Predicting and Learning 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. An overview over approaches to the learning and predicting the performance of a configurable software system. (Coming soon!)

1 Introduction

Todays programs are mostly highly configurable and customizable. Popular applications like Apache or MySQL can have a lot of configuration parameters as can be seen in Fig. 1. With this large amount of configuration options stake holders or customers can be satisfied easier since they can tailor a program to the their specific requirements. But with these big amount of options comes a bigger problem: "unpredictability". Looking at an example of Apache Storm (Fig. 2) shows that the performance of two configurations of a program can differ significantly. figure einfügen shows that solely changing a single parameter TODO 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 generates a function that can give an expected performance for a not measured configuration. This can be used to solve the just mention problem of finding a near optimal solution. Further performance prediction can be used to find default configurations. These should be configurations that fulfills most requirements to an acceptable level. The most straight forward 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 ref section this approach is in general not feasible since TODO 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 [16].

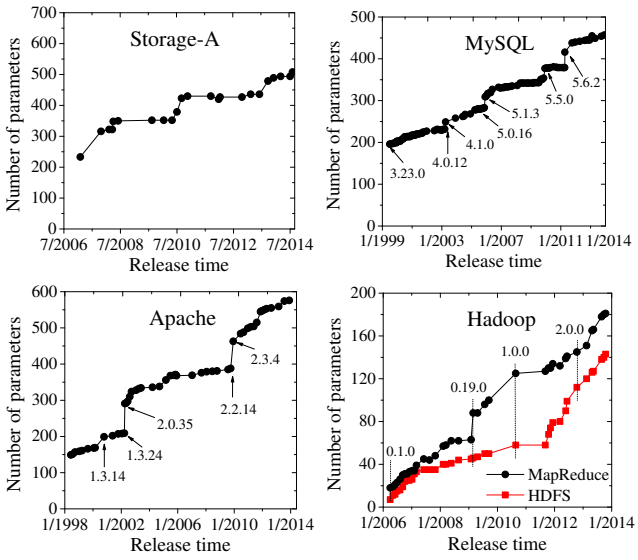
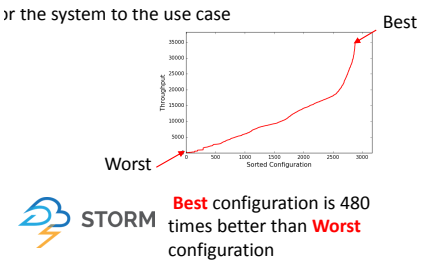


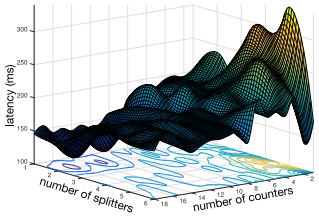
Fig. 1: Number of parameters of different popular programs [20].

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. [18, 7, 16, 10]. They will be explained and compared. More specificly this paper will have a look at 4 different approaches besides *brute-force*.

The first discussed technique is *Automated Feature Interaction Detection* (*AFID*) [18]. 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 behavior 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*) [7] uses random sampling to pick which configurations to compiled and measured. **WHAT** [10], as the next approach is called, 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. [16]. They *Progressive* and *Projective Sampling* are quite similar, since they both take advantage of the fact, that the general formular 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 a optimal size for the actual sample set can be calculated. Both methods also take the cost of measurements and over-/underfitting into consideration.
All these approaches reach an accuracy of over 94% on average in the conducted



(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 that show an configurations can have a significant influence on the performance of a software system.

tests of their corresponding papers. This makes them good enough to be relevant for the topic of this paper.

2 Definitions

Before the actual approaches are discussed it is important to pin point 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 [10]. 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. [10] define a configuration as a function $c : \mathcal{O} \to \{0, 1\}$. It assigns a 1 to each element of $\mathcal{O}$ that is selected and a 0 to those which are not used. Nair et al. [13] and [7] have a similar approach. But instead of using a function to describe c they use a vector or a n-tupel over $\mathbb{Z}_2^+ (= \{0, 1\})$. Each position of those enumerations is associated with exactly one feature. As in [10] 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 codomain 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.

