two factors since it both costs more money and takes time to obtain more samples. In the ideal case, we would like to have performance measurements for all possible configurations of a software system. But this is not practical because certain systems have over 2^{50} unique configurations (see Table 1).

It is entirely possible for the *FindBellwether* method to identify multiple bellwethers (e.g., in the case of 3b the bellwethers were e_1 and e_2). When mutliple bellwethers are found, we may use (a) any one of the bellwether enviroments at random, (b) use all the enviroments, or (c) use heuristics based on human intuition. In this paper, we pick one enviroment from among the bellwethers at random. As long as the chosen project is among the bellwether enviroments, the results remain unchanged.

The BEETLE approach assumes that a *fixed* set of enviroments exist from which we pick one or more bellwethers. But, approach would work just as well where new measurements from new enviroments are added. Specifically, when more environments are added into a project, it is possible that the newly added environment could be the bellwether. Therefore, we recommend repeating *FindBellwether* method prior to using the new enviroment. Note that, repeating *FindBellwether* for new enviroments would add minimal computational overhead since the measurements have already been made for the new enviroments. Also note that, this approach of revisiting *FindBellwether* on availability of new data, has been previously been proposed in other domains in software engineering [23], [20].

4.2 *Transfer*: Using the Bellwether Environments

Once the bellwether environment is identified, it can be used to find the near-optimal configurations of target environments. As shown in Fig. 3c, *FindBellwether* eliminates enviroments that are not potentially bellwethers and returns only the candidate bellwether enviroments. For the remaining target enviroments, we use the model built with the bellwether enviroments to identify the near optimal configurations.

Figure 3e outlines the pseudocode used to perform the transfer. The key steps are listed below:

- *Line 2-3*: We use the *FindBellwether* from 3d to identify the bellwether enviroments.
- *Line 4-6*: If there exists more than one bellwether, we randomly chose one among them be used as the bellwether enviroment.
- *Line 7-8*: The configurations from the bellwether and their corresponding performance measures are used to build a prediction model using regression trees.
- *Line 9-10*: Predict the performances of various configurations from the target enviroment.
- *Line 11-12*: Return the best configuration for the target.

Note that, on *Line 10*, we use *regression trees* to make predictions. It has been the most preferred prediction algorithm in this domain [28], [7], [10]. This is primarily because much of the data used in this domain are a mixture numerical and categorical attributes. Given configuration measurement in the form $\{(c_i, y_i)\}$, c_i is a vector of categorical/numeric values and y_i is a continuous numeric value. For such data, regression trees are the best suited prediction algorithms [12], [29], [32], [26]. This is because the regression trees are built by recursively splitting the configuration vectors into a root node configuration and subsets of children node configurations. This splitting is based on a set of splitting rules that work equally well for both categorical and numeric data.

5 Other Transfer Learning Methods

This section describes the methods we use to compare BEETLE against. These alternatives are (a) two state-of-the-art transfer learners for performance optimization: Valov et al. [26] and Jamshidi et al. [27]; and (b) a non-transfer learner: Nair et al. [12].

5.1 Transfer Learning with Linear Regression

Valov et al. [26] proposed an approach for transferring performance models of software systems across platforms

```
def LinearTransform(source, target, training_coef, budget):
    "Construct a prediction model"
    prediction_model=regTree.train(source, training_coef)

    "Sample random measurements"
    s_samp = source.sample(budget)
    t_samp = target.sample(budget)

    "Get performance measurements"
    s_perf=get_perf(s_samp)
    t_perf=get_perf(t_samp)

    "Train a transfer model with LR"
    transfer_model=linear_model.train(s_perf, t_perf)

    return prediction_model, transfer_model
```

(a) Linear Transformation Transfer [26].

```
def GPTransform(source, target, src_budget, tgt_budget):
    "Sample random configurations"
    s_some = source.sample(src_budget)
    t_some = target.sample(tgt_budget)
    "Get performance measurements"
    s_perf = get_perf(s_some)
    t_perf=get_perf(t_some)
    "Compute correlation"
    perf_correlation = get_correlation(s_perf, t_perf)
    "Compute covariance"
    input_covariance = get_covariance(s_some, t_some)
    "Construct a kernel"
    kernel = input_covariance × perf_correlation
    "Train the Gaussian Process model"
    learner = GaussianProcessRegressor(kernel)
    prediction_model = learner.train(s_some)
    return prediction_model
```

(b) Gaussian Process Transformation Transfer [27].

Fig. 4: Pseudocodes of other transfer learning methods.

with *different hardware settings*. The method consists of the following two components:

- *Performance prediction model:* The configurations on a source hardware are sampled using *Sobol* sampling. The number of configurations is given by $T \times N_f$, where $T = 3, 4, 5$ is the *training coefficient* and N_f is the number of configuration options. These configurations are used to construct a *Regression Tree* model.
- *Transfer Model:* To transfer the predictions from the source to the target, a linear regression model is used since it was found to provide good approximations of the transfer function. To construct this model, a small number of random configurations are obtained from *both the source and the target*. Note that this is a shortcoming since, without making some preliminary measurements on the target, one cannot begin to perform transfer learning.

5.2 Transfer Learning with Gaussian Process

Jamshidi et al. [27] took a slightly different approach to transfer learning. They used Multi-Task Gaussian Processes (GP) to find the relatedness between the performance measures in source and the target. The relationships between input configurations were captured in the GP model using a covariance matrix that defined the kernel function to construct the Gaussian processes model. To encode the relationships between the measured performance of the source and the target, a scaling factor is used with the above kernel.

The new kernel function is defined as follows:

$$k(s, t, f(s), f(t)) = k_t(s, t) \times k_{xx}(f(s), f(t)), \qquad (1)$$

where $k_t(s, t)$ represents the multiplicative scaling factor. $k_t(s, t)$ is given by the correlation between source f(s) and target f(t) function, while k_{xx} is the covariance function for input environments (s & t). The essence of this method is that the kernel captures the interdependence between the source and target environments.

5.3 Non-Transfer Learning Performance Optimization

A performance optimization model with no transfer was proposed by Nair et al. [12] in FSE '17. It works as follows:

1) Sample a small set of measurements of configurations from the target environment.
2) Construct performance model with regression trees.
3) Predict for near-optimal configurations.

The key distinction here is that unlike transfer learners, that use a *different source environment* to build to predict for near-optimal configurations in a target environment, a non-transfer method such as this uses configurations *from within the target* environment to predict for near-optimal configurations.

6 Experimental Setup

6.1 Subject Systems

In this study, we selected five configurable software systems from different domains, with different functionalities, and written in different programming languages. We selected these real-world software systems since their characteristics cover a broad spectrum of scenarios. Briefly,

- SPEAR is an industrial strength bit-vector arithmetic decision procedure and Boolean satisfiability (SAT) solver. It is designed for proving software verification conditions, and it is used for bug hunting. It consists of a binary configuration space with 14 options with 2^{14} or 16384 configurations. We measured how long it takes to solve an SAT problem in all 2^{14} configurations in 10 environments.
- x264 is a video encoder that compresses video files and has 16 configurations options to adjust output quality, encoder types, and encoding heuristics. Due to the cost of sampling the entire configuration space, we randomly sample 4000 configurations in 21 environments.
- SQLITE is a lightweight relational database management system, which has 14 configuration options to change indexing and features for size compression. Due to the cost of sampling and a limited budget, we use 1000 randomly selected configurations in 15 different environments.
- SAC is a compiler for high-performance computing. The SaC compiler implements a large number of high-level and low-level optimizations to tune programs for efficient parallel executions. It has 50 configuration options to control optimization options. We measure the execution time of the program for 846 configurations in 5 enviroments.
- STORM is a distributed stream processing framework which is used for data analytics. We measure the latency of the benchmark in 2,048 randomly selected configurations in 4 environments.

