The Interplay of Sampling and Machine Learning for Software Performance Prediction

Christian Kaltenecker
Saarland University
kaltenec@cs.uni-saarland.de

Alexander Grebhahn
adesso SE
Alexander.Grebhahn@adesso.de

Norbert Siegmund
University of Weimar
norbert.siegmund@informatik.uni-leipzig.de

Sven Apel
Saarland University
apel@cs.uni-saarland.de

***Abstract*—Artificial intelligence has gained considerable momentum in software engineering. One success story is the use of machine learning for performance prediction and optimization of configurable software systems, for which it is often intractable to determine which configuration is optimal. Despite notable success stories, there are major challenges that make this domain special such that "just applying machine-learning techniques out of the box" is infeasible: First, the configuration spaces of real-world software systems are often huge and *highly constrained*. Second, since sampling is such an intricate problem for software configuration spaces, the *sampling strategy* has substantial influence on the performance of the *machine-learning algorithm*, so it is imperative to reason about the *interplay* of sampling and learning. In this article, we review recent advancements in this area, raise awareness of the distinctiveness of software configuration spaces, and provide practical guidelines for modeling, predicting, and optimizing their performance.**

A PROMISE of artificial intelligence is the automation of the design, implementation, optimization, and configuration of efficient and provably correct software. An area that has made considerable progress in adopting artificial intelligence techniques is *highly-configurable software systems*.

Many software systems today are configurable. A *configurable system* provides a number of configuration options that allow programmers, administrators, and users to adapt the system to their functional and non-functional requirements. The benefits of configurability come with a price, though: The sheer size of the configuration space is often overwhelming [1]. For example, the LINUX kernel has over 13 000 configuration options, which span a configuration space whose size cannot even be quantified in reasonable time.

  Published by the IEEE Computer Society

But it is not only the plain size that is challenging. Configuration options may *depend* on other options. For example, the support of multi-user access to a database may require transaction management and specific index structures. Such dependencies *constrain* the set of configurations that arises from combinatorics to a subset of *valid* configurations. For instance, the VP9 video encoder offers 42 configuration options, which give rise to 2^{42} ($\sim 10^{12}$) of combinations, but only 216 000 configurations are valid (i.e., 0.000021 % of all combinations). In larger systems such as the LINUX kernel with more than 300 000 constraints, determining the exact set of valid configurations is notoriously hard.

In any case, knowing and even guaranteeing certain properties, such as performance or energy consumption, for all these configurations is impossible. For example, identifying the optimal configuration of the database system SQLITE for a given workload would require more time than the universe has existed so far. Worse, targeting not only a single but multiple properties (e.g., performance *and* energy consumption) is even harder [2], [3]. Only recently, researchers began to use *machine learning* to address this problem. The key idea is (1) to select and measure a small number of configurations—the *sample* or *learning set*—and (2) to learn a *influence model* based on the sample set. Depending on the learning algorithm, the model can be a neural network [4], regression tree [5], or linear influence model [6], among others. The influence model allows one then to predict the performance (or any other measurable property) of any other configuration [6]. Internally, the influence model captures and combines the influences of all individual configuration options and the relevant influences of their interactions. So, an influence model is not only useful for prediction and optimization but also for *understanding* the external behavior and the inner workings of the system [7].

But here is the caveat: *Which configurations should we sample?* And: *What learning technique works best with the selected samples?* As it turns out, highly-configurable software systems hold several challenges that makes "just applying machine-learning techniques out of the box" impossible. First, due to the *high constrainedness* of software configuration spaces, simple *random sampling* as proposed and used in any machine-learning tutorial or book is challenging. Obtaining a uniformly distributed random sample in the presence of constraints is even known to be computationally intractable [8]. Only recently, researchers began to address the sampling problem with dedicated *sampling strategies* [9], [10], [11]. Beyond random sampling, researchers developed a number of sampling strategies that aim at certain forms of coverage, including *t-wise sampling* (covering all tuples of configuration options' values of size t) and *code coverage sampling* (selecting, at least, every single line of source code once).