3 On the Applicability of the Brute Force Approach

As already mentioned in the Introduction: Brute Force measureing is not feasible in most cases [18]. But whats the actual reason behind this? Let's have a look at an example first:

In one paper Siegmund et al. [18] measured all valid configurations of multiple programs to analyze 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 corresponds to a time of about 10 minutes per configuration measurement. Considering that Berkley DB (C) is a comparably small program this is already a significant amount of time but still a considerable margin that can actually be used to get perfect results.

This changes once one takes a look at larger programs. Modern applications like Apache or MySQL can have hundreds of configurable parameters [20]. Such can be seen in Fig. 1. For example Siegmund [17] says that one version SQLite had 77 features that produced $3 \cdot 10^{77}$ valid configurations. Unfortunately there is no evidence how the latter number was calculated. Yet we can assume that at least the scale of this number in regards to the number of features is correct. This is 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 $2.5 \cdot 10^7 6$ hours. This is obviously not an acceptable duration.

To find out why *brute-force* does not scale well one needs to have a generic look at how many configurations per program are need to be measured. This set of all valid configurations can be written down as a *Feature Model*. An example for this can be seen in Fig. 3. This graph describes the software system called "Database System". *Features Models* can be annotated with further logical expressions for more complicated configuration conditions. Such can also be seen in the just mentioned example. Each *Features 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 as seen in reference Figure is equivalent to the formular

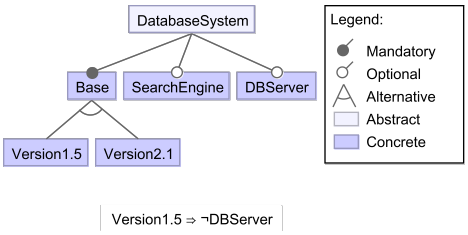


Fig. 3: And example of a feature model.

TODO

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

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

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

Note that these logical expressions also can 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 equal

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

This could also be expanded onto non numeric options that are set by text. So it is established that set of valid configurations can be expressed as a logical formula.

The in this context most interesting property of this formula is the scaling of the number of valid configuration in regards 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, yet already looking at binary options gives a satisfying result. Thats because naturally the total number of possible and valid configurations lies in the exponential space of $\mathcal{O}(2^{\#options})$. In other words: a *configuration space* is an exponentially large. And thats also the reason why brute-force does not scale well.

4 Software Product Lines

As mentioned in the introduction modern software systems are often highly configurable and customizable. To be able to provide such a large amount of customization, software engineers adopted the concept of *mass customization*. A technique mostly known from the automotive industry [14]. It involves the building of a product line that supports the output of highly customizable products. The result of bringing *mass customization* into software development are Software Product Lines (SPL). SPLs are broadly found throughout software development and in all kinds of development environments [3, 8].

4.1 Introduction into Software Product Lines

Pohl et al. [14] describe how a general product-line functions: There is a *common platform*, which is in itself expandable and holds all basic functionalities. It also

provides interfaces for adding on parts (features). Some parts are mandatory and some are optional. For each part there might be different versions or variations that are interchangeable.
Applying the factor software engineering to this general product line leads one to the definition of SPLs by the Software Engineering Institute (SEI, `https://www.sei.cmu.edu/`): "[...] a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way."[5].

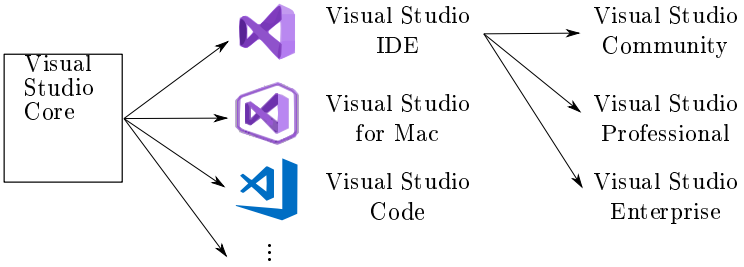


Fig. 4: Partial view of the Visual Studio Product Line [12]. (Assuming a Microsoft is using core libaries for its product. Which is likely since .Net Core runs on all major operating systems.)

