Faster Discovery of Faster System Configurations with Spectral Learning

Vivek Nair · **Tim Menzies**
· **Norbert Siegmund** · **Sven Apel**

Abstract Despite the huge spread and economical importance of configurable software systems, there is unsatisfactory support in utilizing the full potential of these systems with respect to finding performance-optimal configurations. Prior work on predicting the performance of software configurations suffered from either (a) requiring far too many sample configurations or (b) large variances in their predictions. Both these problems can be avoided using the **WHAT** spectral learner. **WHAT**'s innovation is the use of the spectrum (eigenvalues) of the distance matrix between the configurations of a configurable software system, to perform dimensionality reduction. Within that reduced configuration space, many closely associated configurations can be studied by executing only a few sample configurations. For the subject systems studied here, a few dozen samples yield accurate and stable predictors—less than 10 % prediction error, with a standard deviation of less than 2 %. When compared to the state of the art, **WHAT** (a) requires 2 to 10 times fewer samples to achieve similar prediction accuracies, and (b) its predictions are more stable (i.e., have lower standard deviation). Furthermore, we demonstrate that predictive models generated by **WHAT** can be used by optimizers to discover system configurations that closely approach the optimal performance.

Vivek Nair
North Carolina State University, Raleigh, USA
E-mail: vivekaxl@gmail.com

Tim Menzies
North Carolina State University, Raleigh, USA
E-mail: tim.menzies@gmail.com

Norbert Siegmund
University of Weimar, Germany
E-mail: norbert.siegmund@uni-passau.de

Sven Apel
University of Passau, Germany
E-mail: apel@uni-passau.de

1 Introduction

Most software systems today are configurable. Despite the undeniable benefits of configurability, large configuration spaces challenge developers, maintainers, and users. In the face of hundreds of configuration options, it is difficult to keep track of the effects of individual configuration options and their mutual interactions. So, predicting the performance of individual system configurations or determining the optimal configuration is often more guess work than engineering. In their recent paper, Xu et al. documented the difficulties developers face with understanding the configuration spaces of their systems [44]. As a result, developers tend to ignore over 5/6ths of the configuration options, which leaves considerable optimization potential untapped and induces major economic cost [44].

Addressing the challenge of performance prediction and optimization in the face of large configuration spaces, researchers have developed a number of approaches that rely on sampling and machine learning [18, 34, 39]. While gaining some ground, state-of-the-art approaches face two problems: (a) they require far too many sample configurations for learning or (b) they are prone to large variances in their predictions. For example, prior work on predicting performance scores using regression trees had to compile and execute hundreds to thousands of specific system configurations [18]. A more balanced approach by Siegmund et al. is able to learn predictors for configurable systems [39] with low mean errors, but with large variances of prediction accuracy (e.g. in half of the results, the performance predictions for the Apache Web server were up to 50 % wrong). Guo et al. [18] also proposed an incremental method to build a predictor model, which uses incremental random samples with steps equal to the number of configuration options (features) of the system. This approach also suffered from unstable predictions (e.g., predictions had a mean error of up to 22 %, with a standard deviation of up to 46 %). Finally, Sarkar et al. [34] proposed a projective-learning approach (using fewer measurements than Guo at al. and Siegmund et al.) to quickly compute the number of sample configurations for learning a stable predictor. However, as we will discuss, after making that prediction, the total number of samples required for learning the predictor is comparatively high (up to hundreds of samples).

The problems of large sample sets and large variances in prediction can be avoided using the **WHAT** spectral learner, which is our main contribution. **WHAT**'s innovation is the use of the spectrum (eigenvalues) of the distance matrix between the configurations of a configurable system, to perform dimensionality reduction. Within that reduced configuration space, many closely associated configurations can be studied by measuring only a few samples. In a number of experiments, we compared **WHAT** against the state-of-the-art approaches of Siegmund et al. [39], Guo et al. [18], and Sarkar et al. [34] by means of six real-world configurable systems: Berkeley DB, the Apache Web server, SQLite, the LLVM compiler, and the x264 video encoder. We found that **WHAT** performs as well or better than prior approaches, while requiring far fewer samples (just a few dozen). This is significant and most surprising, since some of the systems explored here have up to millions of possible configurations. Overall, we make the following contributions:
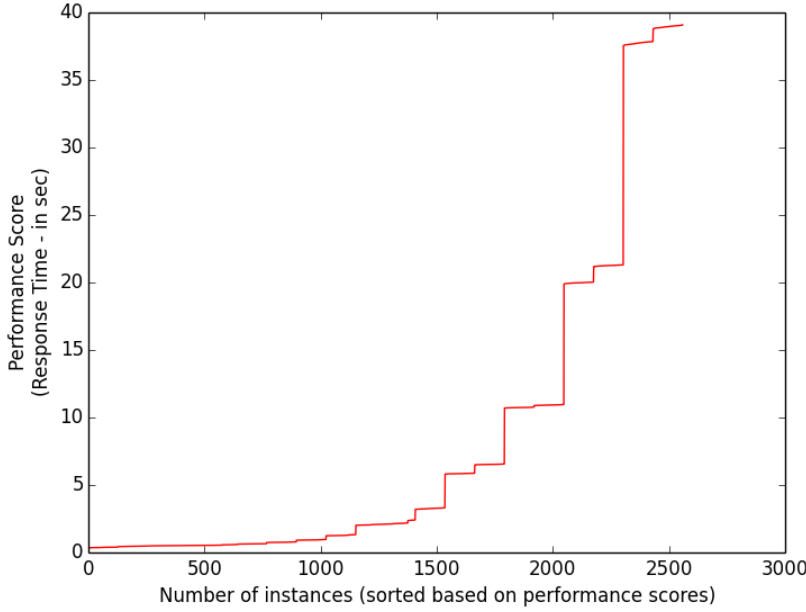
Fig. 1 The variation of performance scores of BDBC.

– We present a novel sampling and learning approach for predicting the performance of software configurations in the face of large configuration spaces. The approach is based on a *spectral learner* that uses an approximation to the first principal component of the configuration space to recursively cluster it, relying only on a few points as representatives of each cluster.
– We demonstrate the practicality and generality of our approach by conducting experiments on six real-world configurable software systems (see Figure 1). The results show that our approach is more accurate (lower mean error) and more stable (lower standard deviation) than state-of-the-art approaches. A key finding is the utility of the principal component of a configuration space to find informative samples from a large configuration space.

  All materials required for reproducing this work are available at `https://goo.gl/689Dve`.

2 Background & Related Work

We use the configurable system, BDBC, as an example to motivate our approach. BDBC is an embedded database system written in C. In this example, we consider 18 features or configuration options of BDBC, which the user can configure. We use the response time to indicate the performance of BDBC in different configurations. These 18 configuration options lead to 2,560 configurations. In Figure 1, we show

the performance distribution of all the configurations of BDBC. It is worth noting the difference between the best performing configuration (lower left corner) and the worst performing (top right corner). The figure shows that having a good configuration reduces the response time by a factor of 40 when compared to the worst possible configuration.

An important point is that, in practice, the configurations are selected often un-informed and may not be the best or near best configuration. More over with more configurations added with ever release [44], it is important to have an automated approach to find the best or near-best configuration. Another aspect to this problem is the cost of evaluation (of a particular configuration), which may be very expensive. So, an ideal method should be able to find the best or near-best performing configuration with the least number of evaluations. Our approach **WHAT** is effective in building accurate as well as stable performance models while using fewer sample configurations than the state-of-the-art.

A configurable software system has a set X of Boolean configuration options,[1] also referred to as features or independent variables in our setting. We denote the number of features of system S as n. The configuration space of S can be represented by a Boolean space $\mathbb{Z}_2^n$, which is denoted by F. All valid configurations of S belong to a set V, which is represented by vectors $\mathbf{C_i}$ (with $1 \leq i \leq |V|$) in $\mathbb{Z}_2^n$. Each element of a configuration represents a feature, which can either be *True* or *False*, based on whether the feature is selected or not. Each valid instance of a vector (i.e., a configuration) has a corresponding performance score associated to it.

The literature offers two approaches to performance prediction of software configurations: a *maximal sampling* and a *minimal sampling* approach: With *maximal sampling*, we compile all possible configurations and record the associated performance scores. Maximal sampling can be impractically slow. For example, the performance data used in our experiments required 26 days of CPU time for measuring (and much longer, if we also count the time required for compiling the code prior to execution). Other researchers have commented that, in real world scenarios, the cost of acquiring the optimal configuration is overly expensive and time consuming [43].

If collecting performance scores of all configurations is impractical, *minimal sampling* can be used to intelligently select and execute just enough configurations (i.e., samples) to build a predictive model. For example, Zhang et al. [46] approximate the configuration space as a Fourier series, after which they can derive an expression showing how many configurations must be studied to build predictive models with a given error. While a theoretically satisfying result, that approach still needs thousands to hundreds of thousands of executions of sample configurations.

Another set of approaches are the four "additive" *minimal sampling* methods of Siegmund et al. [39]. Their first method, called feature-wise sampling (*FW*), is their basic method. To explain *FW*, we note that, from a configurable software system, it is theoretically possible to enumerate many or all of the valid configurations[2]. Since each configuration ($\mathbf{C_i}$) is a vector of n Booleans, it is possible to use this information

---

[1] In this paper, we concentrate on Boolean options, as they make up the majority of all options; see Siegmund et al., for how to incorporate numeric options [38].