The multitude of proposed sampling strategies is a testament to the *distinctiveness of the domain of configurable systems*, which precludes the use of off-the-shelf machine-learning toolkits (e.g., for random sampling). Even more, the choice of the sampling strategy likely influences the performance of the machine-learning procedure. Different machine-learning techniques have different strategies to generalize from a given sample set and to deal with noise and interaction effects, which likely depends on decisions made by the sampling strategy. For example, if a sampling strategy missed configurations with a certain option or combinations of options enabled, the machine-learning procedure cannot possibly learn the corresponding influence, and so its effect is absent in the resulting influence model. Only very recently, researchers—including the author team—began to study and understand the delicate relationship between sampling and learning technique in the domain of highly-configurable software systems. In this article, we share our insights and experience with this issue and provide an overview and practical guidelines for software developers and administrators to model, predict, and optimize the performance of their configurable software systems.

Learning Influence Models

In Figure 1, we illustrate the overall process of learning influence models for configurable software systems. A sampling strategy is used to select (Step I) and measure (Step II) a tractable subset of configurations, which is then used to learn an influence model (Step III). The influence model can then be used to predict the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/MS.2020.2987024, IEEE Software

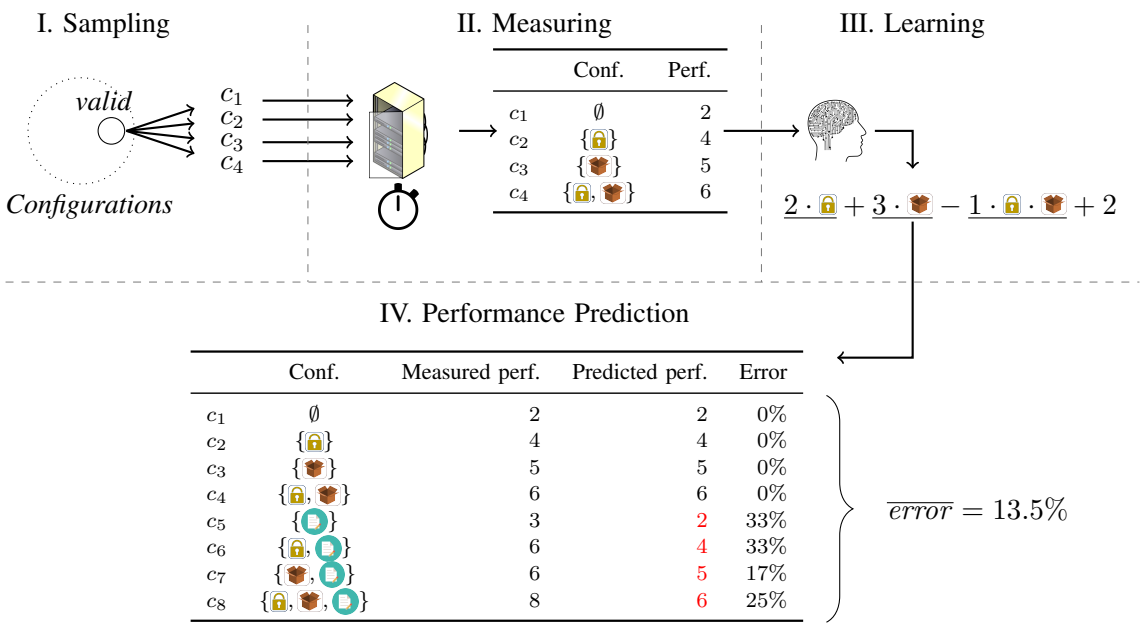

Figure 1. Predicting the performance of configurations: The subject is a configurable software system with the configuration options ENCRYPTION (🔒), COMPRESSION (📦), and LOGGING (🛢). The steps for performance prediction of configurations are *sampling* (I), *measuring* (II), *learning* (III), and *performance prediction* (IV). In Step I, a sampling strategy selects a tractable number of configurations. These configurations are measured in Step II. In Step III, a machine-learning algorithm is used to learn an influence model; terms representing influences of options and interactions among options are underlined. The learned influence model can be used in Step IV to predict the performance of any configuration. This way, one can compare the predicted performance to the measured performance and compute the prediction or model error.

performance of any configuration (Step IV). In our example, the influence model captures the *relevant* influences of configuration options and interactions on performance as a linear combination of the respective terms. An influence is *relevant* if it increases the predictive power of the model on configurations of the sample set. The influence model has four components:

- $2 \cdot$ 🔒 represents the influence of option ENCRYPTION,
- $3 \cdot$ 📦 represents the influence of option COMPRESSION,
- $-1 \cdot$ 🔒 $\cdot$ 📦 represents the influence that arises from the interaction of ENCRYPTION and COMPRESSION, and
- 2 represents the base performance (the *intercept*).