In particular SPLs inherit the behavior of reusing existing components to enhance the development process [6]. A more detailed look at the economical thought behind SPLs is given by Díaz-Herrera et al. [6].
An example of a SPL would be the Visual Studio product line from Microsoft as shown in Figure 4. The common platform are some core libraries that provide shared code to all products of this line. The products themselves add platform or usage specific features (parts) to the core to offer a broad variation of products. The Visual Studio IDE even splits up further into 3 products that all serve a different purpose, but use same code base. Note that figure 1 only displays static variations of the product line. Almost all versions are additionally dynamically expendable via plug-ins and customizable via options and preferences.

4.2 Performance Influences of Feature Binding in SPLs

The following section is based on Combining Static and Dynamic Feature Binding in Software Product Lines by Rosenmüller et al. [15].
This paper will use the terms *feature* and *configuration* as they are defined in [18] "[...], where a feature is a stakeholder-visible behavior or characteristic of a program." and a "a specific set of features, [is] called a configuration".
When designing a SPL one has a choice for each feature to either bind it statically or dynamically. A statically bound feature is compiled directly into the binary of a program. Its always available to the environment when needed but uses up

binary space to do so. When dynamically binding a feature it is not a part of the core binary, yet a part of an external *feature library*. Once such a feature is needed the core loads the according library from an external source. This leads to the core binary being smaller but the size needed for each feature is increased by some meta-data (size, location, format, etc. of the feature library). Multiple features can be combined into one feature library for optimization. Feature libraries are also called *binding units*. Note that [15] uses the *decorator pattern* for the realization of dynamic feature binding.
To describe the effects of binding types we use the terms *functional* and *compositional overhead*. A functional overhead originates form features (or binding units) that are loaded inside a program yet never used. In turn a compositional overhead is found as meta data that is needed to describe dynamic behavior.
An abstract example: A program core consisting out of 3 features and uses up 2MB of (hard-drive) memory. We extract 1 feature that is rarely used into a dynamic link library with the size of 1.1MB. Afterwards our core is only 1MB large. We can already conclude that our original program had a functional overhead of 1MB by subtracting the old and the new binary size. When using the extracted feature our program uses up 2.1MB of memory. So we can say that our compositional overhead for extracting the feature is 0.1MB. That is the data needed to describe the location, format, etc. of the feature library.
Coming back to binding types we have the problem of finding an optimal combination of static and dynamic bindings. Rosenmüller et al. [15] suggest to prefer static binding if no dynamic extensibility is required and resources are not limited. A full static bound program has no compositional overhead and thus only requires additional (persistent and transient) memory. However programs that are more limited in their resources may need to be designed to minimize the overall overhead. This is basically the same task of finding an optimal configuration of a program. Techniques regarding the analysis and prediction will come up later in Section 5. The compositional overhead of binding units can be predicted partially since feature libaries have a constant size header (e.g. 5KB per Dynamic Link Library {DLL}). A general guideline is to avoid a large number and a large size of binding units. Overlapping binding units as well as dynamic splitting and merging of binding units are possible optimizations. Rosenmüller et al. [15] also mention that "[...] there is no optimal size for a binding unit, a domain expert can define binding units per application scenario.".

5 Measuring and Predicting the Performance of Highly Configurable Systems

By learning about the performance difference between multiple configurations it is possible accurately predict the a programs performance. This means that one does not need ot measure all (possibly exponentially many) configurations. Instead a small sample size should be enough to predict a programs performance. Multiple ways using diffrent methods have been proposed over time [13] This paper will take a look at

– Automated Feature Interaction Detection by Siegmund et al. [18],
– an incremental/statistical learning approach by Guo et al. [7],
– **WHAT** a spectral learning approach by Nair et al. [13],

5.1 Approaches to Performance Prediction

Before looking at actual prediction methods one should have a look at general possibilities for prediction methods. There are 2 general approaches to predicting a component-based system's performance [1].

First there is the **measurement-based approach**. A measurement-based approach uses an analysis tool to monitor an application during execution. Based on the measurements of an existing application a performance model is build and modified. This approach is highly dependent on existing software (e.g.: measured application, analysis tool, operating system). [1]

Secondly there is the **model-based approach**. A model-based approach relies on models created by Model Driven Development. It combines multiple models of a system to give a performance prediction. The advantage of this technique is that it does not require a system to exists. Therefore performance modelling can be done before a system is actually composed. Automation and usage profiles are used to improve the prediction accuracy of this method. Approaches that don't use automation are generally found to be reasoning tools rather than applicable approaches. Some approaches do not consider external factors such as external service calls, usage profiles or the execution environment at all. This decreases the accuracy of their predictions. [1]

Mixed approaches are also possible and can occur in many different variations. For example it is possible to parametrize a model based on measured values. [1]

This paper will mainly focus on measurement based approaches.

