Predicting and Learning the Performance of Configurable Software Systems

Lion Wagner

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

Abstract. This is my abstract.

1 Introduction

Todays programs are mostly highly configurable. Whilst buying or downloading a program a user already decides on which version of program (s)he wants to have. On Installation there are mostly multiple options reaching from 'Language' to a selection of software features. Once a program is installed there are usually multiple startup, customization and configuration options provided for a customer. But having such a large amount of options brings some problems into the development of modern applications. This paper will manly focus on the problem of learning and predicting the performance of a highly configurable software system. A developer needs some sort of mechanism to ensure his application has the required performance under most or all configurations. On one hand this is important when it comes to contract terms and conditions with customers. A program should perform as good as the customer requires it to [6]. Otherwise a developer may face fines. On the other hand performance prediction is helpful in finding performance problems or room for improvement. This either helps with the previous point of reaching a set performance target or to make the program more user friendly (less response time, smaller binary size, ...) and therefore more attractive [11].

First all a stable development strategy is needed. Preferably it can offer the creation of multiple similar products with little redundant code or components whilst providing methods to optimize thier performance. In practice software product lines (SPL) are used for this case. This paper will first have a short look at the importance of SPLs in performance engineering.

Furthermore there is the problem of performance prediction. Why do we need to learn and predict the performance of a system? The amount of possible configurations naturally lies in $\mathcal{O}(2^n)$. This scaling makes it hard to test each singular configuration for its performance or correctness. Especially if the configuration under test is unpredictably chosen by a user [**?**] or if feature-interactions impair the performance [6]. For example: Berkley DB (C) is a database management program for embedded systems. It has 18 features and 2560 different configurations. In their paper "*Predicting Performance via Automated Feature-Interaction*

Detection" Siegmund et al. [16] measured each of the possible configurations and it took them 426h (=17,75d)[1]. This and other examples from the same paper show, that it is not practical to brute-force measure each and every configuration possible.
Consequently when one wants to efficiently analysing a software product (line), its performance can only be partially measured and behaviour beyond that has to be predicted. Over time a lot of methods in different disciplines have been prosed.
This paper aims to give a short introduction into the importance of software product lines (SPL) in regards to performance engineering and an overview over solutions for predicting and learning the performance of a highly configurable software system. mention references

TODO

2 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 [2, 7].

2.1 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."[4].
In particular SPLs inherit the behavior of reusing existing components to enhance the development process [5]. A more detailed look at the economical thought behind SPLs is given by Díaz-Herrera et al. [5].
An example of a SPL would be the Visual Studio product line from Microsoft as shown in Figure 1. The common platform are some core libraries that provide

[1] These measurements were done on computers that are (from today's point of view) fairly slow [16, 18].

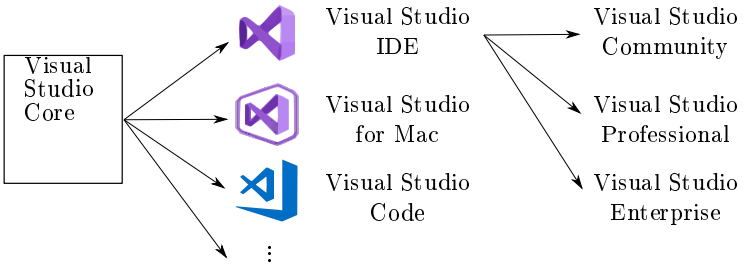

Fig. 1: Partial view of the Visual Studio Product Line [12]. (Assuming a Microsoft is using core libaries for its product. Which is likely since .Net Core runs on all major operating systems.)

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.

2.2 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 [16] "[...], 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.

TODO daten aus paper einfügen

TODO Fig 18 einbauen 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 3. 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.".

3 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. [16],
- an incremental/statistical learning approach by Guo et al. [6],
- **WHAT** a spectral learning approach by Nair et al. [13],
- and Transfer Learning by Jamshidi et al. [9].

3.1 Automated Feature Interaction Detection

Automated feature interaction detection (AFID) is the most straight forward approach to predicting the performance of a highly configurable system. It was developed by Siegmund et al. [16]. 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. [16].

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 [6].
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 [16]. 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. [16] 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 ($\Pi(a \times min(a))$, $\Pi(min(a))$, $\Pi(a \times max(a))$, $\Pi(max(a))$)[16].
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[16, 10]. So it makes sense to look for pair interaction first.

Higher-Order Interactions Heuristic (HO): Siegmund et al. [16] only look at higher order interactions of the rank of three. More on this later.

Hot-Spot Features (HS): Based on [1, 17] Siegmund et al. [16] 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 2 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.

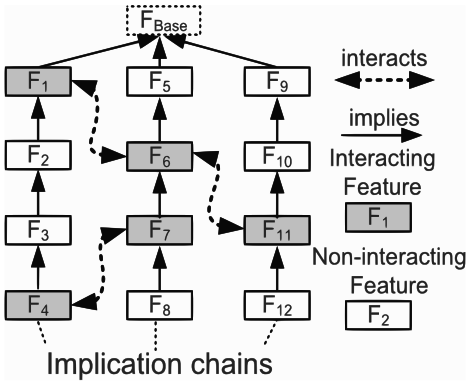

Fig. 2: Implication tree example found in [16]

First the influence of every feature on another chain is measured (PW-heuristic). In the example of Figure 2 the interactions would be measured in this order:"$F1\#F6, F1\#F7, F4\#F6, F4\#F7, F6\#F11, F7\#F11, F1\#F11, F4\#F11$"[16]. 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. [16] 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. [16] 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. [16]

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

3.2 Variability aware Performance Prediction

The following section is based upon [6].