The accuracy of the model depends on two factors: (1) the sample set (Step I) and (2) the machine-learning technique (Step III). Note that other representations for the influence model (e.g., regression trees) are possible and may even result in improved accuracy [5]. However, an important aspect for software engineers is *interpretability* of the model, which is why we resort to the simple, but yet reasonably expressive form of a linear, multi-variable regression model.

Let us start with a closer look at the sampling step. The goal of the sampling step is to select a sample set such that all *relevant* influences are covered. If an important option or interaction among multiple options is not present in the sample set (for example, the interaction among ENCRYPTION and COMPRESSION in our example), the learning step cannot possibly uncover it and include it into the influence model.

The Sampling Problem

Researchers and practitioners have developed a number of strategies to sample the configuration space of a given configurable system. Although incomplete (i.e., it selects only a subset of all

configurations), sampling is the state of the art in practice. For example, in Linux, code coverage sampling is used to create a sample set that covers all lines of source code [12]. Achieving completeness by exploiting parallelism to measure all configurations is infeasible, since the number of configurations grows exponentially with the number of options.

A prominent sampling strategy among researchers and practitioners is *random sampling*. The idea is to select configurations randomly from the configuration space in an unbiased way. More precisely, the probability of selecting a configuration c from configuration space C should be $\frac{1}{|C|}$. Without any knowledge on relevant influences and interaction effects, random sampling is a reasonable choice. Nevertheless, random sampling may miss important information—there is no guarantee that a certain configuration option or interaction is covered in the sample set, which becomes more prevalent when the number of samples is very low compared to the size of the configuration space. Furthermore, obtaining an unbiased random sample is computationally hard. This is due to possible constraints among configuration options. While there has been progress in this direction using binary decision diagrams or satisfiability solvers [13], [14], existing solutions are not ready for industrial adoption yet.

A strategy that is more systematic is *t-wise sampling*. The idea is to select the sample set such that it covers *all* interactions among *all* combinations of t options. *Pair-wise sampling* ($t = 2$) is most popular. It ensures that all *pairs* of configuration options are present (i.e., enabled), at least, once in the sample set. *Option-wise sampling* ($t = 1$) ensures that each individual option is selected, at least, once. Clearly, the larger t is, the more possible interactions we can catch, but the larger the sample set grows. Let us illustrate this fundamental tradeoff with an example—a function that contains preprocessor directives to realize configurability:

```
1
2  void encrypt() {
3  #ifdef ENC
4     prepareData();          // Block 1: ENC
5     #ifdef COMP
6        compressData();      // Block 2: ENC ∧ COMP
7     #endif
8     encryptData();          // Block 3: ENC
9  #endif
10 }
11
```

This example illustrates the interplay of the options ENCRYPTION and COMPRESSION of Figure 1 at code level. Function `encrypt` uses two macros, `ENC` and `COMP`, controlling the inclusion of configuration-dependent code. Block 1 and 3 are included only in configurations that have `ENC` selected. Block 2 is included only if both `ENC` and `COMP` are selected. It can be seen at the code level that, when data are compressed, less data have to be encrypted, which speeds up the encryption process. Option-wise sampling would select only Block 1 and 3, not Block 2, missing the speedup due to the interaction of the two options. Pair-wise sampling would select an additional configuration that includes Block 2, at the cost of a larger sample set, though.

While t-wise sampling is systematic and parametric, it comes with its own challenges. First, computing a sample set even for $t \geq 2$ is computationally expensive and even infeasible for configuration spaces of the size of the LINUX kernel [15]. Second, stoically including all pairs (or triples, quadruples, etc.) may be unnecessarily expensive since practice has shown that not all interactions among options are relevant—actually only few are [7]. To address these challenges, hybrid sampling strategies have been proposed, which combine an element of randomness with coverage criteria such that certain combinations of interactions or certain parts of the code are selected [10], [16], [12]. Sampling strategies that are dedicated to numeric configuration options have been proposed as well [6].

The multitude of possible sampling strategies distinguishes the domain of highly configurable systems from other areas, such as image classification, where machine learning flourishes. The reason is that there are typically many, intricate constraints among configuration options and that obtaining and measuring a sample set is computationally very expensive (which is different from image classification or speech recognition). Practitioners are advised to take a closer look at the sampling strategies mentioned here and beyond to create optimal sample sets for their problems at hand.