5.2 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. [18]. Unlike other methods it does not depend on machine learning but rather tires to directly identify the performance impact of each feature or a combination of features. This method reached a precision of up to 95% in the experiment conducted by Siegmund et al. [18].

Some **Formulars** are needed to descibe a Softwaresystem for AFID . The composition of using two (or more) units/features is denoted by $\cdot$. This composition is also called a configuration [7].

The interaction of two features is denoted by $a\#b$. By combining both we get a feature interaction:

$$a \times b = a\#b \cdot a \cdot b \tag{1}$$

This equation expresses, that when using both a and b we also need to consider their interaction $a\#b$. Note that either a or b can also be a configuration.
Further Sigmund uses an abstract performance function Π that is used to represent some performance value of a configuration:

$$\Pi(a \cdot b) = \Pi(a) + \Pi(b) \tag{2}$$
$$\Pi(a\#b) = \Pi(a \times b) - (\Pi(a) + \Pi(b)) \tag{3}$$
$$\Pi(a \times b) = \Pi(a\#b) + (\Pi(a) + \Pi(b)) \tag{4}$$

Following that the performance of a program $P = a \times b \times c$ can be written down as

$$\Pi(P) = \Pi(a) + \Pi(b) + \Pi(c) + \Pi(a\#b) + \Pi(a\#c) + \Pi(b\#c) + \Pi(a\#b\#c). \tag{5}$$

The Problem with the equations 2-5 is that they assumes that we can measure the performance of a feature in isolation. This is in general not possible [18]. Also we are still in the space of $\mathcal{O}(2^n)$ of possible configurations that we need to measure. To reduce this Sigmund et al. uses a interaction *delta*.

$$\begin{aligned} \Delta a_C &= \Pi(C \times a) - \Pi(C) \\ &= \Pi(a\#C) + \Pi(a) \end{aligned} \tag{6}$$

Where C is a base configuration. This formula describes how the performance influence (Δ) of a on a configuration C can be calculated. Its either the performance difference between using C with and without a, or the performance influence of a itself plus the influence of the interaction between C and a.
As a general approach to reduce its search space AFID looks at:

$$\Delta a_{min} = \Pi(a \times min(a)) - \Pi(min(a)) \tag{7}$$

and

$$\Delta a_{max} = \Pi(a \times max(a)) - \Pi(max(a)) \tag{8}$$

Where $min(a)$ is a valid minimal configuration not containing a but to which a can be added to create another valid configuration. $max(a)$ is a valid maximal configuration not containing a but to which a can be added to create another valid configuration.

For **AFID** one first needs to define when a feature is interacting. For this Siegmund et al. [18] use the definition of

$$a \text{ interacts} \Leftrightarrow \exists C, D | C \neq D \wedge \Delta a_C \neq \Delta a_D. \tag{9}$$

$C = min(a)$ and $D = max(a)$ are chosen to find interacting features and to reduce the search space for C or D from $\mathcal{O}(2^n)$ to $\mathcal{O}(n)$. By measuring $\Delta a_{min(a)} = \Delta a_C$ and $\Delta a_{max(a)} = \Delta a_D$ for each feature some first information about their behavior can be obtained. If both values for a feature a are similar it does not interact with the features of $max(a) \backslash min(a)$. Otherwise a is marked as interacting. In both cases it can still interact with the features of $min(a)$. In total 4 measurements per feature are required ($\Pi(a \times min(a))$, $\Pi(min(a))$, $\Pi(a \times max(a))$, $\Pi(max(a))$)[18].
Since most of the interacting features are known by now one can look for the groups of features whose interaction does have an influence on performance. Again the problem arises that there is an exponential number of possible combinations. Three heuristics are used to simplify the finding of these groups.

Pair-Wise Heuristik (PW):	Most groups of interacting features appear in the size of two[18, 11]. So it makes sense to look for pair interaction first.
Higher-Order Interactions Heuristic (HO):	Siegmund et al. [18] only look at higher order interactions of the rank of three. More on this later.
Hot-Spot Features (HS):	Based on [2, 19] Siegmund et al. [18] assume that hot spot features exist. "[...] There are usually a few features that interact with many features and there are many features that interact only with few features.", these features are the hot spot features.

