Learning and Predicting the Performance of Configurable Software Systems

Seminar - Advanced Software Engineering: Non-Functional Aspects in Software Engineering

Lion Wagner

University of Stuttgart
Institute of Software Technology (ISTE)
70569 Stuttgart, Germany

Abstract. Today's programs are mostly highly configurable and customizable. Popular applications like Apache or MySQL can have hundreds of configurable parameters. Whilst providing flexibility to a customer this also brings some problems as have shown: having so many options can also heavily influence the performance of a software system. And the more options there are, the harder it gets to predict the behavior of software system. This paper will show that a brute force approach to this problem does not work as a general solution. Further it is going to explain and compare four different prediction methods for learning about the performance of highly configurable software systems. Also, the in this context often used machine learning strategy of Classification and Regression Trees is presented.

1 Introduction

As modern programs grow larger and more powerful they also provide many configuration opportunities to their customers. In some cases the number of parameters can be even greater than 500. Examples for this can be found in Fig. 1. With this large amounts of configuration options stakeholders or customers can be satisfied easier since they can tailor a program to their specific requirements. But with this large amount of options comes a bigger problem: "Unpredictability". Looking at an example of Apache Storm (Fig. 2a) shows that the performance of two configurations of a program can differ significantly. Fig. 2b shows that solely changing a single parameter can increase the response time of Apache Storm by up to 100%. Without using prediction methods such results are only visible after executing and measuring multiple, if not all, configurations of a system. Or in other words: when looking only at a single configuration, one cannot conclude whether that configuration is any good for the current requirements in.

This is where prediction comes into play. By learning about the performance of some configurations of the system it tries to generate a function that can give an expected performance for a not measured configuration. This can be used to solve the just mentioned problem of finding a near optimal solution.

Furthermore, performance prediction can be used to find default configurations. These should be configurations that fulfils most requirements to an acceptable level. The most straightforward approach to this problem would be a *brute-force* solution. In this case that would mean measuring each and every single valid configuration. As we will see later in Section 3 this approach is in general not feasible since the amount of valid configurations scales exponentially with the number of parameters. For that reason other approaches had to be found and especially the efficient sampling of a configuration space turned out to be a problem [8].

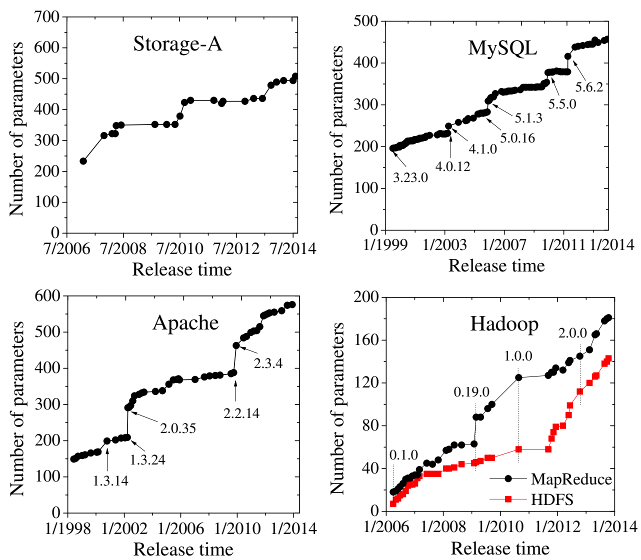
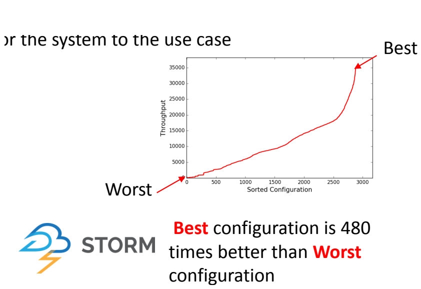
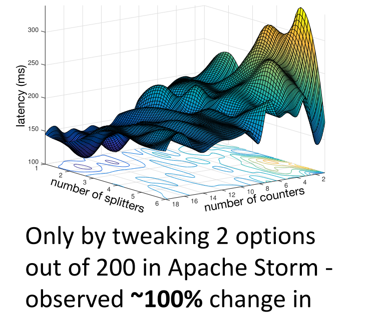


Fig. 1: Number of parameters of different popular programs [12].



(a) Configurations of Apache Storm sorted by measured throughput.



(b) Possible influences of only two options on the latency of Apache Storm.

Fig. 2: Measurements done for Apache Strom which shows that configurations can have a significant influence on the performance of a software system.

This paper will focus on showing different approaches and strategies to predicting the performance of a configurable software system. It will mainly discuss approaches developed by Norbert Siegmund et al. [2, 4, 8, 10]. They will be explained and compared. More specifically, this paper will have a look at four different approaches besides *brute-force*.

The first discussed technique is *Automated Feature Interaction Detection* (*AFID*) [10]. The goal of this approach is to assign a performance influence value to each feature and feature interaction. This is done by observing and measuring the behaviour of certain configurations. The other 4 approaches make use of a CART Tree as their learning choice but differ in the way they choose their sample.

Variability Aware Performance Prediction (*VAPP*) [2] uses random sampling to pick which configurations to compile and measure.

WHAT [4], tires a more mathematical way to find groups of similar configurations without actually measuring them. For this distance based clustering/sampling is used.

The last two sampling approaches are proposed in the same paper by Sarkar et al. [8].

Progressive and *Projective Sampling* are quite similar, since they both take advantage of the fact, that the general formula behind a learning curve is known. With this knowledge they generate a part of the actual curve and fit a function to it. Based on this function an optimal size for the actual sample set can be calculated. Both methods also take the cost of measurements (resources and accuracy) into consideration.
All these approaches can reach accuracies of over 94% on average in the conducted tests of their corresponding papers. This makes them good enough to

be relevant for the topic of this paper. Further their results can be compared straightforwardly since they are all tested on the same set of 6 software systems: Berkeley DB C, Berkeley DB Java, Apache, SQLite, LLVM, x264.