System	Language	$\{|C|, N, |E|\}$	H	W	V	Unchanged
x264	C, Assembly	4000, 16, 21	NUC/4/1.30/15/SSD NUC/2/2.13/7/SSD Station/2/2.8/3/SCSI, AWS/1/2.4/1.0/SSD, AWS/1/2.4/0.5/SSD, Azure/1/2.4/3/SCSI	8/2, 32/11, 128/44	r2389, r2744	Memory, CPU, background services
SPEAR	C, Assembly	16384, 14, 10	NUC/4/1.30/15/SSD, NUC/2/2.13/7/SSD, Station/2/2.8/3/SCSI, AWS/1/2.4/1.0/SSD, AWS/1/2.4/0.5/SSD, Azure/1/2.4/3/SCSI	(in #variables/#clauses), 774/5934, 1008/7728, 1554/11914, 978/7498	1.2, 2.7	Memory, CPU, background services
SQLite	C	1000, 14, 15	NUC/4/1.30/15/SSD, NUC/2/2.13/7/SSD, Station/2/2.8/3/SCSI, AWS/1/2.4/1.0/SSD, AWS/1/2.4/0.5/SSD, Azure/1/2.4/3/SCSI	write–seq, read–batc, read–rand, read–seq	3.7. 6.3, 3.19.0.0	Memory, CPU, background services
SaC	C	846, 50, 5	NUC/4/1.30/15/SSD, NUC/2/2.13/7/SSD, Station/2/2.8/3/SCSI, AWS/1/2.4/1.0/SSD, AWS/1/2.4/0.5/SSD, Azure/1/2.4/3/SCSI	random matrix generator, particle filtering, differential, equation solver, k-means, optimal matching, nbody, simulation, conjugate, gradient, garbage collector.	1.0.0	Memory, CPU, background services
Storm	Clojure	2048, 12, 4	NUC/4/1.30/15/SSD, NUC/2/2.13/7/SSD, Station/2/2.8/3/SCSI, AWS/1/2.4/1.0/SSD, AWS/1/2.4/0.5/SSD, Azure/1/2.4/3/SCSI	WordCount, RollingCount, RollingSort, SOL	Storm 0.9.5 + Zookeeper 3.4.11	JVM machine, Zookeeper Options, Memory, CPU, background services

TABLE 1: Overview of the real-world subject systems. $|C|$:Number of Configurations sampled per environment, N=Number of configuration options, $|E|$: Number of Environments, $|H|$: Hardware, $|W|$: Workloads, and $|V|$: Versions.

Table 1 lists the details of the software systems used in this paper. Here, $|N|$ is the number of configuration options available in the software system. If the options for each configuration is *binary*, then there can be as much as $2^{|N|}$ possible configurations for a given system[2], since it is not possible for us measure the performance of all possible configurations, we measure the performance of a subset of the $2^{|N|}$ samples, this subset is denoted by $|C|$. The performance of each of the $|C|$ configurations are measured under different hardware (H), workloads (W), and software versions (V). A unique combination of H, W, V constitutes an enviroment which is denoted by E. Note that, measuring the performance of $|C|$ configurations in each of the $|E|$ enviroments can be very costly and time consuming. Therefore, instead of all combinations of $H \times W \times V$, we measure the performance in only a subset of the enviroments (the total number is denoted by $|E|$).

6.2 Evaluation Criterion

6.2.1 Inadequacies of Conventional Evaluation Criteria

Typically, performance models are evaluated based on accuracy or error using measures such as *Mean Magnitude of Relative error* (abbreviated as $MMRE$). MMRE is calculated as follows:

$$MMRE = \frac{|predicted - actual|}{actual} \cdot 100$$

2. On the other hand, if there are $|o|$ possible options, then there may be $|o|^N$ possible configurations.

While seemingly intuitive, it has recently been shown that exact measures like MMRE can be somewhat misleading to assess configurations. There has been a lot of criticism leveled against MMRE. MMRE along with other accuracy statistics such as MBRE (which stands for Mean Balanced Relative Error) have been shown to cause conclusion instability [37], [38], [39].

In order to overcome this, recently we have argued against the use of MMRE values, in favor of using *rank-based metrics* that compute the difference between the *relative rankings* of the performance scores [12], [29]. The key intuition behind *relative rankings* is that the raw accuracy (as measured by MMRE) is less important than the rank-ordering of configurations from best to worst. As long as a model can preserve the order of the rankings of the

	Value	Rank
Actual	0.09	100
Predicted	0.11	10
Difference	0.02	90

$$MMRE = \frac{0.11 - 0.09}{0.09} \times 100 = 22\%$$

$$R^{\delta} = |10 - 100| = 90$$

Now, let's say the $min = 0.09$ and $max = 0.11$. Then,

$$NAR = \frac{0.11 - 0.09}{0.11 - 0.09} \times 100 = 100\%$$

Fig. 5: A contrived example to illustrate the challenges with MMRE and rank based measures

configurations (from best to worst), we can still determine which configuration is the most optimum. We can quantify this by measuring the differences in ranking between the actual rank and the predicted rank. More formally, rank-difference R^δ is measured as:

$$R^\delta = |Rank(Predicted) - Rank(Actual)|$$

We note that rank difference, although slightly less susceptible to instability compared to MMRE, is still not particularly informative. This is because it ignores the distribution of performance scores and *a small difference in performance measure can lead to a large rank difference and vice-versa* [40].

To illustrate the challenges with R^δ and $MMRE$, consider the example in Figure 5 where we are trying to find a configuration with the *minimum* value. Here, although the difference between the predicted value and the actual value is only 0.02, the rank difference R^δ is 90. But this does not tell us if $R^\delta = 90$ is good or bad. While, in the same Figure 5, when we calculate MMRE we get an error of only 22%, this may convey a false sense of accuracy. In the same example, let us say that the maximum value permissible is 0.11, then according to Figure 5, our predicted value for the best performance (which recall is supposed to the lowest) is the *highest permissible value* of 0.11.

Therefore, to obtain a realistic estimate of optimality of a configuration, we must use a measure that does not *under-estimate* the difference between the actual best configuration and the predicted best configuration (as with $MMRE$), netiher should we *over-estimate* the difference (as with rank-difference). In other words, we seek a robust measure of optimality.

6.2.2 NAR: A more robust metric

To overcome the challenges above, in this paper, we propose a measure called *Normalized Absolute Residual* (NAR). It represents the ratio of (a) difference between the actual performance value of the optimal configuration and the predicted performace value of the optimal configuration, and (b) The absolute difference between the *maximum* and *minimum* possible performace values. Formally, it can be defined as:

$$NAR = \frac{|min(f(c)) - f(c^*)|}{max(f(c)) - min(f(c))} \cdot 100 \qquad (2)$$

Where $min(f(c))$ is the value of the true minima of configuration c, $f(c^*)$ is the predicted value of the minima, and $max(f(c))$ is the largest performance value of a configuration. This measure is equivalent to Absolute Residual between predicted and actual, normalized to lie between 0% to 100% (hence the name *Normalized Absolute Residual* or NAR). According to this formulation, the *lower the NAR*, the better. Reflecting back on Figure 5, we see that the NAR is 100% which is exact what is expected when a predicted "minima" (0.11) is equal to the actual "maxima" (also 0.11).

Further, it is worth noting that, NAR is actually a variant of the *Generational Distance* or *Inverted Generation Distance* used very commonly in search-based software engineering literature [41], [42], [43].

6.3 Statistical Validation

Our experiments are all subjected to inherent randomness introduced by sampling configurations or by a different source and target environments. To overcome this, we use 30 repeated runs, each time with a different random number seed. The repeated runs provide us with sufficiently large sample size for statistical comparisons. Each repeated run collects the values of NAR.