[2] Though, in practice, this can be very difficult. For example, in models like the Linux Kernel such an enumeration is practically impossible [35].

to isolate examples of how much each feature individually contributes to the total run time:

1. Find a pair of configurations C_1 and C_2, where C_2 uses exactly the same features as C_1, plus one extra feature f_i.
2. Set the run time $\Pi(f_i)$ for feature f_i to be the difference in the performance scores between C_2 and C_1.
3. The run time for a new configuration C_i (with $1 \leq i \leq |V|$) that has not been sampled before is then the sum of the run time of its features, as determined before:
$$\Pi(C_i) = \sum_{f_j \in C_i} \Pi(f_j) \tag{1}$$

When many pairs, such as C_1, C_2, satisfy the criteria of point 1, Siegmund et al. used the pair that covers the *smallest* number of features. Their minimal sampling method, *FW*, compiles and executes only these smallest C_1 and C_2 configurations. Siegmund et al. also offers three extensions to the basic method, which are based on sampling not just the smallest pairs, but also additional configurations covering certain kinds of *interactions* between features. All the following minimal sampling policies compile and execute valid configurations selected via one of three heuristics:

PW (pair-wise): For each pair of features, try to find a configuration that contains the pair and has a minimal number of features selected.

HO (higher-order): Select extra configurations, in which three features, f_1, f_2, f_3, are selected if two of the following pair-wise interactions exist: (f_1, f_2) and (f_2, f_3) and (f_1, f_3).

HS (hot-spot): Select extra configurations that contain features that are frequently interacting with other features.

Guo et al. [18] proposed a progressive random sampling approach, which samples the configuration space in steps of the number of features of the software system in question. They used the sampled configurations to train a regression tree, which is then used to predict the performance scores of other system configurations. The termination criterion of this approach is based on a heuristic, similar to the *PW* heuristics of Siegmund et al.

Sarkar et al. [34] proposed a cost model for predicting the effort (or cost) required to generate an accurate predictive model. The user can use this model to decide whether go ahead and build the predictive model. This method randomly samples configurations and uses a heuristic based on feature frequencies as termination criterion. The samples are then used to train a regression tree; the accuracy of the model is measured by using a test set (where the size of the training set is equal to size of the test set). One of four projective functions (e.g., exponential) is selected based on how correlated they are to accuracy measures. The projective function is used to approximate the accuracy-measure curve, and the elbow point of the curve is then used as the optimal sample size. Once the optimal size is known, Sarkar et al. uses the approach of Guo et al. to build the actual prediction model.

The advantage of these previous approaches is that, unlike the results of Zhang et al., they require only dozens to hundreds of samples. Also, like our approach, they

do not require to enumerate all configurations, which is important for highly configurable software systems. That said, as shown by our experiments (see Section 4), these approaches produce estimates with larger mean errors and partially larger variances than our approach. While sometimes the approach by Sarkar et al. results in models with (slightly) lower mean error rates, it still requires a considerably larger number of samples (up to hundreds), while **WHAT** requires only few dozen.

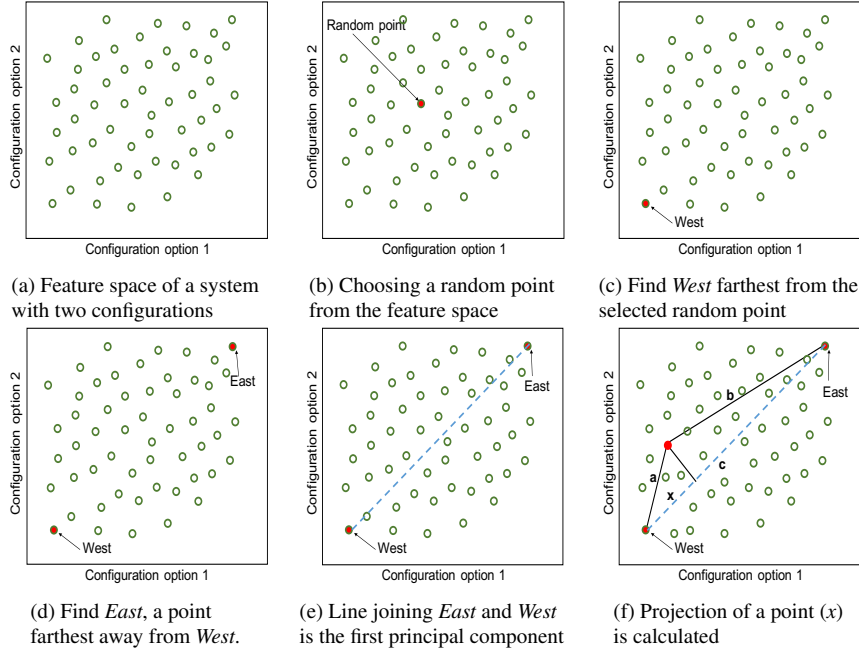## 3 Approach

3.1 Spectral Learning

The minimal sampling method we propose here is based on a spectral-learning algorithm that explores the spectrum (eigenvalues) of the distance matrix between configurations in the configuration space. In theory, such spectral learners are an appropriate method to handle noisy, redundant, and tightly inter-connected variables, for the following reasons: When data sets have many irrelevancies or closely associated data parameters d, then only a few eigenvectors e, $e \ll d$ are required to characterize the data. In this reduced space:

- Multiple inter-connected variables $i, j, k \subseteq d$ can be represented by a single eigenvector;
- Noisy variables from d are ignored, because they do not contribute to the signal in the data;
- Variables become (approximately) parallel lines in e space. For redundancies $i, j \in d$, we can ignore j since effects that change over j also change in the same way over i;

That is, in theory, samples of configurations drawn via an eigenspace sampling method would not get confused by noisy, redundant, or tightly inter-connected variables. Accordingly, we expect predictions built from that sample to have lower mean errors and lower variances on that error.

Spectral methods have been used before for a variety of data mining applications [25]. Algorithms, such as PDDP [4], use spectral methods, such as principle component analysis (PCA), to recursively divide data into smaller regions. Software-analytics researchers use spectral methods (again, PCA) as a pre-processor prior to data mining to reduce noise in software-related data sets [41]. However, to the best of our knowledge, spectral methods have not been used before as a basis of a minimal sampling method.

WHAT is somewhat different from other spectral learners explored in, for instance, image processing applications [36]. Work on image processing does not aim at defining a minimal sampling policy to predict performance scores. Also, a standard spectral method requires an $O(N^2)$ matrix multiplication to compute the components of PCA [23]. Worse, in the case of hierarchical division methods, such as PDDP, the polynomial-time inference must be repeated at every level of the hierarchy. Competitive results can be achieved using an $O(2N)$ analysis that we have developed previously [29], which is based on a heuristic proposed by Faloutsos and Lin [14]

(a) Feature space of a system with two configurations

(b) Choosing a random point from the feature space

(c) Find *West* farthest from the selected random point

(d) Find *East*, a point farthest away from *West*.

(e) Line joining *East* and *West* is the first principal component

(f) Projection of a point (x) is calculated

Fig. 2 Spectral Learning using **WHAT** using the example of a system with two configuration options.

(which Platt has shown computes a Nyström approximation to the first component of PCA [32]).

Figure 2 describes the procedure used to calculate the projection of a configurations. **WHAT** receives N (with $1 \leq |N| \leq |V|$) valid configurations (**C**), $N_1, N_2, ...$, as input (as shown in Figure 2(a)) and then:

1. Picks any point N_i ($1 \leq i \leq |N|$) at random (as shown in Figure 2(b));
2. Finds the point *West* $\in N$ that is furthest away from N_i (as shown in Figure 2(c));
3. Finds the point *East* $\in N$ that is furthest from *West* (as shown in Figure 2(d)).

The line joining *East* and *West* is our approximation for the first principal component (as shown in Figure 2(e)). Using the distance calculation shown in Equation 2, we define c to be the distance between *East* (x) and *West* (y). **WHAT** uses this distance (c) to divide all the configurations as follows: The value x_i is the projection of N_i on the line running from *East* to *West* (as shown in Figure 2(f))[3]. We divide the examples based on the median value of the projection of x_i. Now, we have two clusters of data divided based on the projection values (of N_i) on the line joining *East* and *West*. This process is applied recursively on these clusters until a predefined stopping condition. In our study, the recursive splitting of the N_i's stops when a sub-region contains less

---

[3] The projection of N_i can be calculated in the following way:
$a = dist(East, N_i); b = dist(West, N_i); x_i = \sqrt{\frac{a^2 - b^2 + c^2}{2c}}$.

than $\sqrt{|N|}$ examples.

$$dist(x,y) = \begin{cases} \sqrt{\sum_i (x_i - y_i)^2} & \text{if } x_i \text{ and } y_i \text{ is numeric} \\ \begin{cases} 0, & \text{if } x_i = y_i \\ 1, & \text{otherwise} \end{cases} & \text{if } x_i \text{ and } y_i \text{ is Boolean} \end{cases} \tag{2}$$

We explore this approach for three reasons:

- *It is very fast*: This process requires only $2|n|$ distance comparisons per level of recursion, which is far less than the $O(N^2)$ required by PCA [12] or other algorithms such as K-Means [19].
- *It is not domain-specific*: Unlike traditional PCA, our approach is general in that it does not assume that all the variables are numeric. As shown in Equation 2,[4] we can approximate distances for both numeric and non-numeric data (e.g., Boolean).
- *It reduces the dimensionality problem*: This technique explores the underlying dimension (first principal component) without getting confused by noisy, related, and highly associated variables.