2 Definitions

Before the actual approaches are discussed it is important to pinpoint the definitions of terms which are used in this paper.

This paper often uses the terms “parameters”, (configuration) “options” or “features”. These terms are all equivalent and describe ways to adjust and optimize functional and non-functional properties of a software system [4]. One can divide into different types of options. Binary options usually have a value of 0 or 1 and describe the activation of a feature. Non-binary options support a wider range of values. For example this could be a setting for the stack-size allowed for a program. Non-numeric options support the input of text. Those could be paths or other addresses. The set of all configuration options is denoted as $\mathcal{O}$.

A *configuration* can be defined in multiple different ways. Kaltenecker et al. [4] define a configuration as a function $c : \mathcal{O} \rightarrow \{0, 1\}$. It assigns a 1 to each element of $\mathcal{O}$ that is selected and a 0 to those which are not used. Guo et al. [2] and Nair et al. [7] have a similar approach. But instead of using a function to describe c they use a vector or an n-tuple over $\mathbb{Z}_2^+ (= \{0, 1\})$. Each position of those enumerations is associated with exactly one feature. As in Kaltenecker et al. [4] a 1 indicates an activation of a feature and a 0 means that the feature is not used. These definitions obviously describes binary options only, but can be expanded to support non-binary options by using the co-domain of $\mathbb{N}_0$ instead of $\{0, 1\}$.

The *configuration space* describes all valid configurations of a system. It is denoted as $\mathcal{C}$.

A *sample* is the subset of a *configuration space* that contains all configurations that are complied and measured during the process of sampling and learning. It may also contain the measured performance scores for each configuration.

To describe the quality of an approach an *accuracy* metric is often used. The *accuracy* is defined as 1-*fault rate*. And in turn the fault rate (or error rate) is defined as

$$\text{fault rate} = \frac{|actual - predicted|}{actual}. \tag{1}$$

This definition can be found in [7] and [10].

3 On the Applicability of the Brute Force Approach

As mentioned previously: *brute force* measuring is not feasible in most cases [10]. But what is the actual reason behind this? Let’s have a look at an example first:

In one paper, Siegmund et al. [10] measured all valid configurations of multiple programs to analyse the accuracy of *AFID*. Berkley DB (C) was one of those programs. It is a database management program for embedded systems. It has 19 features and 2560 valid configuration. In the end, it took approximately 426 hours (= 17.75 days) to measure all these configurations. This value calculates to a time of about 10 minutes per configuration measurement. Considering that Berkeley DB (C) is a comparably small program, this is already a significant amount of time but still a time span that might be acceptable be used to get the perfect results of a *brute force* approach.

This changes once one takes a look at larger programs. Modern applications like Apache or MySQL can have hundreds of configuration parameters [12]. Another example was displayed by Siegmund [9]: SQLite has 77 features that can produce $3 \cdot 10^{77}$ valid configurations. Unfortunately, there is no evidence how the latter number was calculated. Yet, it can be assumed that at least the scale of this number in regard to the number of features is correct. This will be explained in the next paragraph. Coming back to the example: Assuming measuring (compiling + profiling) one configuration would take 5 minutes, a brute-force approach would take $2.5 \cdot 10^{76}$ hours. Obviously, this is not an acceptable duration.

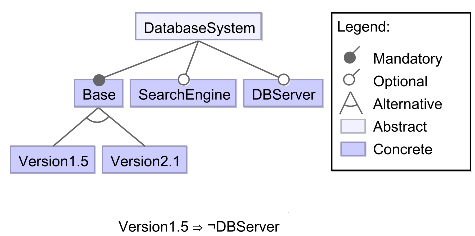


Fig. 3: An example of a feature model.

To find out why *brute-force* does not scale well, a generic look at how many configurations per program are needed to be measured helps. This set of all valid configurations can be written down as a *Feature Model*. An example for this is shown in Fig. 3. This *feature model* describes the software system called “DatabaseSystem”. *Feature Models* can be annotated with further logical expressions to also contain cross-feature constraints. Such can also be seen in the just mentioned example. Each *Feature Model* can also be written down as a logical expression. The atomic variables of this expression are the options of the software system. In this context, a configuration is written down as an assignment of each variable of the expression. The tree in Fig. 3 is equivalent to the function

$$V(C) = \text{Base} \wedge (\text{Version1.5} \oplus \text{Version2.1}) \wedge (\text{Version1.5} \Rightarrow \neg\text{DBServer}).$$

For $V(C) = 1$, a configuration is considered *valid*. Otherwise, it is not accepted. An example configuration could look like this:

$$\begin{aligned} \textbf{DEFAULT} = \{&\text{Base} = 1, \text{Version1.5} = 0, \text{Version2.1} = 1, \\ &\text{DBServer} = 0, \text{SearchEngine} = 0\} \end{aligned}$$

Note that these logical expressions can also be applied to non-binary options. The feature “Base” of the example could also be expressed as a tertiary value:

$$\text{Base} = \begin{cases} 0, \text{Not selected} \\ 1, \text{Version1.5 selected} \\ 2, \text{Version2.1 selected} \end{cases}$$

In this case, $V(C)$ would look like this:

$$V(C) = (\text{Base} \Leftrightarrow 1 \vee \text{Base} \Leftrightarrow 2) \wedge \big((\text{Base} \Leftrightarrow 1) \Rightarrow \neg\text{DBServer}\big).$$

So it is established that set of valid configurations can be expressed as a logical formula.