Using a SAT-Solver an implication graph as seen in Figure 5 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. The three heuristics will be applied in the order of PW $\rightarrow$ HO $\rightarrow$HS.

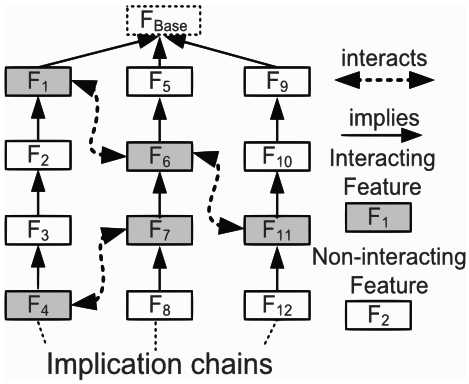


Fig. 5: Implication tree example found in [18]

First the influence of every feature on another chain is measured (PW-heuristic). In the example of Figure 5 the interactions would be measured in this order:"$F1\#F6, F1\#F7, F4\#F6, F4\#F7, F6\#F11, F7\#F11, F1\#F11, F4\#F11$"[18]. If an interaction impact $\Delta a \# b_C$ exceeds a threshold it is recorded.

Secondly, the higher order interaction heuristic is applied. Higher 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 are 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 the hotspot features further third order interactions are explored. Again higher order interactions are not considered to prevent too many measurements.

After applying the three heuristics all detected interacting features or feature combinations are assigned a Δ to represent their performance influence on the program.

Siegmund et al. [18] tested AFID on six different SPLs (Berkely DB C,Berkely DB Java, Apache, SQLite, LLVM, x264). 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 seen in Table 1. Using only the FW approach means that interactions (and the heuristics) are not considered, yet the accuracy is already at about 80% on average. A significant improvement can be made by using the PW heuristic. It uses on average 8.5 times more measurements than the FW approach but improves the accuracy to 91%. Using the HO or HS approach improves the accuracy further by about 2-4%. However for Apache using the HO over the PW approach even deteriorated the average result by 3.9% and doubled the standard variation. As already mentioned using the HS approach gives the best accuracy this is true for all 6 tested applications. Siegmund et al. [18] 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. [18]

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

5.3 Variability aware Performance Prediction

The following section is based upon [7].

Variability aware performance prediction is a model based approach to performance predictions. With the help of *Classication and Regression Trees* (short: CART, see [4]) and some configuration samples a statistical model of a program can be created. In thier own tests Guo et al. [7] reached an average precision of 94%.

The **basic idea** of variability aware performance prediction can be seen in Figure 6. Two cycles can be found.

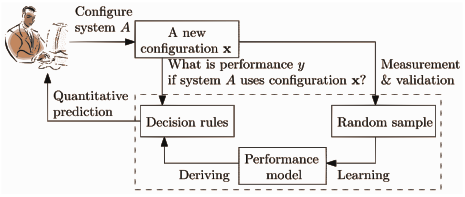


Fig. 6: Overview of the Approach [7]

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 x for System A and asks the predictor (dashed box) for a prediction. It replies with a quantitative prediction for x's performance.

In the second cycle a actual prediction is generated based on decision rules which themselves are created by simplifying a performance model. Random samples are used to learn the performance model.
Like other approach, the target of variability aware performance prediction is to get accurate predictions with only using a small amount of samples for the creation of the performance model.