3.2 Spectral Sampling

When the above clustering method terminates, our sampling policy (which we call S_1) is then applied:

Random sampling (S_1): compile and execute one configuration, picked at random, from each leaf cluster;

We use this sampling policy, because (as we will show later) it performs better than:

East-West sampling (S_2): compile and execute the *East* and *West* poles of the leaf clusters;

Exemplar sampling (S_3): compile and execute all items in all leaves and return the one with lowest performance score.

Note that S_3 is *not* a *minimal* sampling policy (since it executes all configurations). We use it here as one baseline against which we can compare the other, more minimal, sampling policies. In the results that follow, we also compare our sampling methods against another baseline using information gathered after executing all configurations.

3.3 Regression-Tree Learning

After collecting the data using one of the sampling policies (S_1, S_2, or S_3), as described in Section 3.2, we use a CART regression-tree learner [5] to build a performance predictor. Regression-tree learners seek the attribute-range split that most

---

[4] In our study, *dist* accepts pair of configuration (**C**) and returns the distance between them. If x_i and $y_i \in \mathbb{R}^n$, then the distance function would be same as the standard Euclidean distance.

increases our ability to make accurate predictions. CART explores splits that divide N samples into two sets A and B, where each set has a standard deviation on the target variable of σ_1 and σ_2. CART finds the "best" split defined as the split that minimizes $\frac{A}{N}\sigma_1 + \frac{B}{N}\sigma_2$. Using this best split, CART divides the data recursively.

In summary, **WHAT** combines:

- The FASTMAP method of Faloutsos and Lin [14], which rather than N^2 comparisons only performs $2N$ where N is the number of configurations in the configuration space;
- A spectral-learning algorithm initially inspired by Boley's PDDP system [4], which we modify by replacing PCA with FASTMAP (called "WHERE" in prior work [29]);
- The sampling policy that explores the leaf clusters found by this recursive division;
- The CART regression-tree learner that converts the data from the samples collected by sampling policy into a run-time prediction model [5].

That is,

$$
\begin{aligned}
\text{WHERE} \quad &= \quad \text{PDDP} - \text{PCA} + \text{FASTMAP} \\
\textbf{WHAT} \quad &= \quad \text{WHERE} + \{\, S_1, S_2, S_3 \,\} + \text{CART}
\end{aligned}
$$

This unique combination of methods has not been previously explored in the software-engineering literature.

3.4 Approach as a pipeline

Different components of **WHAT** can be used to sample configurations of a configurable software system, which can be then used to generate an accurate and stable performance model. We test **WHAT** in following way:

- All the possible configurations of a system is enumerated
- Split the configurations into training and testing datasets based on a predefined ratio (as discussed in section 4.3) – at this point none of the configurations is measured
- **WHAT** (section 3.1) is used to sample configuration (section 3.2) from the training dataset and for each configuration the corresponding performance is measured.
- Configuration and the corresponding performance score is used to build a performance model using Regression-Tree Learner (section 3.3).
- The accuracy (in terms of MRE) of the built performance model is measured using configurations for the testing set.

In the next section, we formulate our research questions and discuss the experimental setup.

4 Experiments

4.1 Research Questions

We formulate our research questions in terms of the challenges of exploring large complex configuration spaces. As our approach explores the spectral space, our hypothesis is that only a small number of samples is required to explore the whole space. However, a prediction model built from a very small sample of the configuration space might be very inaccurate and unstable, that is, it may exhibit very large mean prediction errors and variances on the prediction error.

Also, if we learn models from small regions of the training data, it is possible that a learner will miss *trends* in the data between the sample points. Such trends are useful when building *optimizers* (i.e., systems that receives one configuration as input and propose an alternate configuration that has, for instance, a better performance). Such optimizers might need to evaluate hundreds to millions of alternate configurations. To speed up that process, optimizers can use a *surrogate model*[5] that mimics the outputs of a system of interest, while being computationally cheap(er) to evaluate [28]. For example, when optimizing performance scores, we might ask a CART for a performance prediction (rather than compile and execute the corresponding configuration). Note that such surrogate-based reasoning critically depends on how well the surrogate can guide optimization.

Therefore, to assess feasibility of our sampling policies, we must consider:

– Performance scores generated from our minimal sampling policy;
– The variance of the error rates when comparing predicted performance scores with actual ones;
– The optimization support offered by the performance predictor (i.e., can the model work in tandem with other off-the-shelf optimizers to generate useful solutions).

The above considerations lead to four research questions:

RQ1: Can **WHAT** *generate good predictions after examining only a small number of configurations?*

Here, by "good" we mean that the predictions made by models that were trained using sampling with **WHAT** are as accurate, or more accurate, as preditions generated from models supplied with more samples.

RQ2: Do less data used in building the predictions models cause larger variances in the predicted performance scores?

RQ3: Can "good" surrogate models (to be used in optimizers) be built from minimal samples?

Note that RQ2 and RQ3 are of particular concern with our approach, since our goal is to sample as little as possible from the configuration space.

RQ4: How good is **WHAT** *compared to the state of the art of learning performance predictors from configurable software systems?*

To answer RQ4, we will compare **WHAT** against approaches presented by Siegmund et al. [39], Guo et al. [18], and Sarkar et al. [34].

---

[5] Also known as response surface methods, meta models, or emulators.

Table 1 Subject systems used in the experiments.

	Description	LOC	# Feat.	Configurations	# Config.	Benchmark Used	Performance Metric
Apache	Apache is a prominent open-source Web server with numerous configuration options.	230,277	9	Base, HostnameLookups, KeepAlive, EnableSendfile, FollowSymLinks, AccessLog, ExtendedStatus, InMemory, Handle	192	We used the tools autobench and httperf to generate load on the Web server. We increased the load until the server could not, handle any, further requests	Maximum load
Berkeley DB C Edition (BDBC)	BDBC is an embedded database system written in C.	219,811	18	HAVE_CRYPTO, HAVE_HASH, HAVE_REPLICATION, HAVE_VERIFY, HAVE_SEQUENCE, HAVE_STATISTICS, DIAGNOSTIC, PAGESIZE, PS1K, PS4K, PS8K, PS16K, PS32K, CACHESIZE, CS32MB, CS16MB, CS64MB, CS512MB	2,560	Benchmark provided by the vendor	Response time
Berkeley DB Java Edition (BDBJ)	BDBJ is a complete re-development of BDBC in Java with full SQL support.	42,596	32	Base, Persistence, IO, OldIO, NewIO, NIOBase, NIOType, ChunkedNIO, SingleWriteNIO, DirectNIO, LogSize, S100MiB, S1MiB, Checksum, BTreeFeatures, INCompressor, IEvictor, Evictor, Critical_Eviction, Verifier, ITracing, Tracing, TracingLevel, Severe, Finest, Statistics	400	Benchmark provided by the vendor	Response time
LLVM	LLVM is a compiler infrastructure written in C++.	47,549	11	time_passes, gvn, instcombine, inline, jump_threading, simplify-cfg, sccp, print_used_types, ipsccp, iv_users, licm	1,024	As benchmark, we measured the time to compile LLVMs test suite	Time to compile LLVMs test suite
SQLite	SQLite is an embedded database system deployed over several millions of devices.	312,625	39	OperatingSystemCharacteristics,SQLITE_SECURE_DELETE, ChooseSQLITE_TEMP_STORE,SQLITE_TEMP_STOREzero, SQLITE_TEMP_STOREone, SQLITE_TEMP_STOREtwo, SQLITE_TEMP_STOREthree, AutoVacuumOff, AutoVacuumOn, SetCacheSize, StandardCacheSize, LowerCacheSize, Higher-CacheSize, LockingMode, ExclusiveLock, NormalLockingMode, PageSize, StandardPageSize, LowerPageSize, HigherPageSize, HighestPageSize	3,932,160	Benchmark provided by the vendor	Response time
x264	x264 is a video encoder in C that provides configuration options to adjust output quality of encoded video files.	45,743	16	no_asm, no_8x8dct, no_cabac, no_deblock, no_fast_pskip, no_mbtree, no_mixed_refs, no_weightb, rc_lookahead, rc_lookahead_20, rc_lookahead_40, rc_lookahead_60, ref, ref_1, ref_5, ref_9	1,152	As benchmark, we encoded the Sintel trailer (735 MB) from AVI to the xH.264 codec	Encoding time

4.2 Subject Systems

The configurable systems we used in our experiments are described in Table 1. All systems are real-world systems and representative of different domains with different configuration mechanisms and implemented using different programming languages. Note, with "predicting performance", we mean predicting performance scores of the subject systems while executing test suites provided by the developers or the community, as described in Table 1. To compare the predictions of our and prior approaches with actual performance measures, we use data sets that have been obtained by measuring *nearly all* configurations[6]. We say *nearly all* configurations, for the following reasoning: For all except one of our subject systems, the total number of valid configurations was tractable (192 to 2560). However, SQLite has 3,932,160 possible configurations (since SQLite has 39 configuration – 2^{39} possible configurations), which is an impractically large number of configurations to test whether our predictions are accurate and stable. Hence, for SQLite, we use the 4500 samples for testing prediction accuracy and stability, which we could collect in one day of CPU time. Taking this into account, we will pay particular attention to the variance of the SQLite results.

4.3 Experimental Rig

RQ1 and RQ2 require the construction and assessment of numerous runtime predictors from small samples of the data. The following rig implements that construction process.

For each configurable software system, we built a table of data, one row per valid configuration. We then ran all configurations of all software systems and recorded the performance scores (i.e., that are invoked by a benchmark). The exception is SQLite for which we measured only the configurations needed to detect interactions and additionally 100 random configurations. To this table, we added a column showing the performance score obtained from the actual measurements for each configuration.