In this context the most interesting property of this formula is the scaling of the number of valid configurations in relation to the number of options. Without loss of generality, one can assume that a system only has numeric binary options. This can be done since all other types of options would just increase the overall number of options. However, already looking at binary options gives a satisfying result. Since naturally the total number of possible assignments for a logical expression is exponentially large, the number of valid configurations also lies in the exponential space of $\mathcal{O}(2^{\#options})$. In other words: a *configuration space* that would have to measured for a *brute force* approach would be exponentially large. That is the reason why brute-force does not scale well and more sophisticated methods are need for efficient predictions.

4 Measuring and Predicting the Performance of Highly Configurable Systems

By learning about the performance difference of multiple configurations it is possible to predict a program's performance accurately. This means, that not all (possibly exponentially many) configurations need to be measured. Instead, a small sample size should be enough to predict a program's performance. Multiple ways that use different methods have been proposed over time [7]. This paper will take a look at

- Automated Feature Interaction Detection by Siegmund et al. [10],
- an incremental/statistical learning approach by Guo et al. [2],
- Cost efficient sampling by Sarkar et al. [8],
- **WHAT** a spectral learning approach by Nair et al. [7],

4.1 General Approach

Guo et al. [2] defines two problems to be solved by prediction approaches:

1. Predicting the performance of a not measured configurations.
2. Finding a function f that shows the correlation between the properties of measured configurations and their performance value and that makes each predicted performance $f(\mathbf{x})$ of a configuration $\mathbf{x}$ as close as possible to its

actual performance.

$$f : \mathcal{C} \rightarrow \mathbb{R} \text{ such that } \sum_{(\mathbf{x},y)\in S} L(y, f(\mathbf{x})) \text{ is minimal} \tag{2}$$

S is a sample and L is a loss function to penalize errors in prediction. $(\mathbf{x}, y)$ is a pair of a configuration and its measured performance value. The function f is sometimes called the *performance model* of the system.

There is a general pattern for the solution of those problems that comes apparent when looking at different prediction approaches. It can be divided into two steps as displayed in Fig. 4.

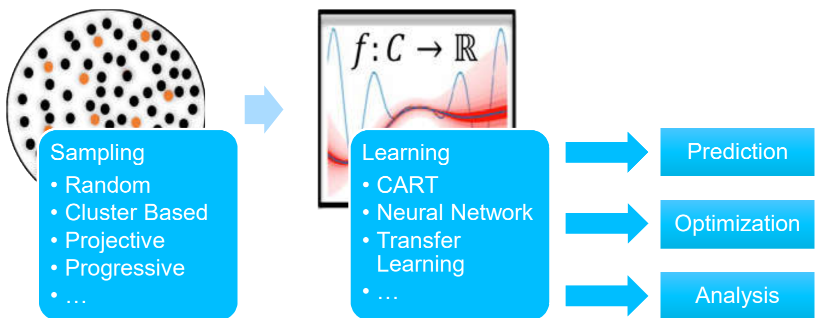


Fig. 4: General pattern of prediction approaches.

The first step is to sample the exponential configuration space. This means finding configurations from which can be learned about the system. This is done by using efficient sampling techniques like spectral sampling [4] or progressive sampling [8].

Once enough and meaningful configurations are found, the learning process starts. Usually the previously chosen configurations are measured, under the condition that this was not already done whilst sampling. The measurement results are fed to a learning process. A lot of different machine learning strategies can be applied [9]. The relative papers typically use *CART*'s. Based on the found performance model continuing tasks like finding near optimal solutions or in-depth performance analysis can be done [9].

4.2 Automated Feature Interaction Detection

Automated feature interaction detection (*AFID*) is a measurement-based approach to predicting the performance of a highly configurable system. It was developed by Siegmund et al. [10]. The following section is also based on the proposing paper of *AFID* [10]. Under the usage of linear regression *AFID* tries

to determine a performance influence value for each feature and feature interaction. A *feature interaction* is defined as a unexpected influence on the performance of a system when using a specific feature combination. In the conducted experiments of Sarkar et al. [8] and Siegmund et al. [10] this method reaches average accuracies of 85% and 95% respectively.
The gereral process of *AFID* is displayed in Fig. 5. It can be divided into two different steps:

1. Finding interacting Features.
2. Measuring the performance influence of feature interactions.

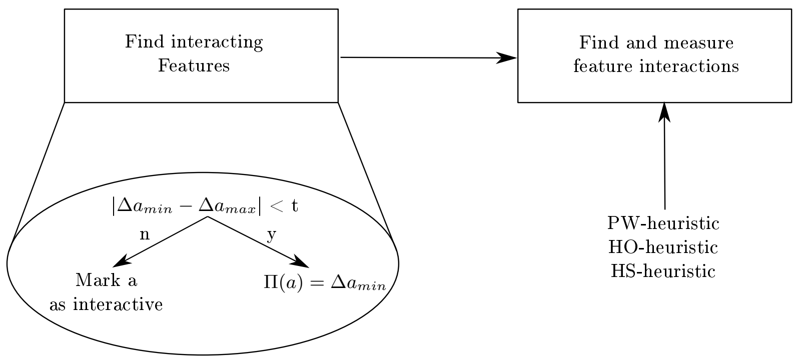


Fig. 5: General steps of *AFID*.

For the first step some notation is needed. For simplicity *AFID* is defined for binary options. The composition of performance influencing units (features or feature interactions) is denoted by a $\cdot$. In case two features are used simultaneously it is denoted by using a $\times$. This would also be another way to describe a configuration. So a program P that uses the two features a and b can be denoted as $P = a \times b$.
If one now wants to know the performance of P, they have to calculate $\Pi(P)$. The exact definition of the performance influence determining function Π can be found in the paper of Siegmund et al. [10]. For now, it is important to note that when calculating $\Pi(P) = \Pi(a \times b)$ not only the performance influence of a and b are necessary but also has $a\#b$ has to be considered. The latter is the performance influence of the possible interaction between a and b. So $\Pi(a \times b) = \Pi(a\#b) + \Pi(a) + \Pi(b)$. This can also go into higher order interaction: When using a program configuration $P_2 = a \times b \times c$ then $\Pi(P_2) = \Pi(a) + \Pi(b) + \Pi(c) + \Pi(a\#b) + \Pi(a\#c) + \Pi(b\#c) + \Pi(a\#b\#c)$. Some interactions do not exist or have an influence on the system, those who actually have to be found and measured. Otherwise, one could end up doing a brute-force solution again.