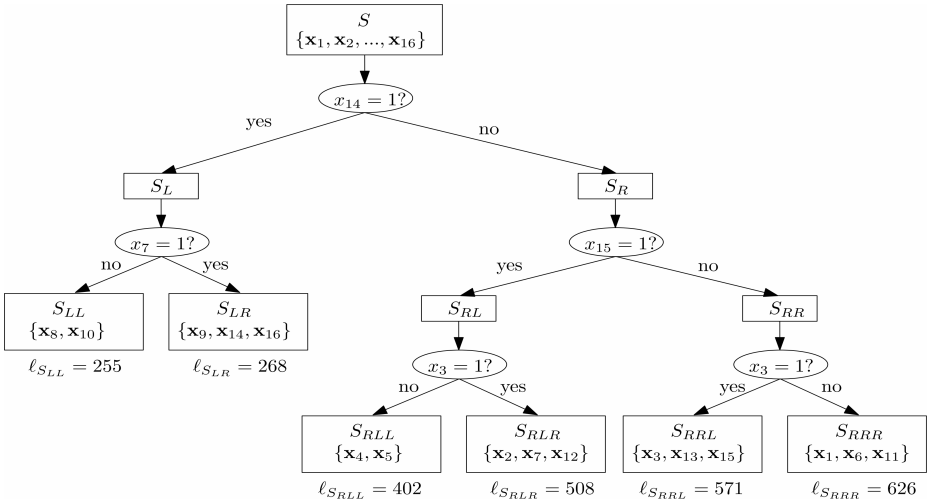


Fig. 7: Example performance model of X264 generated by CART based on the random sample [7].

Statistical Methods are used to perform the actual computation. First of all a configuration is defined as an N-tuple $(x_1, x_2, x_3, ..., x_N)$, where N is the number of all available features. Each x_i represents a feature and can either have the value 1 or 0 depending on whether the feature is selected or not. An actual configuration example would be $\mathbf{x}_j = (x_1 = 1, x_2 = 0, x_3 = 1, \ldots, x_N = 1)$. All valid configurations of a system are denoted as $\mathbf{X}$.
To each configuration $\mathbf{x}_j$ an actual performance value y_j can be assigned. Y denotes the performance of all configurations of a system.
Combining Y with $\mathbf{X}_S \subset \mathbf{X}$ gives a sample S. Now the two problems arise that variability aware performance prediction tries to solve:

1. Predict the performance of the not measured configurations $\hat{=} \mathbf{X} \backslash \mathbf{X}_S$.
2. "Given a sample S, the problem is to find a function f that reveals the correlation between $\mathbf{X}_S$ and Y_S and that makes each configuration's predicted performance $f(x)$ as close as possible to its actual performance y, i.e.:

$$f : \mathbf{X} \to \mathbb{R} \text{ such that } \sum_{\mathbf{x}, y \in S} L(y, f(\mathbf{x})) \text{ is minimal} \tag{10}$$

   where L is a loss function to penalize errors in prediction."[7]

This is done with the help of CART. All sample configurations get categorized into a binary trees leafs. A configurations selection of features determines its location in the tree. The distribution of samples inside the tree is determined with the goal of minimizing the total prediction errors per segment (sub-trees). An example tree can be found in Fig. 7. For each leaf one can determine the *local model* ℓ

$$\ell_{S_i} = \frac{1}{|S_i|} \sum_{y_j \in S_i} y_j \tag{11}$$

As a loss function to penalize the prediction errors Guo et al. [7] choose the sum of squared error loss:

$$\sum_{y_j \in S_i} L(y_i, \ell_{S_i}) = \sum_{y_j \in S_i} (y_j - \ell_{S_i})^2 \tag{12}$$

Therefore the best split for a segment S_i is found when

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

is minimal. To prevent *under-* or *overfitting*[9] the recursive splitting has to be stopped at the right time. This is possible by manual parameter tuning or using a empirical-determined automatic terminator.
Now to the actual calculation of the quantitative prediction. Assuming there are q leafs in our tree than $f(\mathrm{x})$ is defined as:

$$f(\mathrm{x}) = \sum_{i=1}^{q} \ell_{S_i} I(\mathrm{x} \in S_i) \tag{13}$$

where $I(\mathrm{x} \in S_i)$ is an indicator function to indicates that x belongs to a leaf S_i. For the example of Figure 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. [7] test the same software systems as Siegmund et al. [18] (Section 5.2). They also compare their prediction results with the results produced by SPLConquerer under *AFID*.
Unlike *AFID* the size of a sample for variability aware performance prediction can be chosen freely. Guo et al. [7] use 4 different sample sizes based on the size of the program. 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 SPLConquerer's using the PW heuristic. It is found that the prediction accuracy increases linear with the size of the sample. It is also found that for using a small sample with the size of N the prediction accuracy was at 92%. However for Berkeley DB (C) the prediction rate with N sized samples was at $112.4 \pm 354.6\%$[1]. This results into an average accuracy of only $28.6 \pm 68.9\%$. Using a sample size of M significantly improves the average prediction accuracy to 93.8%.
Further Guo et al. [7] comparer their approach with *AFID*. This can be done since N also equals the amount of configurations measured by the FW heuristic. As already established variability aware performance prediction is not accurate for small sample sizes so it is no surprise that *AFID* with the FW heuristic performace better at $20.3 \pm 21.2\%$ with a sample size of N. However, when using samples of size M SPLConquerer's PW heuristic only reaches an average precision of 90.9% compared to the already mentioned 93.9% of Guo et al. [7]'s approach. Using the HO or HS heuristic of *AFID*can produce a precision of up to 95% but requires more measurements. This is not covered by [7].