To rank these 30 numbers collected as above, we use the Scott-Knott test recommended by Mittas and Angelis [44]:

- A list of treatments, sorted by their mean value, are split at the point that maximizes the expected value of the difference in their mean before after the split.
- That split is accepted if, between the two splits, (a) there is a statistically significant difference using a hypothesis test $\mathcal{H}$, and (b) the difference between the two splits is *not* due to a small effect.
- If the split is acceptable, the algorithm then recurses on both splits.
- Once no more splits are found, they are "ranked" smallest to largest (based on their median value).

In our work, in order to judge the statistical significance we use a non-parametric bootstrap test with 95% confidence [45]. Also, to make sure that the statistical significance is not due to the presence of small effects, we use an A12 test [46]. Briefly, the A12 test measures the probability that one split has a lower NAR values than another. If the two splits are equivalent, then $A12 = 0.5$. Likewise if $A12 \geq 0.6$, then 60% of the times, values of one split are significantly smaller that the other. In such a case, it can be claimed that there is a *significant effect* to justify the hypothesis test. We use these two tests (bootstrap and A12) since these are non-parametric and have been previously demonstrated to be informative [47], [48], [49], [50], [51], [52].

7 RESULTS

RQ1: Does there exist a Bellwether Environment?

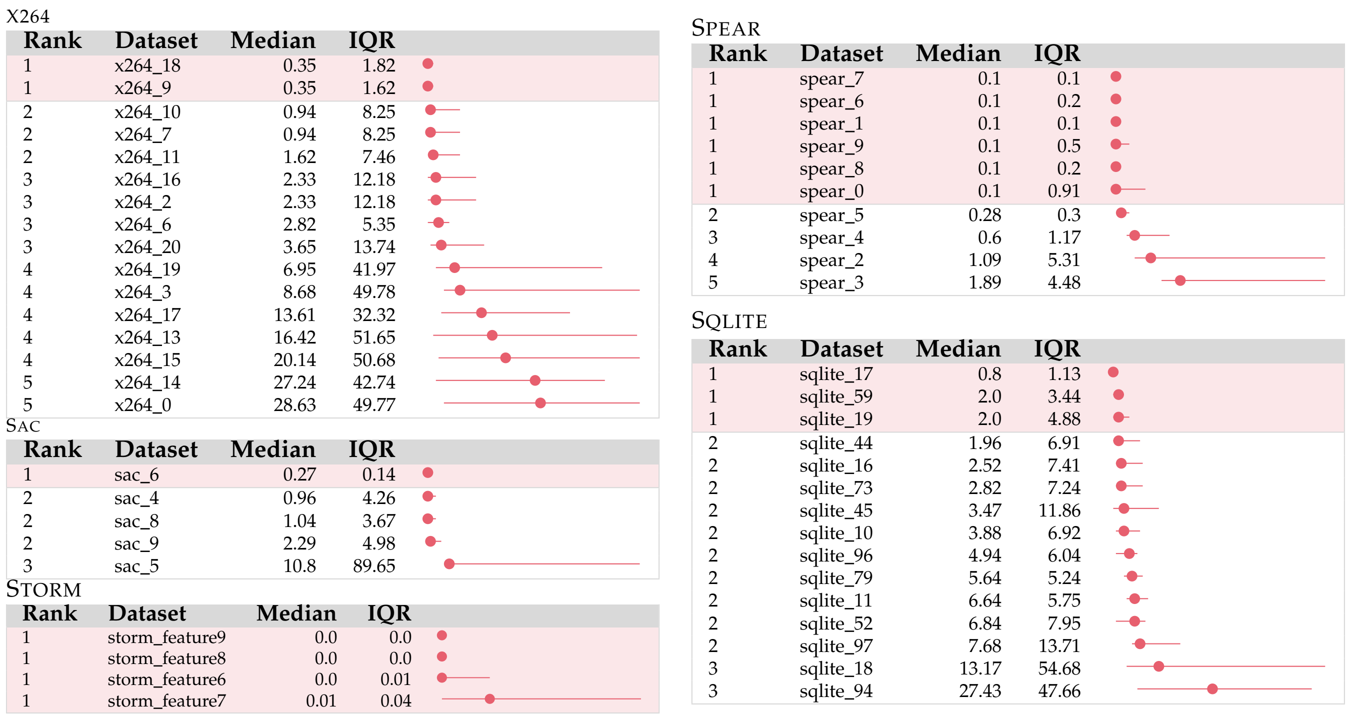
Purpose: The first research question seeks to establish the presence of bellwether environments within different environments of a software system. If there exists a bellwether environment, then identifying that environment can greatly reduce the cost of finding a near-optimal configuration for different environments.

Approach: For each subject software system, we use the environments to perform a pair-wise comparison as follows:

1) We pick one environment as a source and evaluate all configurations to construct a regression tree model.
2) The remaining environments are used as targets. For every target environment, we use the regression tree model constructed above to predict for the best configuration.
3) Then, we measure the NAR of the predictions (see §6.2.2).
4) Afterwards, we repeat steps 1, 2, and 3 for all the other source environments and gather the outcomes.

We repeat the whole process above 30 times and use the Scott-Knott test to rank each environment best to worst.

Result: Our results are shown in Fig. 6. Overall, we find that there is always at least one environment (the bellwether environment) in all the subject systems, that is much superior to others. Note that, STORM is an interesting case,



X264

Rank	Dataset	Median	IQR
1	x264_18	0.35	1.82
1	x264_9	0.35	1.62
2	x264_10	0.94	8.25
2	x264_7	0.94	8.25
2	x264_11	1.62	7.46
3	x264_16	2.33	12.18
3	x264_2	2.33	12.18
3	x264_6	2.82	5.35
3	x264_20	3.65	13.74
4	x264_19	6.95	41.97
4	x264_3	8.68	49.78
4	x264_17	13.61	32.32
4	x264_13	16.42	51.65
4	x264_15	20.14	50.68
5	x264_14	27.24	42.74
5	x264_0	28.63	49.77

SAC

Rank	Dataset	Median	IQR
1	sac_6	0.27	0.14
2	sac_4	0.96	4.26
2	sac_8	1.04	3.67
2	sac_9	2.29	4.98
3	sac_5	10.8	89.65

STORM

Rank	Dataset	Median	IQR
1	storm_feature9	0.0	0.0
1	storm_feature8	0.0	0.0
1	storm_feature6	0.0	0.01
1	storm_feature7	0.01	0.04

SPEAR

Rank	Dataset	Median	IQR
1	spear_7	0.1	0.1
1	spear_6	0.1	0.2
1	spear_1	0.1	0.1
1	spear_9	0.1	0.5
1	spear_8	0.1	0.2
1	spear_0	0.1	0.91
2	spear_5	0.28	0.3
3	spear_4	0.6	1.17
4	spear_2	1.09	5.31
5	spear_3	1.89	4.48

SQLITE

Rank	Dataset	Median	IQR
1	sqlite_17	0.8	1.13
1	sqlite_59	2.0	3.44
1	sqlite_19	2.0	4.88
2	sqlite_44	1.96	6.91
2	sqlite_16	2.52	7.41
2	sqlite_73	2.82	7.24
2	sqlite_45	3.47	11.86
2	sqlite_10	3.88	6.92
2	sqlite_96	4.94	6.04
2	sqlite_79	5.64	5.24
2	sqlite_11	6.64	5.75
2	sqlite_52	6.84	7.95
2	sqlite_97	7.68	13.71
3	sqlite_18	13.17	54.68
3	sqlite_94	27.43	47.66

Fig. 6: Median NAR of 30 repeats. Median NAR is the normalized absolute residual values as described in Equation 2, and IQR the difference between 75th percentile and 25th percentile found during multiple repeats. Lines with a dot in the middle (—●—), show the median as a round dot within the IQR. All the results are sorted by the median NAR: a lower median value is better. The left-hand column (*Rank*) ranks the various techniques where lower ranks are better. Overall, we find that there is always at least one environment, denoted in pink , that is much superior (lower NAR) to others.

here all the environments are ranked 1, which means that all the environments are equally useful as a bellwether environment —in such cases, any randomly selected environment could serve as a bellwether. Further, we note that the variance in the bellwether environments are much lower compared to other environments. Low variance indicates the low median NAR is not an effect of randomness in our experiments and hence increases our confidence in the existence of bellwethers.