Finding these interactions requires to find the related interacting features themselves first. This is done in by intelligently measuring certain configurations. *AFID* defines a feature a as interacting when

$$a \text{ interacts} \Leftrightarrow \exists C, D \subseteq \mathcal{C} | C \neq D \wedge |\Delta a_C - \Delta a_D| \leq t \tag{3}$$

with

$$\begin{aligned} \Delta a_C &= \Pi(C \times a) - \Pi(C) \\ &= \Pi(a\#C) + \Pi(a). \end{aligned} \tag{4}$$

t is a threshold depending on the given performance metric. Using these two equations one can determine whether a feature is interacting with other features using 4 measurements. These include $\Delta a_{min} = \Pi(a \times min(a)) - \Pi(min(a))$ and $\Delta a_{max} = \Pi(a \times max(a)) - \Pi(max(a))$. $min(a)$ is a configuration that contains the minimum possible features without using a. Simultaneously $max(a)$ is also a configuration that contains the maximum amount of possible features without a. Once these measurements are done Eq. (3) can be applied with $C = \Delta a_{min}$ and $D = \Delta a_{max}$. This is done for all features to find interacting ones. If a feature f is not found to be interactive its performance influence $\Pi(f)$ equals Δf_{min}.

Once all interacting features are found the search for the actual interactions starts. In this second step 3 different heuristics are used to determine which interactions are searched for.

Pair-Wise Heuristic (PW): Most groups of interacting features appear in the size of two [5, 10]. So it makes sense to look for pair interaction first.

Higher-Order Interactions Heuristic (HO): Siegmund et al. [10] only look at higher order interactions of the rank of three. Even higher ranks would take up too many measurement resources.

Hot-Spot Features Heuristic(HS): Based on [1] and [11] Siegmund et al. [10] assume that hot spot features exist. At last these specific type of interactions are findable too.

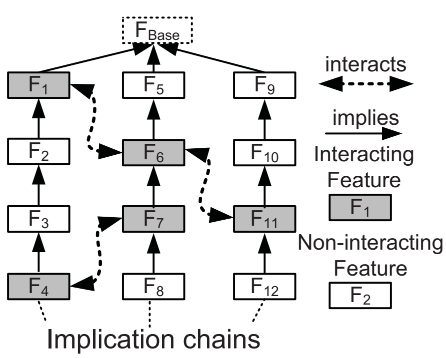


Fig. 6: Implication tree example found in [10].

The three heuristics will be applied in the order of PW $\rightarrow$ HO $\rightarrow$ HS and use the data provided by the previous ones. Using a SAT-Solver an implication-graph as seen in Figure 6 is generated. Each implication-chain in this tree should have at least one interacting feature. When analysing the tree each chain is walked from the top down.

First the influence of every feature on another chain is measured (PW-heuristic). In the example of Figure 6 the interactions would be measured in this order: $F1\#F6$,$F1\#F7$,$F4\#F6$, $F4\#F7$, $F6\#F11$, $F7\#F11$, $F1\#F11$, $F4\#F11$. Once an interaction impact exceeds a threshold it is recorded.

Secondly, the higher order interaction heuristic can be applied. Seconded order interactions can be relatively easily found by looking hat the results of the PW-Heuristik. Three features that interact pair-wise are likely to interact in a third order interaction. For example, looking at features a, b and c. If $\Delta a\#b_{C1}$ and $\Delta b\#c_{C2}$ have been recorded, $\{a\#b, b\#c, a\#c\}$ all have to be non-zero to find a third order interaction. Interactions with and order higher than three are not considered to prevent too many measurements.

Lastly, Hot-Spot features may be detected (HS-heuristic). This is done by counting the interactions per feature. If the number of interactions of a feature is above a certain threshold (e.g. the arithmetic mean) it is categorized as a Hot-Spot feature. Based on these hotspot features further third order interactions are explored. Again higher order interactions are not considered.
After applying the three heuristics all detected interacting features or feature combinations are assigned a Δ to represent their performance influence on the program.

As mentioned, Siegmund et al. [10] tested *AFID* on 6 different software systems. Each program was tested under four approaches: Feature-Wise, Pair-Wise, Higher-Order, Hot-Spot (in this order). Each approach also used the data found by the previous one. Accordingly, the results get better the more heuristics are used as Table 1 shows. Using only the FW-heuristic means, that interactions are not considered at all. However, the accuracy when using this heuristic is already at 80% on average. A significant improvement can be made by using the PW heuristic. It uses 8.5 times more measurements than the FW-heuristic on average, but improves the accuracy to 91%. Using the HO- or HS-heuristics improves the accuracy further by 2-4%. However, for Apache, using the HO over the PW heuristic even deteriorated the average result by 3.9% and doubled the standard deviation. As expected, using the HS-heuristic gives the best accuracy for all 6 tested applications. Siegmund et al. [10] also notes that analysing SQLite only needed about 0.1% of all possible configurations. This hints to the good scalability of *AFID*.

Table 1: Results of average accurcy found by Siegmund et al. [10].

Approach	avg. Accuracy
FW	79.7%
PW	91%
HO	93.7%
HS	95.4%

In conclusion *AFID* measures only a quadric amount of configurations to find the performance model of the software system. Depending on which heuristics are chosen, the accuracy and measurement costs can variate. Siegmund et al. [10] implemented this approach in their tool SPLConquerer[1]. This tool was also used to produce all experiment results.