Note that the following procedure ensures that we *never* test any prediction model on the data that we used to learn this model. Next, we repeated the following procedure 20 times (the figure of 20 repetitions was selected using the Central Limit Theorem): For each subject system in {BDBC, BDBJ, Apache, SQLite, LLVM, x264}

- Randomize the order of the rows in their table of data;
- For *X* in {10, 20, 30, ... , 90};
    - Let *Train* be the first *X* % of the data
    - Let *Test* be the rest of the data;
    - Pass *Train* to **WHAT** to select sample configurations;
    - Determine the performance scores associated with these configurations. This corresponds to a table lookup, but would entail compiling and executing a system configuration in a practical setting.
    - Using the *Train* data and their performance scores, build a performance predictor using CART.

---

[6] http://openscience.us/repo/performance-predict/cpm.html

   – Using the *Test* data, assess the accuracy of the predictor using the error measure of Equation 3 (see below).

The validity of the predictors built by CART is verified on testing data. For each test item, we determine how long it *actually* takes to run the corresponding system configuration and compare the actual measured performance to the *prediction* from CART. The resulting prediction error is then computed using:

$$error = \frac{|\, predicted - actual\,|}{actual} \cdot 100 \tag{3}$$

(Aside: It is reasonable to ask why this metrics and not some of the others proposed in the literature (e.g sum absolute residuals). In short, our results are stable across a range of different metrics. For e.g., the results of this paper have been repeated using sum of absolute residuals and, in those other results, we seen the same ranking of methods; see `http://tiny.cc/sumAR`).

RQ2 requires testing the standard deviation of the prediction error rate. To support that test, we:

- Determine the X-th point in the above experiments, where all predictions stop improving (elbow point);
- Measure the standard deviation of the error at this point, across our 20 repeats.

As shown in Figure 3, all our results plateaued after studying $X = 40\%$ of the valid configurations[7]. Hence to answer RQ2, we will compare all 20 predictions at $X = 40\%$.

RQ3 uses the learned regression tree as a *surrogate model* within an optimizer;

- Take $X = 40\%$ of the configurations;
- Apply **WHAT** to build a CART model using some minimal sample taken from that 40%;
- Use that CART model within some standard optimizer while searching for configurations with least runtime;
- Compare the faster configurations found in this manner with the fastest configuration known for that system.

This last item requires access to a ground truth of performance scores for a large number of configurations. For this experiment, we have access to that ground truth (since we have access to all system configurations, except for SQLite). Note that such a ground truth would not be needed when practitioners choose to use **WHAT** in their own work (it is only for our empirical investigation).

For the sake of completeness, we explored a range of optimizers seen in the literature: DE [40], NSGA-II [9], and our own GALE [26, 47] system. Normally, it would be reasonable to ask why we used those three, and not the hundreds of other optimizers described in the literature [15, 20]. However, as shown below, all these optimizers in this domain exhibited very similar behavior (all found configurations close to the best case performance). Hence, the specific choice of optimizer is not a critical variable in our analysis.

---

[7] Just to clarify one frequently asked question about this work, we note that our rig "studies" 40% of the data. We do not mean that our predictive models require accessing the performance scores from the 40% of the data. Rather, by "study" we mean reflect on a sample of configurations to determine what minimal subset of that sample deserves to be compiled and executed.
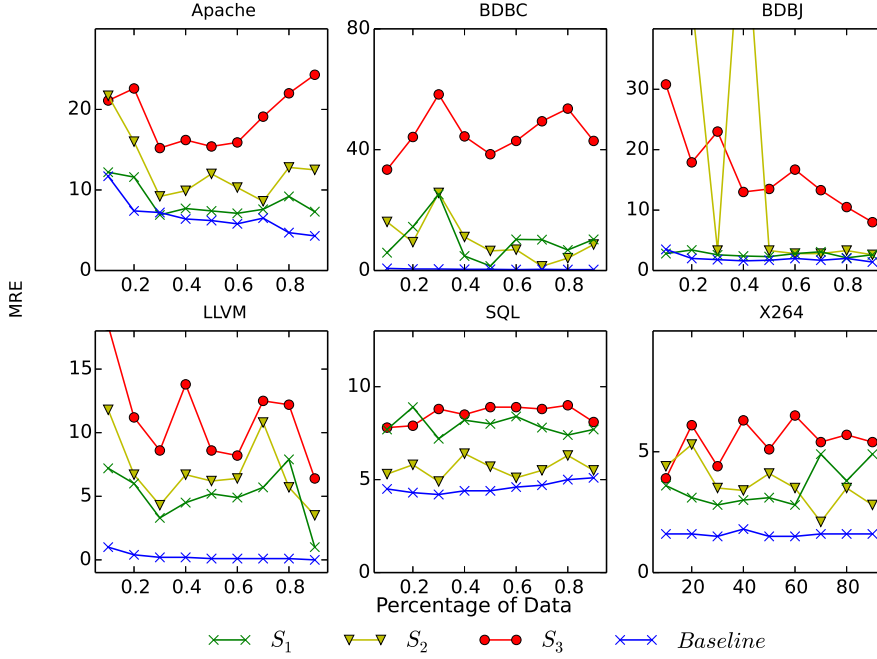
Fig. 3 Errors of the predictions made by **WHAT** with four different sampling policies. Note that, on the y-axis, *lower* errors are *better*.

5 Results

5.1 RQ1

> *Can **WHAT** generate good predictions after examining only a small number of configurations?*

Figure 3 shows the mean errors of the predictors learned after taking $X\%$ of the configurations, then asking **WHAT** and some sampling method (S_1, S_2, and S_3) to (a) find what configurations to measure; then (b) asking CART to build a predictor using these measurements. The horizontal axis of the plots shows what $X\%$ of the configurations are studied; the vertical axis shows the mean relative error (μ) from Equation 3. In this figure:

- The $\times$—$\times$ lines in Figure 3 show a *baseline* result where data from the performance scores of 100 % of configurations were used by CART to build a runtime predictor.
- The other lines show the results using the sampling methods defined in Section 3.2. Note that these sampling methods used runtime data only from a subset of 100 % of the performance scores seen in configurations from 0 to $X\%$.

In Figure 3, *lower* y-axis values are *better* since this means lower prediction errors. Overall, we find that:
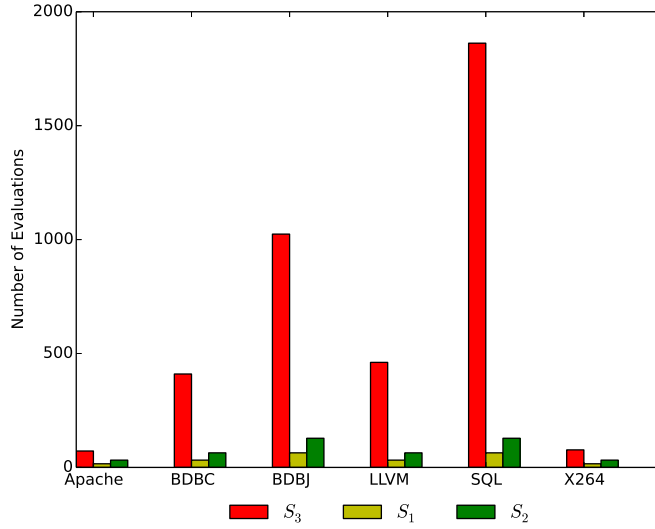
Fig. 4 Comparing evaluations of different sampling policies. We see that the number of configurations evaluated for S_2 is twice as high as S_1, as it selects 2 points from each cluster, where as S_1 selects only 1 point.

- Some of the subject systems exhibit large variances in their error rate, below $X = 40\%$ (e.g., BDBC and BDBJ).
- Above $X = 40\%$, there is little effect on the overall change of the sampling methods.
- Mostly, S_3 shows the highest overall error, so that it cannot be recommended.
- Always, the $\times\!\!-\!\!\times$ baseline shows the lowest errors, which is to be expected since predictors built on the baseline have access to all data.
- We see a trend that the error of S_1 and S_2 are within 5% of the *baseline* results. Hence, we can recommend these two minimal sampling methods.

Figure 4 provides information about which of S_1 or S_2 we should recommend. This figure displays data taken from the $X = 40\%$ point of Figure 3 and displays how many performance scores of configurations are needed by our sampling methods (while reflecting on the configurations seen in the range $0 \leq X \leq 40$). Note that:

- S_3 needs up to thousands of performance scores points, so it cannot be recommended as minimal-sampling policy;
- S_2 needs twice as much performance scores as S_1 (S_2 uses *two* samples per leaf cluster while S_1 uses only *one*).
- S_1 needs performance scores only for a few dozen (or less) configurations to generate the predictions with the lower errors seen in Figure 3.

Combining the results of Figure 3 and Figure 4, we conclude that:

S_1 is our preferred spectral sampling method. Furthermore, the answer to RQ1 is "yes", because applying **WHAT**, we can (a) generate runtime predictors using just a few dozens of sample performance scores; and (b) these predictions have error rates within 5 % of the error rates seen if predictors are built from information about all performance scores.

5.2 RQ2

Do less data used in building prediction models cause larger variances in the predicted values?