Please note, in this specific experiment, we use *all* measured configurations (i.e., 100% of $|C|$ in Table 1) to determine if bellwethers exist. This ensures that the existance of bellwethers is not biased by how we sampled the configuration space. Later, in RQ2, we will restrict our study to determine what fraction of the samples would be adequate to find the bellwethers.

One may be tempted to argue that the answer to this question trivially could be answered as "yes" since it is unlikely that all environments exhibit identical performance and there will always be some environment that can make better predictions. However, observe that the environments ranked first performs much better than the rest (with certain exceptions), and hence, the difference between the bellwether environment and others is not coincidental.

Summary: There exists environments in each subject system, which act as the bellwether environment and hence can be used to find the near-optimal configuration for the rest of the environments.

TABLE 2: Effectiveness of source selection method.

Subject System	Bellwether Environment		Predicted Bellwether Environment	
	Median	IQR	Median	IQR
SQLite	0.8	1.13	1.8	2.48
Spear	0.1	0.1	0.1	0
x264	0.35	1.62	0.9	1.06
Storm	0.0	0.0	0.0	0.0
SaC	0.27	0.14	0.63	7.4

RQ2: How many measurements are required to discover bellwether environments?

Purpose: The bellwether environments found in RQ1 required us to use 100% of the measured performance values from all the environments[3]. Sampling all configurations may not be practical, since that may take an extremely long time [21]. Thus, in this research question, we ask if we can find the bellwether environments sooner using fewer samples.

Approach: We used the racing algorithm discussed in Section §4.1. To see if our proposed method is effective, we compare the performance of bellwether environment with the predicted bellwether environment. It works as follows:

1) We start from 10% of configurations from each environment and assume that every environment is a potential bellwether environment.

3. Note, except for SPEAR, we only have measured a subset of all possible configuration space since we were limited by the time and the cost required to make exhaustive measurements

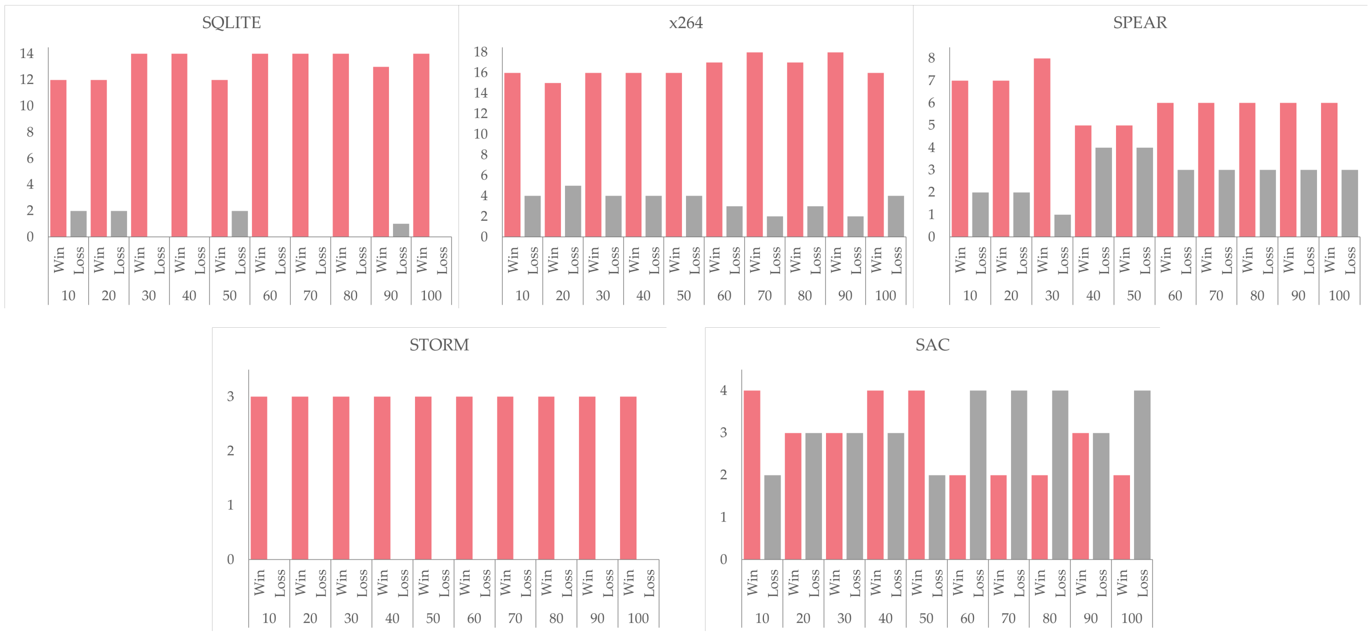


Fig. 7: Win/Loss analysis of learning from the bellwether environment and target environment using Scott Knott. The x-axis represents the percentage of data used to build a model and y-axis is the count. BEETLE wins in all software systems (since BEETLE wins more times than losses) except for SAC– and only when we have measured more that 50% of the data.

2) Then, we increment the number of configurations in steps of 10% and measure the NAR values.
3) We rank the environments and eliminate those that do not show much promise. A detailed description of how this is accomplished can be found in §4.
4) We repeat the above steps until we cannot eliminate any more environments.

Result: Table 2 summarizes our findings. We find that,

- In all 5 cases, using **at most** 10% of the configurations we find one of the bellwether environments that are found using 100% of the measured configurations. In Table 2, the second and the third column represent the median and IQR of the NAR values found using the Bellwether environment, which is found using 100% of the configurations (ground truth). The fourth and the fifth column (under Predicted Bellwether Environment) represent the median, and the IQR values found using the racing algorithm.
- The NAR of the predicted bellwether environments with 10% of the configurations is different by $< 1\%$ from the bellwether found at 100%.

These results are most encouraging in that we need only about 10% of the samples to determine the bellwether:

Summary: The bellwether environment can be recognized using only a fraction of the measurements (under 10%). Encouragingly, the identified bellwether environments have similar NAR values to the bellwether environment with 100% of samples. More running fewer configuration takes less time and is cheaper.

RQ3: How does BEETLE compare to non-transfer methods?

Purpose: We explore how BEETLE compares to a non-transfer learning approach. For our experiment, we use the non-transfer performance optimizer proposed by Nair et al. [12]. Both BEETLE and Nair et al.'s methods seek to achieve the same goal—find the optimal configuration in a target environment. BEETLE uses configurations from a *different source* to achieve this, whereas the non-transfer learner uses configurations from *within the target*. Please note BEETLE can use anywhere between 0%–100% of the configurations from the bellwether environment. In the previous RQs, we showed that 10% was adequate when using the bellwether enviroment.

Approach: Our setup involves evaluating the Win/Loss ratio of BEETLE to the non-transfer learning algorithm while predicting for the optimal configuration. Comparing against true optima, we define "win" as cases where BEETLE has a better (or same) NAR as the non-transfer learner. If the non-transfer learner has a better NAR, that counts as a "loss".