[1] https://github.com/se-passau/SPLConqueror

4.3 Classification and Regression Trees

The next 3 approaches all use a specific type of machine learning strategy to construct their predictors. This method is call Classification and Regression Tree's ($CART$). These trees are typically binary trees that divide the given data points into small enough groups so a direct local prediction can be done. These local prediction are then combined into a global predictor [2]. In the context of configurable software systems each point consists at least out of a configuration and an associated performance score. These data point are then fed into the algorithm seen in Listing 1.1.

The node impurity found in the algorithm is typically calculated by the square mean error. To prevent *under-* or *overfitting*[3] the recursive splitting has to be stopped at the right time. This is possible by manual parameter tuning or using an empirical-determined automatic terminator. The size of the predictor variables set X is usually 1. This means that we decide the branching based on whether a feature is selected or not (or based on its value).

When using the $CART$ found in Fig. 7 a configuration $c = \{\ldots,\ x_3 = 1,\ \ldots,\ x_{14} = 0,\ x_{15} = 0,\ \ldots\}$ would be classified as S_{RRL}. The predicted performance of this class and therefore for c is $\ell_{S_{RRL}} = 571$.

```
1. Start at the root node. Assign all configurations
   to it.
2. For each option, find the set of options X that
   minizes the sum of the node impurities in the two
   child nodes. Divide the configurations assigned to
    the current node based on X into two disjoint
   sets and assign those to the corresponding child
   nodes.
3. If a stopping criterion is reached, exit. Otherwise
   , apply step 2 and 3 to each child node inturn.
```

Listing 1.1: Pseudocode for generating a $CART$. Adopted from Loh [6] to fit software configurations.

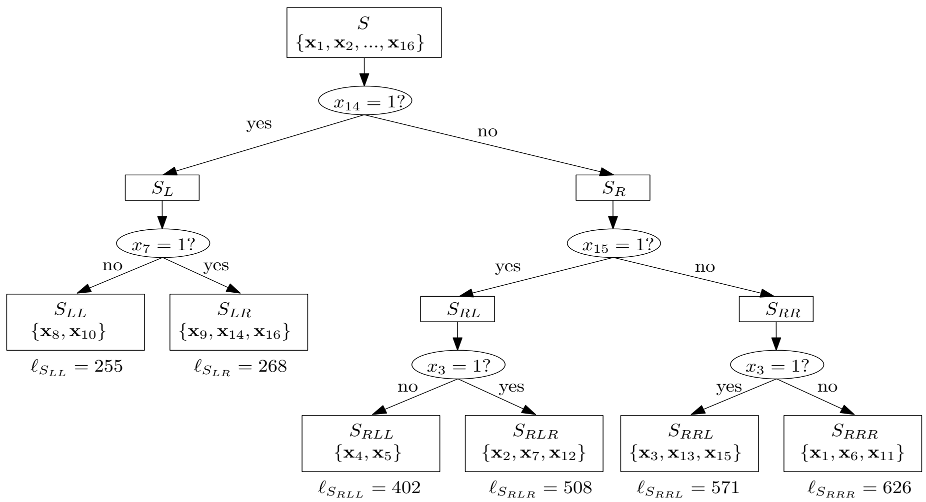


Fig. 7: Example performance model of X264 generated by CART based on the random sampling (N=16), using minimization of the sum of squared error loss [2].

4.4 Variability aware Performance Prediction

Variability aware Performance Prediction (*VAPP*) is a statistics based approach to performance prediction. With the help of random sampling and *CART*s a simple yet effective predictor can be build. The following section is based upon Guo et al. [2]. In their own tests Guo et al. [2] reached an average precision of 94% whilst using a sample as large as the ones *AFID* would be using under the PW-heuristic. Further, tests conducted by Nair et al. [7] with the same sample size showed an accuracy of 92.4%.
The basic idea of variability aware performance prediction is shown in Fig. 8. Two cycles can be found:

- The first cycle is outside of the dashed box and describes the basic input-output behaviour of a predictor. A user configures a new configuration $\mathbf{x}$ for System A and asks the predictor (dashed box) for a prediction. It replies with a quantitative prediction for $\mathbf{x}$'s performance.
- In the second cycle a actual prediction is generated based on decision rules which themselves are in turn created by simplifying a performance model (a *CART*). Random sampling is used to learn the performance model.

Like other approaches, the target of variability aware performance prediction is to get accurate predictions whilst only using a small sample for the creation of the performance model. Nonetheless, *VAPP* offers a free choice of the sample size. The configurations of the sample are chosen randomly out of $\mathcal{C}$.

VAPP uses the tuple-based definition of a configuration. It further defines that each configuration $c_j \in \mathcal{C}$ has an actual performance value y_j. For formal correctness it is assumed that every option of a configuration actually influences the performance of the system. Otherwise, a *CART* could not be applied.

wrap

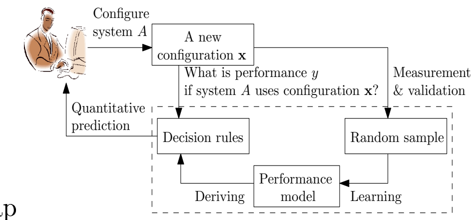


Fig. 8: Overview of the Approach of Variability aware Performance Prediction [2]

In the used *CART* each sub-tree is also called a segment S_i, where i determines the location of the sub-tree. ·This is also shown in Fig. 7.
For the *local model* ℓ of the used *CART* Guo et al. [2] choose the arithmetic average:

$$\ell_{S_i} = \frac{1}{|S_i|} \sum_{y_j \in S_i} y_j \tag{5}$$

As a loss function to penalize the prediction errors (node impurity) the sum of squared error loss is selected.

$$\sum_{y_j \in S_i} L(y_j, \ell_{S_i}) = \sum_{y_j \in S_i} (y_j - \ell_{S_i})^2 \tag{6}$$

Therefore the best split for a segment S_i is found when

$$\sum_{y_j \in S_{iL}} L(y_j, \ell_{S_{iL}}) + \sum_{y_j \in S_{iR}} L(y_i, \ell_{S_{iR}})$$

is minimal.