Table 1. Sampling strategies considered in the empirical study of Grebhahn et al. [17]. We distinguish between binary and numeric as well as structured and unstructured sampling.

Sampling strategy	Abbreviation	Numeric	Structured
Option-wise	OW	✗	✓
Negative Option-wise	NegOW	✗	✓
t-wise ($t \in \{2, 3\}$)	T2, T3	✗	✓
Random (Binary), three sizes (OW, T2, T3)	RB-OW, RB-T2, RB-T3	✗	✗
One Factor at a Time	OFAT	✓	✓
Box-Behnken Design	BBD	✓	✓
Central Composite Inscribed Design	CCI	✓	✓
Plackett-Burman Design	PBD	✓	✓
D-Optimal Design	DOD	✓	✓
Random (Numeric)	RN	✓	✗

Table 2. Machine-learning algorithms considered in the empirical study of Grebhahn et al. [17].

Learning algorithm	Abbreviation	Category
Classification and Regression Trees	CART	Decision tree
k-Nearest-Neighbors Regression	kNN	Regression method
Kernel-Ridge Regression	KRR	Regression method
Multiple Regression	MR	Regression method
Random Forest	RF	Multiple decision trees
Support Vector Regression	SVR	Regression method

The Interplay of Sampling and Learning

Clearly, the choice of the sampling strategy affects the learning algorithm—what information it can extract from the sample set to be included in the influence model. As mentioned previously, the variety of sampling strategies for highly configurable systems is unique. This has immediate implications for the learning step. The question is *whether* and *to what extent* the choice of the sampling strategy affects the ability of different learning algorithms to obtain accurate models. The canonical literature on machine learning and predictive modeling says not much about this issue. A recent empirical study conducted by Grebhahn et al. [17] (the authors) provides first insights.

In this study, we have analyzed the dependencies between 13 sampling strategies (see Table 1) and 6 learning algorithms (see Table 2) on 6 subject systems (see Table 3) in terms of prediction accuracy, stability, and measurement effort using SPL Conqueror [18]. We paid special attention to the fact that different learning algorithms provide different *hyper parameters*. As hyper parameters affect efficiency and accuracy of the learning procedure and even depend on the selected sampling strategy, we included an extension hyper-parameter optimization step, to ensure a fair comparison. For a detailed description, we refer to Grebhahn et al. [17].

In Figure 2, we summarize the results in the form of nested matrix plots. The outer plot shows the error rates achieved by different machine-learning algorithms. The inner plots show the different sampling strategies that have been used in combination with the respective learning algorithm, divided by binary (x axis) and numeric sampling strategies (y axis). Note that the subject systems contain beside binary configuration options also numeric configuration options, such as page size or number of threads, which require dedicated numeric sampling strategies [6].

Note that we distinguish between plots that are on the diagonal of the top-level matrix and plots that are not: Each plot on the diagonal (gray-scale) compares pairs of sampling strategies given a specific learning algorithm. The lighter the gray tone, the higher the accuracy (the lower the error rate) of resulting influence model. For example, in the upper left plot of Figure 2, we *show* the error rates for Classification and Regression Trees (CART) when used with different combinations of sampling strategies. Apparently, the combination to 3-wise sampling and D-Optimal Design (DOD) performs best (lightest tone of gray).

The plots beyond the diagonal *compare* pairs of learning algorithms when combined with the different sampling strategies. Shades of green indicate that the learning algorithm in the row is more accurate than the one in the column (the

Table 3. Subject systems of the empirical study of Grebhahn et al. [17].

Subject system	Application domain	# Configuration options	# Configurations
DUNE MGS	Image processing framework	11	2 304
POLLY	Code optimizer (plugin for LLVM)	19	59 592
HSMGP	Multigrid solver for differential equations	14	3 456
JAVAGC	JAVA garbage collector	11	193 536
TRIMESH	Scientific code library	13	239 360
VP9	Video encoder	20	216 000