Result: Our results are shown in Figures 7 and 8. In Figure 7, the x-axis represents the number of configurations (expressed in %) to train the non-transfer learner and BEETLE, and the y-axis represents the number of wins/losses. We observe:

- *Better performance:* In $\frac{4}{5}$ systems, BEETLE "wins" significantly more than it "losses". This means that BEETLE is better than (or similar to) non-transfer learning methods.
- *Lower cost:* Regarding cost, we note that BEETLE outperforms the non-transfer learner significantly, "winning" at configurations of 10% to 100% of the original sample size. Further, when we look at the trade-off between performance and number of measurements in Figure 8, we note that BEETLE achieves a NAR close to zero with around 100 samples. Also, the non-transfer learning method of Nair et al. [12] has significantly larger NAR while also requiring large sample sizes.

Summary: BEETLE performs better than (or same as) a non-transfer learning approach. BEETLE is also cost/-time efficient as it requires far fewer measurements.

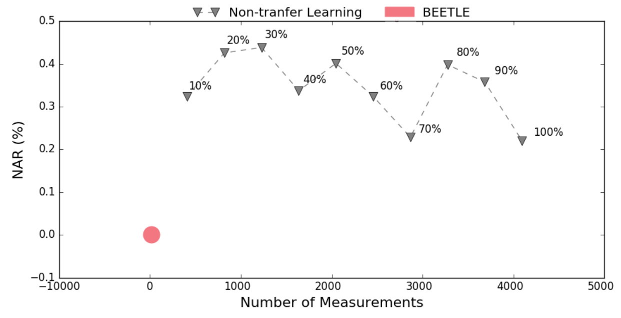


Fig. 8: Trade-off between the quality of the configurations and the cost to build the model for X264. The cost to find a good configuration using bellwethers is much lower than that of non-transfer-learning methods.

SAC

Rank	Learner	Median	IQR
1	Jamshidi et al. [27]	1.58	5.39
2	BEETLE	6.89	99.1
2	Valov et al. [26]	6.99	99.24

SPEAR

Rank	Learner	Median	IQR
1	Jamshidi et al. [27]	0.70	1.29
1	BEETLE	0.79	1.40
1	Valov et al. [26]	1.11	1.98

SQLITE

Rank	Learner	Median	IQR
1	BEETLE	5.41	9.28
2	Valov et al. [26]	6.96	12.91
3	Jamshidi et al. [27]	18.51	50.85

STORM

Rank	Learner	Median	IQR
1	BEETLE	0.04	0.06
1	Jamshidi et al. [27]	0.86	20.69
2	Valov et al. [26]	2.47	53.98

X264

Rank	Learner	Median	IQR
1	BEETLE	8.67	27.01
2	Valov et al. [26]	16.99	41.24
3	Jamshidi et al. [27]	43.58	28.39

Fig. 9: Comparison between state-of-the-art transfer learners and BEETLE. The best transfer learner is shaded gray. The "ranks" shown in the left-hand-side column come from the statistical analysis described in §6.3.

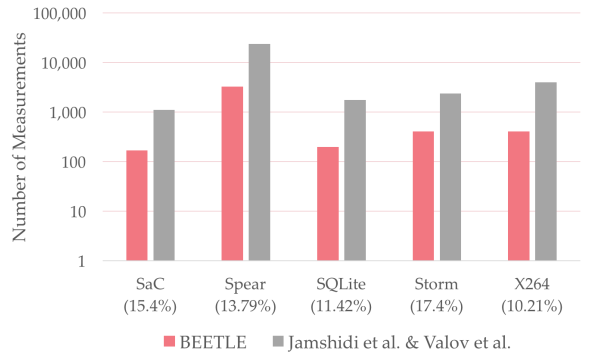


Fig. 10: BEETLE v/s state-of-the-art transfer learners. The numbers in parenthesis represent the numbers of measurements BEETLE uses in comparison to the state-of-the-art learners.

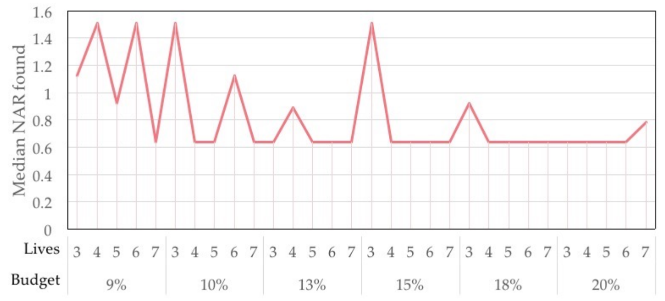


Fig. 11: The trade-off between the budget of the search, the number of lives, and the NAR (quality) of the solutions. We note that the performance is correlated with the budget and number of lives, i.e., as the budget increases the NAR value decreases.

RQ4: How does BEETLE compare to state-of-the-art methods?

Purpose: The main motivation of this work is to show that the source environment can have a significant impact on transfer learning. In this research question, we seek to compare BEETLE with other state-of-the-art transfer learners by Jamshidi et al. [27] and Valov et al. [26].

Approach: We perform transfer learning the methods proposed by Valov et al. [26] and Jamshidi et al. [27] (see §5). Then we measure the NAR values and compare them statistically using Skott-Knott tests. Finally, we rank the methods from best to worst based on their Skott-Knott ranks.

Result: Our results are shown in Fig. 9. In this figure, the best transfer learner is ranked 1. We note that in 4 out of 5 cases, BEETLE performs just as well as (or better than) the state-of-the-art. This result is encouraging in that it points to a significant impact on choosing a good source environment can have on the performance of transfer learners. Further, in Figure 10 we compare the number of performance measurements required to construct the transfer learners (note the logarithmic scale on the vertical axis). The total number of available samples for each software system is shown in the second column of Table 1 (see values corresponding to $|C|$). Of these we used only:

1) *x264*: 10.21% of 4000 samples
2) *SQLite*: 11.42% of 1000 samples
3) *Spear*: 13.79% of 16384 samples
4) *SaC*: 15.4% of 846 samples
5) *Storm*: 17.40% of 2048 samples

Based on these results, we note that BEETLE requires far fewer measurements compared to the other transfer-learning methods. That is,

Summary: In most software systems, BEETLE performs just as well as (or better than) other state-of-the-art transfer learners for performance optimization using far fewer measurements.

8 DISCUSSION

This section discusses some additional questions that may arise in regards to BEETLE and it's applicability to real-world situations.

What is the effect of BEETLE on the day to day business of a software engineer? From an industrial perspective, BEETLE can be used in at least the following ways:

- Consider an organization which has to optimize their software system for different clients (who have different workload and hardware—different AWS subscriptions). While onboarding new clients, the company might not be able to afford to invest extensive resources in finding the near-optimal configuration to appease the client. State-of-the-art transfer learning techniques would expect the organization to provide a source workload (or environment) for this task. But without a subject matter expert (SME) with the relevant knowledge, it is hard for humans to select a suitable source. *BEETLE removes the need for such SMEs since it automates source selection, along with transferring knowledge between the source and the target environment.*

- Consider an organization, which needs to migrate all their workload from a legacy platform to a different cloud platform (e.g., AWS to AZURE or vice versa). Such an organization now has many workloads that they need to optimize; however, they lack experience and performance measurements, on the new platform to accomplish this goal. *In such cases, BEETLE provides a way to discover an ideal source to transfer knowledge to enable efficient migration of workloads.*

How complex is BEETLE compared to other methods? BEETLE is among the easiest transfer learning methods currently available. In comparison with the state-of-the-art methods studied here, we require only few measurements of software systems running under different enviroments, we can build a $FindBellwether$ method that comprises of an all-pairs round-robin comparison followed by elimination of poorly performing enviroments. Then, transfer learning uses one of many off-the-shelf machine learners to build a prediction model (here we use Regression Trees). In this paper, we demonstrate that this method is just as powerful as other methods while being an order of magnitude cheaper in terms of the number of measurements required.