Assuming there are q leafs in a tree than the predictor function $f(\mathrm{x})$ is defined as:

$$f(\mathrm{x}) = \sum_{i=1}^{q} \ell_{S_i} I(\mathrm{x} \in S_i) \tag{7}$$

where I is an indicator function to indicate whether x belongs to a leaf S_i.
For the example of Fig. 7, $f(\mathrm{x})$ unwraps to:

$$\begin{aligned} f(x) = 255 &* I(x_{14} = 1, x_7 = 0) \\ +268 &* I(x_{14} = 1, x_7 = 1) \\ +402 &* I(x_{14} = 0, x_{15} = 1, x_3 = 0) \\ +508 &* I(x_{14} = 0, x_{15} = 1, x_3 = 1) \\ +571 &* I(x_{14} = 0, x_{15} = 0, x_3 = 1) \\ +626 &* I(x_{14} = 0, x_{15} = 0, x_3 = 0) \end{aligned}$$

Every possible configuration x is associated with a leaf of the tree. Therefore, $f(\mathrm{x})$ can always be applied.

For their Experiment Guo et al. [2] test the same software systems as Siegmund et al. [10] (Section 4.2). They also compared their prediction results with the results produced by *AFID*.
Since unlike *AFID* the size of a sample for *VAPP* can be chosen freely, some comparable sample sizes were chosen. Guo et al. [2] use 4 different sample sizes based on the size of the tested programs. For a program with N features they use samples the size of $N, 2N, 3N$ and M. M is the amount of configurations measured by *AFID* using the PW-heuristic. It was found, that the prediction accuracy increases linear with the size of a sample. For a small sample with the size of N the prediction accuracy was at above 92% in 3 cases. However, for Berkeley DB (C) the prediction accuracy with an N sized samples was at 112.4% with a standard deviation of ±354.6%. This shows that *VAPP*is not generally applicable for small samples. Using a sample size of M significantly improves the average prediction accuracy to a stable average of 93.8%.

4.5 Cost-Efficient Sampling

Just how its name suggests, cost-efficient sampling tires to minimize the cost-accuracy rate of a prediction approach. This is done by trying to find a (near) optimal sample size n^* for a given program.
Sarkar et al. [8] apply two different sampling techniques to reach this goal. Their general structure is displayed in Fig. 9.

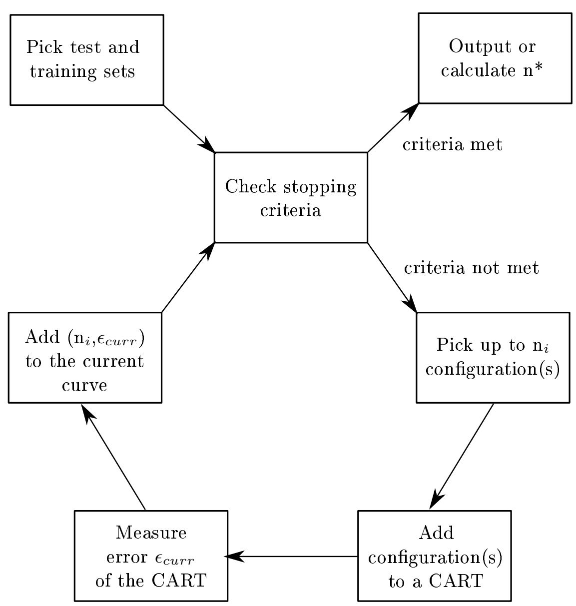


Fig. 9: General procedure of cost-efficient sampling to find an optimal sized n^*.

Firstly a test and a training set are picked from $\mathcal{C}$.

Then, in each iteration of this process, a performance model is build. The used sample is taken from the training set and is increased each iteration. This makes the respective performance model more accurate in each iteration. The error rate of each created performance model is recorded. Together with the size of the used sample a point in a graph is created. As iterations continue this graph resembles an approximation of the learning curve of the performance model of the system. A curve as displayed in Fig. 10 is created. This process continues until a stopping criterion is met. At this point n^* was already found or can be calculated based on the given curve and a cost-function. An optimal sample size should provide a good accuracy without being too large. So naturally n^* lies somewhere at the beginning of the third phase. Four different types of costs are considered by Sarkar et al. [8]:

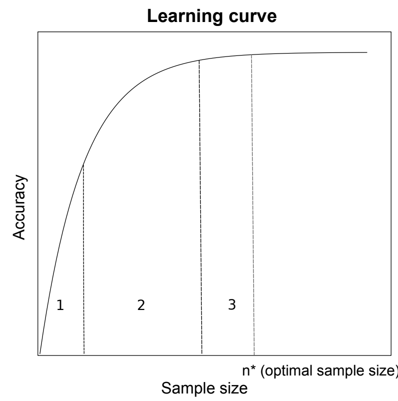


Fig. 10: A typical learnig curve can be divided into three phases: 1. steep incline; 2. gradual incline; 3. sturation/plateau. [8]

$$\begin{aligned} TotalCost = & Cost_{Measuerment(Training)} \\ & + Cost_{Measuerment(Testing)} \\ & + Cost_{ModelBuilding} \\ & + Cost_{PredictionError} \end{aligned} \tag{8}$$