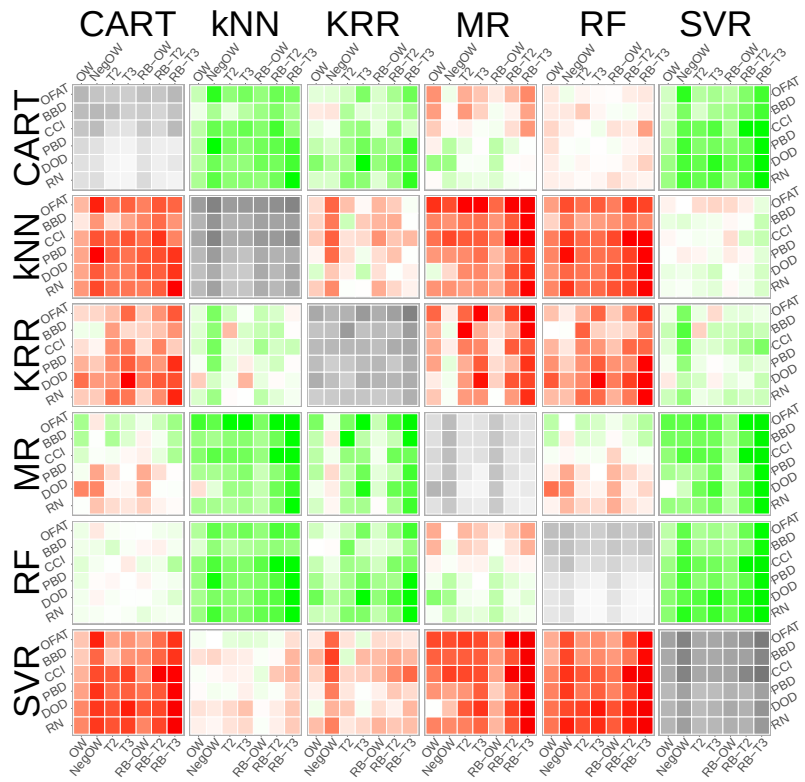

Figure 2. Comparison of combinations of machine-learning algorithms and sampling strategies in terms of predictions accuracy. Plots on the diagonal compare pairs of sampling strategies; the lighter the gray tone, the higher the prediction accuracy. Plots beyond the diagonal compare pairs of learning algorithms; shades of green (red) indicate that the learning algorithm in the row is more (less) accurate than the one in the column.

more intense the green, the larger the effect), and shades of red otherwise. For example, in upper right plot, we compare the error rates achieved when using CART with the error rates of using Support Vector Regression (SVR). It is easy to see that CART outperforms SVR irrespective of the used sampling strategy.

Let us have a closer look at the results. First, let us focus on the machine-learning algorithms independently of the sampling strategy. Not surprisingly, the choice of the algorithm matters and has a significant effect on prediction accuracy. Although a clear ranking is not readily apparent, random forests (RF) outperform the other learning algorithms in most of the cases; Multiple Regression (MR) and CART also perform very well.

Now let us look closer at combinations of sampling strategies and learning algorithms. Despite the rather clear picture that we obtained for learning algorithms, there seems to be no combination that is clearly superior in all cases. Although there is a trend to a specific combination of learning algorithm and sampling strategy, this does not hold for all cases. In many cases, random forests or multiple regression in combination with

option-wise or random sampling perform best, but there are exceptions.

A further notable result, which is not directly visible in Figure 2, is that, despite the rather small sample sets selected by option-wise (e.g., 72 configurations for VP9), random forests and multiple regression are still able to learn comparatively accurate influence models. Increasing the size of the sample set (e.g., from 350 for option-wise to 3 780 for 3-wise for VP9) often increases the prediction accuracy only marginally (e.g., by only 1% for VP9). The paradigm "the more, the better" holds for the domain of configurable software systems only when the more configurations also provide new information (e.g., influences of interactions not seen before). However, due to the possibly exponential number of interactions with respect to the number of configuration options, it is unclear which additional configurations improve accuracy.

Conclusion and Perspectives

Machine learning has proved enormously useful for performance prediction and optimization of configurable software systems. This is a success story par excellence at the intersection between artificial intelligence and software engineering. Nevertheless, this success story shows also that off-the-shelf machine learning approaches alone do not necessarily succeed. The domain of highly configurable software systems is special in various ways, including that there are substantial constraints among configuration options and that data for learning are scarce (measuring a configuration of a real-world software system is not cheap, after all). Researchers and practitioners reacted with the development of tailored sampling strategies, which in turn influence the performance of machine-learning algorithms. A main goal of this article was to raise awareness of this distinctiveness and to provide insights into fundamental tradeoffs of selecting sampling strategies and machine-learning techniques for performance prediction. Recent experiments that we summarized in this article demonstrate that these tradeoffs have indeed a practically relevant effect on prediction accuracy and measurement cost. Our results provide guidance to select a proper combination.