5.4 WHAT

WHAT is a spectral learning approach developt by Nair et al. [13]. It 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 mathematical concept of *eigenvalues/-vectors* of a distance matrix between configurations. This has the advantage of automatic noise reduction as Nair et al. [13] explain: "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."
The main advantage of this approach are a reduced sample size needed and a lower standard deviation compared to previously shown methods [13]. Nair et al. [13] divide **WHAT** into 3 parts.

1. Spectral Learning

This first step is used to cluster all valid configurations. Every configuration is an element of the feature space F. This section uses the same definition for a *configuration* as Section 5.3 [2]. As each configuration is an n-dimensional vector

[1] ± 354.6 indicates the standard deviation
[2] This definition can also be expanded to cover non binary options like a programs stack size. For this $x_i \in \mathbb{R}$ has to be chosen.

(or n-tupel) 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 'most diffrent' means most farthest away. After that a straight through $East$ and $West$ is calculated and all configuration are dived into two cluster by their distance to this line. This process is recursively repeated for each sub-cluster until they reach a threshold size. Nair et al. [13] use the $\sqrt{|N|}$ as their termination value. We end up with leaf clusters that contain configurations which are similar in their chosen feature options. This process runs in linear time[13].

2. Spectral Sampling

For the actual sampling a probabilistic strategy is applied: One configuration randomly picked from each leaf cluster is compiled and executed.
There are also two other sampling strategies mentioned that will get outperformed by the probabilistic strategy.

3. Regression-Tree Learning

This step is similar to Section 5.3. A CART is build from the chosen samples. This time the best split is defined as reaching the minimum of $\frac{A}{N}\sigma_1 + \frac{B}{N}\sigma_2$,[3]. From this CART 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.[13]

6 Conclusion

Nair et al. [13] provide a good overview over the presented approches. It can be found in Fig. 8. Event though Sarkar's approach (which was not covered by 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 ([18]) 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 ([7]) 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

[3] The paper ([13]) defines A and B as sets and N as a (natural) number. So it may is to assume that the formular 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 N.

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. 8: 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." [13]

References

[1] A. A. Abdelaziz, W. M. N, W. Kadir, and A. Osman. Comparative analysis of software performance prediction approaches in context of component-based system. *IJCA*, pages 15–22, 2011.

[2] 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.

[3] J. Bergey, G. Chastek, S. Cohen, P. Donohoe, L. Jones, and L. Northrop. Software product lines: Report of the 2010 u.s. army software product line workshop. Technical Report CMU/SEI-2010-TR-014, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2010.

[4] L. Breiman. *Classification and Regression Trees*. Routledge, New York, 1984.

[5] P. Clements and L. Northrop. Salion, inc.: A software product line case study. Technical Report CMU/SEI-2002-TR-038, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2002.

[6] J. L. Díaz-Herrera, P. Knauber, and G. Succi. Issues and models in software product lines. *International Journal of Software Engineering and Knowledge Engineering*, 10(4):527–539, 2000.

[7] 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.

[8] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid. Dynamic software product lines. *Computer*, 41(4):93–95, April 2008.

[9] 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.

[10] 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.

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

[12] Microsoft. `https://visualstudio.microsoft.com/de/products/`, Accessed: 10.05.2019.

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

[14] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag Berlin Heidelberg, Tiergartenstraße 17, 69121 Heidelberg, 1 edition, 2005.

[15] M. Rosenmüller, N. Siegmund, G. Saake, and S. Apel. Combining static and dynamic feature binding in software product lines. technical report 13, 2009.

[16] 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.

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

[18] 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.

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

[20] 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.