Two competing effects can cause increased or decreased variances in performance predictions. In our study, we report standard deviation (σ) as a measure of variances in the performance predicitons. The less we sample the configuration space, the less we constrain model generation in that space. Hence, one effect that can be expected is that models learned from too few samples exhibit large variances. But, a compensating effect can be introduced by sampling from the spectral space since that space contains fewer confusing or correlated variables than the raw configuration space. Figure 5 reports which one of these two competing effects are dominant. Figure 5 shows that after some initial fluctuations, after seeing $X = 40$ % of the configurations, the variances in prediction errors reduces to nearly zero, which is similar to the results in figure 3.

Based of the results of Figure 5, we answer RQ2 with "no": Selecting a small number of samples does not necessarily increase variance (at least to say, not in this domain).

5.3 RQ3

Can "good" surrogate models (to be used in optimizers) be built from minimal samples?

The results of answering RQ1 and RQ2 suggest to use **WHAT** (with S_1) to build runtime predictors from a small sample of data. RQ3 asks if that predictor can be used by an optimizer to infer what *other* configurations correspond to system configurations with fast performance scores. To answer this question, we ran a random set of 100 configurations, 20 times, and related that baseline to three optimizers (GALE [26], DE [40] and NSGA-II [9]) using their default parameters.

When these three optimizers mutated existing configurations to suggest new ones, these mutations were checked for validity. Any mutants that violated the system's constraints (e.g., a feature excluding another feature) were rejected and the survivors were "evaluated" by asking the CART surrogate model. These evaluations either rejected the mutant or used it in generation $i + 1$, as the basis for a search for more, possibly better mutants.
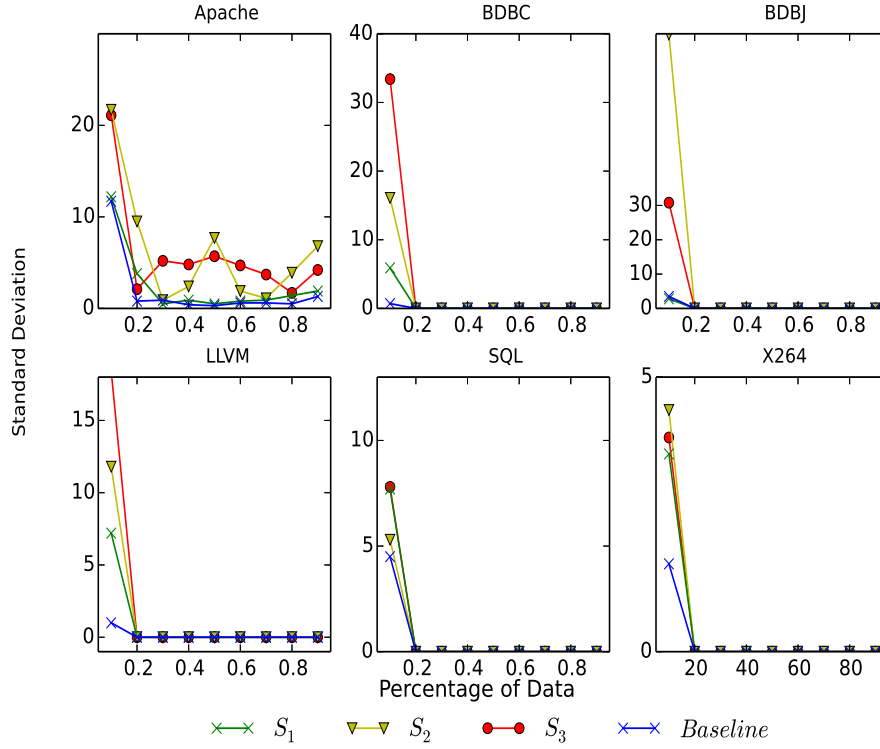
Fig. 5 Standard deviations seen at various points of Figure 3.

Figure 6 shows the configurations found by the three optimizers projected onto the ground truth of the performance scores of nearly all configurations (see Section 4.2). Again note that, while we use that ground truth for the validation of these results, our optimizers used only a small part of that ground-truth data in their search for the fastest configurations (see the **WHAT** + S_1 results of Figure 4).

The important information of Figure 6 is that all the optimized configurations fall within 1 % of the fastest configuration according to the ground truth (see all the left-hand-side dots on each plot). Table 2 compares the performance of the optimizers used in this study. Note that the performances are nearly identical, which leads to the following conclusions:

> Based on the results of figure 6 answer to RQ3 is "yes": For optimizing performance scores, we can use surrogates built from few runtime samples. The choice of the optimizer does not critically effect this conclusion.
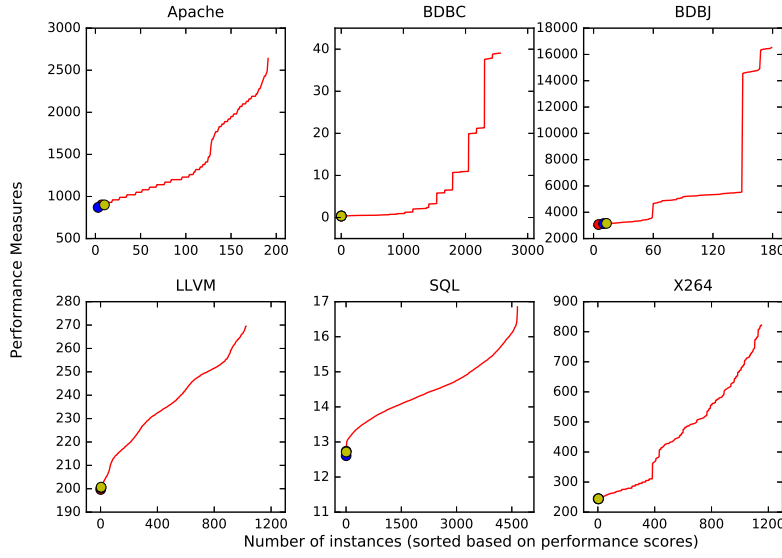
Fig. 6 Solutions found by GALE, NSGA-II, and DE (shown as points) laid against the ground truth (all known configuration performance scores). It can be observed that all the optimizers can find the configuration with lower performance scores.

Table 2 The table shows how the minimum performance scores as found by the learners GALE, NSGA-II, and DE, vary over 20 repeated runs. Mean values are denoted μ and IQR denotes the 25th–75th percentile. A low IQR suggests that the surrogate model build by **WHAT** is stable and can be utilized by off the shelf optimizers to find performance-optimal configurations.

	Searcher					
	GALE		DE		NSGAII	
	Mean	IQR	Mean	IQR	Mean	IQR
Apache	870	0	840	0	840	0
BDBC	0.363	0.004	0.359	0.002	0.354	0.005
BDBJ	3139	70	3139	70	3139	70
LLVM	202	3.98	200	0	200	0
SQLite	13.1	0.241	13.1	0	13.1	0.406
X264	248	3.3	244	0.003	244	0.05

5.4 RQ4

How good is **WHAT** *compared to the state of the art of learning performance predictors from configurable software systems?*

We compare **WHAT** with the three state-of-the-art predictors proposed in the literature [39], [18], [34], as discussed in Section 2. Note that all approaches use regression-trees as predictors, except Siegmund's approach, which uses a regression function derived using linear programming. The results were studied using non-

Rank	Approach	Mean MRE(μ)	STDev(σ)		#Evaluations
Apache					
1	Sarkar	7.49	0.82	•	55
1	Guo(PW)	10.51	6.85	—•—	29
1	Siegmund	10.34	11.68	—•——	29
1	**WHAT**	10.95	2.74	—•—	16
1	Guo(2N)	13.03	15.28	——•——	18
BDBC					
1	Sarkar	1.24	1.46	•	191
2	Siegmund	6.14	4.41	•	139
2	**WHAT**	6.57	7.40	•—	64
2	Guo(PW)	10.16	10.6	—•—	139
3	Guo(2N)	49.90	52.25	———•—	36
BDBJ					
1	Guo(2N)	2.29	3.26	•——	52
1	Guo(PW)	2.86	2.72	—•—	48
1	**WHAT**	4.75	4.46	——•——	16
2	Sarkar	5.67	6.97	———•——	48
2	Siegmund	6.98	7.13	———•—	57
LLVM					
1	Guo(PW)	3.09	2.98	—•—	64
1	**WHAT**	3.32	1.05	•—	32
1	Sarkar	3.72	0.45	•	62
1	Guo(2N)	4.99	5.05	——•——	22
2	Siegmund	8.50	8.28	———•——	43
SQLite					
1	Sarkar	3.44	0.10	•	925
2	**WHAT**	5.60	0.57	•	64
3	Guo(2N)	8.57	7.30	—•——	78
3	Guo(PW)	8.94	6.24	—•—	566
4	Siegmund	12.83	17.0	——•———	566
x264					
1	Sarkar	6.64	1.04	•	93
1	**WHAT**	6.93	1.67	•	32
1	Guo(2N)	7.18	7.07	—•—	32
1	Guo(PW)	7.72	2.33	•—	81
2	Siegmund	31.87	21.24	———•——	81

Fig. 7 Mean MRE(μ) seen in 20 repeats. Mean MRE is the prediction error as described in Equation 3 and STDev (σ) is the standard deviation of the MREs found during multiple repeats. Lines with a a dot in the middle (e.g. —•—) show the mean as a round dot withing the IQR (and if the IQR is very small, only a round dot will be visible). All the results are sorted by the mean values: a lower mean value of MRE is better than large mean value. The left-hand side column (*r*ank) ranks various techniques for e.g. when comparing various techniques for Apache, all the techniques have the same rank since their mean values are not statistically different. *Rank* is computed using Scott-Knott, bootstrap 95% confidence, and A12 test.

parametric tests, which was also used by Arcuri and Briand at ICSE '11 [30]). For testing statistical significance, we used non-parametric bootstrap test 95% confidence [13] followed by an A12 test to check that any observed differences were not trivially small effects; i.e. given two lists X and Y, count how often there are larger numbers in the former list (and there there are ties, add a half mark): $a = \forall x \in X, y \in Y \frac{\#(x>y)+0.5*\#(x=y)}{|X|*|Y|}$ (as per Vargha [42], we say that a "small" effect has $a < 0.6$). Lastly, to generate succinct reports, we use the Scott-Knott test to recursively divide

our optimizers. This recursion used A12 and bootstrapping to group together subsets that are (a) not significantly different and are (b) not just a small effect different to each other. This use of Scott-Knott is endorsed by Mittas and Angelis [30] and by Hassan et al. [16].