Variability aware performance prediction is a model based approach to performance predictions. With the help of *Classication and Regression Trees* (short: CART, see [3]) and some configuration samples a statistical model of a program can be created. In thier own tests Guo et al. [6] reached an average precision of 94%.

The **basic idea** of variability aware performance prediction can be seen in Figure 3. Two cycles can be found.

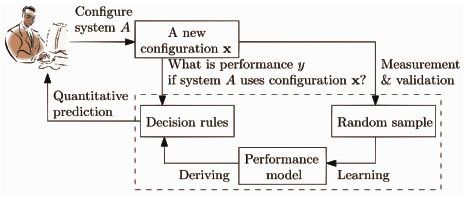

Fig. 3: Overview of the Approach [6]

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.

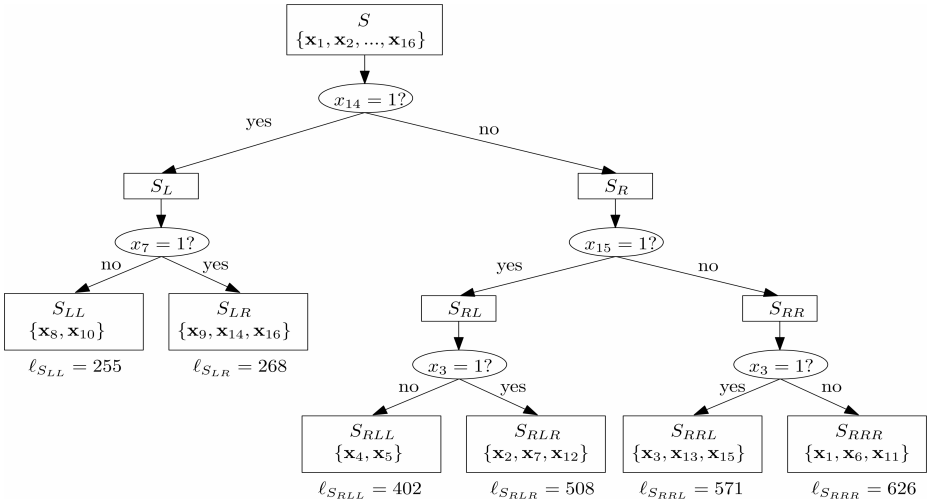

Fig. 4: Example performance model of X264 generated by CART based on the random sample [6].

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."[6]

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 by CART with the goal of minimizing the total prediction errors per segment (subtrees). An example tree can be found in Fig. 4. 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. [6] 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*[8] 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 4 $f(\mathrm{x})$ unwraps to:

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

3.3 WHAT

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. doi: 10.1145/1985793.1985851. URL `http://doi.acm.org/10.1145/1985793.1985851`.

[2] 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. URL `http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=9495`.

[3] L. Breiman. *Classification and Regression Trees*. Routledge, New York, 1984. ISBN 978-1-31513-947-0. doi: 10.1201/9781315139470.

[4] 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. URL `http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=6285`.

[5] 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. doi: 10.1142/S0218194000000286. URL `https://doi.org/10.1142/S0218194000000286`.

[6] 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. ISBN 978-1-4799-0215-6. doi: http://dx.doi.org/10.1109/ASE.2013.6693089.

[7] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid. Dynamic software product lines. *Computer*, 41(4):93–95, April 2008. ISSN 0018-9162. doi: 10.1109/MC.2008.123.

[8] 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. doi: 10.1007/978-0-387-84858-7.

[9] P. Jamshidi, M. Velez, C. Kästner, N. Siegmund, and P. Kawthekar. Transfer learning for improving model predictions in highly configurable software. *CoRR*, abs/1704.00234, 2017. URL `http://arxiv.org/abs/1704.00234`.

[10] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An analysis of the variability in forty preprocessor-based software product lines, 2010. URL `http://doi.acm.org/10.1145/1806799.1806819`.

[11] J. Ludewig and H. Lichter. *Software Engineering Grundlagen, Menschen, Prozesse, Techniken*, volume 3., korrigierte Auflage. dpunkt.verlag, Ringstraße 19 B 69115 Heidelberg, 3 edition, 2013. ISBN 978-3-86490-092-1.

[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. URL `http://arxiv.org/abs/1701.08106`.

[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. ISBN 3540243720. doi: 10.1007/3-540-28901-1.

[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] 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.

[17] C. Taube-Schock, R. Walker, and I. Witten. Can we avoid high coupling? pages 204–228, 07 2011. doi: 10.1007/978-3-642-22655-7_10.

[18] techpowerup. `https://www.techpowerup.com/cpudb/`, Accessed: 17.05.2019.