There are several open issues in this area that deserve more attention. While we were able to demonstrate the dependency between sampling strategy and learning algorithm for performance prediction, it is unclear for which application domains (scientific computing, embedded systems, databases, or even software engineering in general) it is relevant. Worse, we do not know yet whether there are certain combinations of sampling strategies and learning algorithms that are universally (or, at least, most of the time) better than others. While we are aware of the "no free lunch theorem" in machine learning [19] (i.e., if an algorithm performs well on a certain class of problems, then it necessarily pays for that with degraded performance on the set of all remaining problems [19].), for performance prediction and optimization of configurable software systems, we found a certain homogeneity among problem instances for which a single (or few) learning algorithm(s) performs well. Fully characterizing these problem instances (i.e., configuration spaces) is a promising research direction.

Another major issue is that different combinations of sampling strategies and learning algorithms may not only differ in accuracy but also in *interpretability*. After all, a key goal of influence models is not only prediction, but also *comprehension* [7]. Developers, administrators, and users would like to know *why* a certain configuration is fast, not only *whether* it is faster than another. Exploring the influence of sampling and learning on interpretability of influence models is clearly a rich and promising avenue of further work.

Finally, it would be interesting to repeat our experiments in other areas of software engineering, such as defect prediction or code recommendation. Our study provides a blueprint for this endeavor.

Acknowledgements

This work was supported by the German Research Foundation (SI 2171/2, SI 2171/3-1, and AP 206/11-1).

REFERENCES

1. T. Xu, L. Jin, X. Fan, Y. Zhou, S. Pasupathy, and R. Talwadker, "Hey, You Have Given Me Too Many Knobs!: Understanding and Dealing with Over-Designed Configuration in System Software," in *Proceedings of the Joint Meeting of the European Software Engineering*

Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE). ACM, 2015, pp. 307–319.

2. J. Chen, V. Nair, R. Krishna, and T. Menzies, “”Sampling” as a Baseline Optimizer for Search-Based Software Engineering,” *IEEE Transactions on Software Engineering (TSE)*, vol. 45, no. 6, pp. 597–614, 2019.
3. V. Nair, Z. Yu, T. Menzies, N. Siegmund, and S. Apel, “Finding Faster Configurations using FLASH,” *Transactions on Software Engineering (TSE)*, 2018, online first.
4. H. Ha and H. Zhang, “DeepPerf: Performance Prediction for Configurable Software with Deep Sparse Neural Network,” in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1095–1106.
5. A. Sarkar, J. Guo, N. Siegmund, S. Apel, and K. Czarnecki, “Cost-Efficient Sampling for Performance Prediction of Configurable Systems (T),” in *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 342–352.
6. N. Siegmund, A. Grebhahn, S. Apel, and C. Kästner, “Performance-Influence Models for Highly Configurable Systems,” in *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2015, pp. 284–294.
7. S. S. Kolesnikov, N. Siegmund, C. Kästner, A. Grebhahn, and S. Apel, “Tradeoffs in Modeling Performance of Highly-Configurable Software Systems,” *Software and System Modeling (SoSym)*, vol. 18, no. 3, pp. 2265–2283, 2019.
8. S. Chakraborty, D. J. Fremont, K. S. Meel, S. A. Seshia, and M. Y. Vardi, “Distribution-Aware Sampling and Weighted Model Counting for SAT,” in *Proceedings of the AAAI Conference on Artificial Intelligence*. AAAI Press, 2014, pp. 1722–1730.
9. F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel, “A Comparison of 10 Sampling Algorithms for Configurable Systems,” in *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 2016, pp. 643–654.
10. C. Kaltenecker, A. Grebhahn, N. Siegmund, J. Guo, and S. Apel, “Distance-Based Sampling of Software Configuration Spaces,” in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1084–1094.
11. M. Varshosaz, M. Al-Hajjaji, T. Thüm, T. Runge, M. R. Mousavi, and I. Schaefer, “A Classification of Product Sampling for Software Product Lines,” in *Proceedings of the International Systems and Software Product Line Conference (SPLC)*. ACM, 2018, pp. 1–13.
12. R. Tartler, D. Lohmann, C. Dietrich, C. Egger, and J. Sincero, “Configuration Coverage in the Analysis of Large-Scale System Software,” *Operating Systems Review*, vol. 45, no. 3, pp. 10–14, 2011.
13. J. Oh, D. S. Batory, M. Myers, and N. Siegmund, “Finding Near-Optimal Configurations in Product Lines by Random Sampling,” in *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2017, pp. 61–71.
14. S. Chakraborty, K. S. Meel, and M. Y. Vardi, “A Scalable and Nearly Uniform Generator of SAT Witnesses,” in *Proceedings of the International Conference on Computer Aided Verification (CAV)*, 2013, pp. 608–623.
15. A. von Rhein, J. Liebig, A. Janker, C. Kästner, and S. Apel, “Variability-Aware Static Analysis at Scale: An Empirical Study,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 27, no. 4, pp. 18:1–18:33, 2018.
16. J. A. Pereira, M. Acher, H. Martin, and J.-M. Jézéquel, “Sampling Effect on Performance Prediction of Configurable Systems: A Case Study,” in *Proceedings of the International Conference on Performance Engineering (ICPE)*. ACM, 2020.
17. A. Grebhahn, N. Siegmund, and S. Apel, “Predicting Performance of Software Configurations: There is no Silver Bullet,” *Computing Research Repository (CoRR)*, 2019, available at https://arxiv.org/pdf/1911.12643.pdf.
18. N. Siegmund, M. Rosenmüller, M. Kuhlemann, C. Kästner, S. Apel, and G. Saake, “SPL Conqueror: Toward Optimization of Non-Functional Properties in Software Product Lines,” *Software Quality Journal*, vol. 20, no. 3-4, pp. 487–517, 2012.
19. D. H. Wolpert, “The Lack of a Priori Distinctions Between Learning Algorithms,” *Neural Computation*, vol. 8, no. 7, pp. 1341–1390, 1996.