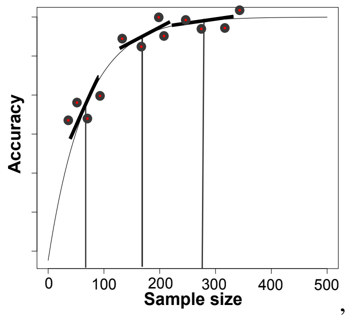
Based on this, the $TotalCost$ for a sample the size of n is defined as:

$$TotalCost(n) = 2n + \epsilon_n \cdot |S| \cdot R \tag{9}$$

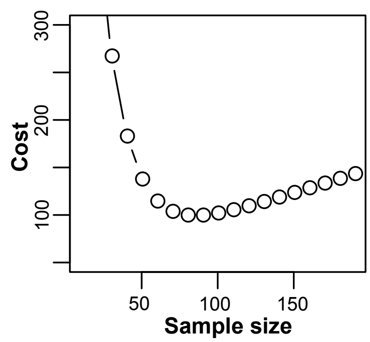
S is a *score set* that contains all configurations, whose performance value will be predicted with the current model. R is tuning parameter for the measuring cost of a training set.

Progressive sampling is the first technique that applies this process. It observes either the learning curve's progress or the related cost-function.

Projective sampling tries to generate the first part of the learning curve. Then function-fitting is applied to find and generate the remainder of the curve. Based on this fitted function, n^* can be calculated.



(a) Gradient-based



(b) Cost minimization

Fig. 11: Stopping criteria of *progressive sampling* [8].

Progressive sampling can be divided into two different strategy types. They differ in the way the next sample size n_{i+1} is calculated:

1. arithmetic: $n_i = n_0 + i \cdot a$
2. geometric: $n_i = n_0 + a^i$

The constant parameter a determines the growth-rate of the samples. On one hand arithmetic progressive-sampling is more precise, but on the other hand we need to build more performance models in comparison to geometric progressive-sampling. For *progressive sampling* there are two types of stopping criteria:

1. Gradient-Based: Additional models around n_i will be build so the gradient around at n_i can be determined. Fig. 11a shows an example of this. Once the gradient or accuracy reaches a certain threshold the iterative process is stopped and n^* set as n_i.
2. Cost Minimization: For each n_i the current error-rate of the performance model is substituted into the cost function of Eq. (9). For well-behaved learning curves the cost function is convex as displayed in Fig. 11b. Once the cost-function increases for the first time, the iterative process is stopped and n^* set as n_{i-1}.

Projective sampling tires to find the actual function behind the learning curve of the performance model of the current system. Typical equations that describe a learning curve and the respective minimum of the cost-function can be found in Table 2. Unlike in *progressive Sampling* the size of the sample set is increased by only a small amount each iteration. The experiments conducted by Sarkar et al. [8] show that even an increment by the size of 1 is enough to get satisfying results. This time the iterative process is stopped once a certain accuracy is reached. Now an algorithm tires to fit the generated points onto the four functions shown in Table 2. The best fitted function is considered the learning curve of the current performance model. Once this function is found, it can be substituted into to cost-function Eq. (9) and the minimum of which can be calculated. These minimums are also shown in Table 2.

Based on the found minimum other techniques that do not have a fixed sample size like *VAPP* can be used. In the experiments conducted by Sarkar et al.

Table 2: Functional representation of typical learning curves and thier optimal sample size based on the given cost function [8].

Name	**Equation**	**Optimal Sample Size**
Logarithmic	$err(n) = a + b.log(n)$	$n^* = -(R \cdot \|S\| \cdot b)/2$
Weiss and Tian	$err(n) = a + bn/(n+1)$	$n^* = \sqrt{(-R \cdot \|S\| \cdot b)/2}$
Power Law	$err(n) = an^b$	$n^* = (\frac{-2}{R \cdot \|S\| \cdot a \cdot b})^{\frac{1}{b-1}}$
Exponential	$err(n) = ab^n$	$n^* = \log_b \left(\frac{-2}{R \cdot \|S\| \cdot a \cdot \ln b} \right)$

[8] *projective sampling* was better than *progressive sampling* in every case. For each of the six tested software systems *projective sampling* had a better accuracy whilst having a lower cost than *progressive sampling*.
For both strategies it is important, that the initial sample generation/picking is successful. Therefore, the picking of representative configurations as n_i samples of each iteration is key. Sarkar et al. [8] use a heuristic based on feature frequencies to optimize the selection. As a general guideline each feature should be selected and deselcted once at least. Looking at a set of configurations S the feature frequency of a feature i is defined as:

$$1 \leq \sum_{j \in S} x_i(j) < |S| \tag{10}$$

Where $x_i \in \{0, 1\}$ describes, whether the feature i is selected in the configuration j or not. This feature frequency is recorded for whilst creating a new sample. In case a certain threshold is reached, the sample generation will be stopped.

4.6 WHAT

WHAT is a spectral learning approach developed by Nair et al. [7]. The following section is based on proposing paper of Nair et al. [7]. *WHAT* aims to find an accurate and stable performance model with fewer samples than the previous methods. To reach this goal it uses spectral and regression tree learning. The idea behind spectral learning is the mathematical concept of *eigenvalues/-vectors* of a distance matrix between configurations. This has the advantage of automatic noise reduction. Nair et al. [7] explain, that when a data set has many irrelevancies or closely associated data parameters d, then only a few eigenvectors $e, e \ll d$ are required to characterize the data. To find these important *eigenvalues/-vectors* a simple clustering algorithm is applied to $\mathcal{C}$. The main advantage of this approach are a reduced sample size and a lower standard deviation compared to previously shown methods [7]. Nair et al. [7] divide *WHAT* into 3 parts.