As seen in Figure 7, the FW heuristic of Siegmund et al. (i.e., the sampling approach using the fewest number of configurations) has the higher errors rate and the highest standard deviation on that error rate (four out of six times). Hence, we cannot recommend this method or, if one wishes to use this method, we recommend using the other sampling heuristics (e.g., HO, HS) to make more accurate predictions (but at the cost of much more measurements). Moreover, the size of the standard deviation of this method causes further difficulties in estimating which configurations are those exhibiting a large prediction error.

As to the approach of Guo et al. (with PW), it does not standout on any of our measurements. Its error results are within 1% of **WHAT**; its standard deviations are usually larger and it requires much more data than **WHAT** (Evaluations column of the figure 7). In terms of the number of measure samples required to build a model, the right-hand column of Figure 7 shows that **WHAT** requires the fewest samples except for two cases: the approach of Guo et al. (with 2N) working on BDBC and LLVM. In both these cases, the mean error and standard deviation on the error estimate is larger than **WHAT**. Furthermore, in the case of BDBC, the error values are $\mu = 14\%$, $\sigma = 13\%$, which are much larger than **WHAT**'s error scores of $\mu = 6\%$, $\sigma = 5\%$.

Although the approach of Sarkar et al. produces an error rate that is sometimes less than the one of **WHAT**, it requires the highest number of measurements. Moreover, **WHAT** 's accuracy is close to Sarkar's approach (1% to 2%) difference). Hence, we cannot recommend this approach, too.

Table 3 shows the number of evaluations used by each approaches. We see that most state-of-the-art approaches often require many more samples than **WHAT**. Using those fewest numbers of samples, **WHAT** has within 1% to 2 % of the lowest standard deviation rates and within 1 to 2 % of lowest error rates. The exception is Sarkar's approach, which has 5 % lower mean error rates (in BDBC, see the Mean MRE column of figure 7). However, as shown in right-hand side of Table 3, Sarkar's approach needs nearly three times more measurements than **WHAT**.

To summarize, there are two cases in Figure 7 where **WHAT** performs worse than, at least, one other method:

- SQLite: The technique proposed by Sarkar et al. does better than **WHAT** (3.44 vs 5.6) but, as shown in the final column of Figure 7, does so at the cost of $\frac{925}{64} \approx 15$ times more evaluations that **WHAT**. In this case, a pragmatic engineer could well prefer our solution over that of Sarkar et al. (since number of evaluations performed by Sarkar et al.more than an order of magnitude than **WHAT**).
- BDBC: Here again, **WHAT** is not doing the best but, compared to the number of evaluations required by all other solutions, it is not doing particularly bad.

Given the overall reduction of the error is small (5 % difference between Sarkar and **WHAT** in mean error), the cost of tripling the data-collection cost is often not feasible in a practical context and might not justify the small additional benefit in accuracy.

Table 3 Comparison of the number of the samples required with the state of the art. The grey colored cells indicate the approach that requires the lowest number of samples. We notice that WHAT and Guo (2N) uses less data compared to the other approaches. The high fault rate of Guo (2N) accompanied with high variability in the predictions makes WHAT our preferred method.

	Samples				
	Siegmund	Guo (2N)	Guo (PW)	Sarkar	WHAT
Apache	29	181	29	55	16
BDBC	139	36	139	191	64
BDBJ	48	52	48	57	16
LLVM	62	22	64	43	32
SQLite	566	78	566	925	64
X264	81	32	81	93	32

> Based on the results of figure 7, we answer **RQ4** with "yes", since **WHAT** yields predictions that are similar to or more accurate than prior work, while requiring fewer samples.

6 Why does it work?

In this section, we present an in-depth analysis to understand why our sampling technique (based on a spectral learner) achieves such low mean fault rates while being stable (low variance). We hypothesize that the configuration space of the system configuration lie on a low dimensional manifold.

6.1 History

Menzies et. al [29] demonstrated how to exploit the underlying dimension to cluster data to find local homogeneous data regions in an otherwise heterogeneous data space. The authors used an algorithm called WHERE (see section 3.3), which recurses on two dimensions synthesized in linear time using a technique called FASTMAP [14]. The use of underlying dimension has been endorsed by various other researchers [1, 2, 10, 45]. There are numerous other methods in the literature, which are used to learn the underlying dimensionality of the data set such as Principal Component Analysis (PCA) [24] [8], Spectral Learning [36] and Random Projection [3]. These algorithms use different techniques to identify the underlying, independent/orthogonal dimensions to cluster the data points and differ with respect to the computational complexity and accuracy. We use WHERE since it computationally efficient $O(2N)$, while still being accurate.

---

[8] WHERE is an approximation of the first principal component
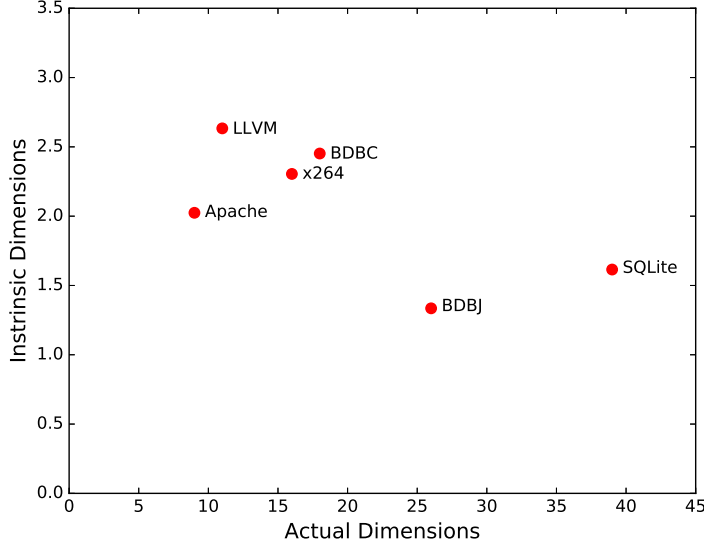
Fig. 8 The actual dimensions are shown on the x-axis and intrinsic dimensionality is shown on the y-axis. The points are annotated with the names of the corresponding software system. The intrinsic dimensionality of the systems are much lower than the actual dimensionality (number of columns in the dataset).

6.2 Testing Technique

Given our hypothesis the configuration space lies in a lower dimensional hyperplane — it is imperative to demonstrate that the intrinsic dimensionality of the configuration space is less than the actual dimension. To formalize this notion, we borrow the concept of correlation dimension from the domain of physics [17]. The correlation dimension of a dataset with k items is found by computing the number of items found at distance withing radius r (where r is the Euclidean distance between two configurations) while varying r. This is then normalized by the number of connections between k items to find the expected number of neighbors at distance r. This can be written as:

$$C(r) = \frac{2}{k(k-1)} \sum_{i=1}^{n} \sum_{j=i+1}^{n} I(||x_i, x_j|| < r) \qquad (4)$$

$$where : I(x < y) = \begin{cases} 1, & \text{if x <y} \\ 0, & \text{otherwise} \end{cases}$$

Given the dataset with k items and range of distances $[r_0 – r_{max}]$, we estimate the intrinsic dimensionality as the mean slope between $\ln(C(r))$ and $\ln(r)$.
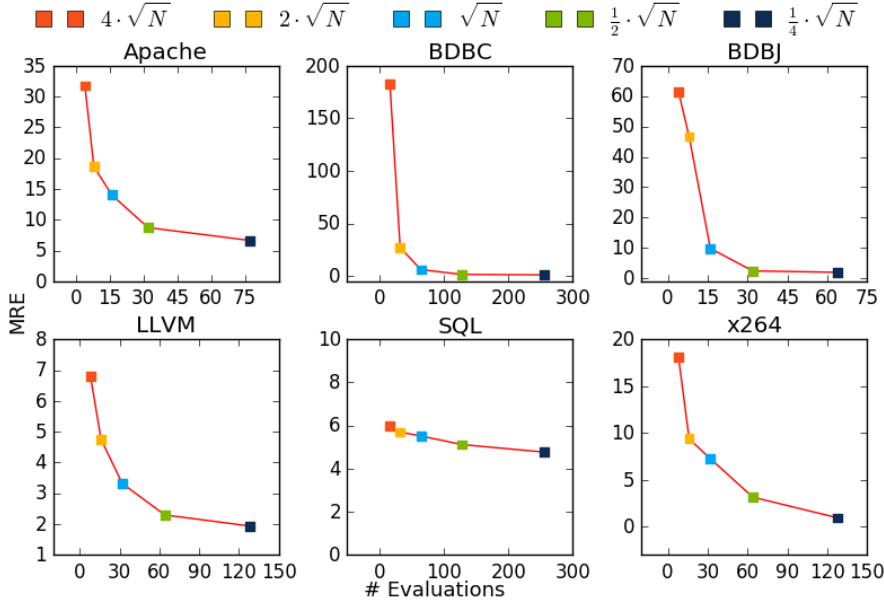
Fig. 9 The trade-off between the number of evaluations (affected by the size of the sub-region) and the performance (MRE) of the model generated.