Christian Kaltenecker is a PhD student at the Chair of Software Engineering at Saarland University, Germany, under the supervision of Prof. Sven Apel. He received a Bachelor degree in 2014 and a Master degree in 2016 from the University of Passau, Germany. His research interests include sampling of software configuration spaces, performance prediction, and performance evolution of configurable software systems. One of his key contributions is the distance-based sampling strategy, which is an efficient and promising sampling strategy for sampling configurations of highly configurable software systems for per-

formance prediction. Contact him at kaltenec@cs.uni-saarland.de.

Alexander Grebhahn is a Big Data Engineer at the adesso SE in Berlin, Germany. He received a Bachelor degree and a Master degree from the University of Magdeburg, Germany. Before, he worked for several years in the group of Prof. Sven Apel at University of Passau, Germany. His research interests include machine learning, configurable software systems, and high-performance computing. Contact him at Alexander.Grebhahn@adesso.de.

Norbert Siegmund holds the Chair of Software Systems at Leipzig University, Germany. Prof. Siegmund received his PhD with distinction in 2012 from the Otto-von-Guericke University Magdeburg. His research aims at the automation of software engineering by combining methods from software analysis, machine learning, and meta-heuristic optimization. His special interests include software product lines and configurable software systems, performance and energy optimization, and digitization based on MicroServices and Web technologies. He is author and co-author of more than 70 peer-reviewed scientific publications. He regularly serves in program committees of top-ranked international conferences and is in the review board of the renowned journal IEEE Transactions on Software Engineering. He is a founding member of the Java User Group Thuringia. Contact him at norbert.siegmund@informatik.uni-leipzig.de.

Sven Apel holds the Chair of Software Engineering at Saarland University, Germany. Prof. Apel received his PhD in Computer Science in 2007 from the University of Magdeburg, Germany. His research interests include software product lines, software analysis, optimization, and evolution, as well as empirical methods and the human factor in software engineering. He is the author or co-author of over a hundred peer-reviewed scientific publications. He serves regularly in program committees of top-ranked international conferences and he is a member of the editorial boards of IEEE Transactions on Software Engineering, IEEE Software, and Empirical Software Engineering. He was program-committee co-chair of the 31st International Conference on Automated Software Engineering (ASE) and the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE). Contact him at apel@cs.uni-saarland.de.