What are the impact of different hyperparameter choices? With all the transfer learners and predictors discussed here, there are a number of internal parameters that may (or may not) have a significant impact on the outcomes of this study. We indentify two key hyperparameters that affect BEETLE namely, *Budget* and *Lives*. As shown in 3d, both these hyperparameters determine when to stop sampling the source and declare the bellwethers. These bellwethers subsequently affect tranfer learning. To study the effect of these hyperparameters, we plot the trade-off between the budget and lives versus NAR. This is shown in Figure 11. Here we make the following observations:

- *Budget:* There is discernible impact of larger budget on the performance of bellwethers. We note that the performance is correlated to the budget, i.e., as the budget increases the NAR value decreases (lower NAR values are better). This is to be expected, an increased budget permits an larger sample to construct tranfer learners, thereby improving the likelihood of finding a near optimal solution.
- *Lives:* Although lower lives seems to correspond to larger NAR (worse). The correlation between the number of lives and NAR is less pronounced than that between Budget and NAR. That said, we noted that having 5 or lives generally corresponds to better NAR values. Thus, in all the experiments in this paper, we use 5 lives as default.

Is BEETLE applicable in other domains? In principle, yes. BEETLE could be applied to any transfer learning application, where the choice of the source data impacts the performance of transfer learning. This can be applied to problems such as configuring big data systems [3], finding suitable cloud configuration for a workload [53], [54], configuring hyperparameters of machine learning algorithms [55], [56], [57], runtime adaptation of robotic systems [27]. In these applications, the correct choice of source datasets using bellwethers can help to reduce the amount of time it takes to discover a near-optimal configuration setting.

Can BEETLE identify bellwethers in completely dissimilar environments? In theory, yes. Given a software system, BEETLE currently looks for an environment which can be used to find a near-optimal configuration for a majority of other environments for *that* software system. Therefore, given performance measurements in a variety of environments, BEETLE can assist in discovering a suitable source environment to transfer knowledge. Our work shows that knowledge can be transferred between environments comprised of completely different hardware, software versions, and workloads.

When are bellwethers ineffective? The existence of bellwethers depends on the following:

- *Metrics used:* Finding bellwether using metrics that are not justifiable, may be unsuccessful, for example, discovering bellwethers in performance optimization, by measuring MMRE instead of NAR may fail (see http://tiny.cc/bw_metrics) [12].
- *Different Software System:* Bellwethers of a certain software system 'A' may not work for software system 'B.' In other words, it cannot be used for cases where the configuration spaces across environment are not consistent.
- *Different Performance Measures:* Bellwether discovered for one performance measure (time) may not work for other performance measures (throughput).

9 Threats to Validity

Internal validity: Due to the size of configuration spaces, we could only measure configurations exhaustively in one subject system (SPEAR) and had to rely on sampling (with substantial sampling size) for the others, which may miss effects in parts of the configuration space that we did not sample. We did not encounter any surprisingly different observation in our exhaustively measured SPEAR dataset. Also, all other existing performance optimization work suffer from this threat. Measurement noise in benchmarks can be reduced but not avoided. We performed benchmarks on dedicated systems and repeated each measurement three times. We repeated experiments when we encountered unusually large deviations.

External validity: We selected a diverse set of subject systems and a large number of environment changes, but, as usual, one has to be careful when generalizing to other subject systems and environment changes. Even though we tried to run our experiment on a variety of software systems from different domains, we should take not to claim that our results generalize beyond the specific case studies explored here.

That said, a theoretical case can be made that BEETLE would be useful for systems larger than those studied here. When assessing the scalability of the performance optimization method, we ask primarily two questions. Firstly, if the **sampling technique** used in the presented method is scalable. There as various methods used in the literature (such as Feature-frequency based sample selection [7] and Spectral-based sample selection [10]). However, these strategies are linear time algorithms. Given the progressive sampling nature of the performance optimization, the complexity of the sampling techniques (as proposed in [7], [10] is $O(N)$, where N is the number of valid configuration. This can be very expensive because N could be exponentially large. Oh et al. [8], have demonstrated the need to have variants of random sampling to make it scalable. Based on Oh et al.'s suggestion, BEETLE uses random sampling as its preferred sampling strategy.

Secondly, is the **machine learning model** used in the method is scalable. There are various models used in performance optimization such as Gaussian Process [27], Regression Trees [28], [7], [29], and Bagging, Random Forest, and Support Vector Machines (SVMs) [5]. Recently, Nair et al. [29] showed that Gaussian Processes are not scalable (hence should be avoided) and showed that Regression Trees (such as CART) is scales to a larger dataset (in our case, software system). Also, there exists a large literature on methods to make regression trees scalable [58]. Hence, the regression tree is our machine learning model of choice.

Since BEETLE uses *random sampling* (which is efficient) and *regression trees* (which scales), we suggest that BEETLE could scale up to much larger configuration spaces than those explored here.

10 Related Work

Performance Optimization: Modern software systems come with a large number of configuration options. For example, in APACHE (a popular web server) there are around 600 different configuration options and in HADOOP, as of version 2.0.0, there are around 150 different configuration options, and the number of options is constantly growing [1]. These configuration options control the internal properties of the system such as memory and response times. Given the large number of configurations, it becomes increasingly difficult to assess the impact of the configuration options on the system's performance. To address this issue, a common practice is to employ performance prediction models to estimate the performance of the system under these configurations [28], [59], [60], [61], [34], [62]. To leverage the full benefit of a software system and its features, researchers augment performance prediction models to enable *performance optimization* [12], [8].

Performance optimization is an essential challenge in software engineering. As shown in the next few paragraphs, this problem has attracted much recent research interest. Approaches that use meta-heuristic search algorithms to explore the configuration space of Hadoop for high-performing configurations have been proposed [9]. It has been reported that such meta-heuristic search can find configurations options that perform significantly better than baseline default configurations. In other work, a control-theoretic framework called *SmartConf* to automatically set and dynamically adjust performance-sensitive configurations to optimize configuration options [63]. For the specific case of deep learning clusters, a job scheduler called *Optimus* has been developed to determine configuration options that optimize training speed and resource allocations [64]. Performance optimization has also been extensively explored in other domains such as Systems Research [65], [66] and Cloud Computing [67], [68], [54], [53], [11].

Much of the performance optimization tasks introduced above require access to measurements of the software system under various configuration settings. However, obtaining these performance measurements can cost a significant amount of time and money. For example, in one of the software systems studied here (X264), it takes over 1536 hours to obtain performance measurements for 11 out the 16 possible configuration options [26]. This is in addition to other time-consuming tasks involved in commissioning these systems such as setup, tear down, etc. Further, making performance measurements can cost an exorbitant amount of money. In our case, for X264, obtaining performance measurements of 2048 configurations under different environments on Amazon AWS `c4.large` cluster cost us several thousand dollars.

Transfer Learning: When a software system is deployed in a new environment, not every user can afford to repeat the costly process of building a new performance model to find an optimum configuration for that new environment. Instead, researchers propose the use of transfer learning to reuse the measurements made for previous environments [26], [27], [69], [70]. Jamshidi et al. [27], conducted a preliminary exploratory study of transfer learning in performance optimization to identify transferable knowledge between a source and a target environment, ranging from easily exploitable relationships to more subtle ones. They demonstrated that information about influential configuration options could be exploited in transfer learning and that knowledge about performance behavior can be transferred between environments.

Following this, a number of transfer learning methods were developed to predict for the optimum configurations in a new *target* environment, using the performance measures of another *source* environment as a proxy. Several researchers have shown that transfer learning can decrease the cost of learning significantly [21], [26], [27], [69].