6.3 Evaluation

On the configuration space of our subject systems, we observe that the intrinsic dimensionality of the software system is much lower than the actual dimension. Figure 8 presents the intrinsic dimensionality along with the actual dimensions of the software systems. If we take a look at the intrinsic dimensionality and compare it with the actual dimensionality, then it becomes apparent that the configuration space lies on a lower dimensional hyperplane. For example, SQLite has 39 configuration options, but the intrinsic dimensionality of the space is just 1.61 (this is a fractal dimension). At the heart of **WHAT** is WHERE (a spectral clusterer), which uses the approximation of the first principal component to divide the configuration space and hence can take advantage of the low intrinsic dimensionality.

As a summary, our observations indicate that the intrinsic dimension of the configuration space is much lower that its actual dimension. Hence, clustering based on the intrinsic dimensions rather than the actual dimension would be more effective. In other words, configurations with similar performance values lie closer to the intrinsic hyperplane, when compared to the actual dimensions, and may be the reason as to why **WHAT** achieves empirically good results.

7 Discussion

7.1 What is the trade-off between the MRE and the number of measurements?

WHAT requires that the practitioner to define a stopping criterion, (size of the sub-region) before the process commences. The stopping criterion preempts the process of recursive division of regions based on projection values of the configurations. In our experiments, the number of measurements or the size of the training set depends on the stopping criterion. An early termination of the sampling process would lead to a very inaccurate performance model, while late termination would result in resource wastage. Hence, it is very important to discuss the trade-off between the upper bound of the size of the sub-region and the MRE of the model built. In Figure 9, we show the trade-off between the MRE found and the number of measurements (size of training set). The trade-off characterizes the relationship between two conflicting objectives, for example, point in Apache, (size of sub-region=$4 \cdot \sqrt{N}$) requires very few measurements but the MRE of the model built is the highest, whereas point (size of sub-region=$\frac{1}{4} \cdot \sqrt{N}$) requires large number of measurements, but the MRE of the model built is the lowest. Since our objective is to minimize the number of measurements while reducing MRE, we assign the value of $\sqrt{N}$ to the upper bound of the size of the sub-region for the purposes of our experiments.

7.2 What is the relationship between intrinsic dimensionality and difficulty of a problem (or dataset)?

Houle et al. [22] observe a clear correlation between dimensionality of a problem space and loss of performance, that is a problem represented in lower dimensions is easier to model than the same problem represented in higher dimensions. In a similar vein, Domingos [11] explains how our intuitions fail in higher dimensions and algorithms that work on lower dimensions does not work on higher dimensions. This is because the size of the training data required to create a generalized 'model' for such a high dimensional space is exponentially large [9]. This is generally referred to as the "curse of dimensionality", but what counteracts this is the "blessing of non-uniformity". Blessing of non-uniformity refers to the fact that the possible valid solutions in a space is not spread uniformly across the problem space but concentrated on or near a lower dimensional manifold. Hence, it is a rule of thumb of machine learning practitioners to reduce the dimension of a data set by projecting the data onto a lower dimensional orthogonal subspace that captures the variation in the data. Bruges [6] mentions that, if data lies in a lower dimensional space (with lower intrinsic dimensions), then modeling the data directly in lower dimensional manifold make it much easier to model. Our results are inline with the observation made by Bruges, as we show that few samples are enough to model a large (sometimes millions) space, which can be attributed to the low intrinsic dimensionality of the space.

There are several other techniques (similar to **WHAT**) that also exploit the non-uniformity of the data points, such as Random Projections [8] and Auto-encoders [21].

---

[9] Another challenge of having high dimensional search space is the amount of noise induced by irrelevant dimensions.

The central intuition is similar to our work: problems that contain intrinsic lower dimensions should be easier/cheaper to model than those with higher intrinsic dimensionality. That said, to the best of our knowledge, we are the first to propose exploring the lower intrinsic dimensionality of configuration spaces, and exploit those lower dimensions for the purposes of sampling.

7.3 What are the limitations of WHAT?

The limitations of **WHAT** are:

– **WHAT** cannot be used for non-numeric configuration options. However, it can be used for numeric configurations options, not just Boolean options (most related work only supports Boolean options).
– The configurable systems used in this paper are fairly easy to model using machine learning techniques such as CART, but there exist software systems, which cannot be model using CART (even using 40% of all possible configurations). For these systems, **WHAT** cannot be used to build accurate performance models.
– The effectiveness of **WHAT** depends on projecting the configurations on the approximated first principal component. The approximation of the first principal component require calculating the farthest points (points which is most dissimilar) in the configuration space using euclidean distance. However, there maybe systems where Euclidean distance (as used in this paper) cannot find the most dissimilar points [7]. For such systems, **WHAT** in current form will not be effective (which is a part of our future work).
– Finding near-optimal configuration can become challenging when the configuration space is non-convex. However, we did not find such systems during our empirical evaluations. The other point we would like to stress is: we wanted to build a tool, which can differentiate between good and not-so-good configurations using few evaluations and our goal is not to find the best configuration but rather near optimal solutions.

8 Reliability and Validity

Reliability refers to the consistency of the results obtained from the research. For example, how well independent researchers could reproduce the study? To increase external reliability, we took care to either clearly define our algorithms or use implementations from the public domain (SciKitLearn) [31]. Also, all the data used in this work are available on-line in the PROMISE[10] code repository and all our algorithms are on-line at github.com/ai-se/where.

Validity refers to the extent to which a piece of research actually investigates what the researcher purports to investigate [37]. *Internal validity* checks if the differences found in the treatments can be ascribed to the treatments under study.

---

[10] http://openscience.us/repo/performance-predict/cpm.html

One threat to internal validity of our experiments is the choice of *training and testing* data sets discussed in Figure 1. Recall that, while all our learners used the same *testing* data set, our untuned learners were only given access to *training* data.

Another threat to internal validity is *instrumentation*. The very low μ and σ error values reported in this study are so small that it is reasonable to ask whether they are due to some instrumentation quirk, rather than due to using a clever sample strategy:

- Our low μ values are consistent with prior work [34];
- As to our low σ values, we note that, when the error values are so close to 0%, the standard deviation of the error is "squeezed" between zero and those errors. Hence, we would expect that experimental rigs that generate error values on the order of 5 % and Equation 3 should have σ values of $0 \leq \sigma \leq 5$ (e.g., like those seen in our introduction).

Regarding SQLite, we cannot measure all possible configurations in reasonable time. Hence, we sampled only 100 configurations to compare prediction and actual performance values. We are aware that this evaluation leaves room for outliers. Also, we are aware that measurement bias can cause false interpretations [29]. Since we aim at predicting performance for a special workload, we do not have to vary benchmarks.

We aimed at increasing the *external validity* by choosing software systems from different domains with different configuration mechanisms and implemented with different programming languages. Furthermore, our subject systems are deployed and used in the real world. Nevertheless, assuming the evaluations to be automatically transferable to all configurable software systems is not fair. To further strengthen external validity, we run the model (generated by **WHAT** $+ S_1$) against other optimizers, such as NSGA-II and differential evolution [40]. That is, we validated whether the learned models are not only applicable for GALE style of perturbation. In Table 2, we see that the models developed are valid for all optimizers, as all optimizers are able to find the near optimal solutions.

9 Related Work

In 2000, Shi and Maik [36] claimed the term "spectral clustering" as a reference to their normalized cuts image segmentation algorithm that partitions data through a spectral (eigenvalue) analysis of the Laplacian representation of the similarity graph between instances in the data.

In 2003, Kamvar et al. [25] generalized that definition saying that "spectral learners" were any data-mining algorithm that first replaced the raw dimensions with those inferred from the spectrum (eigenvalues) of the affinity (a.k.a. distance) matrix of the data, optionally adjusted via some normalization technique).

Our clustering based on first principal component splits the data on a approximation to an eigenvector, found at each recursive level of the data (as described in §3.1). Hence, this method is a "spectral clusterer" in the general Kamvar sense. Note that, for our data, we have not found that Kamvar's normalization matrices are needed.

Regarding sampling, there are a wide range of methods know as experimental designs or designs of experiments [33]. They usually rely on fractional factorial designs as in the combinatorial testing community [27].

Furthermore, there is a recent approach that learns *performance-influence models* for configurable software systems [38]. While this approach can handle even numeric features, it has similar sampling techniques for the Boolean features as reported in their earlier work [39]. Since we already compared to that earlier work and do not consider numeric features, we did not compare our work to performance-influence models.

10 Conclusions & Future Work

Configurable software systems today are widely used in practice, but they impose challenges regarding finding performance-optimal configurations. State-of-the-art approaches require too many measurements or are prone to large variances in their performance predictions. To overcome these limitations, we have proposed a fast spectral learner, called **WHAT**, along with three new sampling techniques. The key idea of **WHAT** is to explore the configuration space with eigenvalues of the features used in a configuration to determine exactly those configurations for measurement that reveal key performance characteristics. This way, we can study many closely associated configurations with only a few measurements.