1. Spectral Learning
This first step of spectral learning is used to cluster all valid configurations. As each configuration is an n-dimensional vector (or n-tuple) it can be placed in an n-dimensional space.
WHAT gets N different valid configurations as input and is picks a random configuration N_i and two configurations $West$ and $East$. $West$ is the configuration that is most different to N_i and $East$ is the configuration most different to $West$. In mathematical terms the 'most different' can mean the geometrically distance or another difference value ($\in \mathbb{R}$) based on the type of options (Boolean, Paths, ...). After that a straight through $East$ and $West$ is calculated and all configuration are dived into two clusters. This division is based on the median value of all calculated distances. This process is recursively repeated for each sub-cluster until they reach a threshold size. Nair et al. [7] use $\sqrt{|N|}$ as their termination value. Unlike other clustering algorithms like K-Means, spectral learning runs in linear time of $\mathcal{O}(2|n|)$.

2. Spectral Sampling
For the actual sampling a probabilistic strategy can be applied. The most important is *random sampling*, where one configuration of each leaf cluster is randomly picked as a representative. This representative is then compiled and measured.
There are also two other sampling strategies mentioned. *East-West sampling* complies and executes $East$ and $West$ of the current leaf-cluster. Whereas *Exemplar sampling* measures all configurations of a leaf-cluster, but only returns the lowest performance score as a representative. Both of these strategies get outperformed by the random sampling strategy.

3. Regression-Tree Learning
In this step a CART is build from the chosen samples. This time the best split is defined as reaching the minimum of $\frac{A}{N}\sigma_1 + \frac{B}{N}\sigma_2$'[2]. σ_1 and σ_2 are the standard deviations of the corresponding sub-trees. From this CART again decision rules can be derived.

Results of testing *WHAT* on the programs we introduced earlier show that it has an average precision of 93.4%. Also the standard deviation is comparably low.[7]

5 Conclusion

Nair et al. [7] provide a good overview over the presented approches. It can be found in Fig. 12. Event though Sarkar's approach (which was not covered by

[2] The paper ([7]) defines A and B as sets and N as a (natural) number. So it may be assumed that the formula actually should be $\frac{|A|}{N}\sigma_1 + \frac{|B|}{N}\sigma_2$. This does make sense, since this formula weights both standard deviations σ proportional to total number of current configurations N.

this paper) was rated best in 4 of the 6 case studies it needs significantly larger samples to do so. In the case of SQLite the WHAT needs about 15 times less evaluations to get a result that differs only by 2%. Siegmunds's approach ([10]) is in last place on 4 occasions and has the greatest standard variation in almost all cases. One might argue that *AFID*is the worst of the presented approaches since it does have worse predictions than the others in most cases and always uses the second most (in 5 out of 6 cases) or most evaluations. The table also demonstrates that Guo's approach ([2]) is very inconsistent for small sample size. Guo(2N) has significantly worse predictions than other methods when its analysing Berkley DB C (18 Features) or Apache (9 Features). However, when looking at Berkley DB Java (26) or LLVM (11) it has an acceptable mean error rate. Further more its also visible that the side of *WHAT*'s usage of spectral learning lives up to its expectations. The standard deviation of *WHAT*is significantly lower than its competitors on 4 occasions. In the 2 left cases it sits in between Guo's and Siegmund's approaches.

In the end none of the presented methods has a edge over the others. Some need too large of a sample size and some are just not reliable or generally feasible. Gou's and Sarkar's approaches can do well under the usage of large sample sizes. Siegmund's approach is rather robust but suffers from large standard deviations the larger a system gets. *WHAT*seems to be the most consistent out of the 3 candidates. It has the lowest standard deviation and a rather low mean error fault rate on most tested occasions. Furthermore it needs the least samples to do so. At last the decision on which approach one should use to learn and predict the performance of a configurable software system should be done based on the properties of a software system. By looking at the properties of the in this paper presented approaches a suitable method should be findable.

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. 12: Overview and comparison over the mean error rate (mean MRE, μ) and the standard deviation (σ) of the presented approaches. Siegmund stands for *AFID*and Guo for *Variablity aware performance prediction*. Sarkar's approach was not discussed in this paper. "[The column] Rank is computed using Scott-Knott, bootstrap 95% confidence, and A12 test." [7]

References

[1] S. Apel and D. Beyer. Feature cohesion in software product lines: An exploratory study. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 421–430, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0445-0.

[2] J. Guo, K. Czarnecki, S. Apel, N. Siegmund, and A. Wasowski. Variability-aware performance prediction: A statistical learning approach. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 301–311. IEEE Press, 2013.

[3] T. Hastie, R. Tibishirani, and J. Friedman. *The Elements of Statistical Learning*. Springer-Verlag New York, New York, 2 edition, 2009. ISBN 978-0-387-84857-0.

[4] C. Kaltenecker, A. Grebhahn, N. Siegmund, J. Guo, and S. Apel. Distance-based sampling of software configuration spaces. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 1084–1094, Piscataway, NJ, USA, 2019. IEEE Press.

[5] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An analysis of the variability in forty preprocessor-based software product lines, 2010.

[6] W.-Y. Loh. Classification and regression trees. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 1(1):14–23, 2011.

[7] V. Nair, T. Menzies, N. Siegmund, and S. Apel. Faster discovery of faster system configurations with spectral learning. *CoRR*, abs/1701.08106, 2017.

[8] A. Sarkar, J. Guo, N. Siegmund, S. Apel, and K. Czarnecki. Cost-efficient sampling for performance prediction of configurable systems (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 342–352, Nov 2015.

[9] N. Siegmund. Challenges and insights from optimizing configurable software systems, 2019.

[10] N. Siegmund, S. S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, and G. Saake. Predicting performance via automated feature-interaction detection. In *In Proc. of ICSE*, pages 167–177. IEEE, 2012.

[11] C. Taube-Schock, R. Walker, and I. Witten. Can we avoid high coupling? pages 204–228, 07 2011.

[12] 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 *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 307–319, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3675-8.