All transfer learning methods place implicit faith in the quality of the source. A poor source can significantly deteriorate the performance of transfer learners.

Source Selection with Bellwethers: It is advised that the source used for transfer learning must be chosen with care to ensure optimum performance [71], [72], [57]. An incorrect choice of the source may result in the all too common *negative transfer* phenomenon [57], [73], [74], [75]. A negative transfer can be particularly damaging in that it often leads to performance degradation [21], [57]. A preferred way to avoid negative transfer is with *source selection*. Many methods have been proposed for identifying a suitable source for transfer learning [20], [23], [57]. Of these, source selection using the bellwether effect is one of the simplest. It has been effective in several diverse domains of software engineering such as defect prediction, effort estimation, and code-smell detection [20], [24], [25].

Besides negative transfer, previous approaches suffer from a lack of scalability. For example, Google Visor [70] Jamshidi et al. [27] rely on a Gaussian process which known to not scaling to large amounts of data in high dimensional spaces [76] Accordingly, in this work, we introduce the notion of source selection with bellwether effect for transfer learning in performance optimization. With this, we develop a Bellwether Transfer Learner called BEETLE. We show that, for performance optimization, BEETLE can outperform both non-transfer and the transfer learning methods.

11 Conclusion

Our approach, BEETLE, exploits the bellwether effect—there are one or more bellwether environments which can be used to find good configurations for the rest of the environments. We also propose a new transfer learning method, called BEETLE, which exploits this phenomenon. As shown in this paper, BEETLE can quickly identify the bellwether environments with only a few measurements ($\approx 10\%$) and use it to find the near-optimal solutions in the target environments. Further, after extensive experiments with five highly-configurable systems demonstrating, we show that BEETLE:

- Identifies suitable sources to construct transfer learners;
- Finds near-optimal configurations with only a small number of measurements (an average of $\leq 13.5\% \approx \frac{1}{7}^{th}$ of the available number of samples);
- Performs as well as non-transfer learning approaches; and
- Performs as well as state-of-the-art transfer learners.

Based on our experiments, we demonstrate our initial problem–"whence to learn?" is an important question, and,

> A *good source* with a simple transfer learner is *better than source agnostic* complex transfer learners.

We show BEETLE can help answer this question effectively.

References

[1] 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 *Foundations of Software Engineering*. ACM, 2015.

[2] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang, "Automatic database management system tuning through large-scale machine learning," in *International Conference on Management of Data*. ACM, 2017.

[3] P. Jamshidi and G. Casale, "An uncertainty-aware approach to optimal configuration of stream processing systems," in *Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. IEEE, 2016.

[4] N. Siegmund, S. S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, and G. Saake, "Predicting performance via automated feature-interaction detection," in *International Conference on Software Engineering*. IEEE, 2012.

[5] 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*. ACM, 2015, pp. 186–190.

[6] N. Siegmund, A. Grebhahn, S. Apel, and C. Kästner, "Performance-influence models for highly configurable systems," in *FSE'15*, 2015, pp. 284–294.

[7] A. Sarkar, J. Guo, N. Siegmund, S. Apel, and K. Czarnecki, "Cost-efficient sampling for performance prediction of configurable systems (t)," in *International Conference on Automated Software Engineering*. IEEE, 2015.

[8] J. Oh, D. Batory, M. Myers, and N. Siegmund, "Finding near-optimal configurations in product lines by random sampling," in *Foundations of Software Engineering*. ACM, 2017.

[9] C. Tang, K. Sullivan, and B. Ray, "Searching for high-performing software configurations with metaheuristic algorithms," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, 2018, pp. 354–355.

[10] V. Nair, T. Menzies, N. Siegmund, and S. Apel, "Faster discovery of faster system configurations with spectral learning," *Automated Software Engineering*, 2017.

[11] C.-J. Hsu, V. Nair, T. Menzies, and V. Freeh, "Micky: A cheaper alternative for selecting cloud instances," *IEEE Cloud*, 2018.

[12] V. Nair, T. Menzies, N. Siegmund, and S. Apel, "Using bad learners to find good configurations," *Foundations of Software Engineering*, 2017.

[13] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *International Conference on Software Engineering*. IEEE, 2013.

[14] J. Nam and S. Kim, "Heterogeneous defect prediction," in *Foundations of Software Engineering*. ACM Press, 2015.

[15] X. Jing, F. Wu, X. Dong, F. Qi, and B. Xu, "Heterogeneous Cross-Company Defect Prediction by Unified Metric Representation and CCA-Based Transfer Learning Categories and Subject Descriptors," *Foundations of Software Engineering*, 2015.

[16] E. Kocaguneli and T. Menzies, "How to find relevant data for effort estimation?" in *Empirical Software Engineering and Measurement*. IEEE, 2011.

[17] E. Kocaguneli, T. Menzies, and E. Mendes, "Transfer learning in effort estimation," *Empirical Software Engineering*, 2015.

[18] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano, "On the relative value of cross-company and within-company data for defect prediction," *Empirical Software Engineering*, 2009.

[19] F. Peters, T. Menzies, and L. Layman, "LACE2: Better privacy-preserving data sharing for cross project defect prediction," in *International Conference on Software Engineering*, 2015.

[20] R. Krishna and T. Menzies, "Bellwethers: A baseline method for transfer learning," *IEEE Transactions on Software Engineering*, pp. 1–1, 2018.

[21] P. Jamshidi, N. Siegmund, M. Velez, C. Kästner, A. Patel, and Y. Agarwal, "Transfer learning for performance modeling of configurable systems: An exploratory analysis," in *International Conference on Automated Software Engineering*. IEEE Press, 2017.

[22] S. J. Pan and Q. Yang, "A survey on transfer learning," *Transactions on knowledge and data engineering*, 2010.

[23] R. Krishna, T. Menzies, and W. Fu, "Too much automation? the bellwether effect and its implications for transfer learning," in *International Conference on Automated Software Engineering*. ACM, 2016.

[24] S. Mensah, J. Keung, S. G. MacDonell, M. F. Bosu, and K. E. Bennin, "Investigating the significance of bellwether effect to improve software effort estimation," in *International Conference on Software Quality, Reliability and Security, QRS*. IEEE, 2017.

[25] S. Mensah, J. Keung, M. F. Bosu, K. E. Bennin, and P. K. Kudjo, "A stratification and sampling model for bellwether moving window," in *International Conference on Software Engineering and Knowledge Engineering*, 2017.

[26] P. Valov, J.-C. Petkovich, J. Guo, S. Fischmeister, and K. Czarnecki, "Transferring performance prediction models across different hardware platforms," in *International Conference on Performance Engineering*. ACM, 2017.

[27] P. Jamshidi, M. Velez, C. Kästner, N. Siegmund, and P. Kawthekar, "Transfer learning for improving model predictions in highly configurable software," in *Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE, 2017.

[28] J. Guo, K. Czarnecki, S. Apel, N. Siegmund, and A. Wasowski, "Variability-aware performance prediction: A statistical learning approach," in *International Conference on Automated Software Engineering*. IEEE, 2013.

[29] V. Nair, Z. Yu, T. Menzies, N. Siegmund, and S. Apel, "Finding faster configurations using flash," *Transactions on Software Engineering (Accepted)*, 2018.

[30] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu, "Starfish: a self-tuning system for big data analytics." in *Conference on Innovative Data Systems Research*, 2011.

[31] A. Saltelli, K. Chan, E. M. Scott *et al.*, *Sensitivity analysis*. Wiley New York, 2000, vol. 1.

[32] J. Guo, D. Yang, N. Siegmund, S. Apel, A. Sarkar, P. Valov, K. Czarnecki, A. Wasowski, and H. Yu, "Data-efficient performance learning for configurable systems," *Empirical Software Engineering*, 2017.