We evaluated our approach on six real-world configurable software systems borrowed from the literature. Our approach achieves similar to lower error rates, while being stable when compared to the state of the art. In particular, with the exception of Berkeley DB, our approach is more accurate than the state-of-the-art approaches by Siegmund et al. [39] and Guo et al. [18]. Furthermore, we achieve a similar prediction accuracy and stability as the approach by Sarkar et al [34], while requiring a far smaller number of configurations to be measured. We also demonstrated that our approach can be used to build cheap and stable surrogate prediction models, which can be used by off-the-shelf optimizers to find the performance-optimal configuration. We use the correlation dimension to demonstrate how the high dimensional configuration space of our subject systems has a low intrinsic dimensionality, which might be the reason why **WHAT** performs so well on these datasets.

As to future work, we plan to explore the implications of **WHAT**. Currently **WHAT** uses a static number of evaluations based on the total number of possible configurations ($\sqrt{N}$), which may not be useful for systems that are more difficult to model that the system used in this study. Hence, we need a progressive strategy, which can progressively sample new configurations and stop the sampling process based on either the performance score achieved or the budget allocated. Finally the current version of **WHAT**, assumes that all the features are of similar importance and uses a Euclidean distance to differentiate between good and 'not-so-good' solutions. There are certainly systems where not all the features are equally important or there are redundancy in terms of configuration options. Hence, using feature weighting techniques to find weight (importance) of configuration options and use that information to differentiate between configurations.

References

1. Nicolas Bettenburg, Meiyappan Nagappan, and Ahmed E Hassan. Think locally, act globally: Improving defect and effort prediction models. In *Proc. of IEEE Working Conference on Mining Software Repositories*, pages 60–69. IEEE, 2012.
2. Nicolas Bettenburg, Meiyappan Nagappan, and Ahmed E Hassan. Towards improving statistical modeling of software engineering data: Think locally, act globally! *Empirical Software Engineering*, 20(2):294–335, 2015.
3. Ella Bingham and Heikki Mannila. Random projection in dimensionality reduction: Applications to image and text data. In *Proc. of SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 245–250. ACM, 2001.
4. Daniel Boley. Principal direction divisive partitioning. *Data Mining and Knowledge Discovery*, 2(4):325–344, 1998.
5. Leo Breiman, Jerome Friedman, Charles J Stone, and Richard A Olshen. *Classification and Regression Trees*. CRC, 1984.
6. Christopher JC Burges. Dimension reduction: A guided tour. *Foundations and Trends in Machine Learning*, 2(4):275–365, 2010.
7. Jianfeng Chen, Vivek Nair, Rahul Krishna, and Tim Menzies. Is "sampling" better than "evolution" for search-based software engineering? *arXiv:1608.07617*, 2016.
8. Sanjoy Dasgupta. Experiments with random projection. In *Proc. of conference on Uncertainty in Artificial Intelligence*, pages 143–151. Morgan Kaufmann Publishers Inc., 2000.
9. Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multi-objective genetic algorithm: NSGA-II. *Transactions on Evolutionary Computation*, 6(2):182–197, 2002.
10. Constanze Deiters, Andreas Rausch, and Mirco Schindler. Using spectral clustering to automate identification and optimization of component structures. In *Proc. of International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*, pages 14–20. IEEE, 2013.
11. Pedro Domingos. A few useful things to know about machine learning. *Communications of the ACM*, 55(10):78–87, 2012.
12. Qian Du and James E Fowler. Low-complexity principal component analysis for hyperspectral image compression. *International Journal of High Performance Computing Applications*, 22(4):438–448, 2008.
13. Bradley Efron and Robert J Tibshirani. *An Introduction to the Bootstrap*. CRC, 1994.
14. Christos Faloutsos and King-Ip Lin. Fastmap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In *Proc. of International Conference on Management of Data*, pages 163–174. ACM, 1995.
15. Roger Fletcher. *Practical Methods of Optimization*. John Wiley & Sons, 2013.
16. Baljinder Ghotra, Shane McIntosh, and Ahmed E Hassan. Revisiting the impact of classification techniques on the performance of defect prediction models. In *Proc. of International Conference on Software Engineering*, pages 789–800. IEEE, 2015.
17. Peter Grassberger and Itamar Procaccia. Measuring the strangeness of strange attractors. *Physica D: Nonlinear Phenomena*, 9(1–2):189–208, 1983.
18. Jianmei Guo, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, and Andrzej Wasowski. Variability-aware performance prediction: A statistical learning approach. In *Proc. of International Conference on Automated Software Engineering*, pages 301–311. IEEE, 2013.
19. Greg Hamerly. Making k-means even faster. In *Proc. of International Conference on Data Mining*, pages 130–140. SIAM, 2010.
20. Mark Harman, S Afshin Mansouri, and Yuanyuan Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys*, 45(1):11, 2012.
21. Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.
22. Michael E Houle, Hisashi Kashima, and Michael Nett. Generalized expansion dimension. In *Proc. International Conference on Data Mining Workshops (ICDMW)*, pages 587–594. IEEE, 2012.

23. Alexander Ilin and Tapani Raiko. Practical approaches to principal component analysis in the presence of missing values. *Journal of Machine Learning Research*, 11(Jul):1957–2000, 2010.
24. Ian Jolliffe. *Principal Component Analysis*. Wiley Online Library, 2002.
25. Kamvar Kamvar, Sepandar Sepandar, Klein Klein, Dan Dan, Manning Manning, and Christopher Christopher. Spectral learning. In *Proc. of International Joint Conference of Artificial Intelligence*. Stanford InfoLab, 2003.
26. Joseph Krall, Tim Menzies, and Misty Davies. Gale: Geometric active learning for search-based software engineering. *Transactions on Software Engineering*, 41(10):1001–1018, 2015.
27. D Richard Kuhn, Raghu N Kacker, and Yu Lei. *Introduction to Combinatorial Testing*. CRC, 2013.
28. Ilya Gennadyevich LOSHCHILOV. *Surrogate-Assisted Evolutionary Algorithms*. PhD thesis, Citeseer, 2013.
29. Tim Menzies, Andrew Butcher, David Cok, Andrian Marcus, Lucas Layman, Forrest Shull, Burak Turhan, and Thomas Zimmermann. Local versus global lessons for defect prediction and effort estimation. *Transactions on Software Engineering*, 39(6):822–834, 2013.
30. Nikolaos Mittas and Lefteris Angelis. Ranking and clustering software cost estimation models through a multiple comparisons algorithm. *Transactions on Software Engineering*, 39(4):537–551, 2013.
31. Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12(Oct):2825–2830, 2011.
32. John Platt. Fastmap, metricmap, and landmark mds are all nystrom algorithms. In *Proc. of International Conference on Artificial Intelligence and Statistics*, 2005.
33. Friedrich Pukelsheim. *Optimal Design of Experiments*. SIAM, 2006.
34. Atri Sarkar, Jianmei Guo, Norbert Siegmund, Sven Apel, and Krzysztof Czarnecki. Cost-efficient sampling for performance prediction of configurable systems. In *Proc. of International Conference on Automated Software Engineering*, pages 342–352. IEEE, 2015.
35. Abdel Salam Sayyad, Joseph Ingram, Tim Menzies, and Hany Ammar. Scalable product line configuration: A straw to break the camel's back. In *Proc. of International Conference on Automated Software Engineering*, pages 465–474. IEEE, 2013.
36. Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. *Transactions on Pattern Analysis and Machine Intelligence*, 22(8):888–905, 2000.
37. Janet Siegmund, Norbert Siegmund, and Sven Apel. Views on internal and external validity in empirical software engineering. In *Proc. of International Conference on Software Engineering*, volume 1, pages 9–19. IEEE, 2015.
38. Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. Performance-influence models for highly configurable systems. In *Proc. of International Conference on Foundations of Software Engineering*, pages 284–294. ACM, 2015.
39. Norbert Siegmund, Sergiy S Kolesnikov, Christian Kästner, Sven Apel, Don Batory, Marko Rosenmüller, and Gunter Saake. Predicting performance via automated feature-interaction detection. In *Proc. of International Conference on Software Engineering*, pages 167–177. IEEE, 2012.
40. Rainer Storn and Kenneth Price. Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization*, 11(4):341–359, 1997.
41. Christopher Theisen, Kim Herzig, Patrick Morrison, Brendan Murphy, and Laurie Williams. Approximating attack surfaces with stack traces. In *Proc. of International Conference on Software Engineering*, pages 199–208. IEEE, 2015.
42. András Vargha and Harold D Delaney. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.
43. Gary M Weiss and Ye Tian. Maximizing classifier utility when there are data acquisition and modeling costs. *Data Mining and Knowledge Discovery*, 17(2):253–282, 2008.
44. Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. Hey, you have given me too many knobs!: Understanding and dealing with over-designed configuration in system software. In *Proc. of International Conference on Foundations of Software Engineering*, pages 307–319. ACM, 2015.
45. Feng Zhang, Quan Zheng, Ying Zou, and Ahmed E Hassan. Cross-project defect prediction using a connectivity-based unsupervised classifier. In *Proc. of International Conference on Software Engineering*, pages 309–320. ACM, 2016.

46. Yi Zhang, Jianmei Guo, Eric Blais, and Krzysztof Czarnecki. Performance prediction of configurable software systems by fourier learning. In *Proc. of International Conference on Automated Software Engineering*, pages 365–373. IEEE, 2015.

47. Marcela Zuluaga, Guillaume Sergent, Andreas Krause, and Markus Püschel. Active learning for multi-objective optimization. In *Proc. of International Conference in Machine Learning*, volume 28, pages 462–470. ICML, 2013.