[33] *AWS EC2 Document History*, http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/DocumentHistory.html.

[34] P. Valov, J. Guo, and K. Czarnecki, "Empirical comparison of regression methods for variability-aware performance prediction," in *International Conference on Software Product Line*. ACM, 2015.

[35] M. Birattari, T. Stützle, L. Paquete, and K. Varrentrapp, "A racing algorithm for configuring metaheuristics," in *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*. Morgan Kaufmann Publishers Inc., 2002, pp. 11–18.

[36] P.-L. Loh and S. Nowozin, "Faster hoeffding racing: Bernstein races via jackknife estimates," in *International Conference on Algorithmic Learning Theory*. Springer, 2013, pp. 203–217.

[37] I. Myrtveit and E. Stensrud, "Validity and reliability of evaluation procedures in comparative studies of effort prediction models," *Empirical Software Engineering*, 2012.

[38] I. Myrtveit, E. Stensrud, and M. Shepperd, "Reliability and validity in comparative studies of software prediction models," *IEEE Transactions on Software Engineering*, 2005.

[39] T. Foss, E. Stensrud, B. Kitchenham, and I. Myrtveit, "A simulation study of the model evaluation criterion mmre," *IEEE Transactions on Software Engineering*, 2003.

[40] C. Trubiani, P. Jamshidi, J. Cito, W. Shang, Z. M. Jiang, and M. Borg, "Performance issues? hey devops, mind the uncertainty!" *IEEE Software*, 2018.

[41] S. Wang, S. Ali, T. Yue, Y. Li, and M. Liaaen, "A practical guide to select quality indicators for assessing pareto-based search algorithms in search-based software engineering," in *ICSE'16*. IEEE, 2016, pp. 631–642.

[42] J. Chen, V. Nair, R. Krishna, and T. Menzies, "" sampling" as a baseline optimizer for search-based software engineering," *Transactions on Software Engineering*, 2018.

[43] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE transactions on evolutionary computation*, vol. 6, no. 2, pp. 182–197, 2002.

[44] N. Mittas and L. Angelis, "Ranking and clustering software cost estimation models through a multiple comparisons algorithm," *Transactions on Software Engineering*, 2013.

[45] R. J. Tibshirani and B. Efron, *An introduction to the bootstrap*. Chapman and Hall New York, 1993, vol. 57.

[46] A. Vargha and H. D. Delaney, "A critique and improvement of the cl common language effect size statistics of mcgraw and wong," *Journal of Educational and Behavioral Statistics*, 2000.

[47] N. L. Leech and A. J. Onwuegbuzie, "A call for greater use of nonparametric statistics." in *Annual Meeting of the Mid-South Educational Research Association*. ERIC, 2002.

[48] S. Poulding and J. A. Clark, "Efficient software verification: Statistical testing using automated search," *Transactions on Software Engineering*, 2010.

[49] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *ICSE'11*, 2011, pp. 1–10.

[50] M. J. Shepperd and S. G. MacDonell, "Evaluating prediction systems in software project estimation," *Information & Software Technology*, 2012.

[51] V. B. Kampenes, T. Dybå, J. E. Hannay, and D. I. K. Sjøberg, "A systematic review of effect size in software engineering experiments," *Information & Software Technology*, 2007.

[52] E. Kocaguneli, T. Zimmermann, C. Bird, N. Nagappan, and T. Menzies, "Distributed development considered harmful?" in *International Conference on Software Engineering*, 2013.

[53] C.-J. Hsu, V. Nair, T. Menzies, and V. W. Freeh, "Scout: An Experienced Guide to Find the Best Cloud Configuration," *arXiv preprint arXiv:1803.01296*, 2018.

[54] C.-J. Hsu, V. Nair, V. W. Freeh, and T. Menzies, "Low-level augmented bayesian optimization for finding the best cloud vm," *International Conference on Distributed Computing Systems*, 2018.

[55] W. Fu, T. Menzies, and X. Shen, "Tuning for software analytics: Is it really necessary?" *Information and Software Technology*, 2016.

[56] W. Fu, V. Nair, and T. Menzies, "Why is differential evolution better than grid search for tuning defect predictors?" *arXiv:1609.02613*, 2016.

[57] M. J. Afridi, A. Ross, and E. M. Shapiro, "On automated source selection for transfer learning in convolutional neural networks," *Journal of Pattern Recognition*, 2018.

[58] P. Domingos and G. Hulten, "Mining high-speed data streams," in *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2000, pp. 71–80.

[59] K. Hoste, A. Phansalkar, L. Eeckhout, A. Georges, L. K. John, and K. De Bosschere, "Performance prediction based on inherent program similarity," in *International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 2006.

[60] F. Hutter, L. Xu, H. H. Hoos, and K. Leyton-Brown, "Algorithm runtime prediction: Methods & evaluation," *Artificial Intelligence*, 2014.

[61] E. Thereska, B. Doebel, A. X. Zheng, and P. Nobel, "Practical performance models for complex, popular applications," in *ACM SIGMETRICS Performance Evaluation Review*. ACM, 2010.

[62] D. Westermann, J. Happe, R. Krebs, and R. Farahbod, "Automated inference of goal-oriented performance prediction functions," in *International Conference on Automated Software Engineering*. ACM, 2012.

[63] S. Wang, C. Li, H. Hoffmann, S. Lu, W. Sentosa, and A. I. Kistijantoro, "Understanding and auto-adjusting performance-sensitive configurations," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2018, pp. 154–168.

[64] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, "Optimus: an efficient dynamic resource scheduler for deep learning clusters," in *Proceedings of the Thirteenth EuroSys Conference*. ACM, 2018, p. 3.

[65] Y. Zhu, J. Liu, M. Guo, Y. Bao, W. Ma, Z. Liu, K. Song, and Y. Yang, "Bestconfig: tapping the performance potential of systems via automatic configuration tuning," in *Symposium on Cloud Computing*. ACM, 2017.

[66] S. W. C. Li, H. H. S. Lu, W. Sentosa, and A. I. Kistijantoro, "Understanding and auto-adjusting performance-sensitive configurations," 2018.

[67] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics." in *Symposium on Networked Systems Design and Implementation*, 2017.

[68] N. J. Yadwadkar, B. Hariharan, J. E. Gonzalez, B. Smith, and R. H. Katz, "Selecting the best vm across multiple public clouds: A data-driven performance modeling approach," in *Symposium on Cloud Computing*. ACM, 2017.

[69] H. Chen, W. Zhang, and G. Jiang, "Experience transfer for the configuration tuning in large-scale computing systems," *Transactions on Knowledge and Data Engineering*, 2011.

[70] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. Karro, and D. Sculley, "Google vizier: A service for black-box optimization," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2017, pp. 1487–1495.

[71] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson, "How transferable are features in deep neural networks?" in *Advances in neural information processing systems*, 2014.

[72] M. Long, Y. Cao, J. Wang, and M. Jordan, "Learning transferable features with deep adaptation networks," in *International Conference on Machine Learning*, 2015.

[73] S. Ben-David and R. Schuller, "Exploiting task relatedness for multiple task learning," in *Learning Theory and Kernel Machines*. Springer, 2003.

[74] M. T. Rosenstein, Z. Marx, L. P. Kaelbling, and T. G. Dietterich, "To transfer or not to transfer," in *NIPS 2005 workshop on TL*, 2005.

[75] S. J. Pan and Q. Yang, "A survey on transfer learning," *Transactions on knowledge and data engineering*, 2010.

[76] C. E. Rasmussen, "Gaussian processes in machine learning," in *Advanced lectures on machine learning*. Springer, 2004, pp. 63–71.